CryoGrid

Table of contents

Section 1: Quick start

Software requirements

CryoGrid is written in Matlab. Version 2018x or higher is required.

Known Limitations

Multi-tile (3D) functionality not yet implemented.

Quick start - model structure

CryoGrid is a simulation tool to calculate ground temperatures and volumetric water/ice contents (as well as salt concentration, etc. depending on the selected SUBSURFACE classes) in single-tile (1D) stratigraphies. Multi-tile (3D) models can be realized by coupling several 1D stratigraphies.

A stratigraphy is realized by stacking one or several SUBSURFACE classes, which each account for different physical processes. The different SUBSURFACE classes typically have specific state variables and model parameters and use different constitutive equations to calculate those variables.

The stratigraphy of SUBSURFACE classes is sandwiched between dedicated TOP and BOTTOM classes marking the top and bottom of the stratigraphy.

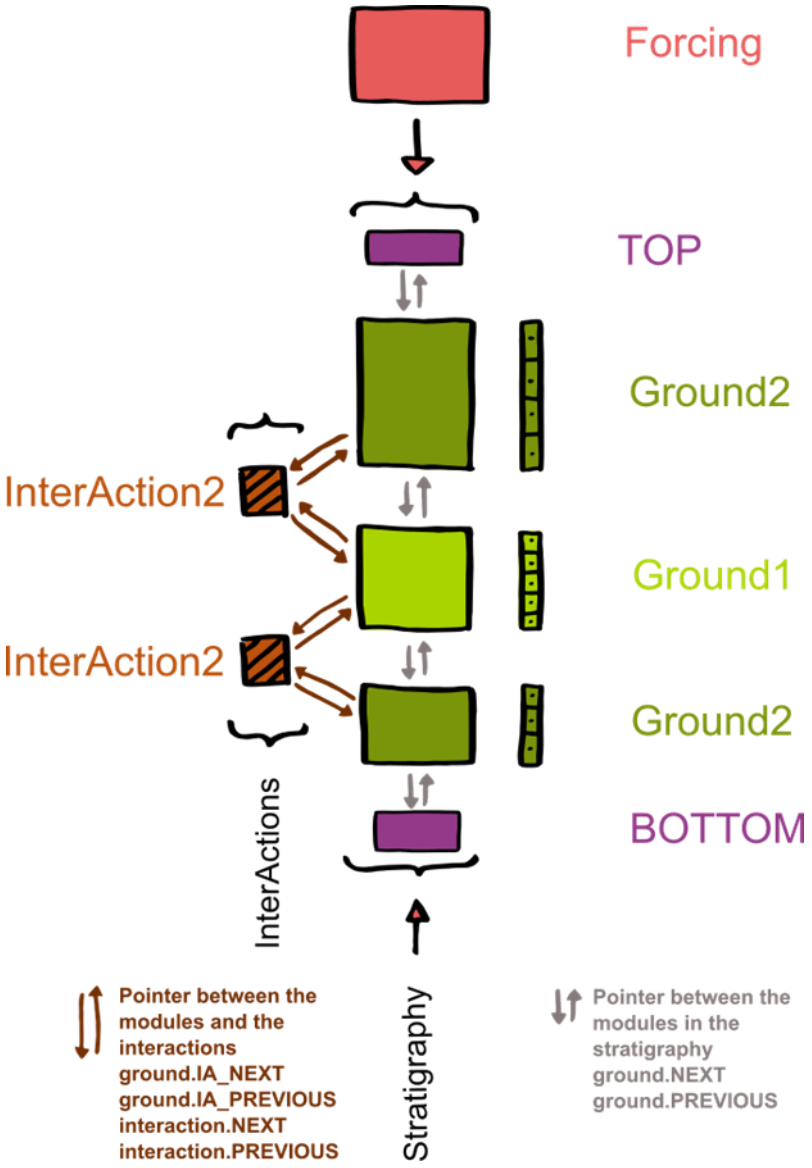

Fig. 1: Example of a CryoGrid stratigraphy, consisting of the SUBSURFACE classes Ground1 and Ground2 (example names) coupled by the INTERACTION class InterAction2 (also example name). This example assumes that Ground1 and Ground2 are compatible with each other irrespective of their order in the stratigraphy, which is not the case for all SUBSURFACE classes.

The interactions between the SUBSURFACE classes are realized with INTERACTION classes that compute fluxes across the boundaries of pairs of SUBSURFACE classes.
Forcing data driving the CryoGrid simulations are provided by dedicated FORCING classes which e.g. read the forcing data from a file and interpolate them for each timestamp.

Quick start - Terminology

- *parameter file*: spreadsheet used to set up a CryoGrid run, including definition of CryoGrid stratigraphy, forcing data, model grid, output format, etc. At this point, the parameter file can be set up in Excel and YAML format.
- *class*: Matlab class that realizes object-oriented programming in Matlab (https://se.mathworks.com/help/matlab/object-oriented-programming.html).
- *CryoGrid* SUBSURFACE *class*: Matlab classes with mandatory functions and properties which contain the defining equations for the time evolution for different elements of the Cryosphere, such as normal ground, a lake, or the snow cover. For clarity, SUBSURFACE class names start with a keyword GROUND, LAKE, SNOW, etc., depending on their functionality, and we refer to them as GROUND, LAKE, and SNOW classes in the following. Note that many of the classes in the code are not CryoGrid SUBSURFACE classes, but do something else. See Section 4: “Categories of CryoGrid classes”.
- *CryoGrid INTERACTION (IA) class*: Matlab class with mandatory functions and properties which contains the defining equations for interactions and fluxes between pairs of SUBSURFACE classes. Two SUBSURFACE classes are compatible with each other, if an IA class for them is available. See Fig. 1.
- *CryoGrid stratigraphy*: stratigraphy of vertically connected CryoGrid SUBSURFACE classes with INTERACTION classes between each pair of SUBSURFACE classes. See Fig. 1.
- *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 SUBSURFACE classes in the stratigraphy.

- *CryoGrid tile:* a model realization defined by its stratigraphy, model forcing, lateral interactions, etc.
- *TIER*: level structure for SUBSURFACE and INTERACTION class code to reduce code redundancy. TIER0, base level; TIER1: library level; TIER2 and 3: fully functional SUBSURFACE and INTERACTION class levels. See Section 2: CryoGrid code structure - TIER levels and inheritance.

Files and folder structure

- *run_CG_1D.m* : main file to be executed for single tile (1D) model runs
- **modules**: contains the model code (see Section 2: “CryoGrid code structure - Folder structure” for details)
- **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, for example provided as *.mat*-files. The forcing data must be provided in the format defined in the employed FORCING class.
- **analyze_display**: contains service functions to display model output, such as *read_display_out.m*, which displays the instance *out* produced by the OUT class *OUT_all.m and OUT_all_lateral.m*.

Quick Start - how to set up a new single tile (1D) run

1. Set the variable *init_format* in *run_CG_1D.m to 'EXCEL'.*
2. Change the runningNumber in *run_CG_1D.m* to *yourRunningNumber*
3. Create a new subfolder *yourRunningNumber* in the folder “results“
4. Create the files *CONSTANT_excel.xlsx* and the parameter file *yourRunningNumber.xlsx* in that folder. For each CryoGrid SUBSURFACE class, a fully functional parameter file named after the class is found in the subfolder *param_files_templates*. This can be copied to the folder *yourRunningNumber* and renamed to *yourRunningNumber.xlsx.* If several classes are needed, combine the parameter files in a single one. The file *CONSTANT_excel.xlsx* is also provided *in param_files_templates.*

5. Edit the parameter file *yourRunningNumber.xlsx*:
    a. Set all necessary *parameters* and *state variables* in *yourRunningNumber.xlsx.*
    b. List the desired LATERAL INTERACTION classes under the parameter “lateral_interactions” in the section “TILE_IDENTIFICATION” and for each class, set up the parameters in the respective LATERAL_IA sections.
    c. Select the desired OUT (i.e. output) class which determines the output formats and set the required parameters. See “Documentation of OUT classes”.
    d. Select the desired FORCING class which determines the input data, and prepare the input data files required for the FORCING class. See “Documentation of FORCING classes”.
6. Run main program *run_CG_1D_excel.m.* The CryoGrid output is stored in the folder *yourRunningNumber.* Output frequency and properties of the output data set is determined by the choice of the OUT-class (Step 5c above).
7. (optional) The folder analyze_display contains service functions for plotting CryoGrid output. Display your results with the appropriate script (needs to match your selected OUT class) in the *analyze_display* folder. If you want to analyze your results in a new matlab instance, you will need to run the PARAMETER_PROVIDER function to initialize the OUT class before you can open your results.

For running from parameter files in YAML format (file ending .yml), the procedure is similar, but in step 1 you need to set *init_format* in *run_CG_1D.m to ‘YAML’.*

Section 2: CryoGrid code structure

Folder structure

The CryoGrid model code is contained in the folder “modules”, in which it is organized as follows:

- **modules/TIER0**: base level: contains the basic class definitions for CryoGrid SUBSURFACE 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 SUBSURFACE class level: inherits from TIER1 library classes**,** contains fully functional CryoGrid classes
- **modules/TIER3:** second SUBSURFACE class level: inherits from TIER2 SUBSURFACE classes**,** contains fully functional CryoGrid classes. TIER3 in particular contains the SUBSURFACE classes (GROUND, LAKE, etc. classes) that can interact with a (SUBSURFACE) SNOW class.
- **modulesTIER**XX**/INTERACTION:** INTERACTION (IA) classes defining interactions and fluxes between pairs of SUBSURFACE classes, same TIER structure as for SUBSURFACE 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 process forcing data and provide it to the SUBSURFACE classes
- **modules/BUILDERS:** contains BUILDER classes which assemble the initial state including the stratigraphy for the CryoGrid model run. BUILDER classes depend on the different PROVIDER classes for obtaining initialization parameters.
- **modules/IO:** all functionality related to model input and output
- **modules/IO/PARAMETER_PROVIDER:** PROVIDER classes reading information from the parameter files (excel og yaml) or other custom backend data source, and provide the information to model classes when requested.

