GLIMMER 1.5.1 Documentation

Magnus Hagdorn[1], Ian Rutt[2], Tony Payne[3] and Felix Hebeler[4]

April 19, 2010

[1]Magnus.Hagdorn@ed.ac.uk
[2]I.C.Rutt@bristol.ac.uk
[3]A.J.Payne@bristol.ac.uk
[4]fhebeler@geo.unizh.ch

Contents

Chapter 1

User Guide

1.1 Introduction

GLIMMER[1] is a set of libraries, utilities and example climate drivers used to simulate ice sheet evolution. At its core, it implements the standard, shallow-ice representation of ice sheet dynamics. This approach to ice sheet modelling is well-established, as are the numerical methods used. What is innovative about GLIMMER is its design, which is motivated by the desire to create an ice modelling system which is easy to interface to a wide variety of climate models, without the user having to have a detailed knowledge of its inner workings. This is achieved by several means, including the provision of a well-defined code interface to the model[2], as well as the adoption of a very modular design. The model is coded almost entirely in standards-complient Fortran 95, and extensive use is made of the advances features of that language.

1.1.1 Overview

GLIMMER consists of several components:

- **GLIDE: G**eneral **L**and **I**ce **D**ynamic **E**lements: the core of the model. This component is the actual ice sheet model. GLIDE is responsible for calculating ice velocities, internal ice temperature distribution, isostatic adjustment and meltwater production. GLIDE needs some representation of the climate to run, provided by a *driver* program. The user may write their own driver code, or may use one of the four supplied drivers (see section 1.1.2 below).

- **SIMPLE:** Simple climate drivers that implement the experiments of the first phase of the EISMINT project, with idealised geometry.

- **GLINT: GL**IMMER **Int**erface. Originally developed for the GENIE[3] Earth Systems Model, GLINT allows the core ice model to be coupled to a variety of global climate models, or indeed any source of time-varying climate data on a lat-long grid. An example driver is provided to illustrate the use of GLINT, which uses temperature and precipitation data to drive a positive degree day (PDD) mass-balance model.

- **EIS: E**dinburgh **I**ce **S**heet climate driver based on a parameterisation of the equilibrium line altitude, sea-level surface temperatures and eustatic sea-level change.

[1] *GLIMMER* was originally an acronym, reflecting the project's origin within GENIE. The meaning of the acronym is no longer important, however.
[2] The *API*, in computer-speak.
[3] Grid-ENabled Integrated Earth-system model

- **EISMINT3:** An implementation of a later part of the EISMINT project, concerning the modelling of the Greenland ice sheet.
- **GLUM: G**Limmer **U**seful **M**odules, various utility procedures used by the other components.
- Visualisation programs using GMT[4].

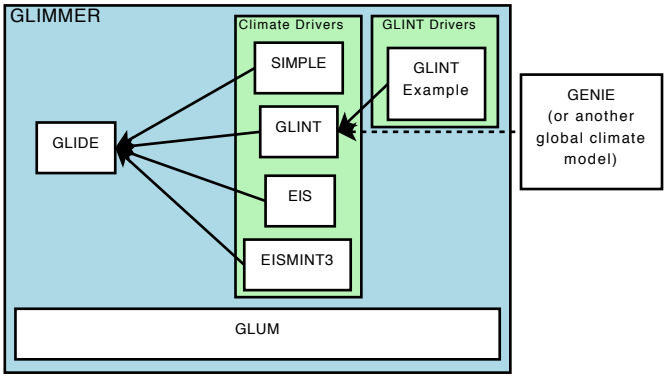

Figure 1.1: Relationship between the various GLIMMER components.

The relationship between the GLIMMER components is illustrated in Figure 1.1.

1.1.2 Climate Drivers

The core ice sheet model, GLIDE, is connected to the climate via the surface mass balance and temperature fields and (optionally) a scalar value for eustatic sea level. These drivers can be derived from simple assumptions, e.g. uniform mass balance or EISMINT tests, or from climate model output, e.g. GENIE or a regional climate model. These components, and how they relate to each other, are outlined in Figure 1.2.

1.1.3 Configuration, I/O and Visualisation

In general terms, each component is configured using a configuration file similar to Windows `.ini` files. At run-time, model configuration is printed to a log file.

2D and 3D data is read/written to/from netCDF files using the CF (Climate-Forecast) metadata convention[5]. NetCDF is a scientific data format for storing multidimensional data in a platform- and language-independent binary format. The CF conventions specify the metadata used to describe the file contents.

Many programs can process and visualise netCDF data, e.g. OpenDX, Matlab, IDL, etc. Additionally, the GLIMMER code bundle contains GMT scripts written in Python to visualise the output.

1.2 Getting and Installing GLIMMER

GLIMMER is a relatively complex system of libraries and programs which build on other libraries. This section documents how to get GLIMMER and its prerequisites, compile and install it. Please report problems and bugs to the GLIMMER mailing list.

[4] Generic Mapping Tools
[5] `http://www.cgd.ucar.edu/cms/eaton/cf-metadata/`

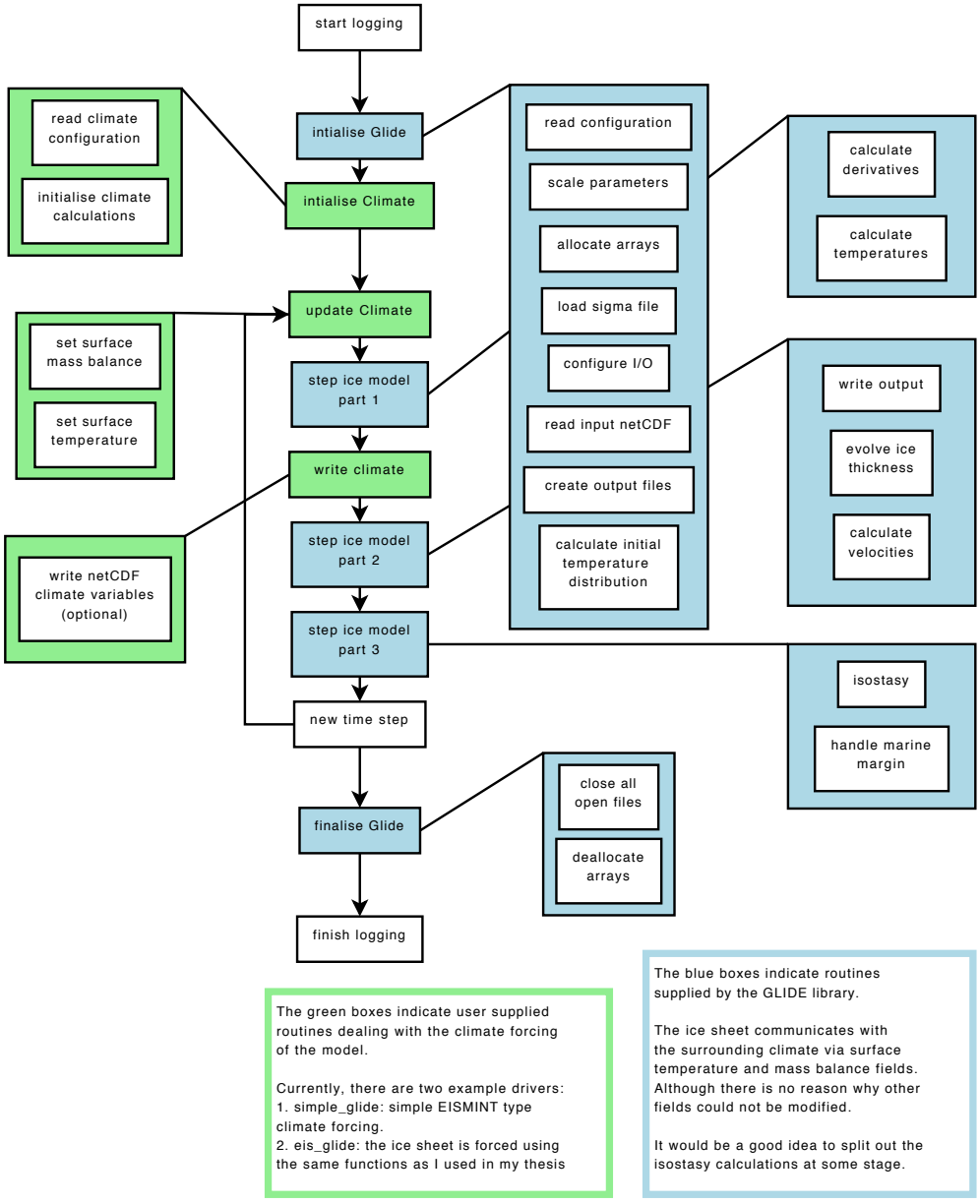

Figure 1.2: Outline of the GLIDE and Climate components.

1.2.1 Prerequisites

GLIMMER is distributed as source code; a sane build environment is therefore required to compile the model. On UNIX systems GNU make is suggested, since the Makefiles may rely on some GNU make specific features. There are two ways of getting the source code:

1. download a *released* version from the GLIMMER website[6], or
2. download the latest developers' version of GLIMMER and friends from NeSCForge using CVS.

For beginners, the latest release is recommended. More experienced users may want to try the CVS version, as it will have all the latest bug-fixes and new features.

In either case, a good f95 compiler is required. GLIMMER is known to work with the NAGware f95, Intel ifort and later versions of GNU gfortran compilers. GLIMMER does not compile with the SUN WS 7.0 f95 compiler due to a compiler bug. The current SUN f95 compiler might work, but has not been tested yet.

The other important prerequisite is the netCDF library, which GLIMMER uses for data I/O. You will most likely need to compile and install the netCDF library yourself, since the binary packages usually do not contain the Fortran 90 bindings which are used by GLIMMER.

Additional packages are required if you want to build GLIMMER from CVS. You need GNU autoconf and automake to generate the build system, as well as Python, which is used for analysing dependencies and for automatically generating parts of the code. Furthermore, the Python scripts rely on language features which were only introduced with Python version 2.3.

1.2.2 The GLIMMER Directory Structure

The following commands describe the setup if you use the `bash` shell. The setup works similarly for other shells. We suggest that you install glimmer and friends in its own directory, e.g. `/home/user/glimmer`. Assign the shell variable `$GLIMMER_PREFIX` to this directory, i.e. `export GLIMMER_PREFIX=/home/user/glimmer`. This directory will contain the following sub–directories:

`$GLIMMER_PREFIX/bin`	executables are installed in this directory. Set your path to include this directory, i.e. `export PATH=$PATH:$GLIMMER_PREFIX/bin`.
`$GLIMMER_PREFIX/include`	include and f95 module files will be installed in this directory. If you want to compile your own climate drivers set the compiler search path to include this directory.
`$GLIMMER_PREFIX/lib`	the libraries get installed here. Set your linker to look in this directory for the GLIMMER libraries if you want to compile your own climate drivers.
`$GLIMMER_PREFIX/share`	data files get installed here.
`$GLIMMER_PREFIX/src`	this is the only directory you need to create yourself. Unpack the GLIMMER sources here.

1.2.3 Installing a Released Version of GLIMMER

Download the GLIMMER tarball from the GLIMMER site and unpack it in the `$GLIMMER_PREFIX/src` directory using

```
tar -xvzf glimmer-VERS.tar.gz
```

where `VERS` is the package version.

The package is then compiled using the usual GNU sequence of commands:

[6] `http://glimmer.forge.nesc.ac.uk`

```
./configure --prefix=$GLIMMER_PREFIX [other_options]
make
make install
```

The options and relevant environment variables are described in Table 1.1.

Variable	Description
`FC`	f95 compiler to be used
`FCFLAGS`	flags passed to the f95 compiler
`LDFLAGS`	linker flags
Option	**Description**
`--help`	print help
`--prefix=`*prefix*	the installation prefix, e.g. `GLIMMER`
`--with-netcdf=`*location*	prefix where the netCDF library is installed
`--with-blas=`*location*	extra libraries used to provide BLAS functionality. A built–in, non–optimised version of BLAS is used if this option is not used.
`--enable-doc`	build documentation.
`--enable-profile`	enable profiling of GLIMMER (see Sec. 1.2.5)
`--enable-restarts`	enable full restarts (see Sec. 1.2.6)

Table 1.1: Environment variables and `configure` options used by GLIMMER.

1.2.4 Installing from CVS

Revisions of GLIMMER are managed using CVS. You can download the latest development version of GLIMMER using the following sequence of cvs commands:

```
cvs -d:pserver:anonymous@forge.nesc.ac.uk:/cvsroot/glimmer login
cvs -z3 -d:pserver:anonymous@forge.nesc.ac.uk:/cvsroot/glimmer co glimmer
```

The cvs version does not include some automatically generated files. In order to be able to compile the cvs version you need the GNU autotools and python. The build scripts are generated by running

```
./bootstrap
```

in the `$GLIMMER_PREFIX/src` directory. The package is then configured and built as described in Section 1.2.3.

