**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 single tile (1D) 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.

**Files and 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 basic class definitions 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 folder 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 in the stratigraphies stored in *out* must be initialized prior to calling *read\_display\_out.m*).

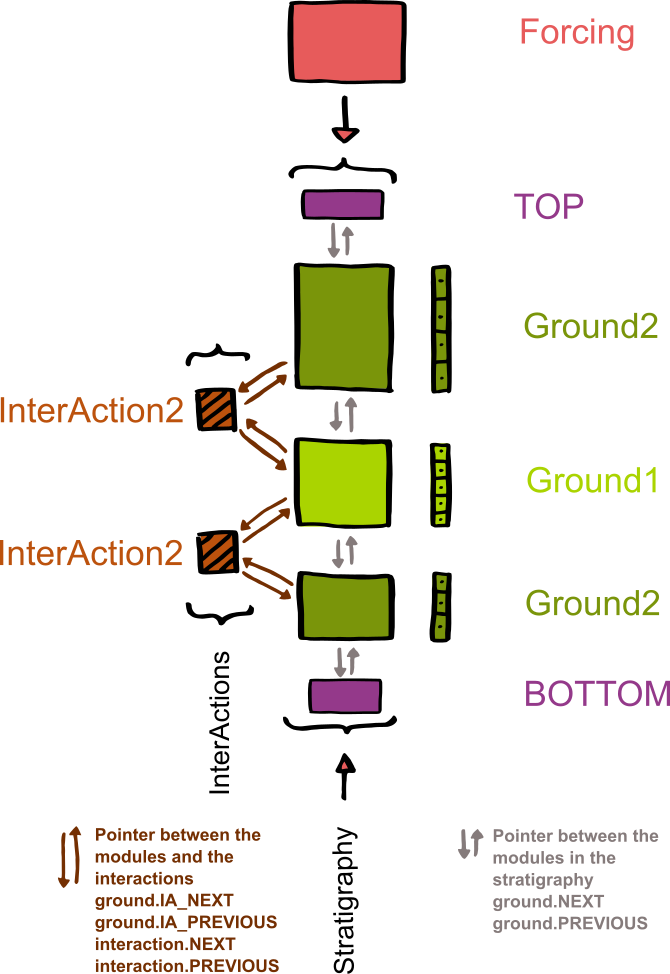
**TIER levels and inheritance**

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 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 variables and functions. When defining a new variable 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 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 (GROUND classes 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. INTERACTION classes follow a similar structure with TIER0 to TIER2. They are located in the subfolder INETRACTION under TIER1/2.

**Basic structure of the subsurface stratigraphy**

Stacking different CryoGrid GROUND classes to form a stratigraphy is facilitated NEXT and PREVIOUS variables which represent pointers to the CryoGrid classes below and above. 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 GROUND 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*. 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 GORUND class (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**

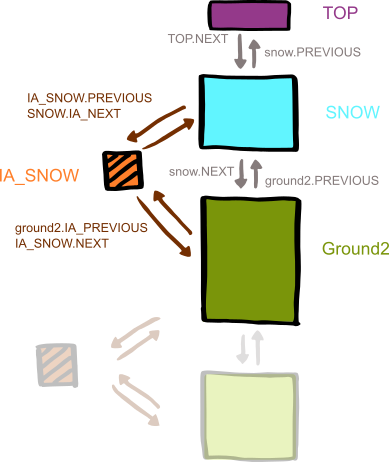
All GROUND classes can in principle handle whichever state variable or boundary condition. However, the state variables boundary conditions need to be compatible with the used interaction class(es) and the used forcing. For example, if you have temperature as a state variable and the interaction between 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. The choice of the correct INTERACTION class is accomplished by a compatibility matrix in *get\_IA\_class.m.* If you write a new GROUND or INTERACTION class, the compatibility matrix must be updated.

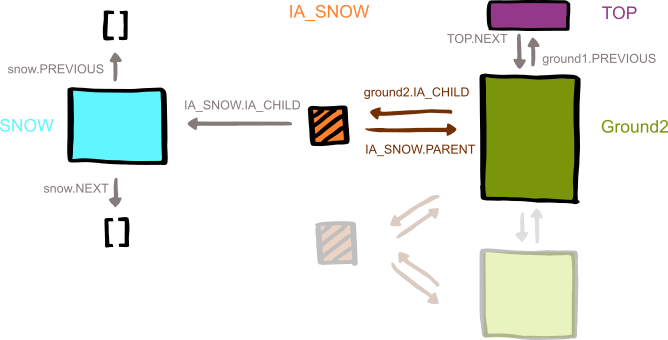
Note that the compatibility in general depends on the position of the two classes in the stratigraphy. For example, a snow class is only compatible with a GROUND class, when it is on top, i.e. appears above the GROUND class in the stratigraphy. The same is true for hydrologically active GROUND classes, 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 INTERACTION 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 grid 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)

SEBASTIAN: This picture is wrong/outdated and needs to be changed. CHILD points directly to the snow class, while IA\_SNOW points to the interaction class. In the interaction class, it is the normal pointers NEXT and PREVIOUS pointing to SNOW and GROUND. The SNOW class has an additional pointer PARENT pointing to the GROUND class. There is no pointer from the snow class to the IA\_class

**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 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 class.

**GROUND classes**

In the GROUND classes (TIER 2 and 3), the time integration of prognostic state variables and computation of the dependent diagnostic variables is realized. *Prognostic* state variables arevariables 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. Model parameters (stored as fields in PARA) must not change in 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, otherwise they should be defined as parameters in PARA.

*Mandatory functions* must be contained in each valid GROUND class (no exceptions!)