- **modules/IO/CONSTANT_PROVIDER:** provider classes reading the information from the CONSTANT file (or other backend data source) containing physical constants, and provide the information to model classes when requested.
- **modules/IO/FORCING_PROVIDER:** provider classes reading the information from the forcing file (or other backend data source), and provide the information to model classes when requested.
- **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:** depreciated folder, will be removed after code clean-up

TIER levels and inheritance

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

1. The TIER0 BASE classes provide the mandatory variables and mandatory functions for each class; in general, they are empty arrays/ empty functions. All BASE classes inherit from *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 class variables and functions. When defining a new variable (i.e. “property” in the Matlab class) in TIER1, it *must* have a unique name that is different from the variables in all other TIER1 (and TIER2/3) classes. In TIER1, classes comprise all functions related to specific processes, such as heat conduction, water fluxes, etc. There is clearly some redundancy between the functions in some of the classes, but the idea is to retain as much as possible of the code within each function to make it easier to understand. Examples of TIER1 classes are HEAT_CONDUCTION (containing functions related to conductive heat transfer), WATER_FLUXES (functions related to vertical water fluxes) or SEB (functions related to the surface energy balance).

3. TIER2 SUBSURFACE 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 additional variables. In TIER2, all classes are fully functional CryoGrid SUBSURFACE classes. SUBSURFACE class names start with a keyword GROUND, LAKE, SNOW, etc. to clarify which element of the Cryosphere they represent, followed by a set of further keywords describing the processes represented by the class (see Section 5: Detailed documentation).
4. TIER3 SUBSURFACE classes inherit from generally one TIER2 class, but add more functionality. At this stage, coupling to SNOW classes to represent the seasonal build-up and ablation of the snow cover is added in TIER3, but other applications might arise in the future.
5. INTERACTION classes follow a similar structure with TIER0 to TIER2. They are located in the subfolder INTERACTION under TIER1/2.

An example of the TIER structure is shown in Fig. 2. Note that there are many more TIER1/2/3 classes than those shown - the full list of TIER1 classes that a TIER2 inherits from is defined in the *class_def* statement (first line) of the TIER2 class.

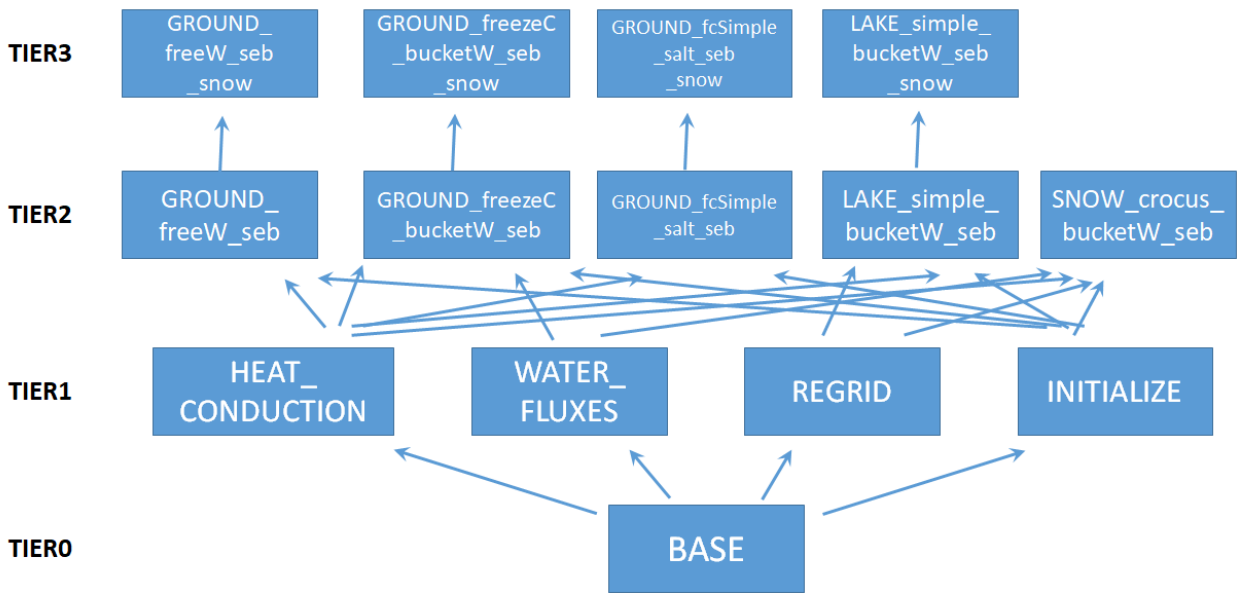

Fig. 2: Example of the inheritance structure of CryoGrid with the different TIER levels. Only a selection of available classes in the TIER1 to 3 levels is shown.

Basic structure of a subsurface stratigraphy

Stacking different CryoGrid SUBSURFACE classes to form a stratigraphy is facilitated by NEXT and PREVIOUS variables which represent pointers to the CryoGrid classes below and above (Fig. 1). Dedicated TOP and BOTTOM classes are used as confining classes for the stratigraphy. In a similar way, the interaction classes (e.g. for exchange of heat, salt, water) are addressed by pointers from the two SUBSURFACE classes: 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* (Fig. 1). In this way, the INTERACTION class can directly access variables in both involved SUBSURFACE classes.

A SUBSURFACE class can delete itself, move itself to a new position, and initialize and insert a new SUBSURFACE class. 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 SUBSURFACE and INTERACTION classes involved must be repositioned in the correct order. A SUBSURFACE class is deleted by the automatic garbage collection in Matlab when all pointers from still existing classes/variables are deleted.

Time integration and state variables

The purpose of CryoGrid is to compute the time evolution for state variables, such as ground temperature, water contents, etc. We distinguish between *prognostic* state variables for which a time derivative is calculated and which are then integrated in time to advance to the next timestep. For the prognostic state variables, CryoGrid uses the very simple time integration scheme “first-order forward Euler”, i.e. the new model state is computed as the old model state plus time derivatives times the model timestep. When the new model state of the prognostic state variables is obtained, *diagnostic* state variables are calculated from the prognostic variable by constitutional relationships. This calculation is instantaneous, i.e. does not depend on the employed timestep. Both prognostic and diagnostic variables are stored in fields of the variable STATVAR without distinction between the two. It is possible that a physical quantity (e.g. temperature) is a prognostic variable in some SUBSURFACE classes and diagnostic in others.

Stability and accuracy of the time integration is ensured by automatically selecting an appropriate timestep. Most importantly, the calculation of a suitable timestep is not accomplished by well-known stability criteria for the first-order forward Eule scheme, but explicitly takes the physics represented by the individual SUBSURFACE classes into account to ensure both a stable and accurate simulation.
Time integration follows a defined protocol with mandatory functions for the different steps (see Section 4: Categories of CryoGrid classes). First, the time derivatives (which in general are proportional to fluxes of matter or energy) of all prognostic state variables are calculated based on the old model state (defined by both prognostic and diagnostic state variables) and the forcing data of the respective timestamp, which determine the upper boundary condition for the uppermost SUBSURFACE class. Second, a suitable timestep is calculated, taking both the derivatives and the old model state into account. Third, the prognostic variables are integrated in time to obtain the new model state of the prognostic variables. Finally, the new model state of the diagnostic state variables is calculated based on the new model state of the prognostic variables.
If possible, prognostic state variables should be extensive properties, see here: https://en.wikipedia.org/wiki/Intensive_and_extensive_properties. In addition, bulk properties should be used as state variables, i.e. the total water volume in a grid cell [unit m^3], and the total internal energy [unit Joules]. Likewise fluxes/derivatives should be bulk fluxes, i.e. unit m^3/sec for water fluxes, and unit Joules/sec = Watt for energy fluxes.

Compatibility of SUBSURFACE classes

Compatibility of two SUBSURFACE classes (i.e. they can appear on top of to each other in the stratigraphy) is ensured by providing a dedicated INTERACTION class. In particular, INTERACTION classes compute fluxes across the boundaries between SUBSURFACE classes, which are required for computing time derivatives of state variables in the prognostic step of the time integration. For example, if two SUBSURFACE classes have temperature as state variables and the interaction between two modules is through conductive heat flow, the INTERACTION class computes the conductive heat flux between the adjacent grid cells of the two modules. If two SUBSURFACE classes feature different state variables, the

INTERACTION class must contain the necessary code to compute the correct fluxes for both involved classes. For example, if only one of the classes is hydrologically active, while water(+ice) contents are static for the other class, the INTERACTION class must provide a zero water flux boundary condition for the hydrologically active class to reflect the fact that water flow through the boundary is not possible.
The function *get_IA_class.m* in modules/TIER2/INTERACTION/ is used to determine the correct INTERACTION class for each pair of SUBSURFACE classes and create and initialize it. The choice of the correct INTERACTION class is accomplished by a compatibility matrix in *get_IA_class.m.* If you write a new SUBSURFACE or INTERACTION class, *get_IA_class.m and* the compatibility matrix must be updated. Note that the compatibility in general depends on the position of the two classes in the stratigraphy. For example, a (SUBSURFACE) SNOW class is only compatible with another (SUBSURFACE) GROUND class when it appears above the GROUND class in the stratigraphy. The same is true for hydrologically active SUBSURFACE classes, which are in general only compatible with non-hydrologically-active SUBSURFACE classes when they appear above it in the stratigraphy. Most of the compatibility limitations are determined by common sense (snow generally appears on top of the ground), or the applications of the model. If desired, it is possible to make two classes compatible by creating a dedicated INTERACTION class and updating *get_IA_class.m.*

Seasonal build-up and disappearance of snow cover

The snow cover is simulated by dedicated TIER2 SUBSURFACE classes with class names *SNOW_XX.m*. The basic principle is that the SNOW class is added between the TOP class and the uppermost SUBSURFACE class, when a snow cover is present (Fig. 3). Interactions between the SNOW and SUBSURFACE class are again realized by dedicated INTERACTION classes, which facilitate the exchange of energy, water, etc., depending on the capabilities of the involved classes.