1.2.5 Profiling

If you run the `configure` script with the option `--enable-profile` you enable profiling of the model. By default times are integrated over 100 time steps. You can cheange this behaviour by setting the variable `PROFILE_PERIOD`. The timing data is written to the file `glide.profile` which contains 5 columns of data (see Table 1.2). A python script using the PyGMT library to

Column 1	total CPU time elapsed when data is written to file
Column 2	accumulated time spent on this block of calculations
Column 3	integer ID used to identify this block of calculations
Column 4	model year
Column 5	description of this block of calculations

Table 1.2: File format of profile data file.

visualise the profile is provided.

1.2.6 Restarts

GLIMMER provides *two* mechanisms for initialising the state of the model from results of a previous run, written to a file:

- **Hotstarts:** This is the simpler of the two mechanisms. A NetCDF file containing *hotstart* data may be written as part of the regular output from the model, along with other output files. The variables written to the hotstart file are limited to those describing the state of the ice sheet, such as thickness, temperature distribution, etc — only those that are necessary to initialise the model cleanly. The model may be initialised from any of the time-slices in the hotstart file during the usual initialisation sequence. For most applications, hotstarting should be entirely adequate.
- **Restarts:** Sometimes, however, it is desireable to be able to write the entire state of the model, including all temporary variables, accumulation arrays, etc, to file. The *restart* mechanism enables this to be done. The difference in this case is that the model doesn't need to be initialised in the usual way - its state is simply read from file, and the run continues as before. The implementation of this is complex, as it involves writing all elements of each derived type and all its sub-types to file. Consequently, full restarts are disabled by default at build-time, and must be enabled with the `--enable-restarts` option when the build is configured.

A full description of both methods are provided later in this manual.

1.3 GLIDE

GLIDE is the actual ice sheet model. GLIDE comprises three procedures which initialise the model, perform a single time step and finalise the model. The GLIDE configuration file is described in Section 1.3.1. The GLIDE API is described in Appendix B.2. The simple example driver explains how to write a simple climate driver for GLIDE. Download the example from the GLIMMER website or from CVS:

```
cvs -d:pserver:anonymous@forge.nesc.ac.uk:/cvsroot/glimmer login
cvs -z3 -d:pserver:anonymous@forge.nesc.ac.uk:/cvsroot/glimmer co glimmer-example
```

1.3.1 Configuration

The format of the configuration files is similar to Windows `.ini` files and contains sections. Each section contains key, values pairs.

- Empty lines, or lines starting with a `#`, `;` or `!` are ignored.
- A new section starts with the the section name enclose with square brackets, e.g. `[grid]`.
- Keys are separated from their associated values by a `=` or `:`.

Sections and keys are case sensitive and may contain white space. However, the configuration parser is very simple and thus the number of spaces within a key or section name also matters. Sensible defaults are used when a specific key is not found.

`[grid]`
Define model grid. Maybe we should make this optional and read grid specifications from input netCDF file (if present). Certainly, the input netCDF files should be checked (but presently are not) if grid specifications are compatible.
continued on next page

continued from previous page

`ewn`	(integer) number of nodes in x–direction
`nsn`	(integer) number of nodes in y–direction
`upn`	(integer) number of nodes in z–direction
`dew`	(real) node spacing in x–direction (m)
`dns`	(real) node spacing in y–direction (m)
`sigma_file`	(string) Name of file containing σ coordinates. Alternatively, the sigma levels may be specified using the `[sigma]` section decribed below. If no sigma coordinates are specified explicitly, they are calculated based on the value of `sigma_builtin`
`sigma_builtin *`	If sigma coordinates are not specified in this configuration file or using the `sigma_file` option, this specifies how to compute the sigma coordinates. **0** Use Glimmer's default spacing $\sigma_i = \frac{1-(x_i+1)^{-n}}{1-2^{-n}}$ with $x_i = \frac{\sigma_i - 1}{\sigma_n - 1}, n = 2.$ 1 Use evenly spaced layers 2 Use the spacing defined for Pattyn's model

`[sigma]`	
Define the sigma levels used in the vertical discretization. This is an alternative to using a separate file (specified in section `[grid]` above). If neither is used, the levels are calculated as described above. **This does not work in version 1.0.0 — a bugfix will be incorporated into v.1.0.2.**	
`sigma_levels`	(real) list of sigma levels, in ascending order, separated by spaces. These run between 0.0 and 1.0

`[time]`	
Configure time steps, etc. Update intervals should probably become absolute values rather than related to the main time step when we introduce variable time steps.	
`tstart`	(real) Start time of the model in years
`tend`	(real) End time of the model in years
`dt`	(real) size of time step in years
`ntem`	(real) time step multiplier setting the ice temperature update interval
`nvel`	(real) time step multiplier setting the velocity update interval

`[options]`	
Parameters set in this section determine how various components of the ice sheet model are treated. Defaults are indicated in bold.	
`ioparams`	(string) name of file containing netCDF I/O configuration. The main configuration file is searched for I/O related sections if no file name is given (default).
`temperature`	0 isothermal **1** full
`flow_law`	**0** Patterson and Budd 1 Patterson and Budd (temp=-10degC) 2 constant value, taken from default_flwa *

continued on next page

continued from previous page

`basal_water`	0	local water balance
	1	local water balance + const flux
	2	none
`marine_margin`	0	ignore marine margin
	1	Set thickness to zero if floating
	2	Set thickness to zero if relaxed bedrock is below a given depth
	3	Lose fraction of ice when edge cell
	4	Set thickness to zero if present-day bedrock is below a given depth
`slip_coeff`	**0**	zero
	1	set to a non–zero constant everywhere
	2	set constant where the ice base is melting
	0	$\propto$ basal water
`evolution`	**0**	pseudo-diffusion
	1	ADI scheme
	2	diffusion
	3 *	Higher-order incremental remapping
`vertical_integration`	**0**	standard
	1	obey upper BC
`topo_is_relaxed`	**0**	relaxed topography is read from a separate variable
	1	first time slice of input topography is assumed to be relaxed
	2	first time slice of input topography is assumed to be in isostatic equilibrium with ice thickness.
`periodic_ew`	**0**	switched off
	1	periodic lateral EW boundary conditions (i.e. run model on torus)
`periodic_ns`	**0**	switched off
	1	periodic lateral NS boundary conditions (i.e. run model on torus)
`hotstart`	Hotstart the model if set to 1. This option only affects the way the initial temperature and flow factor distribution is calculated.	

`[ho_options] *`

Parameters set in this section determine how various components of the higher-order extensions to the ice sheet model are treated. Defaults are indicated in bold. To enable higher-order computation, set diagnostic_scheme to something other than 0. The computed velocities will only be used prognistically, however, if an evolution scheme that can make use of them has been enabled (currently only incremental remapping)

`diagnostic_scheme`	**0**	No higher-order diagnostics
	1	Pattyn/Bocek diagnostic, computed on the ice grid
	2	Pattyn/Bocek diagnostic, computed on the velocity grid
	3	Payne/Price diagnostic

continued on next page

continued from previous page

`basal_stress_input`	**0**	Ice glued to the bed (beta field is all NaN)
	1	Beta field is 1/soft
	2	Beta field is 1/btrc
	3	Beta field is read from input NetCDF independent of shallow-ice sliding law
	4	Use slip ratio, described in ISMIP-HOM F (**?**)
`basal_stress_type`	**0**	Linear bed
	1	Plastic bed
`which_ho_source`	**0**	Use vertically averaged formulation of the shelf front source term
	1	Use a vertically explicit formulation of the shelf front source term (currently not working)
	2	Turn off the ice shelf front and treat those locations as a land margin instead
`guess_specified`	**0**	Use a model-defined initial guess (SIA for Pattyn/Bocek, zero for Payne/Price)
	1	Read the initial velocity guess from uvelhom and vvelhom
`include_thin_ice`	0	Do not include ice below the ice dynamics limit in the higher-order diagnostic
	1	Compute higher-order diagnostic for all ice, even ice below the ice dynamics limit
`which_ho_sparse`	**0**	Solve sparse linear system with LU-preconditioned biconjugate gradient method
	1	Solve sparse linear system with LU-preconditioned GMRES method
	2	Solve sparse linear system with UMFPACK (not always available)
`which_ho_sparse_fallback`	Specifies a sparse solver package to use if the package specified in `which_ho_sparse` fails. The options are the same, though setting to -1 disables the fallback (this is the default).	

The `[ho_options]` parameter `which_ho_efvs` provides different options for the Pattyn/Bocek (`diagnostic_scheme=1 or 2`) and the Payne/Price (`diagnostic_scheme=3`) implementations of the higher-order model. The Pattyn/Bocek (`diagnostic_scheme=1 or 2`) options are:

`which_ho_efvs`	**0**	Use full nonlinear viscosity
	1	Apply a linear viscosity

The `[ho_options]` parameters described below are used only with the Payne/Price implementation of the higher-order model (`diagnostic_scheme=3`).

`which_ho_efvs`	**0**	Use the effective strain rate when calculating the effective viscosity, where strain rates are calculated using velocity fields from the previous iteration.
	1	Use a constant value. This can be useful for debugging, and is essentially the same as setting n=1 except that the specified value to use for the viscosity is simple hardcoded into the subroutine 'findefvsstr' in 'glam_strs2.F90' as case(2).

continued on next page

continued from previous page

`which_ho_babc`	("which higher-order basal boundary condition") Note that all basal boundary conditions using the Payne/Price core are through some form of a betasquared sliding law, TauB = betasquared * u_b (where TauB is the basal traction and u_b is the rate of basal sliding). For example, if the chosen value for betasquared is >> the expected basal traction, sliding will be negligible, effectively enforcing a zero slip basal boundary condition. If, on the other hand, if betasquared << the basal traction, the sliding rate will be large. Other cases are discussed below. 0 specify some constant value, hard coded into glam_strs2.F90, into subroutine 'calcbetasquared' under "case(0)". Useful for debugging. 1 specify some simple pattern, also hardcoded into glam_strs2.F90, into subroutine 'calcbetasquared' under "case(1)". Useful for debugging. 2 The value of 'betasquared' is treated in such a way as to enforce sliding over a subglacial till with Coulomb-friction (plastic) rheology, analagous to the methods used in SSA models by Schoof (ref) and Bueler and Brown (ref). The yield stress is specified in the 2d (x,y) array 'minTauf'. This is essentially the same as assuming that betasquared is non-linear, with some dependence on the sliding velocity. Note that betasquared has units of Pa s/m. Here, we are setting betasquared equal to a given map of the yield stress (with units of Pa) divided by the magnitude of the sliding velocity from the previous iteration (with units of m/s). Thus, the units are consistent. The regularization constant "smallnum" (with units of velocity) avoids division by 0 in early iterations and enforces no (or negligible) slip for cases where u,v ¡ smallnum; the expression then reduces to betasquared = minTauf / smallnum, which for smallnum ¡¡ 1, will result in a very large value for betasquared. 3 assign betasquared in the pattern required for the circular ice shelf test case. 4 no slip everywhere in the domain (betasquared set to some very large value everywhere). **5** Use none of the "hardcoded" options. Instead, use the betasquared field that is passed in to the higher-order solver from main portion of the code (e.g. from an input netCDF file).

continued on next page

continued from previous page

`which_ho_resid`	Determines calculation method for velocity residual, which is defined as: $resid = \frac{\lvert vel_{k-1} - vel_k \rvert}{vel_k}$ where "k" is the iteration index (i.e. "k-1" refers to the velocity from the previous iteration) and "||" is the absolute value operator. **0** Report the maximum value of the residual over the whole domain. That is, do not halt the linear iterations until the maximum residual over the entire 3d velocity field falls below some tolerance. 1 The same as option 0 but omitting the basal velocities from the residual calculation. This can be useful in some cases where one is enforcing no slip basal boundary conditions over some portion of the domain by specifying a very large value for betasquared. In this case, while the basal velocities are essentially 0 everywhere, there may be a few locations where the "sliding" velocity jumps around from, e.g. 1e-10 to 1e-12 m/yr, which with option 1 would be reported as a residual of 100. The result w/o this fix is that the solution appears to NOT be converging when in fact it has converged. 2 Report the mean value of the residual over the whole domain. Similar to option 1, this is useful in some cases if/when one wants to confirm that the solution is converging over the majority of the domain, but is failing to converge for one or a few grid cells. Note that Another useful way of testing for this is to select option 1 and enable the debugging line just before the "return" statement in the subroutine "mindcrshstr" in "glam_strs2.F90". In addition to the residual, this line will report the coordinates (as k,i,j = z,x,y index) of the max residual. If the solution is not converging and the location of the max residual remains the same for a number of iterations, try setting "which_ho_resid = 2" to see if the mean residual converges. This is an indication that the solution may still be occurring over the majority of the domain (and, e.g., there is some problem with boundary conditions at the offending grid point that will not converge).