Initialization:

1. Constructor class must inherit from the constructor class in TIER1 INITIALIZE.
2. provide\_PARA(): defines all parameters, which are read automatically from the parameter file by the PROVIDER/BUILDER classes. 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 initialized automatically from the parameter file by the PROVIDER/BUILDER classes. Also here, 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 from the constants file by the PROVIDER/BUILDER classes. The variable names defined in provide\_CONST() must be identical to the variable names in the 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 if the class is the uppermost one in the stratigraphy.
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 if the class is the lowermost one 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 dertermined in the previous functions.
6. *compute\_diagnostic\_first\_cell():* computes diagnostic variables when the 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 necessarily a mandatory function, but 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 variables:*

*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:*  temprorary variables, in particular the time derivatives of the prognosticstate variables.

*PREVIOUS:*  pointer to the GROUND class above in the stratigraphy.

*NEXT:* pointer to the GROUND class below in the stratigraphy.

*IA\_PREVIOUS:* pointer to the INTERACTION class which facilitates exchange with the GROUND class above in the stratigraphy.

*IA\_NEXT:* pointer to the INTERACTION class which facilitates exchange with the GROUND class below in the stratigraphy.

If a GROUND class is compatible with classes representing snow cover (SNOW\_XX.m), it also contains the fields *CHILD* and *IA\_CHILD*, which represent pointers to the SNOW\_XX class when it is still a CHILD and the INTERACTION class between the SNOW\_XX and the GROUND classes, respectively.

The SNOW\_XX classes feature a field PARENT, which during the CHILD phase points to the GROUND class.

**TOP and BOTTOM classes**

The TOP / BOTTOM classes *Top.m* / *Bottom.m* in particular features the variable NEXT / PREVIOUS which represent pointers the ground class below / above. The TOP / BOTTOM classes represent the ends of each stratigraphy which makes it possible to loop over all classes in the stratigraphy. 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\_tirgger()*), as each class is connected to *Top* by pointers. Furthermore, LATERAL represents a pointer to the LATERAL class, so that also this one can be accessed form the stratigraphy.

**INTERACTION classes**

INTERACTION classes handle the boundary conditions for all modules in the stratigraphy that share an inner boundary with another module. INTERACTION classes have only one *mandatory function* that must be contained in each valid class:

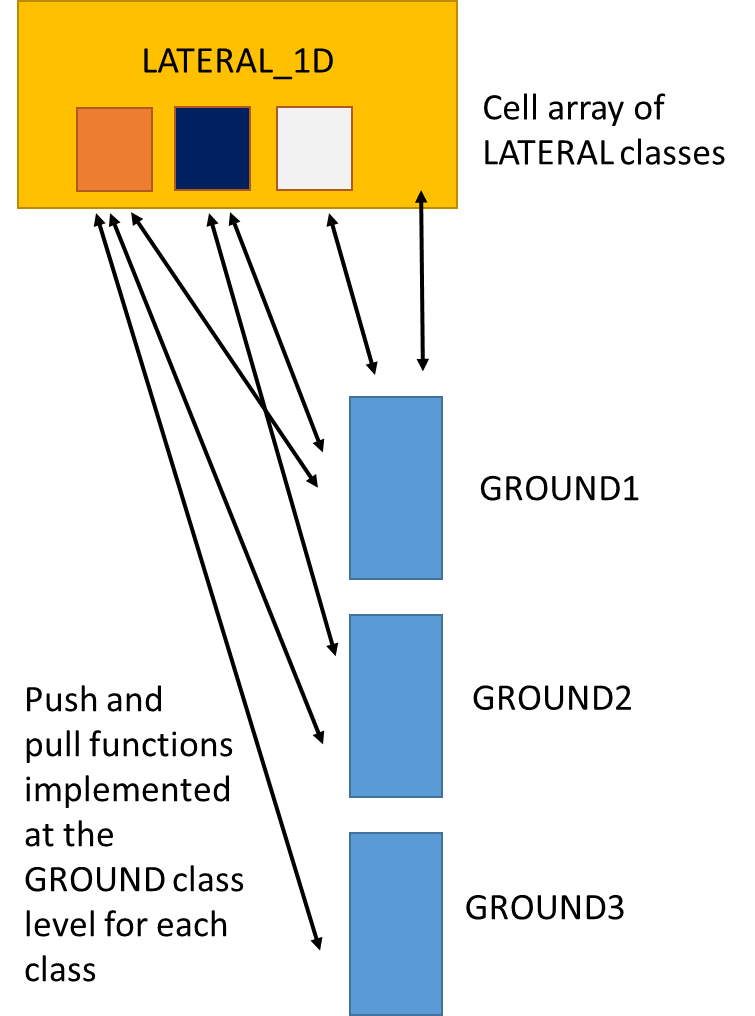
*get\_boundary\_condition\_m():* calculates exchange fluxes (i.e. the part of the time derivative of prognostic state variable, that is determined by fluxes between two classes) between two classes. As an example, it calculates heat fluxes between two classes.

INTERACTION classes can contain additional functions, in particular functions related to trigger events which lead to the creation of new classes or annihilation of existing 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 GROUND classses itself.

INTERACTION classes only contain pointers to the GROUND class above (PREVIOUS) and below (NEXT) as *mandatory variables*, i.e. they do not store state variables

**LATERAL classes**

Lateral classes



**FORCING classes**

**OUT classes**

**GRID classes**

**STARTIGRAPHY classes**

**PROVIDER classes**

**BUILDER classes**

**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 gets defined in *yourRunningNumber.xlsx* seperate for each used class.

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.

**Documentation of GROUND classes**

**GROUND\_fcSimple\_salt\_seb** and **GROUND\_fcSimple\_salt\_seb\_snow**

-fcSimple: simple freeze curve/soil freezing characteristic – 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 guearanteed.

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

-seb: surface energy balance as upper boundary condition for energy fluxes.