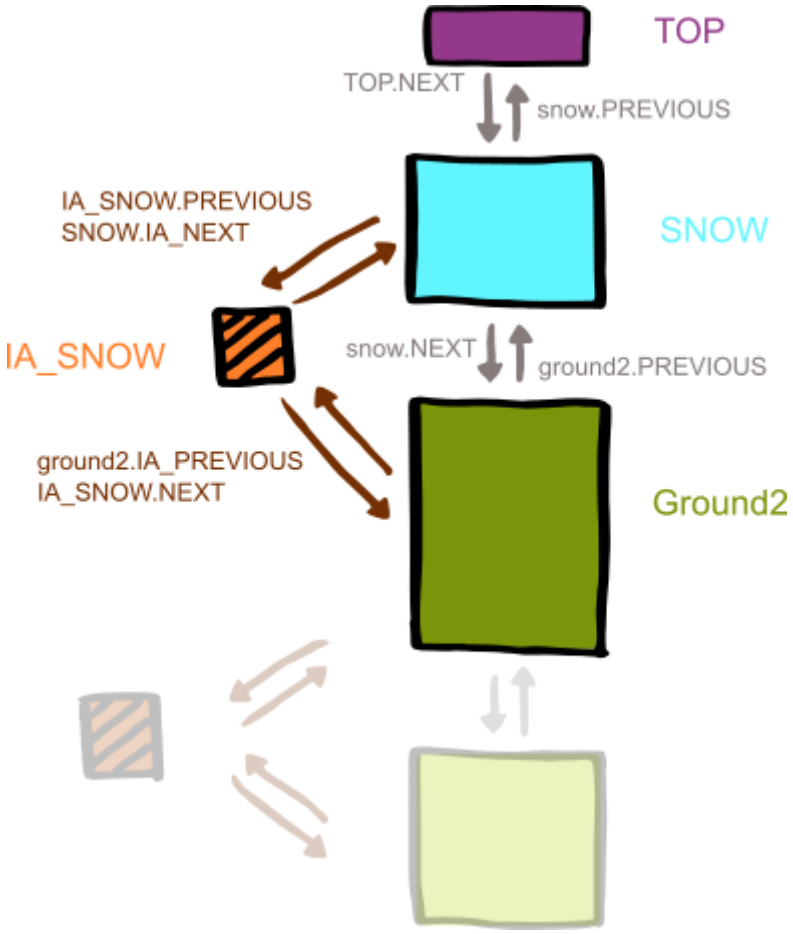

Fig. 3: CryoGrid stratigraphy with SNOW class added as regular SUBSURFACE class.

A significant problem for the numerical scheme is to handle very shallow snow covers, as this results in a small grid cell size and thus very small timesteps. In CryoGrid, the SNOW classes are attached and detached in two stages to the stratigraphy. After the first snowfall, the SNOW class is first added as a CHILD of the uppermost SUBSURFACE class (Fig. 4), then becoming a full class within the stratigraphy when enough snow has accumulated (Fig. 3). In the CHILD phase, both the physics and the interactions with the SUBSURFACE class are handled by special CHILD-phase functions, both in the SNOW class and partly also in the INTERACTION class coupling the SNOW to the SUBSURFACE class. When the snow depth is large enough that it can be handled without numerical problems the SNOW class and the associated INTERACTION class are simply rearranged, so that it becomes part of the regular stratigraphy (Fig. 3). The procedure is repeated in the opposite direction during snowmelt.

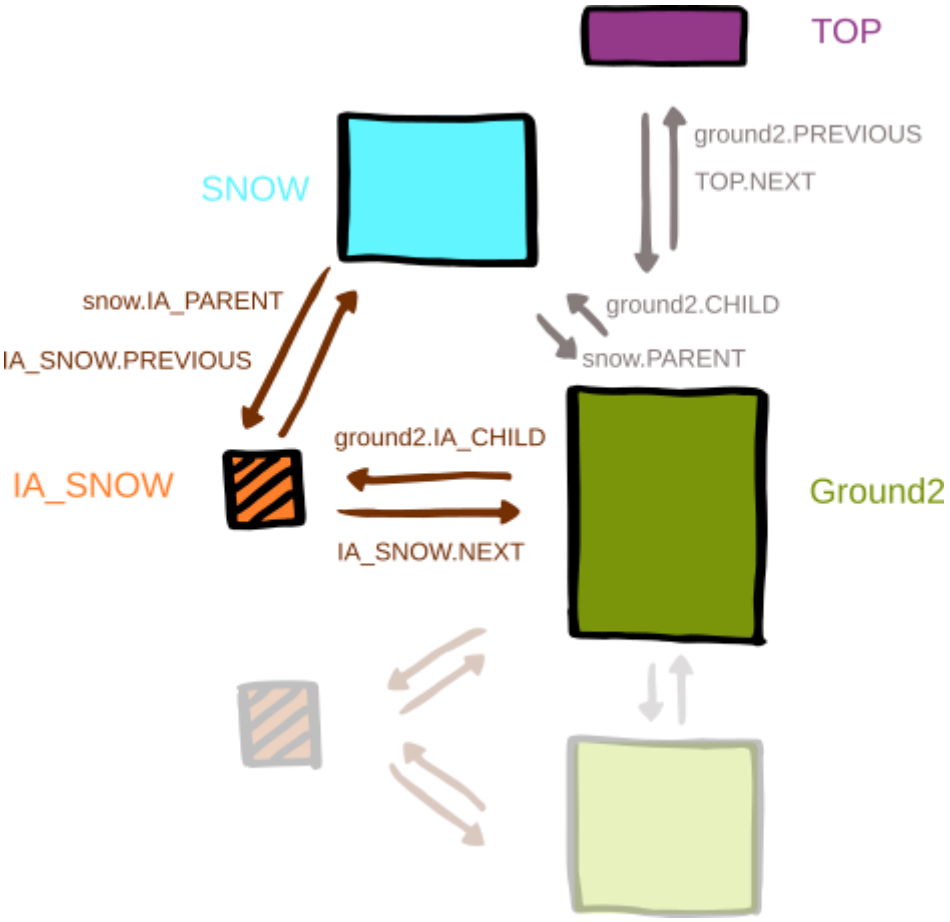

Fig. 4: CryoGrid stratigraphy with SNOW class added as a CHILD to the uppermost SUBSURFACE class Ground2.

Section 3: Adding new modules and functionality to CryoGrid

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!!

Class documentation

Every author should indicate her/his name and the date in the header of each class (same for files that are not a 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 to make it understandable. In addition, each CryoGrid class should be described in this manual. Every author is responsible for the documentation of new CryoGrid classes.

Not so Quick Start - how to create a new SUBSURFACE class

There is no definite scheme for compiling new SUBSURFACE classes, so this is only a rough guide. Discuss your plans with an experienced developer prior to setting off!

Most important: Design and create new classes without changing anything (!) in existing functions or classes! Even if only a minor change in e.g. a function in a TIER1 class would be necessary, add a new function to the TIER1 class instead of changing the existing function. This ensures that existing functionality is not affected by the new class. If changes to existing code become necessary, this must be discussed with all developers

1. Always develop and test a new SUBSURFACE class *myClass* without taking the snow cover into account. Normally (there are exceptions), the variable *snowfall* provided by the FORCING class should not be used at all. This means that development should start at the TIER2 level.
2. Choose the TIER2 class *oldClass* which is closest in functionality to the new class and copy + rename this class to *newClass*.
3. Change the class name in the *initialize* function. Add additional parameters (*PARA*), state variables (*STATVAR*) and constants (*CONST*) that must be initialized form the parameter and constant files in the appropriate private methods of *newClass* (i.e. *provide_PARA*, *provide_STATVAR* and *provide_CONST*). The variables in these functions must be declared empty ([]) and will be automatically populated during the initialization process. Dependent variables (i.e. variables computed from the variables populated during initialization) should not be declared here, but the appropriate code computing dependent variables should be added to the *finalize_init()* function.
4. Add statements to the mandatory functions in the new SUBSURFACE class. In general, additional functionality should be coded as functions in TIER1 classes, and then called as a single line statement in the TIER2 class. If adequate, make a new TIER1 class and add it to the list of classes after the *class_def* statement.
5. Test the new class without interactions with other classes, i.e. by making a parameter file in which the stratigraphy consists only of *newClass* (i.e. no other classes).
6. Compatibility with other classes: check the interaction classes used for *oldClass* and, if changes are necessary, copy, rename and modify them in the same way as for the SUBSURFACE class, paying attention to the TIER1 and TIER2 levels. Add *newClass* and potential new INTERACTION classes to the

function *get_IA_class()* in *TIER2/INTERACTION*. Carefully update the compatibility matrix in *get_IA_class.m*. Make sure you don't change anything related to existing SUBSURFACE class combinations!

7. Test all combinations with other classes in dedicated test examples.
8. Compatibility with snow cover: To add snow cover representation to *newClass*, copy one of the TIER3 classes (*oldClass_snow.m*) and rename it *newClass_snow.m.* Search and replace all occurrences of "*oldClass*" by "*newClass*", as well as *oldClass_snow with newClass_snow* in th *class_def* statement and the constructor. In general, all TIER3 classes will work as template, the only rule is not to mix *Xice* classes and non-*Xice* classes (if *myClass* features *Xice*, use an *oldClass* which has *Xice*). Redo the previous two steps for *newClass_snow,* but also ensure compatibility with the SNOW classes through dedicated INTERACTION classes.
9. Lateral interactions: IMPORTANT! All new code related to lateral fluxes needs to be added to TIER2 *newClass.m* and related TIER1 classes, not in the LATERAL classes. Also here, check the dedicated functions (in general *push_...* and *pull_…*) in *oldClass* and modify them. To ensure compatibility with a lateral class in *LATERAL/LAT1D* or *LAT3D*, check which SUBSURFACE class functions are called. In general, this will be in the functions *push* and *pull*. Add these functions to TIER2 *newClass.m*, placing the actual code in a suitable TIER1 class.
10. Comment the new code and add a description to "Section 5: Detailed Documentation" in this document.