continued on next page

continued from previous page

`[parameters]`	
Set various parameters.	
`log_level`	(integer) set to a value between 0, no messages, and 6, all messages are displayed to stdout. By default messages are only logged to file.
`ice_limit`	(real) below this limit ice is only accumulated; ice dynamics are switched on once the ice thickness is above this value.
`marine_limit`	(real) all ice is assumed lost once water depths reach this value (for `marine_margin`=2 or 4 in `[options]` above). Note, water depth is negative.
`calving_fraction`	(real) fraction of ice lost due to calving.
`geothermal`	(real) constant geothermal heat flux.
`flow_factor`	(real) the flow law is enhanced with this factor
`hydro_time`	(real) basal hydrology time constant
`isos_time`	(real) isostasy time constant
`basal_tract_const`	constant basal traction parameter. You can load a nc file with a variable called `soft` if you want a specially variying bed softness parameter.
`basal_tract`	(real(5)) basal traction factors. Basal traction is set to $B = \tanh(W)$ where the parameters (1) width of the tanh curve (2) W at midpoint of tanh curve [m] (3) B minimum [$\mathrm{ma}^{-1}\mathrm{Pa}^{-1}$] (4) B maximum [$\mathrm{ma}^{-1}\mathrm{Pa}^{-1}$] (5) multiplier for marine sediments
`default_flwa` *	Flow law parameter A to use in isothermal experiments (flow_law set to 2). Default value is 10^{-16}.
`[isostasy]`	
Isostatic adjustment is only enabled if this section is present in the configuration file. The options described control isostasy model.	
`lithosphere`	**0** local lithosphere, equilibrium bedrock depression is found using Archimedes' principle 1 elastic lithosphere, flexural rigidity is taken into account
`asthenosphere`	**0** fluid mantle, isostatic adjustment happens instantaneously 1 relaxing mantle, mantle is approximated by a half-space
`relaxed_tau`	characteristic time constant of relaxing mantle (default: 4000.a)
`update`	lithosphere update period (default: 500.a)
`[projection]`	
Specify map projection. The reader is referred to Snyder J.P. (1987) *Map Projections - a working manual.* USGS Professional Paper 1395.	
`type`	This is a string that specifies the projection type (`LAEA`, `AEA`, `LCC` or `STERE`).
`centre_longitude`	Central longitude in degrees east
`centre_latitude`	Central latitude in degrees north
`false_easting`	False easting in meters
`false_northing`	False northing in meters

continued on next page

continued from previous page	
`standard_parallel`	Location of standard parallel(s) in degrees north. Up to two standard parallels may be specified (depending on the projection).
`scale_factor`	non-dimensional. Only relevant for the Stereographic projection.
`[elastic lithosphere]`	
Set up parameters of the elastic lithosphere.	
`flexural_rigidity`	flexural rigidity of the lithosphere (default: 0.24e25)

`[GTHF]`	
Switch on lithospheric temperature and geothermal heat calculation.	
`num_dim`	can be either `1` for 1D calculations or 3 for 3D calculations.
`nlayer`	number of vertical layers (default: 20).
`surft`	initial surface temperature (default 2°C).
`rock_base`	depth below sea-level at which geothermal heat gradient is applied (default: -5000m).
`numt`	number time steps for spinning up GTHF calculations (default: 0).
`rho`	The density of lithosphere (default: 3300kg m^{-3}).
`shc`	specific heat capcity of lithosphere (default: 1000J kg^{-1} K^{-1}).
`con`	thermal conductivity of lithosphere (3.3 W m^{-1} K^{-1}).

NetCDF I/O can be configured in the main configuration file or in a separate file (see `ioparams` in the `[options]` section). Any number of input and output files can be specified. Input files are processed in the same order they occur in the configuration file, thus potentially overwriting priviously loaded fields.

`[CF default]`	
This section contains metadata describing the experiment. Any of these parameters can be modified in the `[output]` section. The model automatically attaches a time stamp and the model version to the netCDF output file.	
`title`	Title of the experiment
`institution`	Institution at which the experiment was run
`references`	References that might be useful
`comment`	A comment, further describing the experiment

`[CF input]`	
Any number of input files can be specified. They are processed in the order they occur in the configuration file, potentially overriding previously loaded variables.	
`name`	The name of the netCDF file to be read. Typically netCDF files end with `.nc`.
`time`	The time slice to be read from the netCDF file. The first time slice is read by default.

`[CF output]`	
This section of the netCDF parameter file controls how often selected variables are written to file.	
`name`	The name of the output netCDF file. Typically netCDF files end with `.nc`.
`start`	Start writing to file when this time is reached (default: first time slice).
`stop`	Stop writin to file when this time is reached (default: last time slice).
`frequency`	The time interval in years, determining how often selected variables are written to file.

continued on next page

continued from previous page	
`variables`	List of variables to be written to file. See Appendix **??** for a list of known variables. Names should be separated by at least one space. The variable names are case sensitive. Variable `hot` selects all variables necessary for a hotstart.

1.4 Example Climate Drivers

GLIMMER comes with three climate drivers of varying complexity:

1. `simple_glide`: an EISMINT type driver
2. `eis_glide`: Edinburh Ice Sheet driver.
3. `libglint`: Interface to global climate data or model

These drivers are described in some detail here.

1.4.1 EISMINT Driver

Configuration

`[EISMINT-1 fixed margin]`	
EISMINT 1 fixed margin scenario.	
`temperature`	(real(2)) Temperature forcing $$T_{\text{surface}} = t_1 + t_2 d$$ where $$d = \max\{\lvert x - x_{\text{summit}}\rvert, \lvert y - y_{\text{summit}}\rvert\}$$
`massbalance`	(real) Mass balance forcing
`period`	(real) period of time–dependent forcing (switched off when set to 0) $$\Delta T = 10 \sin \frac{2\pi t}{T}$$ and $$\Delta M = 0.2 sin \frac{2\pi t}{T}$$

`[EISMINT-1 moving margin]`	
EISMINT 1 moving margin scenario.	
`temperature`	(real(2)) Temperature forcing $$T_{\text{surface}} = t_1 - t_2 H$$ where H is the ice thickness

continued on next page

continued from previous page	
`massbalance`	(real(3)) Mass balance forcing $$M = \min\{m_1, m_2(m_3 - d)\}$$ where $$d = \sqrt{(x - x_{\text{summit}})^2 + (y - y_{\text{summit}})^2}$$
`period`	(real) period of time–dependent forcing (switched off when set to 0) $$\Delta T = 10 \sin \frac{2\pi t}{T}$$ and $$M = \min\left\{m_1, m_2\left(m_3 + 100 sin \frac{2\pi t}{T} - d\right)\right\}$$

1.4.2 EIS Driver

Configuration

`[EIS ELA]`	
Mass balance parameterisation of the EIS driver. The Equilibrium Line Altitude is parameterised with $$z_{\text{ELA}} = a + b\lambda + c\lambda^2 + \Delta z_{\text{ELA}},$$ where λ is the latitude in degrees north. The mass balance is then defined by $$M(z^*) = \begin{cases} 2M_{\max}\left(\frac{z^*}{z_{\max}}\right) - M_{\max}\left(\frac{z^*}{z_{\max}}\right)^2 & \text{for} \quad z^* \le z_{\max} \\ M_{\max} & \text{for} \quad z^* > z_{\max} \end{cases},$$ where z^* is the vertical distance above the ELA.	
`ela_file`	name of file containing ELA variation with time, Δz_{ELA}
`ela_ew`	name of file containing longitudinal variations of ELA field. File contains list of longitude, ELA pairs. The ELA perturbations are calculated by linearly interpolating values from file.
`ela_a`	ELA factor a
`ela_b`	ELA factor b
`ela_c`	ELA factor c
`zmax_mar`	The elevation at which the maximum mass balance is reached, $z_{\max}$ for marine conditions
`bmax_mar`	The maximum mass balance, $M_{\max}$ for marine conditions
`zmax_cont`	The elevation at which the maximum mass balance is reached, $z_{\max}$ for continental conditions
`bmax_cont`	The maximum mass balance, $M_{\max}$ for continental conditions

`[EIS CONY]`
The mass balance function can be modified with a continentality value. The continentality value is determined by finding the ratio between points below sea level and the total number of points in a circle of given search radius. This value between 0 (maritime) and 1 (continental) is used to interpolate between continental and maritime mass balance curves.

continued on next page

continued from previous page	
`period`	how often continentality is updated (default 500a).
`radius`	the search radius (default 600000m).
`file`	set to **1** to load continentality from file.

`[EIS Temperature]`	
Temperature is also assumed to depend on latitude and the atmospheric lapse rate.	
`temp_file`	name of file containing temperature forcing time series.
`type`	
	0 polynomial: $T(t) = \sum_{i=0}^{N} a_i(t)\lambda^i + bz$. where λ is the latitude.
	1 exponential: $T(t) = a_0 + a_1 \exp(a_2(\lambda - \lambda_0))$
`lat0`	λ_0 (only used when exponential type temperature)
`order`	order of polynomial, N (only used when using polynomial type temperature).
`lapse_rate`	lapse rate, b.

`[EIS SLC]`	
Global sea–level forcing	
`slc_file`	name of file containing sea–level change time series.

1.4.3 GLINT driver

Overview

GLINT is the most complex of the drivers supplied as part of GLIMMER. It was originally developed as an interface between GLIDE and the GENIE Earth-system model, but is designed to be flexible enough to be used with a wide range of global climate models. Perhaps the most distinctive feature of GLINT is the way it uses the object-oriented GLIDE architecture to enable multiple ice models to be coupled to the same climate model. This means that regional ice models can be run at high resolution over several parts of the globe, but without the expense of running a global ice model.

GLINT automates the processes required in coupling regional models to a global model, particularly the down- and up-scaling of the fields that form the interface between the two models. The user may specify map projection parameters for each of the ice models (known as *instances*), and choose one of several alternative mass-balance schemes to use in the coupling. The differing time-steps of global model, mass-balance scheme, and ice model are handled automatically by temporal averaging or accumulation of quantities (as appropriate). This is illustrated schematically in figure 1.3.

Prerequisites

If you plan to use GLINT, the following should be borne in mind:

- Global input fields must be supplied on a latitude-longitude grid. The grid does not have to be uniform in latitude, meaning that Gaussian grids may be used. Irregular grids (e.g. icosahedral grids) are not supported currently. The boundaries of the grid boxes may be specified; if not, they are assumed to lie half-way between the grid-points in lat-lon space.
- In the global field arrays, latitude must be indexed from north to south – i.e. the first row of the array is the northern-most one. Again, some flexibility might be introduced into this in the future.
- The global grid must not have grid points at either of the poles. This restriction is not expected to be permanent, but in the meantime can probably be overcome by moving the location of the polar points to be fractionally short of the pole (e.g. at 89.9° and -89.9°).

Initialising and calling

The easiest way to learn how GLINT is used is by way of an example. GLINT should be built automatically as part of GLIMMER, and we assume here that this has been achieved successfully.