Section 4: Categories of CryoGrid classes

In this section, the different types of CryoGrid classes are described. If a class can be initialized through a parameter file (in Excel or YAML format), the class keyword is provided in brackets.

SUBSURFACE classes (class keyword SUBSURFACE_CLASS)

In the SUBSURFACE classes (TIER 2 and 3), the time integration of prognostic state variables and computation of the dependent diagnostic variables is realized. *Prognostic* state variables are variables for which a time derivative is calculated and which are then integrated in time to advance to the next timestep. *Diagnostic* variables, on the other hand, are calculated from the prognostic variable by constitutional relationships. Both prognostic and diagnostic variables are stored in fields of the variable STATVAR without distinction. It is possible that a physical quantity (e.g. temperature) is a prognostic variable in some classes and diagnostic in others. As a general rule, extensive physical parameters should be used as prognostic variables, while intensive variables should rather be diagnostic variables. See https://en.wikipedia.org/wiki/Intensive_and_extensive_properties for a distinction between the two. Furthermore, the CryoGrid code largely uses bulk properties as state variables, i.e. the total water volume in a grid cell [unit m3], and the total internal energy [unit Joules]. While there are exceptions, implementing new SUBSURFACE classes should adopt this paradigm, especially as this ensures compatibility with existing INTERACTION and LATERAL classes.
Model parameters (stored as fields in PARA) must not change with time, otherwise they should be defined as state variables in STATVAR. Some physical quantities, such as soil water contents, can be static in some classes, but they are still defined as STATVAR and not as PARA to ensure compatibility with classes with variable water contents. Physical constants are stored as fields in the variable CONST. These should normally not be changed by the user. If changes are needed in specific cases, they should be defined as parameters in PARA.

Mandatory functions must be contained in eachSUBSURFACE class (no exceptions!)
Initialization:

1. Constructor method must inherit from the constructor method in TIER1 INITIALIZE.

2. *provide_PARA*(): defines all parameters, which are populated automatically by the PROVIDER/BUILDER classes based on the chosen provider backend data source. For example, in the case of an Excel parameter file backend (PARA_PROVIDER_EXCEL), the variable names in the parameter file must be identical to the variable names defined in provide_PARA().
3. *provide_STATVAR*(): defines all state variables some of which are populated automatically by the PROVIDER/BUILDER classes based on the chosen provider backend data source. Here again, in the case of e.g. an Excel parameter file backend (PARA_PROVIDER_EXCEL), the variable names in the parameter file must be identical to the variable names defined in provide_PARA().
4. *provide_CONST*(): defines all physical constants which are initialized automatically by the PROVIDER/BUILDER classes. The variable names defined in provide_CONST() must be identical to the variable names in the backend data source (e.g. constant file).
5. *finalize_init*(): class-specific final step of the initialization – after this function is executed, the class is fully initialized and time integration can start.

Time integration (functions called in this order!):

1. *get_boundary_condition_u():* calculates time derivatives of state variables (in general fluxes) related to the upper boundary of the stratigraphy. This function is only called for the uppermost SUBSURFACE class in the stratigraphy, but it must be implemented for all SUBSURFACE classes.
2. *get_boundary_condition_l():* calculates time derivatives of state variables (in general fluxes) related to the lower boundary of the stratigraphy. This function is only called for the lowermost SUBSURFACE in the stratigraphy.
3. *get_derivatives_prognostic():* calculates time derivatives of state variables due to fluxes between grid cells within a class.
4. *get_timestep():* calculates the optimal timestep for time integration, considering the time derivatives of state variables and stability and accuracy requirements.
5. *advance_prognostic():* integrates prognostic state variables in time, using the time derivatives and the timestep determined in the previous steps.
6. *compute_diagnostic_first_cell():* computes diagnostic variables when the SUBSURFACE class is the uppermost one in the stratigraphy. In particular, the Obukhov length related to atmospheric stability is computed here.

7. *compute_diagnostic():* computes diagnostic state variables based on the integrated prognostic state variables. After this function, all state variables must have been advanced to the next timestep.
8. *check_trigger():* checks if the conditions for trigger events are fulfilled, and if yes, executes the trigger. Triggers in particular comprise changes to the stratigraphy, such as creation/deletion of classes.
9. *penetrate_SW*(): not a mandatory function, but it must be available if the class shall be used in a stratigraphy together with classes that represent penetration of short-wave radiation, such as some SNOW and LAKE classes.

   Selected mandatory fields:

   CONST: physical constants that are in general not to be changed by the user.

   PARA: model parameters that are to be changed by the user, but should not change throughout the model run (otherwise it should become a state variable).

   STATVAR: state variables (prognostic and diagnostic), i.e. main model output

   TEMP: temporary variables, in particular the time derivatives of the prognostic state variables.

   PREVIOUS: pointer to the SUBSURFACE class above in the stratigraphy.

   NEXT: pointer to the SUBSURFACE class below in the stratigraphy.

   IA_PREVIOUS: pointer to the INTERACTION class which facilitates exchange with the SUBSURFACE class above in the stratigraphy.

   IA_NEXT: pointer to the INTERACTION class which facilitates exchange with the SUBSURFACE class below in the stratigraphy.

   If a SUBSURFACE class is compatible with SNOW class, it also contains the fields *CHILD* and *IA_CHILD*, which represent pointers to the *SNOW* class when it is still a CHILD and the INTERACTION class between the *SNOW* and the SUBSURFACE classes, respectively.

   SNOW classes feature a mandatory field PARENT, which during the CHILD phase points to the SUBSURFACE class by which it is created (generally the uppermost SUBSURFACE in the stratigraphy).

INTERACTION classes

INTERACTION classes handle the boundary conditions for all SUBSURFACE classes in the stratigraphy that share an inner boundary with another SUBSURFACE

class. INTERACTION classes have only one *mandatory function* that must be contained in each valid class:
get_boundary_condition_m(): calculates exchange fluxes between two SUBSURFACE classes, i.e. the part of the time derivative of prognostic state variable, that is determined by fluxes to/from the adjacent SUBSURFACE class. As an example, it calculates heat fluxes between two SUBSURFACE classes, so that heat conduction can act on the entire stratigraphy.
INTERACTION classes can contain additional functions, in particular functions related to trigger events which lead to the creation of new or annihilation of existing SUBSURFACE classes. All functionality which is specific for a combination of two classes *must* be coded in the respective INTERACTION class, not in one of the SUBSURFACE classses itself.
INTERACTION classes only contain pointers to the SUBSURFACE class above (PREVIOUS) and below (NEXT) as *mandatory variables*, i.e. they do not store state variables.

TOP and BOTTOM classes

The TOP and BOTTOM classes (named *Top.m* and *Bottom.m*) represent the ends of each stratigraphy. They are referenced by explicit variables TOP and BOTTOM in *run_CG_1D.m*, which makes it possible to loop over all SUBSURFACE classes in the stratigraphy. The TOP / BOTTOM classes in particular features the fields NEXT / PREVIOUS which represent pointers the SUBSURFACE class below / above. The TOP class has two additional variables: STORE is a pointer to the list of all classes which have been initialized by the PROVIDER/BUILDER classes, including those which are not (yet) part of the stratigraphy. Such classes can be inserted later (e.g. following trigger events by the function *check_trigger()*), as each class SUBSURFACE is connected to *Top* by a series of pointers, when following the stratigraphy upwards. Furthermore, the field LATERAL in the TOP class represents a pointer to the LATERAL class, so that also this one can be accessed from the stratigraphy.

LATERAL classes (class keyword: LATERAL)

LATERAL classes simulate lateral exchange of energy and matter (e.g. water or snow), either with external reservoirs (single-tile 1D simulations), or between different CryoGrid stratigraphies (multi-tile 3D simulations), each of which can again be coupled to external reservoirs (see Figs. 5 and 6). Lateral fluxes are only added/subtracted after a user-determined fixed interaction timestep, so they are not part of the for time integration scheme of the regular stratigraphy. This is especially relevant for multi-tile simulations which use parallel computing to distribute CryoGrid stratigraphies to different cores. To compute lateral fluxes between different CryoGrid stratigraphies, the cores exchange information through MPI, which requires a constant (lateral interaction) timestep. For MPI, see https://en.wikipedia.org/wiki/Message_Passing_Interface. In general, interaction timesteps should be significantly longer than the typical CryoGrid timestep, which means that the lateral fluxes should be small compared to the vertical fluxes within the CryoGrid stratigraphies. The lateral interaction timestep must be manually adjusted, seeking for a balance between runtime and model accuracy and stability. LATERAL classes do not have any functionality in modeling a specific flux or process, but they are rather containers for LATERAL INTERACTION classes (see below) in which the actual functionality is encoded and which are selected by the user (as a list under “lateral_interactions” in TILE_IDENTIFICATION in the parameter file). The user-selected LATERAL INTERACTION classes are stored in a cell array in the field IA_CLASSES of the LATERAL class. All LATERAL_IA classes feature a pointer field *PARENT* to the LATERAL class, through which they can access the variables of the LATERAL class. LATERAL classes run through a defined series of functions (coded in the LATERAL_IA classes) which are called for each LATERAL_IA class in the cell array. The three most essential steps (coded in LATERAL_IA classes, called by the LATERAL class) are:

1. *pull()* - pull the required information from the SUBSURFACE classes in the CryoGrid stratigraphy, combine them in a suitable way and assign them to fields (STATVAR, STATVAR2ALL, STATVAR_PRIVATE) in the LATERAL class.

2. pack, send, receive and unpack - pack information that must be exchanged with other cores in a single vector, send it out to the other cores, receive information vectors from the other cores and unpack them.
3. *get_derivatives()* - compute the fluxes/ time derivatives for the state variables for the entire CryoGrid stratigraphy.(i.e. not distinguishing between different SUBSURFACE classes).
4. *push()* - push the fluxes/ time derivatives back to the SUBSURFACE classes in the CryoGrid stratigraphy and add/subtract them for each SUBSURFACE class.