Typically, GLINT will be called from the main program body of a climate model. To make this possible, the compiler needs to be told to use the GLINT code. Use statements appear at the very beginning of f90 program units, before even `implicit none`:

```
use glint_main
```

The next task is to declare a variable of type `glint_params`, which holds everything relating to the model, including any number of ice-sheet instances:

```
type(glint_params) :: ice_sheet
```

Before the ice-sheet model may be called from the climate model, it must be initialised. This is done with the following subroutine call:

```
call initialise_glint(ice_sheet,lats,lons,time_step,paramfile)
```

In this call, the arguments are as follows:

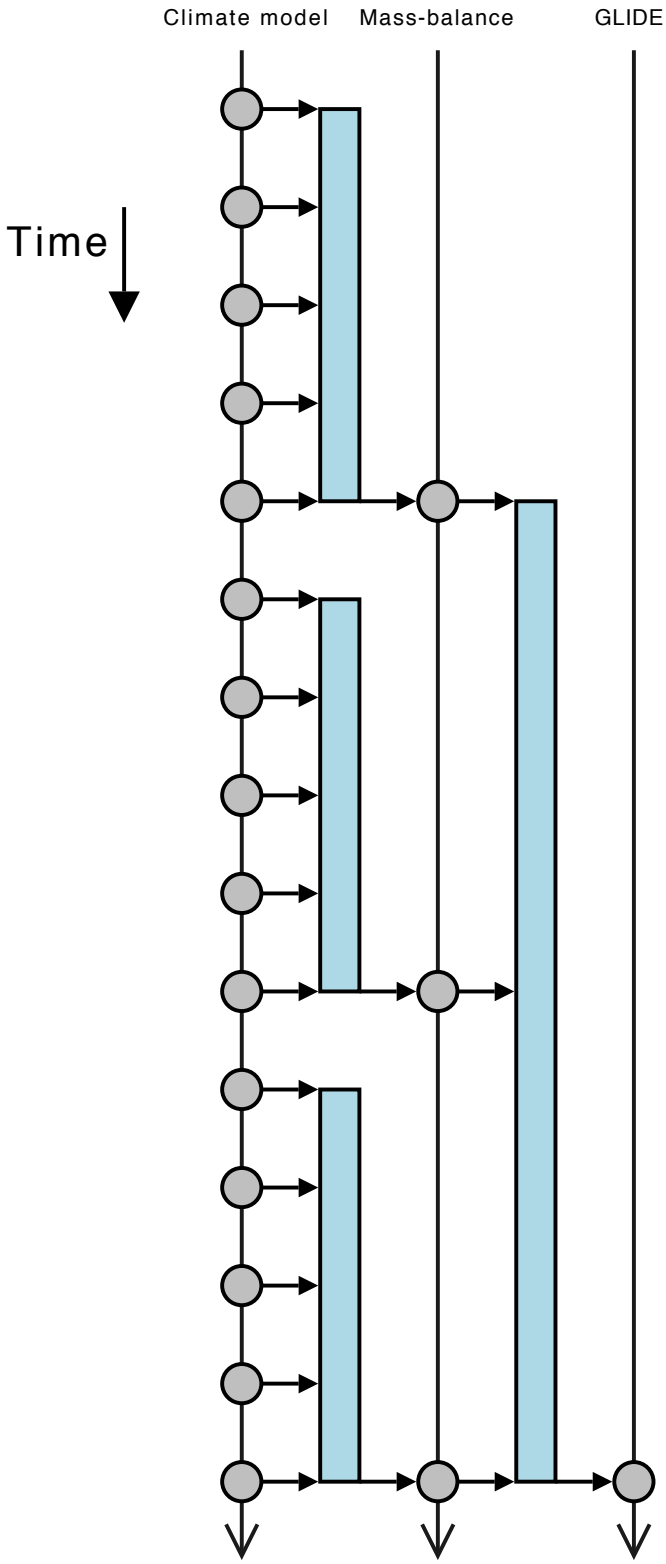

Figure 1.3: Relationship between the timesteps in GLINT. The filled circles represent timesteps, the rectangles represent averaging/accumulation, and the arrows, flow of coupling fields.

- `ice_sheet` is the variable of type `glint_params` defined above;
- `lats` and `lons` are one-dimensional arrays giving the locations of the global grid-points in latitude and longitude, respectively;
- `time_step` is the intended interval between calls to GLINT, in hours. This is known as the *forcing timestep*.
- `paramfile` is the name of the GLINT configuration file.

The contents of the configuration file will be dealt with later. Having initialised the model, it may now be called as part of the main climate model time-step loop:

```
call glint(ice_sheet,time,temp,precip,orog)
```

The arguments given in this example are the compulsory ones only; a large number of optional arguments may be specified – these are detailed in the reference section below. The compulsory arguments are:

- `ice_sheet` is the variable of type `glint_params` defined above;
- `time` is the current model time, in hours;
- `temp` is the daily mean 2 m global air temperature field, in °C;
- `precip` is the global daily accumulated precipitation field, in mm (water equivalent, making no distinction between rain, snow, etc.);
- `orog` is the global orography field, in m.

Two mass-balance schemes, both based on the positive degree day (PDD) method, are supplied with GLIMMER, and are available through GLINT. One of these calculates the mass-balance for the whole year (the *Annual PDD scheme*), while the other calculates on a daily basis (the *Daily PDD scheme*). The annual scheme incorporates a stochastic temperature variation to account for diurnal and other variations, which means that if this scheme is to be used, GLINT should be called such that it sees a seasonal temperature variation which has had those variations removed. In practice, this means calling GLINT on a monthly basis, with monthly mean temperatures. For the daily scheme, no such restriction exists, and the scheme should be called at least every 6 hours.

Finishing off

After the desired number of time-steps have been run, GLINT may have some tidying up to do. To accomplish this, the subroutine `end_glint` must be called:

```
call end_glint(ice_sheet)
```

API

A detailed description of the GLINT API may be found in the appendices.

Configuration

GLINT uses the same configuration file format as the rest of GLIMMER. In the case where only one GLIDE instance is used, all the configuration data for GLINT and GLIDE can reside in the same file. Where two or more instances are used, a top-level file specifies the number of model instances and the name of a configuration file for each one. Possible configuration sections specific to GLINT are as follows:

[GLINT]	
Section specifying number of instances.	
n_instance	(integer) Number of instances (default=1)

[GLINT instance]	
Specifies the name of an instance-specific configuration file. Unnecessary if we only have one instance whose configuration data is in the main config file.	
name	Name of instance-sepcific config file (required).

[GLINT climate]	
GLINT climate configuration	
precip_mode	Method of precipitation downscaling: **1** Use large-scale precipitation rate 2 Use parameterization of *Roe and Lindzen*
acab_mode	Mass-balance model to use: **1** Annual PDD mass-balance model (see section 1.5.2) 2 Annual accumulation only 3 Hourly energy-balance model (RAPID - not yet available) 4 Daily PDD mass-balance model (no docs yet)
ice_albedo	Albedo of ice — used for coupling to climate model (default=0.4)
lapse_rate	Atmospheric temperature lapse-rate, used to correct the atmospheric temperature onto the ice model orography. This should be *positive* for temperature falling with height ($\mathrm{K\,km^{-1}}$) (default=8.0)
data_lapse_rate	Atmospheric temperature vertical lapse rate, to be used in the calculation of temperature at sea-level. The variable lapse_rate is then used to adjust the temperature to the surface of the local ice sheet topography. If data_lapse_rate is not set, it is set to the value of lapse_rate by default.
ice_tstep_multiply	Ice time-step multiplier: allows asynchronous climate-ice coupling. See below for full explanation of GLINT time-stepping. (default = 1)
mbal_accum_time	Mass-balance accumulation time (in years, default is equal to mass-balance timestep). See below for full explanation of GLINT time-stepping.

GLINT timestepping — an explanation

By default, the model accepts input on each forcing timestep (as specified in the call to initialise_glint, above). These are accumulated over the course of a mass-balance time-step, whereupon the mass-balance model is called. The output from the mass-balance model is accumulated over the course of an ice model time-step, and finally the ice model is called.

This default behaviour can be altered, in two ways:

1. The number of ice sheet time-steps executed for each accumulated mass-balance field may be increased - thus accelerating the icesheet relative to the forcing. To do this, set ice_tstep_multiply in the [GLINT climate] config section - must be an integer. This is only possible if the mass-balance is accumulated over an integer number of years.

2. The mass-balance accumulation period can be altered by setting mbal_accum_time in the [GLINT climate] config section — this is a floating-point value in years.

The interaction of these two parameters is fairly complex, and permits a reasonably sophisticated control of how the ice sheet model is forced. Various checks are made at run-time to make sure sensible/possible values are selected. Most importantly, all relevant time-steps must divide

into one another appropriately - the model will (should...) stop if an un-sensible combination of values is detected.

GLINT timestepping — further examples

To aid understanding of the time-stepping controls, here are some examples. First, suppose we have these time-step values:

forcing time-step:	6 hours
mass-balance time-step:	1 day
ice time-step:	0.5 year

By default, the model will accumulate 6 months' worth of mass-balance calculations, and force the ice sheet model based on that. This might not be desirable, so you could set:

```
mbal_accum_time = 1.0
```

This would make GLINT accumulate 1 year's worth of mass-balance output before forcing the ice sheet (at which point it would execute *two* ice sheet time-steps of 0.5 years each).

Having done that, you could accelerate the ice model by a factor of ten, by setting:

```
ice_tstep_multiply = 10
```

In this scenario, 20 ice sheet time-steps of 0.5 years each would be done after each 12-month accumulation of mass-balance data.

For the second example, we consider the contrasting situation where we don't want to calculate a mass-balance on all the available data (perhaps to save time). Consider these time-step values:

forcing time-step:	6 hours
mass-balance time-step:	1 day
ice time-step:	10 years

(Clearly this a fairly numerically stable and/or low-resolution ice sheet).

To avoid running the daily PDD scheme c.3600 times (depending on the value of `days_in_year`), we can set to only use the first two years of data:

```
mbal_accum_time = 2.0
```

GLINT accumulates mass-balance for 2 years, then waits for 8 years (incoming data are ignored during this time), before calling the ice sheet. Ice sheet acceleration may be enabled with `ice_tstep_multiply` as before.

1.5 Supplied mass-balance schemes

1.5.1 Overview

The user is, of course, free to supply their own mass-balance model for use with GLIDE. However, GLIMMER includes within it a annual positive-degree-day model for mass balance, shortly to be augmented by a similar daily model and an hourly energy balance model. This section gives details of how to configure and call these models.

1.5.2 Annual PDD scheme

The annual PDD scheme is contained in the f90 module `glimmer_pdd`, and the model parameters are contained in the derived type `glimmer_pdd_params`. Configuration data is contained in a standard GLIMMER config file, which needs to be read from file before initialising the mass-balance model. The model is initialised by calling the subroutine `glimmer_pdd_init`, and the mass-balance may be calculated annually by calling `glimmer_pdd_mbal`.

Example of use:

```
use glimmer_pdd
use glimmer_config

...

type(glimmer_pdd_params) :: pdd_scheme
type(ConfigSection),pointer :: config

...

call glimmer_pdd_init(pdd_scheme,config)

...

call glimmer_pdd_mbal(pdd_scheme,artm,arng,prcp,ablt,acab)
```

In the subroutine call to `glimmer_pdd_mbal`, apart from the parameter variable `pdd_scheme`, there are three input fields (`artm`, `arng` and `prcp`), which are, respectively, the annual mean air temperature, annual temperature half-range, and annual accumulated precipitation fields. The final two arguments are output fields — annual ablation (`ablt`) and annual mass-balance (`acab`). All arrays are of type `real(sp)`. Temperatures are degrees Celcius, and precipitation, ablation and mass-balance are measured in m of water equivalent.

Day-degree calculation

The greater part of the information held in the `glimmer_pdd_params` derived type comprises a look-up table (the *PDD table*). The model is implemented this way for computational efficiency.

The table has two dimensions: mean annual air temperature (T_a) (as the second index) and annual air temperature half range (i.e., from July's mean to the annual mean ΔT_a) (as the first index). Following *Huybrechts and others* [1991], daily air temperatures (T'_a) are assumed to follow a sinusoidal cycle

$$T'_a = T_a + \Delta T_a \cos\left(\frac{2\pi t}{A}\right) + \mathbf{R}(0,\sigma) \tag{1.1}$$

where A is the period of a year and R is a random fluctuation drawn from a normal distribution with mean 0 °C and standard deviation σ °C. *Huybrechts and others* [1991] indicate that the number of positive degree days (D, °C days) for this temperature series can be evaluated as

$$D = \frac{1}{\sigma\sqrt{2\pi}} \int_0^A \int_0^{T'_a+2.5\sigma} T_a \times \exp\left(\frac{-(T_a - T'_a)^2}{2\sigma^2}\right) dT dt \tag{1.2}$$

where t is time. The table is completed by evaluating this integral using a public-domain algorithm (Romberg integration), by *Bauer* [1961]. The inner and outer integrals are coded

as two subroutines (`inner_integral` and `pdd_integrand`), which call the Romburg integration recursively.

The main parameter needed is the assumed standard deviation of daily air temperatures, which can be set in the configuration file (the default is 5 °C).

The positive-degree days are then looked up in the table (as a function of T_a and ΔT_a). We take care to check that this look up is in done within the bounds of the table. The final value of P is determined using bi-linear interpolation given the four nearest entries in the table to the actual values of T_a and ΔT_a.

The remainder of the loop completes the calculation of the ablation and accumulation given this value for P.

Mass balance calculation

We use the following symbols: a is total annual ablation; a_s is potential snow ablation; b_0 is the capacity of the snowpack to hold meltwater by refreezing; the total number of positive degree days (D); day-degree factors for snow and ice (f_s and f_i); and the fraction of snowfall that can be held in the snowpack as refrozen meltwater (W_max). Note that the day-degree factors have been converted from ice to water equivalents using the ratio of densities.

First, determine the depth of superimposed ice (b_0) that would have to be formed before runoff (mass loss) occurs as a constant fraction (W_{max}) of precipitation (P)

$$b_0 = W_{max} P. \tag{1.3}$$

Now determine the amount of snow melt by applying a constant day-degree factor for snow to the number of positive day-degrees

$$a_s = f_s D. \tag{1.4}$$

We now compare the potential amount of snow ablation with the ability of the snow layer to absorb the melt. Three cases are possible. First, all snow melt is held within the snowpack and no runoff occurs ($a = 0$). Second, the ability of the snowpack to hold meltwater is exceeded but the potential snow ablation is still less than the total amount of precipitation so that $a = a_s - b_0$. Finally, the potential snow melt is greater than the precipitation (amount of snow available), so that ice melt (a_i) has to be considered as well. The total ablation is therefore the sum of snow melt (total precipitation minus meltwater held in refreezing) and ice melt (deduct from total number of degree days, the number of degree days needed to melt all snowfall and convert to ice melt)

$$a = a_s + a_i = P - b_0 + f_i \left(D - \frac{P}{f_s} \right). \tag{1.5}$$

We now have a total annual ablation, and can find total net mass balance as the difference between the total annual precipitation and the total annual ablation.

Note that this methodology is fairly standard and stems from a series of Greenland papers by Huybrechts, Letreguilly and Reeh in the early 1990s.

Configuration

The annual PDD scheme is configured using a single section in the configuration file:

`[GLIMMER annual pdd]`	
Specifies parameters for the PDD table and mass-balance calculation	
`dx`	Table spacing in the x-direction (°C) (default=1.0)
`dy`	Table spacing in the y-direction (°C) (default=1.0)
`ix`	Lower bound of x-axis (°C) (default=0.0)

continued on next page

continued from previous page	
`iy`	Lower bound of y-axis (°C) (default=-50.0)
`nx`	Number of values in x-direction (default=31)
`ny`	Number of values in x-direction (default=71)
`wmax`	Fraction of melted snow that refreezes (default=0.6)
`pddfac_ice`	PDD factor for ice (m day^{-1} °C^{-1}) (default=0.008)
`pddfac_snow`	PDD factor for snow (m day^{-1} °C^{-1}) (default=0.003)

References

Bauer (1961) *Comm. ACM* **4**, 255.
Huybrechts, Letreguilly and Reeh (1991) *Palaeogeography, Palaeoclimatology, Palaeoecology (Global and Planetary Change)* **89**, 399-412.
Letreguilly, Reeh and Huybrechts (1991) *Palaeogeography, Palaeoclimatology, Palaeoecology (Global and Planetary Change)* **90**, 385-394.
Letreguilly, Huybrechts and Reeh (1991) *Journal of Glaciology* **37**, 149-157.

1.5.3 Daily PDD scheme

The other PDD scheme supplied with GLIMMER is a daily scheme. This is simpler than the annual scheme in that it does not incorporate any stochastic variations. The mass-balance is calculated on a daily basis, given the daily mean temperature and half-range, and assuming a sinusoidal diurnal cycle. Consequently, the firn model is more sophisticated than with the annual scheme, and includes a snow-densification parameterization.

Configuration

The daily PDD scheme is configured using a single section in the configuration file:

`[GLIMMER daily pdd]`	
Specifies parameters for the PDD table and mass-balance calculation	
`wmax`	Fraction of melted snow that refreezes (default=0.6)
`pddfac_ice`	PDD factor for ice (m day^{-1} °C^{-1}) (default=0.008)
`pddfac_snow`	PDD factor for snow (m day^{-1} °C^{-1}) (default=0.003)
`rain_threshold`	Temperature above which precipitation is held to be rain (°C) (default=1.0)
`whichrain`	Which method to use to partition precipitation into rain and snow: **1** Use sinusoidal diurnal temperature variation 2 Use mean temperature only
`tau0`	Snow densification timescale (s) (default=10 years)
`constC`	Snow density profile factor C (m^{-1}) (default=0.0165)
`firnbound`	Ice-firn boundary as fraction of density of ice (default=0.872)
`snowdensity`	Density of fresh snow (kg m^{-3}) (default=300.0)

Chapter 2

Tutorial

2.1 Introduction

This tutorial section aims to provide a set of more practical, step-by-step instructions on how to first get GLIMMER started after the successful installation and familiarise yourself with the different climate driver options. In general, this tutorial is intended to address the question:

'I have successfully compiled GLIMMER, now what? Do I have to write my own config files, climate drivers etc? I want to see some ice sheet modelling pronto!'

The really short version of an answer to this is type

```
glide_launch.py myconfig.config
```

where `myconfig.config` is a configuration file for GLIMMER as described in the documentation. If you have a config file and all the necessary data ready, this is how you get GLIMMER started.

Assuming that if you are reading this, you probably won't yet have your own config file ready, so you might want to read on:

2.2 EISMINT: using `glimmer-example`

As you hopefully already know, the heart of GLIMMER is the actual ice sheet model GLIDE. This is where ice physics are resolved etc. To model an ice sheet using GLIDE, you at least need to provide it with information about the mass balance. To get you started with a real simple example climate driver, download `glimmer-example` from the project homepage or via CVS, cd into the directory and type

```
glide_launch.py example.config
```

this will kick off a simple EISMINT-1 moving margin type model run. The results are written to `example.nc`, use a viewer like ncview to visualise them. Take a look at the `example.config` file printed below and read the documentation on the EISMINT type climate driver (section 1.4.1) to better understand what is happening:

```
# configuration for the EISMINT-1 test-case # moving margin

[EISMINT-1 moving margin]

[grid]
# grid sizes
ewn = 31
```

```
nsn = 31
upn = 11
dew = 50000
dns = 50000

[options]
temperature = 1
flow_law = 2
isostasy = 0
sliding_law = 4
marine_margin = 2
stress_calc = 2
evolution = 2
basal_water = 2
vertical_integration = 0

[time]
tend = 200000.
dt = 10.
ntem = 1.
nvel = 1.
niso = 1.

[parameters]
flow_factor = 1
geothermal = -42e-3

[CF default]
title: EISMINT-1 moving margin

[CF output]
name: example.nc
frequency: 1000
variables: thk uflx vflx bmlt temp
uvel vvel wvel
```

The line `[EISMINT-1 moving margin]` sets the model type for this run to be EISMINT (simple_glide binary). This can also be achieved by specifying the correct binary using the `-m` flag, e.g.

```
glide_launch.py -m simple_glide example.config
```

It is probably advisable to use the `-m` option instead of specifying the binary using a keyterm, as this will only work for EIS and EISMINT model types. For ease of use, the option was integrated in the config file for this example.

The `[grid]` section sets up the topography for the model run.

As this is an EISMINT testcase, there is no 'real' input topography, but ice is building up on a flat surface, which is why nothing more but the grid dimensions need to be specified. Be aware that this only works for EISMINT type model runs using simple_glide. In this case, the mass balance is parameterised as a function of distance from the grid center, resulting in a point symmetric ice sheet. The grid used here has a size of 31x31 cells (`ewn x nsn`), comprises of 11 vertical layers (`upn = 11`) and an internal cell spacing of 5000 (`dew` and `dns`).

The `[options]` sections determines the basic behaviour of the model:

`temperature = 1` resolves the temperature over the whole of the 11 layers of ice (instead of assuming ice to be isothermal), `isostasy = 0` turns off the isostasy component, etc. (check the documentation).

In the `[time]` section, the end time of the model run is set to 200000 with a timestep size of 10 and keeping all internal update processes (temperature and velocity) in line with the timesteps by setting their multiplier to 1.

Flow factor and geothermal heat flux parameters are set in the `[parameters]` section.

Finally, in the `[CF output]` section, the name of the file to store the results is given, together with the `variables` that should be dumped to the file and the `frequency` with which they are written to it (every 1000 years). In this example, ice thickness (`thk`), basal melt temperature `bmlt`, ice temperature `temp` etc is output to the result file every 1000 years. Note that this output frequency is independent of the modelling timesteps.

You might want to try and change some of the parameters, e.g. speed up ice flow by increasing the flow factor, and re-run the model to see what happens. This is fairly simple and straight forward example of how to get GLIMMER to do some basic modelling. If you want to see a bit more of what GLIMMER can do, try the next section.

2.3 EIS: using `glimmer-tests`

`glimmer-tests` provides more example configurations, that include both the EISMINT and EIS climate drivers. If you have not already done so, download `glimmer-tests` via the nescforge page or CVS, and do the usual

```
./configure -with-glimmer-prefix=/path/to/GLIMMER/installation
(e.g. /usr/local/GLIMMER)
```

(if you updated GLIMMER via CVS, you need to do `./bootstrap` first.)

`glimmer-tests` is not (yet) a test suite, but will exemplarily show what GLIMMER can do (see the `glimmer-tests` README file for detailed information on the tests).

Basically `glimmer-tests` runs GLIMMER using the EISMINT 1 and 2 (and 3) climate driver (fixed and moving margin type ice sheets with no external mass balance forcing), as well as the Edinburgh Ice Sheet (EIS) climate driver, using mass balance parameterisation via ELA and temperature forcing. There are a couple of other tests running besides this, e.g. some benchmarks. If you want to run all the examples, simply do a `make` in the `glimmer-tests` directory, but be aware that running all tests will take a good 12+ hours on a single CPU 3 GHZ machine. If you're too impatient for this, simply do a `make` in one of the subdirectories, e.g. EISMINT1 and GLIDE will be launched using the EISMINT climate driver, which should deliver you a number of netcdf files with the model results, eg. `e1.fm.1.nc` containing the EISMINT1 fixed margin results 1, etc. Again, to visualise the results use a viewer like ncview.

If you want a more sophisticated results, try `make` in the `eis` directory, which will repeat the results of Hagdorn (2003) reconstructing the Fennoscandian ice sheet during the last glacial maximum, using the EIS driver.

2.3.1 A short introduction to the EIS driver parameterisation

Again, check the config file `fenscan.config` to see the basic parameters for this model run. Have a look at the `mb2.data` (mass balance forcing via ELA), `temp-exp.model` (exponential type temperature forcing) and `specmap.data` (sealevel change) data files and compare them to the EIS driver documentation (section 1.4.2) to get an idea of how things are done.

The first column in every data file is the model time at which the new parameter values are applied. For the temperature model, the records in the `temp-exp.model` file

...

```
 -97000.000000  -17.858964  23.158964   -0.051329
 -96000.000000  -20.074036  24.674036   -0.051329
...
```

correspond to the timesteps -97000 and -96000 (first column - model usually ends at time 0) where the parameters a0 (2nd column), a1 (3rd column) and a2 (last column) of the exponential temperature model $T(t) = a_0 + a_1 \exp(a_2(\lambda - \lambda_0))$ (page 15) are updated to reflect an approximate change in temperature of -2 degrees Celsius.

For EIS, the mass balance is parameterised via the ELA, according to

$$z_{\text{ELA}} = a + b\lambda + c\lambda^2 + \Delta z_{\text{ELA}},$$

given the parameters in the according config file section:

```
...
[EIS ELA]
ela_file = mb2.data
bmax_mar = 4.
ela_a = 14430.069930
ela_b =-371.765734
ela_c = 2.534965
...
```

Factors a, b and c are specified together with the maximum mass balance of 4. The latitude λ in degrees North is read from the input topography grid. In order to do the ELA forcing over time, the parameter Δz_{ELA} is varied over time using the ela file `mb2.data`:

```
...
 -109000 225
 -105000 350
...
```

Similar to the temperature forcing, Δz_{ELA} (column 2) is changed at timestep -10900 (column 1), to reflect an ELA 225m above the altitude value calculated using the factors a, b, c and the latitude λ. At timestep -10500, ELA is rising to 350m above the calculated value.

Where a globally changing Δz_{ELA} is insufficient to reflect disparities in ELA, there are two options to fine tune ELA behaviour. First, continentality can be used to introduce a dependency of mass balance with distance to oceans. The according settings are supplied using the `[EIS CONY]` section of the config file (see section 1.4.2). In short, an index is calculated for every grid cell reflecting the ratio of below sealevel cells to land cells within a certain `range` (defaults to 600km). Maximum mass balance values are then scaled between the values given in the `[EIS ELA]` section for `bmax_mar` (marine conditions, all cells within `range` are below sea level) and `bmax_cont` (continenal conditions, all cells within `range` are above sea level). Alternatively, continentality values between 0 and 1 can be input using a file. Set the according flag `file` to 1 and specify the file containing the cony data using a `[CF input]` section in the configuration file (see example for ELA file below). .

In case a more detailed spatial distribution of ELA altitudes is needed, e.g. to reflect special orographic effects, a map of Δz_{ELA} can be input to the model using a netcdf file, containing a variable 'ela' on a grid the same size and coordinates as the input topography grid the model is running on. This ela file is coupled using a `[CF input]` section in the configuration file

```
[CF input]
name: ela_1k.nc
```

resulting in a spatial distribution of Δz_{ELA} being applied to the model. The variation of ELA over time using a global Δz_{ELA} is still applied on top of this ELA forcing file.

Note: (Maybe an example containing an ELA forcing file should be added to GLIMMER test/examples?)

Sealevel changes are forced upon the model in an according way using the `specmap.data` file.