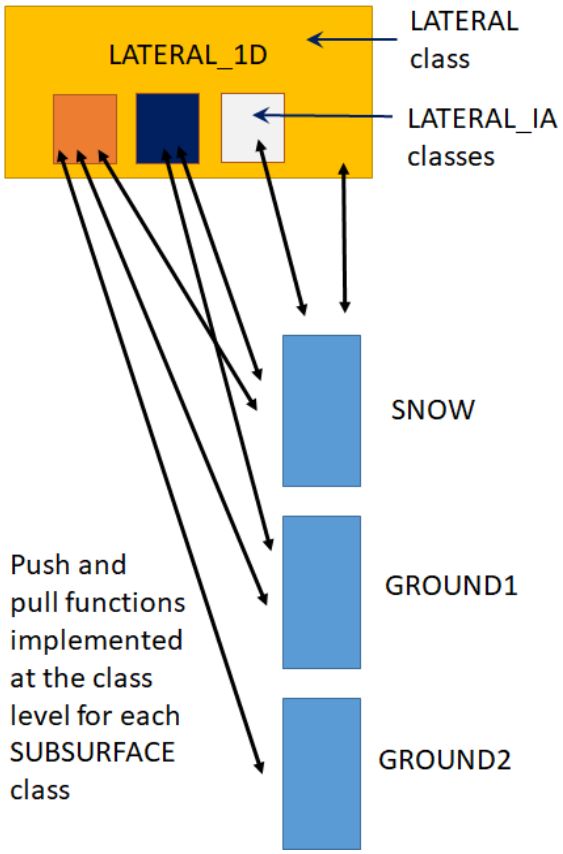

Fig. 5: Schematic setup of LATERAL and LATERAL_IA classes and their communication with the SUBSURFACE classes of the CryoGrid stratigraphy, for the case of single-tile (1D) runs using the LATERAL class LATERAL_1D. In this example, three different LATERAL_IA classes are employed. Note that LALTERAL_IA classes do not necessarily exchange information with all SUBSURFACE stratigraphies. As an example, LATERAL_IA classes computing lateral water fluxes do not exchange information with non-hydrologically-active SUBSURFACE classes.

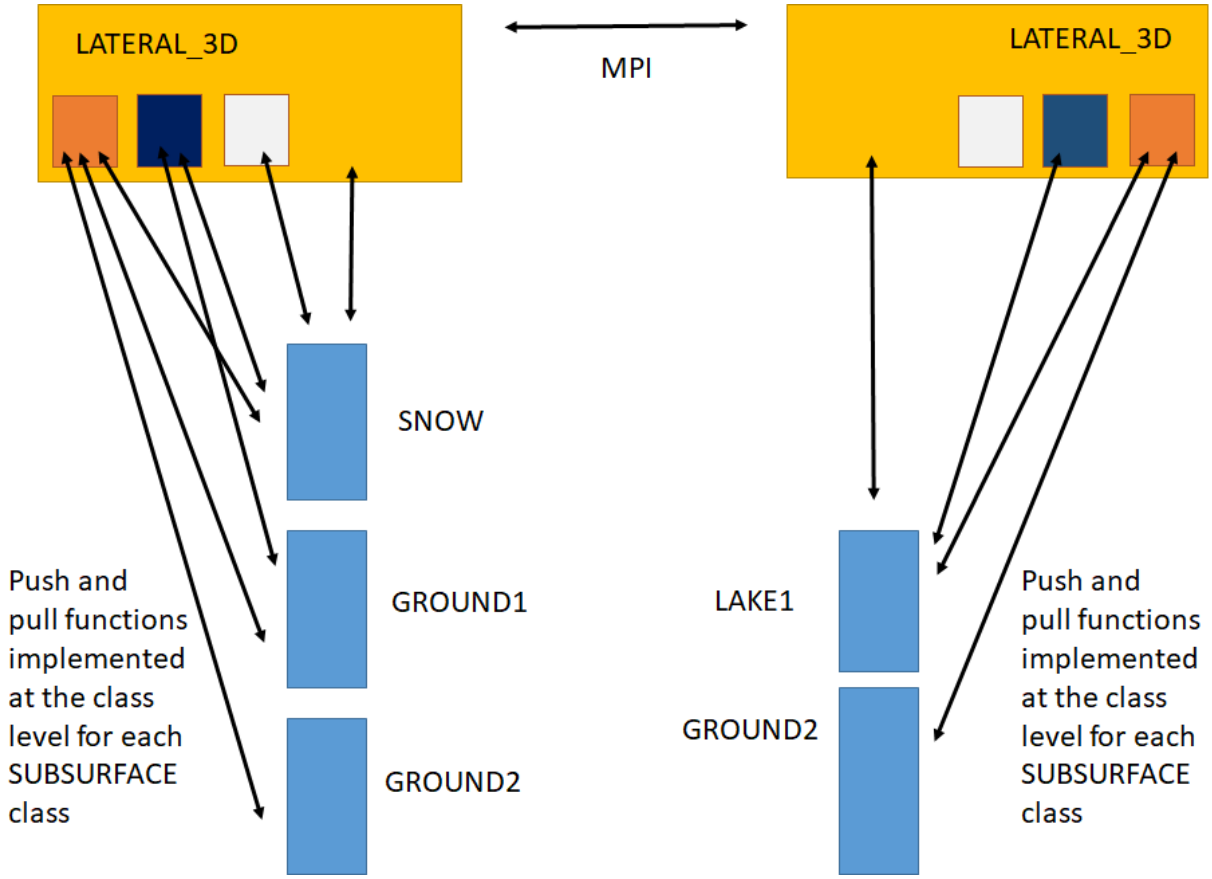

Fig. 6: same as Fig. 4, but for multi-tile (3D) runs, using the LATERAL classes LATERAL_3D. In this example, two CryoGrid stratigraphies are coupled together. The procedure using LATERAL and LATERAL_IA classes is designed to flexibly handle fluxes between different SUBSURFACE classes, irrespective of their positions in the CryoGrid stratigraphy.

Depending on the employed LATERAL class, this basic procedure can be strongly educed: for single-tile runs (class LATERAL_1D), all fluxes can be directly computed and added/subtracted in a single function, so only the *push()* function is required.

LATERAL INTERACTION classes (class keyword: LATERAL_IA)

Each LATERAL INTERACTION classes represents a specific physical process, such as lateral fluxes of heat, water and snow. Especially classes exchanging fluxes with

an external reservoir are specific for certain situations, e.g. for a water reservoir with fixed water table or a seepage face. In addition, LATERAL INTERACTION classes are typically specific for either single-tile or multi-tile runs (i.e. using either LATERA_1D or LATERAL_3D as LATERAL class).
LATERAL INTERACTION classes feature a series of mandatory functions which are called by the LATERAL class. In addition to service functions, the most important ones as *push()*, *pull()* and *get_derivative()* (see LATERAL class description for details). The *push()* and *pull()* functions are of particular importance, as they exchange information with the SUBSURFACE classes of the stratigraphy. However, they do not contain any functionality themselves, but (or each SUBSURFACE class in the CryoGrid stratigraophy) call *push_...* and *pull_…*-functions implemented in each SUBSURFACE CLASS that is compatible with the respective LATERAL_IA class.
For making a new SUBSURFACE CLASS compatible with an existing LATERAL_IA class, the only thing to do is to implement the required *push_…* and *pull_…* functions in the SUBSURFACE class
For implementing a new LATERA_IA class, the following steps should be followed (only a coarse recommendation!):

1. Always copy an existing LATERA_IA class as template and replace the mandatory functions by the new code. For single-tile runs, use a single-tile LATERAL_IA class as template (folder LAT1D), but use a multi-tile LATERAL_IA class (folder LAT3D) as template for multi-tile runs.
2. Implement the function *get_derivative().*
3. In each of the SUBSURFACE classes that is to be compatible with the new LATERAL_IA class implement *push_...* and *pull_...* functions, that are called by *push()* and *pull()* in the LATERAL_IA class. Make use of the TIER1 library level, which contains dedicated classes for for lateral processes, such as *WATER_FLUXES_LATERAL, HEAT_FLUXES_LATERAL and SNOW_FLUXES_LATERAL.*

Step 3 means that compiling a new LATERAL_IA class can be a lot of work if it should be compatible with a large number of SUBSURFACE classes. Still, it should be the goal to keep all LATERAL_IA classes compatible with all SUBSURFACE classes (at least for combinations for which this makes sense).

FORCING classes (class keyword: FORCING)

FORCING classes are a standardized way to provide the model forcing for CryoGrid simulations. In particular, each FORCING class must provide a function *myForcing = interpolate_forcing(t, myForcing)* which provides (generally interpolated) forcing data at an arbitrary MATLAB timestamp t. These forcing data are stored to the mandatory field *myForcing.TEMP,* from where they are accessed by the CryoGrid code.
FORCING classes must contain the mandatory field *myForcing.PARA* which contains parameters that are assigned from the parameter file in the standardized initialization procedure, in the same way as for SUBSURFACE classes. In general, FORCING classes also contain the field *myForcing.DATA* which stores the full time series of model forcing data.
FORCING classes are tailored to fit the requirements of SUBSURFACE classes: for SUBSURFACE classes computing the surface energy balance, for example, the FORCING class must provide all required driving variables (NOTE: this is currently the only option implemented). Therefore, implementing a new SUBSURFACE class might make it necessary to implement a new FORCING class, i.e. if the new SUBSURFACE class requires model forcing that cannot be provided by any of the existing FORCING classes.

OUT classes (class keyword: OUT)

OUT classes provide the users with a standardized way to define the desired model output and the format and frequency in which it is stored. In particular, each OUT class must provide a function *myOut = store_OUT(out, t, TOP_CLASS, …)* which computes output fields from the CryoGrid stratigraphy and stores them in a way specific for each OUT class. OUT classes contain the mandatory field myOut.PARA which contains parameters that are assigned from the parameter file in the standardized initialization procedure, in the same way as for SUBSURFACE classes.
The choice of the OUT class controls the size of the output files. For model testing, users might prefer to store all fields and variables of SUBSURFACE classes in the CryoGrid stratigraphy, while storing only a small subset of target variables might be

more adequate for long-term model runs. NOTE: at this stage, only a limited number of OUT classes is available.