2.4 GLINT: using glint-example

If finally you want to see what GLIMMER can do using the GLINT climate driver, download the `glint-example` and try one of the provided example setups. CD into the directory and try any of the config examples. Start glint_example by typing

```
glint_example
```

You will then be asked for a climate configuration file and an ice model configuration file. For the climate file, a global example including precipitation and temperature timeseries is provided. To let glint know about it, type

```
glint_example.config
```

For the ice model config, there are two examples, Greenland and North America. To chose either one, type

```
gland20.config
```

or

```
namerica20.config
```

respectively at the prompt asking for the config file, to start the model. Both models are outputting three files each, containing different variables. Every 100 years, a file `namerica20.hot.nc` or `gland20.hot.nc`, respectively is output, which can be used to hotstart the model later from any of the recorded stages.

As mentioned above, the model type (binary) to use can be stated in the configuration file, or given using the `-m` option. Currently, the three model binaries that come with GLIMMER are `simple_glide, eis_glide` and `glint_example`. The simple_glide and eis_glide drivers that are started using the glide_launch.py Python script, which needs to know which binary to address. The model binary can also be set as an environment variable $GLIDE_MODEL. However, as glint is called directly using the compiled binary `glint_example` here, it is not necessary to further specify the model.

Chapter 3

Numerics

This part describes the numerical implementation of GLIMMER in some detail. It is hoped that more parts will be added in the future.

3.1 Ice Thickness Evolution

The evolution of the ice thickness, H, stems from the continuity equation and can be expressed as

$$\frac{\partial H}{\partial t} = -\nabla \cdot (\overline{\boldsymbol{u}} H) + B, \tag{3.1}$$

where $\overline{\boldsymbol{u}}$ is the vertically averaged ice velocity, B is the surface mass balance and ∇ is the horizontal gradient operator (Payne and Dongelmans, 1997).

For large–scale ice sheet models, the *shallow ice approximation* is generally used. This approximation states that bedrock and ice surface slopes are assumed sufficiently small so that the normal stress components can be neglected (Hutter, 1983). The horizontal shear stresses (τ_{xz} and τ_{yz}) can thus be approximated by

$$\begin{aligned} \tau_{xz}(z) &= -\rho g (s - z) \frac{\partial s}{\partial x}, \\ \tau_{yz}(z) &= -\rho g (s - z) \frac{\partial s}{\partial y}, \end{aligned} \tag{3.2}$$

where ρ is the density of ice, g the acceleration due to gravity and $s = H + h$ the ice surface.

Strain rates $\dot{\epsilon}_{ij}$ of polycrystalline ice are related to the stress tensor by the non–linear flow law:

$$\dot{\epsilon}_{iz} = \frac{1}{2}\left(\frac{\partial u_i}{\partial z} + \frac{\partial u_z}{\partial i}\right) = A(T^*)\tau_*^{(n-1)}\tau_{iz} \qquad i = x, y, \tag{3.3}$$

where τ_* is the effective shear stress defined by the second invariant of the stress tensor, n the flow law exponent and A the temperature–dependent flow law coefficient. T^* is the absolute temperature corrected for the dependence of the melting point on pressure ($T^* = T + 8.7 \cdot 10^{-4}(H + h - z)$, T in Kelvin, Huybrechts, 1986). The parameters A and n have to be found by experiment. n is usually taken to be 3. A depends on factors such as temperature, crystal size and orientation, and ice impurities. Experiments suggest that A follows the Arrhenius relationship:

$$A(T^*) = f a e^{-Q/RT^*}, \tag{3.4}$$

where a is a temperature–independent material constant, Q is the activation energy for creep and R is the universal gas constant (Paterson, 1994). f is a tuning parameter used to 'speed–up' ice flow and accounts for ice impurities and the development of anisotropic ice fabrics (Payne, 1999; Tarasov and Peltier, 1999, 2000; Peltier et al., 2000).

Integrating (3.4) with respect to z gives the horizontal velocity profile:

$$\boldsymbol{u}(z) - \boldsymbol{u}(h) = -2(\rho g)^n |\boldsymbol{\nabla} s|^{n-1} \boldsymbol{\nabla} s \int_h^z A(s-z)^n dz, \tag{3.5}$$

where $\boldsymbol{u}(h)$ is the basal velocity (sliding velocity). Integrating (3.5) again with respect to z gives an expression for the vertically averaged ice velocity:

$$\overline{\boldsymbol{u}} H = -2(\rho g)^n |\boldsymbol{\nabla} s|^{n-1} \boldsymbol{\nabla} s \int_h^s \int_h^z A(s-z)^n dz dz'. \tag{3.6}$$

The vertical ice velocity stems from the conservation of mass for an incompressible material:

$$\frac{\partial u_x}{\partial x} + \frac{\partial u_y}{\partial y} + \frac{\partial u_z}{\partial z} = 0. \tag{3.7}$$

Integrating (3.7) with respect to z gives the vertical velocity distribution of each ice column:

$$w(z) = -\int_h^z \boldsymbol{\nabla} \cdot \boldsymbol{u}(z) dz + w(h), \tag{3.8}$$

with lower, kinematic boundary condition

$$w(h) = \frac{\partial h}{\partial t} + \boldsymbol{u}(h) \cdot \boldsymbol{\nabla} h + S, \tag{3.9}$$

where S is the melt rate at the ice base given by Equation (3.55). The upper kinematic boundary is given by the surface mass balance and must satisfy:

$$w(s) = \frac{\partial s}{\partial t} + \boldsymbol{u}(s) \cdot \boldsymbol{\nabla} s + B. \tag{3.10}$$

3.1.1 Numerical Grid

The continuous equations descrining ice physics have to be discretised in order to be solved by a computer (which is inherently finite). This section describes the finite–difference grids employed by the model.

Horizontal Grid

The modelled region ($x \in [0, L_x]$, $y \in [0, L_y]$) is discretised using a regular grid so that $x_i = (i-1)\Delta x$ for $i \in [1, N]$ (and similarly for y_j). The model uses two staggered horizontal grids in order to improve stability. Both grids use the same grid spacing, Δx and Δy, but are off-set by half a grid (see Fig. 3.1). Quantities calculated on the (r, s)–grid are denoted with a tilde, i.e. $\tilde{F}$. Quantities are transformed between grids by averaging over the surrounding nodes, i.e. a quantity in the (i, j)–grid becomes in the (r, s) grid:

$$\tilde{F}_{r,s} = \tilde{F}_{i+\frac{1}{2}, j+\frac{1}{2}} = \frac{1}{4}(F_{i,j} + F_{i+1,j} + F_{i+1,j+1} + F_{i,j+1}) \tag{3.11a}$$

and similarly for the reverse transformation:

$$F_{i,j} = F_{r-\frac{1}{2}, s-\frac{1}{2}} = \frac{1}{4}(\tilde{F}_{r-1,s-1} + \tilde{F}_{r,s-1} + \tilde{F}_{r,s} + \tilde{F}_{r-1,s}) \tag{3.11b}$$

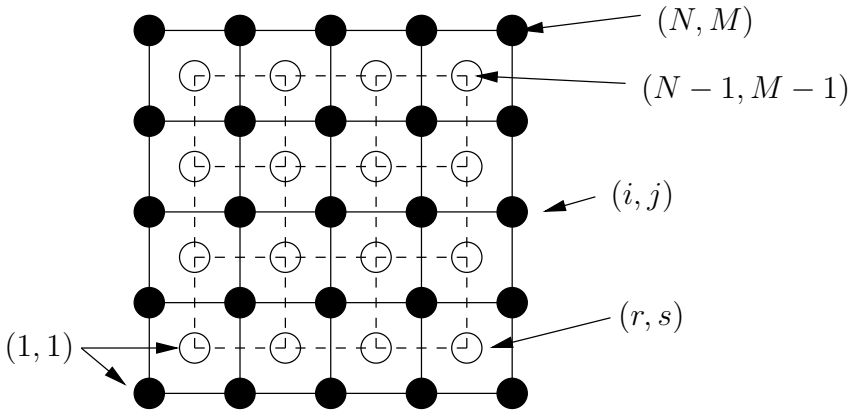

Figure 3.1: Horizontal Grid.

In general, horizontal velocities and associated quantities like the diffusivity are calculated on the (r, s) grid, ice thickness, temperatures and vertical velocities are calculated on the (i, j)–grid.

Horizontal gradients are calculated on the (r, s)–grid, i.e. surface gradients are:

$$\left(\frac{\partial s}{\partial x}\right)_{r,s} = \tilde{s}^x_{r,s} = \frac{s_{i+1,j} - s_{i,j} + s_{i+1,j+1} - s_{i,j+1}}{2\Delta x} \tag{3.12a}$$

$$\left(\frac{\partial s}{\partial y}\right)_{r,s} = \tilde{s}^y_{r,s} = \frac{s_{i,j+1} - s_{i,j} + s_{i+1,j+1} - s_{i+1,j}}{2\Delta y} \tag{3.12b}$$

Ice thickness gradients, $\tilde{H}^x_{r,s}$ and $\tilde{H}^y_{r,s}$, are formed similarly. Gradients in the (r, s)–grid are formed in a similar way, e.g.

$$\left(\frac{\partial u}{\partial x}\right)_{i,j} = u^x_{i,j} = \frac{\tilde{u}_{r,s-1} - \tilde{u}_{r-1,s-1} + \tilde{u}_{r,s} - \tilde{u}_{r-1,s}}{2\Delta x} \tag{3.13}$$

Periodic Boundary Conditions

The model can be run with horizontal periodic boundary conditions, i.e. the western edge of the modelled region is joined with the eastern edge. Figure 3.2 illustrates the numeric grid when the model is run in torus mode.

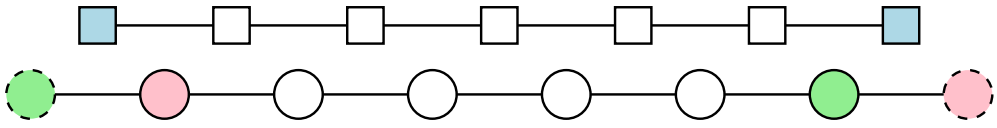

Figure 3.2: A row of the numeric grid when the model is used in torus mode. Circles indicate points in (i, j)–grid and squares indicate points in the (r, s)–grid. Points with the same colour are logically the same.

These boundary conditions are enforced by exchanging points for the temperature and vertical velocity calculations. The ice thicknesses are calculated explicitly at the ghostpoints.

σ–Coordinate System

The vertical coordinate, z, is scaled by the ice thickness analogous to the s–coordinate in numerical weather simulations (e.g. Holton, 1992). A new vertical coordinate, σ, is introduced

so that the ice surface is at $\sigma = 0$ and the ice base at $\sigma = 1$ (see Fig. 3.3), i.e.

$$\sigma = \frac{s - z}{H}. \tag{3.14}$$

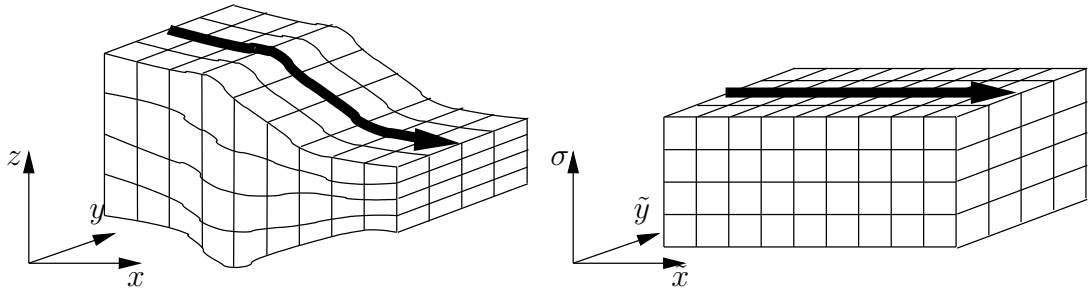

Figure 3.3: Vertical scaling of the ice sheet model. The vertical axis is scaled to unity. The horizontal coordinates are not changed.

The derivatives of a function f in (x, y, z, t) become in the new $(\tilde{x}, \tilde{y}, \sigma, \tilde{t})$ system:

$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial \tilde{x}} + \frac{1}{H}\Delta_{\tilde{x}}\frac{\partial f}{\partial \sigma}, \tag{3.15a}$$

$$\frac{\partial f}{\partial y} = \frac{\partial f}{\partial \tilde{y}} + \frac{1}{H}\Delta_{\tilde{y}}\frac{\partial f}{\partial \sigma}, \tag{3.15b}$$

$$\frac{\partial f}{\partial t} = \frac{\partial f}{\partial \tilde{t}} + \frac{1}{H}\Delta_{\tilde{t}}\frac{\partial f}{\partial \sigma}, \tag{3.15c}$$

$$\frac{\partial f}{\partial z} = -\frac{1}{H}\frac{\partial f}{\partial \sigma}, \tag{3.15d}$$

where the geometric factors, $\Delta_{\tilde{x}}$, $\Delta_{\tilde{y}}$ and $\Delta_{\tilde{t}}$, are defined by

$$\Delta_{\tilde{x}} = \left(\frac{\partial s}{\partial \tilde{x}} - \sigma\frac{\partial H}{\partial \tilde{x}}\right), \tag{3.16a}$$

$$\Delta_{\tilde{y}} = \left(\frac{\partial s}{\partial \tilde{y}} - \sigma\frac{\partial H}{\partial \tilde{y}}\right), \tag{3.16b}$$

$$\Delta_{\tilde{t}} = \left(\frac{\partial s}{\partial \tilde{t}} - \sigma\frac{\partial H}{\partial \tilde{t}}\right). \tag{3.16c}$$

The integral of z becomes in the σ–coordinate system:

$$\int_h^z f dz = -H\int_1^\sigma f d\sigma \tag{3.17}$$

The vertical coordinate is discretised using an irregular grid spacing to reflect the fact that ice flow is more variable at the bottom of the ice column. In the vertical the index k is used.