GRID classes (class keyword: GRID)

GRID classes are used to provide the model grid for the entire CryoGrid stratigraphy, irrespective of the used SUBSURFACE classes. GRID classes contain the mandatory field myGrid.PARA which contains parameters that are assigned from the parameter file in the standardized initialization procedure, in the same way as for SUBSURFACE classes. NOTE: at this stage, only a single GRID class is available.

STRATIGRAPHY classes (class keyword: STRATIGRAPHY)

Two types of STRATIGRAPHY classes can be distinguished. First, STRATIGRAPHY_STATVAR classes provide a standardized way to initialize state variables (STATVAR) to the model grids in SUBSURFACE classes. More than one STRATIGRAPHY_STATVAR class can be used, if different state variables are to be initialized in different ways (for example some state variables via interpolation of values at known depths, others as layers with constant values). NOTE: when initializing with the CryoGrid parameter file, all state variables listed in the provide_STATVAR()-functions of all employed SUBSURFACE classes must be initialized through one or several STRATIGRAPHY_STATVAR classes. Second, STRATIGRAPHY_CLASSES provide a standardized way to initialize the CryoGrid stratigraphy, i.e. define the initial stratigraphy of SUBSURFACE classes. Furthermore, information on additional classes that are added to the stratigraphy during a run (e.g. a SNOW class) can be provided.

PROVIDER classes

PROVIDER classes have been implemented in order to abstract the process of obtaining parameters needed for the model to run away from the main code, and make it easier to implement new ways of obtaining model parameters. For example, the typical Excel parameter or constants files are easy to work with for manual modifications when working with single or few tile models. However, when working with thousands of tiles in regional or pan-arctic models, the user may need to obtain parameters from a database or structured file format like netcdf.

The PROVIDER classes expose a consistent and well-defined interface to the main code, which allows any model class to request parameters from the PROVIDER class in a consistent way, without knowing the type of backend data source that contains the values of the requested parameters.

BUILDER classes

BUILDER classes assemble the initial state of the model, including the stratigraphy of SUBSURFACE classes and vertical and lateral interactions, for a CryoGrid model run**.** Currently only one BUILDER class is available, TILE_BUILDER, which initializes a single 1D tile. Several additional BUILDER classes are anticipated, e.g.: 2D_TILE_PROFILE_BUILDER, 3D_TILE_GRID_BUILDER, and 3D_HEX_TILE_GRID_BUILDER.

BUILDER classes rely on PROVIDER classes to obtain information about model setup, and parameter and constant values.

For simple 1D model runs, tile setup information is available as a separate section in the parameter data source (e.g. excel or yaml file).

Currently there is no accepted definition of how 3D geometry and 3D lateral interactions should be defined, read and provided. 3D is not directly supported by the current backends, and thus a file format specifically for this purpose must be defined.

Section 5: Detailed documentation

Documentation of CryoGrid parameter files

For setting up a CryoGrid run, a parameter file in either Excel or YAML format is employed. The parameter file controls all aspects of the run and must be placed in the subfolder *yourRunningNumber* in the folder “results“ and named *yourRunningNumber.xlsx or yourRunningNumber.yml.*

The file is scanned for key words (e.g. SUBSURFACE _CLASS) and then for pre-defined ("allowed") parameter names. Empty lines and lines with dashes can be used to visually divide different classes and sections in the parameter file, but they are not read and have no impact on the run. The index besides the class name allows for different modules based on the same class, e.g. two SUBSURFACE modules with the same functionalities, but different parameter values, e.g.different albedos.The same structure is used for parameter setting of the FORCING, the GRID and the STRATIGRAPHY, using the dedicated classes for each of them.

Templates for parameter files for different SUBSURFACE classes are found in the folder *param_file_templates* from where they can be copied to the results folder and modified/adapted subsequently. We recommend to always start with these templates and also use them for trouble-shooting in case of errors. In the following, we document both mandatory and non-mandatory sections for the example of an Excel parameter file. The statements and sections can appear in any order, but we recommend to follow the standard provided by the templates for clarity and easier troubleshooting:.

1. Keyword TILE_IDENTIFICATION: provides meta-data related to the CryoGrid stratigraphy. In particular, in the field *lateral_interactions* a list of selected LATERAL INTERACTION classes is provided as a line vector between the keywords LIST and END. The TILE_IDENTIFICATION section is terminated by the keyword TILE_IDENTIFICATION_END.
2. Keyword OUT: The selected OUT class must be inserted in the cell below the keyword. The OUT section is terminated by the keyword OUT_END. In general, there should only be a single OUT section.

3. Keyword FORCING: The selected FORCING class must be inserted in the cell below the keyword. The FORCING section is terminated by the keyword FORCING_END. In general, there should only be a single FORCING section.
4. Keyword GRID: The selected GRID class must be inserted in the cell below the keyword. The GRID section is terminated by the keyword GRID_END. In general, there should only be a single GRID section.
5. Keyword STRATIGRAPHY: The selected STRATIGRAPHY class must be inserted in the cell below the keyword. The STRATIGRAPHY section is terminated by the keyword STRATIGRAPHY_END. There must be at least two STRATIGRAPHY sections, one of which must be of STRATIGRAPHY_CLASSES, which defines the stratigraphy of SUBSURFACE classes. The other stratigraphy section(s) are of type STRATIGRAPHY_STATVAR and provide the initial profile for the required state variables for the CryoGrid stratigraphy.
6. Keyword SUBSURFACE_CLASS: The selected SUBSURFACE class must be inserted in the cell below the keyword. The SUBSURFACE_CLASS section is terminated by the keyword CLASS_END. All SUBSURFACE classes that are listed in STRATIGRAPHY_CLASSES must have a separate SUBSURFACE_CLASS section. There can be additional SUBSURFACE_CLASS sections with unused SUBSURFACE classes, which are not read from the parameter file.
7. Keyword LATERAL_CLASS: The selected LATERAL class must be inserted in the cell below the keyword. The LATERAL_CLASS section is terminated by the keyword CLASS_END.There should only be a single GRID section, either *LATERAL_1D* or *LATERAL_3D*.
8. Keyword LATERAL_IA_CLASS: The selected LATERAL INTERACTION class must be inserted in the cell below the keyword. The LATERAL_IA_CLASS section is terminated by the keyword CLASS_END. All LATERAL INTERACTION classes that are listed in TILE_IDENTIFICATION must have a separate LATERAL_IA_CLASS section. There can be additional LATERAL_IA_CLASS sections with unused LATERAL INTERACTION classes, which are not read from the parameter file.

Documentation of SUBSURFACE classes

SUBSURFACE class names consist of a set of keywords which define the functionality and physical processes represented by the class. Example are GROUND_fcSimple_salt_seb_snow, SNOW_crocus_bucketW_seb, or LAKE_simple_bucketW_seb

Initial keyword: GROUND, SNOW; LAKE indicate the physical system represented by the class. This keyword should be as much as possible self-explanatory.

Freeze curve: *freeW*, *fcSimple*, *freezeC* describe the freezing characteric of the material.

1. *freeW*: freezing characteric of free water, phase change at 0 degree C.
2. *fcSimple*: simple freeze curve/soil freezing characteristic that can accomodate for the effect of salt, as well as (not implemented in any GROUND class yet) in principle for excess ice layers. For zero salt content, the freeze curve is computed as a linear function crossing through the temperature defined in the STATVAR deltaT (end of freezing) and 0 degrees (onset of freezing). For high non-zero salt contents, it is assumed that all salt freezes out into the remaining unfrozen water/brine, which establishes a functional relationship between water/ice contents and salt concentration in the brine. The freeze curve is then defined by freezing temperature of the brine. Between the two regimes, a smooth transition is guaranteed.
3. *freezeC*: Freezing characteristic similar to the original CryoGrid 3 model, based on the “freezing = drying” assumption and described in Dall’Amico et al. 2011 and Westermann et al., 2016. The STATVAR soil_type defines the shape of the freezing characteristic, using integer numbers for different soil types. At the moment, five different soil_type codes are supported: 1: sand; 2: silt: 3: clay: 4: peat; 5: water (i.e. approximation of free water, very large-pore ground material).

(Upper) Boundary condition of energy exchange: *seb* (surface energy balance) is currently the only option, but a temperature (i.e. Dirichlet) boundary condition (as for the old CryoGrid 2) could be implemented. Note that an adequate FORCING class providing the necessary forcing data must be selected!

1. *seb*: surface energy balance, as in the old CryoGrid 3 (Westermann et al., 2016).

Water fluxes: *bucketW* and *RichardsEq* represent different schemes governing vertical water fluxes. *RichardsEq* is currently in an experimental state and should not be used.

1. *bucketW*: Bucket scheme for water in which water moves downwards if the volumetric content in a grid cell exceeds the field capacity (i.e. STATVAR field_capacity) of the ground.The hydraulic conductivity is calculated as the saturated conductivity (a PARA at this stage!) times the liquid water saturation times a penalty factor for freezing ground as described in Dall'Amico et al. 2011. The hydraulic conductivity has only little impact on the results, but the employed equation should be checked and if necessary changed
2. *RichardsEq:* Vertical water fluxes are governed by Richards equation, with hydraulic conductivity and matric potential following the Mualem-vanGenuchten model with the modification for freezing soil as described in Dall'Amico et al. 2011.

Salt diffusion: *salt* indicates that salt diffusion is enabled. Note that advective transport of salt by water fluxes is not necessarily represented, this requires an additional "water fluxes" key word (not yet implemented together with salt as a class).

1. *salt*: salt diffuses between adjacent grid cells following the salt concentration gradient within the unfrozen water/brine.

Excess ground ice representation: *Xice* indicates that grid cells can contain excess ground ice.

1. *Xice*: the initial excess ground ice values are initially provided as the fraction of the non-excess ice ground material (i.e. soil matrix plus pore ice) - as an example, setting Xice to 1 in the parameter file indicates that half of the ground volume is filled by excess ice. The excess ice fraction features the freezing characteristic of free water, i.e. excess ground ice does not melt before a temperature of 0 degree C is reached. When melted, excess ground ice

becomes excess liquid water, which is routed upwards within the grid cells of the class. The uppermost grid cell can permanently contain a non-zero excess water content which indicates the presence of surface water. Depending on the functionality of the class, this can for example trigger the creation of a LAKE class which is added to the top of the stratigraphy.