3.1.2 Ice Sheet Equations in σ–Coordinates

The horizontal velocity, Equation (3.5), becomes in the σ–coordinate system

$$\boldsymbol{u}(\sigma) = -2(\rho g)^n H^{n+1}|\boldsymbol{\nabla} s|^{n-1}\boldsymbol{\nabla} s\int_1^\sigma A\sigma^n d\sigma + \boldsymbol{u}(1) \tag{3.18}$$

and the vertically averaged velocity

$$\overline{\boldsymbol{u}}H = H\int_0^1 \boldsymbol{u}d\sigma + \boldsymbol{u}(1)H \tag{3.19}$$

The vertical velocity, Equation (3.8), becomes

$$w(\sigma) = -\int_1^\sigma \left(\frac{\partial \boldsymbol{u}}{\partial \sigma}\cdot(\boldsymbol{\nabla}s - \sigma\boldsymbol{\nabla}H) + H\boldsymbol{\nabla}\cdot\boldsymbol{u}\right)d\sigma + w(1) \tag{3.20}$$

and lower boundary condition

$$w(1) = \frac{\partial h}{\partial t} + \boldsymbol{u}(1)\cdot\boldsymbol{\nabla}h + S. \tag{3.21}$$

3.1.3 Calculating the Horizontal Velocity and the Diffusivity

Horizontal velocity and diffusivity calculations are split up into two parts:

$$\boldsymbol{u}(\sigma) = c\boldsymbol{\nabla}s + \boldsymbol{u}(1) \tag{3.22a}$$

$$D = H\int_0^1 cd\sigma \tag{3.22b}$$

$$\boldsymbol{q} = D\boldsymbol{\nabla}s + H\boldsymbol{u}(1) \tag{3.22c}$$

with

$$c(\sigma) = -2(\rho g)^n H^{n+1}|\boldsymbol{\nabla}s|^{n-1}\int_1^\sigma A\sigma^n d\sigma \tag{3.22d}$$

Quantities $\boldsymbol{u}$ and D are found on the velocity grid. Integrating from the ice base ($k = N-1$), the discretised quantities become

$$\tilde{c}_{r,s,N} = 0 \tag{3.23a}$$

$$\tilde{c}_{r,s,k} = -2(\rho g)^n H_{r,s}^{n+1}\left((\tilde{s}_{r,s}^x)^2 + (\tilde{s}_{r,s}^y)^2\right)^{\frac{n-1}{2}} \sum_{\kappa=N-1}^{k} \frac{A_{r,s,\kappa} + A_{r,s,\kappa+1}}{2}\left(\frac{\sigma_{\kappa+1}+\sigma_\kappa}{2}\right)^n(\sigma_{\kappa+1}-\sigma_\kappa) \tag{3.23b}$$

$$\tilde{D}_{r,s} = H_{r,s}\sum_{k=0}^{N-1}\frac{\tilde{c}_{r,s,k} + \tilde{c}_{r,s,k+1}}{2}(\sigma_{k+1}-\sigma_k) \tag{3.23c}$$

Expressions for $\boldsymbol{u}_{i,j,k}$ and $\boldsymbol{q}_{i,j}$ are straight forward.

3.1.4 Solving the Ice Thickness Evolution Equation

Equation (3.1) can be rewritten as a diffusion equation, with non–linear diffusion coefficient D:

$$\frac{\partial H}{\partial t} = -\boldsymbol{\nabla}\cdot D\boldsymbol{\nabla}s + B = -\boldsymbol{\nabla}\cdot\boldsymbol{q} + B \tag{3.24}$$

This non–linear partial differential equation can be linearised by using the diffusion coefficient from the previous time step. The diffusion coefficient is calculated on the (r, s)–grid, i.e. staggered in both x and y direction. Figure 3.4 illustrates the staggered grid. Using finite differences, the fluxes in x direction, q^x become

$$q^x_{i+\frac{1}{2},j} = -\frac{1}{2}(\tilde{D}_{r,s} + \tilde{D}_{r,s-1})\frac{s_{i+1,j} - s_{i,j}}{\Delta x} \tag{3.25a}$$

$$q^x_{i-\frac{1}{2},j} = -\frac{1}{2}(\tilde{D}_{r-1,s} + \tilde{D}_{r-1,s-1})\frac{s_{i,j} - s_{i-1,j}}{\Delta x} \tag{3.25b}$$

and the fluxes in y direction

$$q^y_{i,j+\frac{1}{2}} = -\frac{1}{2}(\tilde{D}_{r,s} + \tilde{D}_{r-1,s})\frac{s_{i,j+1} - s_{i,j}}{\Delta y} \tag{3.25c}$$

$$q^y_{i,j-\frac{1}{2}} = -\frac{1}{2}(\tilde{D}_{r,s-1} + \tilde{D}_{r-1,s-1})\frac{s_{i,j} - s_{i,j-1}}{\Delta y}. \tag{3.25d}$$

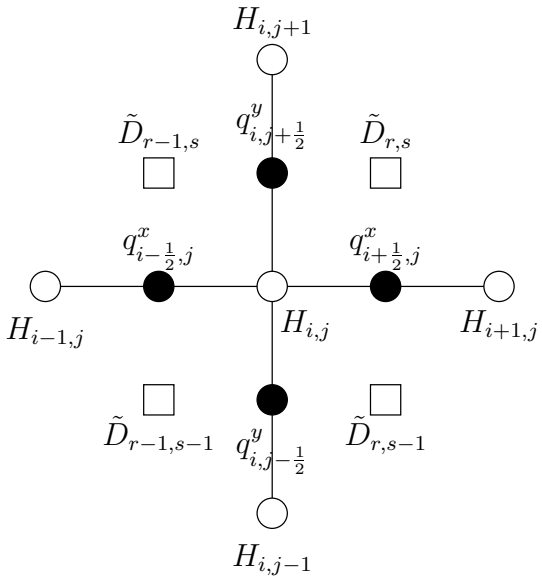

Figure 3.4: Illustration of the staggered grid used to calculate ice thicknesses, diffusivities and mass fluxes.

ADI Scheme

The alternating–direction implicit method (ADI) uses the concept of operator splitting where Equation (3.24) is first solved in the x–direction and then in the y–direction, (Press et al., 1992). The time step Δt is devided into two time steps $\Delta t/2$. The descretised version of Equation (3.24) becomes (Huybrechts, 1986):

$$2\frac{H^{t+\frac{1}{2}}_{i,j} - H^t_{i,j}}{\Delta t} = -\frac{q^{x,t+\frac{1}{2}}_{i+\frac{1}{2},j} - q^{x,t+\frac{1}{2}}_{i-\frac{1}{2},j}}{\Delta x} - \frac{q^{y,t}_{i,j+\frac{1}{2}} - q^{y,t}_{i,j-\frac{1}{2}}}{\Delta y} + B_{i,j} \tag{3.26a}$$

$$2\frac{H^{t+1}_{i,j} - H^{t+\frac{1}{2}}_{i,j}}{\Delta t} = -\frac{q^{x,t+\frac{1}{2}}_{i+\frac{1}{2},j} - q^{x,t+\frac{1}{2}}_{i-\frac{1}{2},j}}{\Delta x} - \frac{q^{y,t+1}_{i,j+\frac{1}{2}} - q^{y,t+1}_{i,j-\frac{1}{2}}}{\Delta y} + B_{i,j} \tag{3.26b}$$

Gathering all $t+\frac{1}{2}$ terms on the left side, Equation (3.26a) can be expressed as a tri–diagonal set of equations for each row j:

$$-\alpha_{i,j}H^{t+\frac{1}{2}}_{i-1,j}+(1-\beta_{i,j})H^{t+\frac{1}{2}}_{i,j}-\gamma_{i,j}H^{t+\frac{1}{2}}_{i+1,j}=\delta_{i,j} \tag{3.27}$$

with

$$\alpha_{i,j}=\frac{\tilde{D}_{r-1,s}+\tilde{D}_{r-1,s-1}}{4\Delta x^2}\Delta t \tag{3.28a}$$

$$\beta_{i,j}=-\frac{\tilde{D}_{r,s}+2\tilde{D}_{r-1,s}+\tilde{D}_{r-1,s-1}}{4\Delta x^2}\Delta t=-(\alpha_{i,j}+\gamma_{i,j}) \tag{3.28b}$$

$$\gamma_{i,j}=\frac{\tilde{D}_{r,s}+\tilde{D}_{r,s-1}}{4\Delta x^2}\Delta t \tag{3.28c}$$

and the RHS,

$$\delta_{i,j}=H^t_{i,j}-\frac{\Delta t}{2\Delta y}\left(q^{y,t}_{i,j+\frac{1}{2}}-q^{y,t}_{i,j-\frac{1}{2}}\right)+\frac{\Delta t}{2}B_{i,j}+\alpha_{i,j}h_{i-1,j}-\beta_{i,j}h_{i,j}+\gamma_{i,j}h_{i+1,j}. \tag{3.28d}$$

A similar tri–diagonal system is found for each column, i of Equation (3.26b).

Linearised Semi–Implicit Scheme

Using the Crank–Nicolson scheme, the semi–implicit temporal discretisation of (3.24) is then:

$$\frac{H^{t+1}_{i,j}-H^t_{i,j}}{\Delta t}=\frac{q^{x,t+1}_{i+\frac{1}{2},j}-q^{x,t+1}_{i-\frac{1}{2},j}}{2\Delta x}+\frac{q^{y,t+1}_{i,j+\frac{1}{2}}-q^{y,t+1}_{i,j-\frac{1}{2}}}{2\Delta y}+\frac{q^{x,t}_{i+\frac{1}{2},j}-q^{x,t}_{i-\frac{1}{2},j}}{2\Delta x}+\frac{q^{y,t}_{i,j+\frac{1}{2}}-q^{y,t}_{i,j-\frac{1}{2}}}{2\Delta y}+B_{i,j} \tag{3.29}$$

The superscripts ^{t} and $^{t+1}$ indicate at what time the ice thickness H is evaluated. Collecting all H^{t+1} terms of (3.29) on the LHS and moving all other terms to the RHS we can rewrite (3.29) as

$$-\alpha_{i,j}H^{t+1}_{i-1,j}-\beta_{i,j}H^{t+1}_{i+1,j}-\gamma_{i,j}H^{t+1}_{i,j-1}-\delta_{i,j}H^{t+1}_{i,j+1}+(1-\epsilon_{i,j})H^{t+1}_{i,j}=\zeta_{i,j} \tag{3.30}$$

with the RHS,

$$\zeta_{i,j}=\alpha_{i,j}H^t_{i-1,j}+\beta_{i,j}H^t_{i+1,j}+\gamma_{i,j}H^t_{i,j-1}+\delta_{i,j}H^t_{i,j+1}+(1+\epsilon_{i,j})H^t_{i,j}+2(\alpha_{i,j}h_{i-1,j}+\beta_{i,j}h_{i+1,j}+\gamma_{i,j}h_{i,j-1}+\delta_{i,j}h_{i,j+1}+\epsilon_{i,j}h_{i,j})+B_{i,j}\Delta t \tag{3.31}$$

with the elements of the sparse matrix

$$\alpha_{i,j}=\frac{\tilde{D}_{r-1,s}+\tilde{D}_{r-1,s-1}}{4\Delta x^2}\Delta t \tag{3.32a}$$

$$\beta_{i,j}=\frac{\tilde{D}_{r,s}+\tilde{D}_{r,s-1}}{4\Delta x^2}\Delta t \tag{3.32b}$$

$$\gamma_{i,j}=\frac{\tilde{D}_{r,s-1}+\tilde{D}_{r-1,s-1}}{4\Delta y^2}\Delta t \tag{3.32c}$$

$$\delta_{i,j}=\frac{\tilde{D}_{r,s}+\tilde{D}_{r-1,s}}{4\Delta y^2}\Delta t \tag{3.32d}$$

$$\epsilon_{i,j}=-(\alpha_{i,j}+\beta_{i,j}+\gamma_{i,j}+\delta_{i,j}) \tag{3.32e}$$

This matrix equation is solved using an iterative matrix solver for non-symmetric sparse matrices. The solver used here is the bi–conjugate gradient method with incomplete LU decomposition preconditioning provided by the SLAP package.