Coupling to SNOW classes facilitating snow cover representation: *snow* indicates that the class automatically couples to SNOW classes, so that the seasonal snow cover can be represented in a transit way.

1. *snow*: This keyword is only used in TIER3 classes - compatibility with all SNOW classes is guaranteed by a standardized procedure which must be followed when compiling a new SUBSURFACE class (read “how to compile a new SUBSURFACE class”).

Additional keywords for SNOW classes:

Snow physics scheme: *simple*, *crocus* and *crocus2*

1. *simple*: This snow scheme features a constant initial snow density (specified by a PARA) and refreezing of melt water. The snow does not compact. It does not represent penetration of short-wave radiation. Together with the *bucketW* hydrological scheme, it is close to the original CryoGrid 3 snow scheme (only without penetration of short-wave radiation).
2. *crocus*: Snow scheme of the CROCUS snow model (Vionnet et al., 2012), featuring snow microphycics, compaction, variable albedo and penetration of short-wave radiation. Meltwater exceeding the available pore space within the snow cover is automatically removed.
3. *crocus2*: Same as *crocus*, but meltwater exceeding the available pore space is not removed, but pools up within the uppermost grid cell, so that the entire snow water equivalent is conserved. *crocus2* is designed to work with SUBSURFACE classes with *Xice* functionality.

Additional keywords for LAKE classes:

The LAKE classes have a different setup than GRUND and SNOW classes: there is a different class for the summer (unfrozen lake, indicated by the additional keyword

unfrozen) and winter (frozen lake, no additional keyword) states. The switch between the two classes is accomplished by automatic triggers, which annihilate the old and create the new class.

Lake physics scheme: *simple* indicates a similar lake physics as in Westermann et al., 2016. In the future, the more sophisticated *fLake* scheme may become available.

1. *simple*: fully mixed stratification for unfrozen conditions, stratified water column for ice-covered conditions.The ice cover always stays on top of the water column. Penetration of short-wave radiation in both water and ice is implemented, but should be refined.

Available SUBSURFACE classes:

GROUND_fcSimple_salt_seb and **GROUND_fcSimple_salt_seb_snow**

State variables that must be initialized:

Ground temperature - *T [unit degreeC]*

Volumetric water + ice contents - *waterIce [unitless, volumetric fraction]*

Volumetric mineral content - *mineral [unitless, volumetric fraction]*

Volumetric organic content - *organic [unitless, volumetric fraction]*

Total concentration of ions dissolved in the soil water phase - *saltConc [mol/m3]*

Onset of soil freezing - *deltaT [degree C]*

Compatible with LATERAL INTERACTION classes: LAT3D_HEAT

GROUND_freeW_seb and **GROUND_freeW_seb_snow**

State variables that must be initialized:

Ground temperature - *T [unit degreeC]*

Volumetric water + ice contents - *waterIce [unitless, volumetric fraction]*

Volumetric mineral content - *mineral [unitless, volumetric fraction]*

Volumetric organic content - *organic [unitless, volumetric fraction]*

Compatible with LATERAL INTERACTION classes: LAT3D_HEAT

GROUND_freeW_bucketW_seb and **GROUND_freeW_bucketW_seb_snow**

State variables that must be initialized:

Ground temperature - *T [unit degreeC]*

Volumetric water + ice contents - *waterIce [unitless, volumetric fraction]*

Volumetric mineral content - *mineral [unitless, volumetric fraction]*

Volumetric organic content - *organic [unitless, volumetric fraction]*

Field capacity in volumetric fraction of the entire volume, if volumetric water content exceeds the field capacity, this water is considered mobile - *field_capacity [unitless, volumetric fraction]*

Compatible with LATERAL INTERACTION classes: LAT3D_HEAT, LAT3D_WATER_UNCONFINED_AQUIFER, LAT3D_WATER_RESERVOIR, LAT3D_WATER_SEEPAGE_FACE, LAT_REMOVE_SURFACE_WATER, LAT_REMOVE_SUBSURFACE_WATER, LAT_SEEPAGE_FACE, LAT_WATER_RESERVOIR

GROUND_freezeC_seb and **GROUND_freezeC_seb_snow**

State variables that must be initialized:

Ground temperature - *T [unit degreeC]*

Volumetric water + ice contents - *waterIce* [unitless, volumetric fraction]

Volumetric mineral content - *mineral* [unitless, volumetric fraction]

Volumetric organic content - *organic* [unitless, volumetric fraction]

Field capacity in volumetric fraction of the entire volume, if volumetric water content exceeds the field capacity, this water is considered mobile - *field_capacity* [unitless, volumetric fraction]

Soil type 1: sand; 2: silt; 3: clay; 4: peat; 5: approximation of free water - *soil_type* [integer values from 1 to 5]

Compatible with LATERAL INTERACTION classes: LAT3D_HEAT

GROUND_freezeC_bucketW_seb and **GROUND_freezeC_bucketW_seb_snow**

State variables that must be initialized:

Ground temperature - *T [unit degreeC]*

Volumetric water + ice contents - *waterIce* [unitless, volumetric fraction]

Volumetric mineral content - *mineral* [unitless, volumetric fraction]

Volumetric organic content - *organic* [unitless, volumetric fraction]

Field capacity in volumetric fraction of the entire volume, if volumetric water content exceeds the field capacity, this water is considered mobile - *field_capacity* [unitless, volumetric fraction]

Soil type 1: sand; 2: silt; 3: clay; 4: peat; 5: approximation of free water - *soil_type* [integer values from 1 to 5]

Compatible with LATERAL INTERACTION classes: LAT3D_HEAT, LAT3D_WATER_UNCONFINED_AQUIFER, LAT3D_WATER_RESERVOIR, LAT3D_WATER_SEEPAGE_FACE, LAT_REMOVE_SURFACE_WATER, LAT_REMOVE_SUBSURFACE_WATER, LAT_SEEPAGE_FACE, LAT_WATER_RESERVOIR

GROUND_freezeC_bucketW_Xice_seb and
GROUND_freezeC_bucketW_Xice_seb_snow

State variables that must be initialized:

Ground temperature - *T [unit degreeC]*

Volumetric water + ice contents - *waterIce* [unitless, volumetric fraction]

Volumetric mineral content - *mineral* [unitless, volumetric fraction]

Volumetric organic content - *organic* [unitless, volumetric fraction]

Excess ice content in volumetric fraction relative to the excess ice free part of the ground - *Xice* [unitless, volumetric fraction]

Field capacity in volumetric fraction of the excess ice free part of the ground, if volumetric water content in excess ice free part of the ground exceeds the field capacity, this water is considered mobile - *field_capacity* [unitless, volumetric fraction]

Soil type 1: sand; 2: silt; 3: clay; 4: peat; 5: approximation of free water - *soil_type* [integer values from 1 to 5]

Note: excess ice melts at o degree C, while ground ice bound in the excess ice free part of the ground melts according to the soil freezing characteristic of the respective soil type.

Compatible with LATERAL INTERACTION classes: LAT3D_HEAT, LAT3D_WATER_UNCONFINED_AQUIFER, LAT3D_WATER_RESERVOIR, LAT3D_WATER_SEEPAGE_FACE, LAT_REMOVE_SURFACE_WATER, LAT_REMOVE_SUBSURFACE_WATER, LAT_SEEPAGE_FACE, LAT_WATER_RESERVOIR

LAKE_simple_seb and **LAKE_simple_seb_snow**

State variables that must be initialized:

Ground temperature - *T [unit degreeC]*

Volumetric water + ice contents - *waterIce* [unitless, volumetric fraction] - set this to 1

Volumetric mineral content - *mineral* [unitless, volumetric fraction] - set this to 0

Volumetric organic content - *organic* [unitless, volumetric fraction] -set this to 0

NOTE: at this stage, the class does not employ mineral and organic contents for anything so they should be set to 0 at all times. In the future iteration, these variables could be related to light penetration in the water column..

Compatible with LATERAL INTERACTION classes: LAT3D_HEAT

LAKE_simple_bucketW_seb and **LAKE_simple_bucketW_seb_snow**

State variables that must be initialized:

Ground temperature - *T [unit degreeC]*

Volumetric water + ice contents - *waterIce* [unitless, volumetric fraction] - set this to 1

Volumetric mineral content - *mineral* [unitless, volumetric fraction] - set this to 0

Volumetric organic content - *organic* [unitless, volumetric fraction] -set this to 0

NOTE: at this stage, the class does not employ mineral and organic contents for anything so they should be set to 0 at all times. In the future iteration, these variables could be related to light penetration in the water column and influenced e.g. by lateral advection.

Compatible with LATERAL INTERACTION classes: LAT3D_HEAT

SNOW_simple_seb

Called from the CryoGrid stratigraphy, so no initialization required.

Compatible with LATERAL INTERACTION classes:
LAT_REMOVE_SURFACE_WATER

SNOW_simple_bucketW_seb

Called from the CryoGrid stratigraphy, so no initialization required.

Compatible with LATERAL INTERACTION classes:
LAT_REMOVE_SURFACE_WATER, LAT_SEEPAGE_FACE,
LAT_WATER_RESERVOIR

SNOW_crocus_bucketW_seb

Called from the CryoGrid stratigraphy, so no initialization required.

Compatible with LATERAL INTERACTION classes: LAT3D_SNOW_CROCUS,
LAT3D_HEAT, LAT3D_WATER_UNCONFINED_AQUIFER,
LAT3D_WATER_RESERVOIR, LAT3D_WATER_SEEPAGE_FACE,
LAT_REMOVE_SURFACE_WATER, LAT_SEEPAGE_FACE,
LAT_WATER_RESERVOIR

SNOW_crocus2_bucketW_seb

Called from the CryoGrid stratigraphy, so no initialization required.

Compatible with LATERAL INTERACTION classes: LAT3D_SNOW_CROCUS, LAT3D_HEAT, LAT3D_WATER_UNCONFINED_AQUIFER, LAT3D_WATER_RESERVOIR, LAT3D_WATER_SEEPAGE_FACE, LAT_SEEPAGE_FACE, LAT_WATER_RESERVOIR

Documentation of LATERAL and LATERAL_IA classes

The LATERAL class *LATERAL_1D* is designed for single-tile (1D) runs. The following LATERAL INTERACTION classes can be used with *LATERAL_1D*:

LAT_REMOVE_SURFACE_WATER: Removes all water that has pooled up above the ground/snow surface, i.e. water that is in excess of the pore space of the first grid cell.

LAT_REMOVE_SUBSURFACE_WATER: Removes all water that is in excess of the field capacity.

LAT_SEEPAGE_FACE_WATER: simulates a seepage face between defined elevations through which water can drain.

LAT_WATER_RESERVOIR: simulates water exchange with a water reservoir at a defined elevation. Both in and outlow is possible, depending on the water table elevation in the CryoGrid stratigraphy.

The LATERAL class *LATERAL_3D* is designed for multi-tile (3D) runs. The following LATERAL INTERACTION classes can be used with *LATERAL_3D*:

LAT3D_WATER_UNCONFINED_AQUIFER: simulates water flow between two connected CryoGrid stratigraphies. Only water flow between the uppermost unconfined aquifers is accounted for.

LAT3D_WATER_RESERVOIR: simulates water exchange of a CryoGrid stratigraphies with a water reservoir at a defined elevation. Both in and outlow is possible, depending on the water table elevation in the CryoGrid stratigraphy. NOTE: *LAT3D_WATER_RESERVOIR* must be used together with *LAT3D_WATER_UNCONFINED_AQUIFER*, and it must be placed after LAT3D_WATER_UNCONFINED_AQUIFER in the list of LATERAL INTERACTION classes.

LAT3D_WATER_SEEPAGE_FACE: simulates a seepage face between defined elevations through which water can drain. NOTE: *LAT3D_WATER_SEEPAGE_FACE* must be used together with *LAT3D_WATER_UNCONFINED_AQUIFER*, and it must be placed after LAT3D_WATER_UNCONFINED_AQUIFER in the list of LATERAL INTERACTION classes.

LAT3D_HEAT: simulates conductive heat flow between two connected CryoGrid stratigraphies.

LAT3D_SNOW_CROCUS: simulates redistribution of snow due to wind drift between CryoGrid stratigraphies.

Documentation of OUT classes

***OUT_all*:** makes an identical copy of the stratigraphy at each output timestep, but stores the SUBSURFACE classes in a cell array in the variable STRATIGRAPHY. While it stores the raw model state including parameters and temporary variables, processing is required in order to analyze and display the output. Functions for this are found in the *analyze_display* folder.

OUT_all_lateral: identical to *OUT_all*, but also stores the state variables of the lateral classes (as separate cell array). This can for example be used to create runoff/discharge curves if a lateral water class is used.

Documentation of FORCING classes

***FORCING_seb*:** simple model forcing for SUBSURFACE classes computing the surface energy balance (keyword "*seb*"). The data must be stored in a Matlab ".mat" file with a struct *FORCING* with field "*data*", which contains the time series of the actual forcing data, e.g. *FORCING.data.Tair* contains the time series of air temperatures. Have a look at the existing forcing files in the folder "forcing" and prepare new forcing files in the same way. The mandatory forcing variables are air temperature (*Tair,* in degree Celsius), incoming long-wave radiation (*Lin*, in W/m2), incoming short-wave radiation (*Sin*, in W/m2), specific humidity (*q*, in kg water vapor / kg moist air), wind speed (*wind*, in m/sec), rainfall (*rainfall*, in mm/day), snowfall (*snowfall*, in mm/day) and timestamp (t_span, in Matlab time - increment 1 corresponds to one day). IMPORTANT POINT: the time series must be equally spaced in time, and this must be really exact. When reading the timestamps from an existing data set (e.g. an Excel file), rounding errors can result in small differences in the forcing timestep, often less than a second off. In this case, it is better to manually compile a new, equally spaced timestep in Matlab.

Planned future FORCING classes:

FORCING_seb_slopeAspect: same as FORCING_seb, but adjusts incoming short- and long-wave radiation to different aspects and slopes (Juditha Schmidt, UiO)

Potentially, a FORCING class implementing Toposcale (Fiddes et l., 2013) could be created, which directly reads netcdf files from for example ERA reanalysis.

Documentation of STRATIGRAPHY_CLASSES classes

STRAT_classes:currently the only STRATIGRAPHY_CLASSES class and used to provide information on the initial stratigraphy of SUBSURFACE classes, as well as other SUBSURFACE classes that are (potentially) needed during the run. The initial stratigraphy of SUBSURFACE classes is provided as a matrix with depth (i.e. the upper depth below the subsurface for each class - the last class extends to the bottom of the model domain) in the first column, the class name in the second and the class index in the third column. *Sleeping classes* are SUBSURFACE classes that

are not part of the initial stratigraphy, but become added by a trigger of another SUBSURFACE class during the run. An example is a LAKE class that gets added above a GROUND class when enough surface water has accumulated. Sleeping classes are added to the bottom of list of the initial stratigraphy classes, but leaving the depth field empty. The SNOW class to be used by the run is selected in the fields snow_classname and snow_class_index. If no SNOW class is used, these fields can be left empty. NOTE: each SUBSURFACE class listed in STRAT_classes must be initialized separately in the parameter file (see SUBSURFACE classes).

Documentation of STRATIGRAPHY_STATVAR classes

STRAT_layers: used to initialize the initial depth profile of state variable (stored in the field STATVAR) as layers with constant values. In the matrix, the variable names (that must match variables mentioned in the *provide_STATVAR*() functions of the SUBSURFACE classes) must be inserted in first line. The first row provides the start depth of each layer, and the last layer extends to the bottom of the model domain. NOTE: the state variables are in general provided in “human-readable” units. In the automatic initialization procedure, they are converted to the true unit of the state variable used in the simulations.

STRAT_linear: same as STRAT_layers, but values of the state variable for certain depths are provided. Between these depths, values are linearly interpolated.

Documentation of PROVIDER classes

PROVIDER classes have been implemented in order to abstract the process of obtaining parameters needed for the model to run away from the main code, and make it easier to implement new ways of obtaining model parameters. For example, the typical Excel parameter or constants file is easy to work with for manual modifications when working with single or few tile models. However, when working with thousands of tiles in regional or pan-arctic models, the user may need to obtain parameters from a database or structured file format like netcdf.

The PROVIDER classes expose a consistent and well-defined interface to the main code, which allows any model class to request parameters from the PROVIDER class in a

consistent way, without knowing the type of backend data source that contains the values of the requested parameters.

CryoGrid currently implements three types of PROVIDER classes:

- **PARA_PROVIDER**s: classes that provides values of parameters to the model
- **FORCING_PROVIDER**s: classes that provides forcing data to the model
- **CONST_PROVIDER**s: classes that provides values of constants to the model

The following PROVIDER class implementations are currently available:

- **PARA_PROVIDER_EXCEL**: a PROVIDER class that provides parameters from the classic Excel style parameter file
- **PARA_PROVIDER_YAML**: a PROVIDER class that provides parameters from a YAML text file. YAML is a text file format which is easy to programmatically read and write, and YAML parameter files may therefore be easily programmatically generated e.g. for spatially distributed models or sensitivity analyses that need to vary values of specific parameters.
- **FORCING_PROVIDER_MAT**: a PROVIDER class that provides forcing data stored in a MAT file.
- **CONST_PROVIDER_EXCEL:** a PROVIDER class that provides values of constants stored in an Excel file.

PROVIDER classes public interface:

New PROVIDER classes must be implemented in such a way that all necessary information to identify and access the backend data source is passed to the class during initialization. The currently implemented classes are all file based, and take a path and a filename as input arguments during initialization.

The PROVIDER classes must implement one public method, ***populate_struct***, which is called in order to obtain parameter values from the provider class.

The call signature is:

structure = populate_struct(structure, [possible additional arguments])

Where ***structure*** is a structure with named fields to be populated by the data source and returned to the caller. The PROVIDER class will iterate through each field in ***structure*** and populate the field value from the data source, based on the name of the field.

The fields of ***structure*** may themselves be structures with named fields. In such cases, the PROVIDER classes should interpret the field names of the substructure as column names of a table, and populate the substructure accordingly. Each field of the substructure will thus be populated with an array of values (corresponding to the rows of the table).

FORCING_PROVIDER and CONST_PROVIDER classes take no additional arguments beyond the structure to be populated.

The ***populate_struct*** method of PARA_PROVIDER classes takes three additional arguments:

structure = populate_struct(structure, section, name, index)

section is a string specifying which type of information to obtain (which section if the file, in case of excel or yaml parameter files, e.g. 'FORCING', 'OUT', 'SUBSURFACE_CLASS', etc.)
name is the name of the class for which to obtain parameters
index is the unique index (in the data source) of the class for which to obtain parameters (backends can define multiple copies of the same class with different parameter values, each must have a unique index)

These additional arguments are needed specifically for navigation of excel and yaml parameter files. Other future PARA_PROVIDER backend implementations may not need all these arguments, but should accept (and silently ignore) them for compatibility.

Currently, the PARA_PROVIDER class also define some helper functions:
get_class_list(section): *Get a list (order preserving) of all classes defined in the parameter backend datasource (e.g. excel or yaml file).*
get_class_id_by_name_and_index(section, name, index): Get the class id corresponding to the class name and unique class index passed. The class id is an integer describing the location at which the class definition occurs in the list of classes defined in the backend data source / parameter file (e.g. id=5 is the 5th class definied in the parameter file)

get_class_name_and_index_by_id(section, id): Get the class name and unique class index of the id'th class defined in the given section of the backend data source (e.g. excel or yaml file).

These methods are currently also required. But future development efforts should aim at decoupling the PROVIDER public interface more from the original excel file data structure, and implement it in a way that is meaningful also in other contexts (e.g. for a database backend or similar).