Non–Linear Scheme

The non–linearity of Equation (3.24) arises from the dependance of D on s. A non–linear scheme for (3.24) can be formulated using Picard iteration, which consists of two iterations: an outer, non–linear and an inner, linear equation. The scheme is started off with the diffusivity from the previous time step, i.e.

$$D^{(0),t+1} = D^t \tag{3.33a}$$

and Equation (3.30) becomes

$$\begin{aligned} -\alpha_{i,j}^{(\xi),t+1} H_{i-1,j}^{t+1} - \beta_{i,j}^{(\xi),t+1} H_{i+1,j}^{(\xi+1),t+1} - \gamma_{i,j}^{(\xi),t+1} H_{i,j-1}^{(\xi+1),t+1} \\ - \delta_{i,j}^{(\xi),t+1} H_{i,j+1}^{(\xi+1),t+1} + (1 - \epsilon_{i,j}^{(\xi),t+1}) H_{i,j}^{(\xi+1),t+1} = \zeta_{i,j}^{(0),t} \end{aligned} \tag{3.33b}$$

Equation (3.33b) is iterated over ξ until the maximum ice thickness residual is smaller than some threshold:

$$\max\left(\left|H^{(\xi+1),t+1} - H^{(\xi),t+1}\right|\right) < H_{\text{res}} \tag{3.34}$$

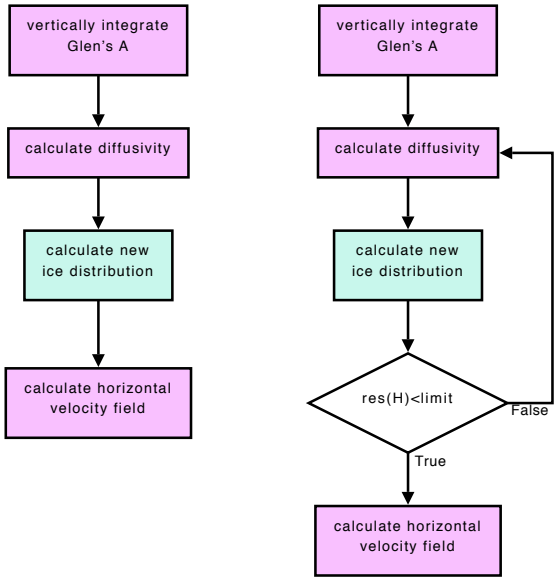

Figure 3.5: Flow diagram showing how the linearised solver (on the left) and the non–linear solver work. The inner, linear iteration is contained within the box labeled "calculate new ice distribution".

3.1.5 Calculating Vertical Velocities

Grid Velocity

The vertical grid moves as a consequence of using a σ–coordinate system. The grid velocity is

$$w^{\text{grid}}(\sigma) = \frac{\partial s}{\partial t} + \boldsymbol{u} \cdot \boldsymbol{\nabla} s - \sigma \left(\frac{\partial H}{\partial t} + \boldsymbol{u} \cdot \boldsymbol{\nabla} H \right) \tag{3.35}$$

The numerical implementation of Equation (3.35) is straight–forward.

Vertical Velocity

The discretised version of the vertical velocity equation (3.20) is slightly more compilicated because the horizontal velocities are calculated on the (r, s) grid. The vertical velocity at the ice base is $w_{i,j,N} = w^{\text{grid}}_{i,j,N} - b_{i,j}$, where $b_{i,j}$ is the basal melt rate. Integrating from the bottom, the vertical velocity is then

$$\begin{aligned} w_{i,j,k} = -\sum_{\tilde{k}=N-1}^{1} \Bigg\{ \mathcal{H}_{i,j} \left(\frac{u^x_{i,j,k} + u^x_{i,j,k+1}}{2} + \frac{v^y_{i,j,k} + v^y_{i,j,k+1}}{2} \right) (\sigma_{k+1} - \sigma_k) \\ + (\tilde{u}_{i,j,k+1} - \tilde{u}_{i,j,k}) \left(\tilde{s}^x_{i,j} - \frac{1}{2}(\sigma_{k+1} + \sigma_k) \tilde{H}^x_{i,j} \right) \\ + (\tilde{v}_{i,j,k+1} - \tilde{v}_{i,j,k}) \left(\tilde{s}^y_{i,j} - \frac{1}{2}(\sigma_{k+1} + \sigma_k) \tilde{H}^y_{i,j} \right) \Bigg\} + w_{i,j,N} \end{aligned} \tag{3.36}$$

with the weighted ice thickness

$$\begin{aligned} \mathcal{H}_{i,j} = \frac{4H_{i,j} + 2(H_{i-1,j} + H_{i+1,j} + H_{i,j-1} + H_{i,j+1})}{16} \\ + \frac{H_{i-1,j-1} + H_{i+1,j-1} + H_{i+1,j+1} + H_{i-1,j+1}}{16} \end{aligned}$$

This scheme produces vertical velocities at the ice divide which are too small. The vertical velocities on the ice surface are given by the upper kinematic boundary condition, Equation (3.10). Equation (3.36) can be corrected with:

$$w^*_{i,j,k} = w_{i,j,k} - (1 - \sigma_k)(w_{i,j,k} - w_{si,j}), \tag{3.37}$$

where $w_{si,j}$ is the vertical velocity at the ice surface given by (3.10). Figure 3.6 shows the different vertical velocities at the ice surface. The difference between the vertical velocities

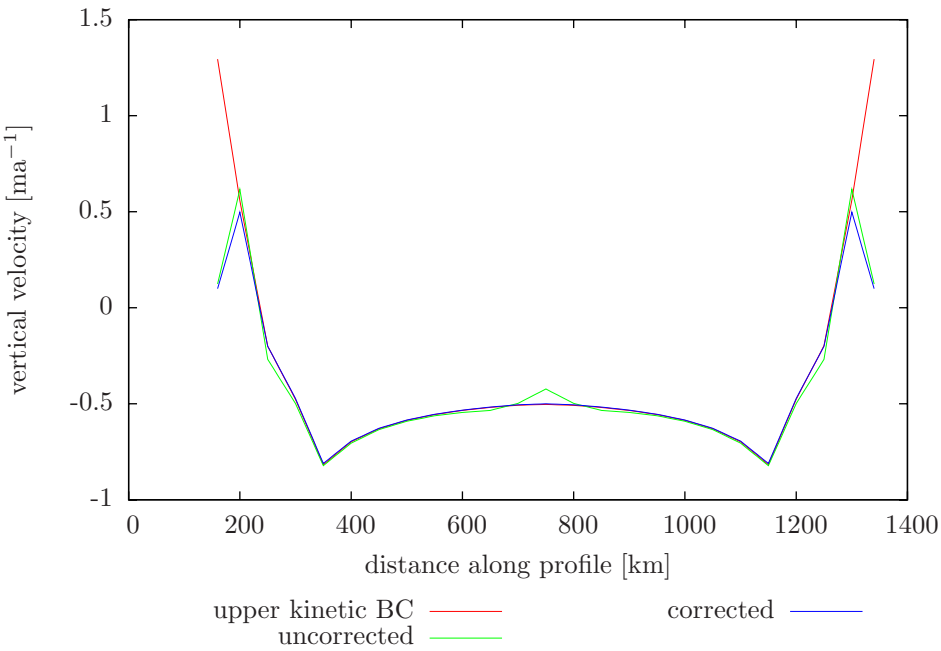

Figure 3.6: Vertical ice surface velocities of the EISMINT-1 moving margin experiment.

calculated by the model and the vertical velocities given by (3.10) at the ice margin are due

to the fact that temperatures and velocities are only calculated when the ice is thicker than a certain threshold value which is not met at the ice margin.

Figure 3.7 shows vertical profiles of the vertical velocity at the ice divide and a point half–way between the divide and the domain margin. A corresponding temperature profile is also shown since the vertical velocity determines the vertical temperature advection (see Section 3.2.4).

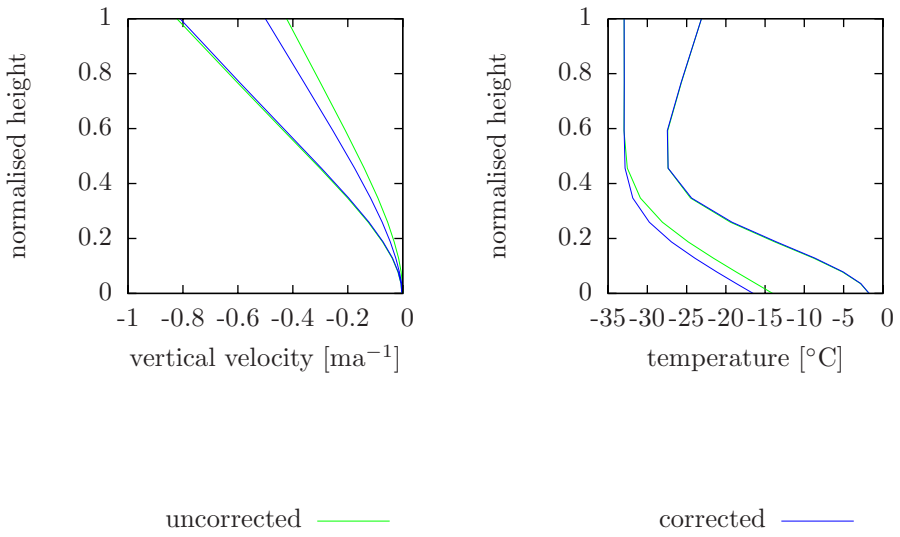

Figure 3.7: Vertical velocity and temperature distribution for columns at the ice divide and a point half–way between the divide and the domain margin.

3.2 Temperature Solver

The flow law, Equation (3.3), depends on the temperature of ice. It is, therefore, necessary to determine how the distribution of ice temperatures changes with a changing ice sheet configuration. The thermal evolution of the ice sheet is described by

$$\frac{\partial T}{\partial t} = \frac{k}{\rho c}\nabla^2 T - \boldsymbol{u}\cdot\boldsymbol{\nabla} T + \frac{\Phi}{\rho c} - w\frac{\partial T}{\partial z}, \tag{3.38}$$

where T is the absolute temperature, k is the thermal conductivity of ice, c is the specific heat capacity and Φ is the heat generated due to internal friction. In the σ–coordinate system, Equation (3.38), becomes

$$\frac{\partial T}{\partial t} = \frac{k}{\rho c H^2}\frac{\partial^2 T}{\partial \sigma^2} - \boldsymbol{u}\cdot\boldsymbol{\nabla} T + \frac{\sigma g}{c}\frac{\partial \boldsymbol{u}}{\partial \sigma}\cdot\boldsymbol{\nabla} s + \frac{1}{H}\frac{\partial T}{\partial \sigma}\left(w - w_{\text{grid}}\right) \tag{3.39}$$

The terms represents (1) vertical diffusion, (2) horizontal advection, (3) internal heat generation due to friction and (4) vertical advection and a correction due to the sigma coordinate system. Let's rewrite (3.39) to introduce some names:

$$\frac{\partial T}{\partial t} = a\frac{\partial^2 T}{\partial \sigma^2} + b(\sigma) + \Phi(\sigma) + c(\sigma)\frac{\partial T}{\partial \sigma}, \tag{3.40}$$

where

$$a = \frac{k}{\rho c H^2} \tag{3.41a}$$

$$b(\sigma) = -\boldsymbol{u} \cdot \boldsymbol{\nabla} T \tag{3.41b}$$

$$\Phi(\sigma) = \frac{\sigma g}{c} \frac{\partial \boldsymbol{u}}{\partial \sigma} \cdot \boldsymbol{\nabla} s \tag{3.41c}$$

$$c(\sigma) = \frac{1}{H} (w - w_{\text{grid}}) \tag{3.41d}$$

3.2.1 Vertical Diffusion

Discretisation of $\partial^2 T / \partial \sigma^2$ is slightly complicated because the vertical grid is irregular. Using Taylor series the central difference formulas are

$$\left.\frac{\partial T}{\partial \sigma}\right|_{\sigma_{k-1/2}} = \frac{T_k - T_{k-1}}{\sigma_k - \sigma_{k-1}} \tag{3.42a}$$

and

$$\left.\frac{\partial T}{\partial \sigma}\right|_{\sigma_{k+1/2}} = \frac{T_{k+1} - T_k}{\sigma_{k+1} - \sigma_k} \tag{3.42b}$$

The second partial derivative is then, also uning central differences:

$$\left.\frac{\partial^2 T}{\partial \sigma^2}\right|_{\sigma_k} = \frac{\partial T/\partial \sigma|_{\sigma_{k+1/2}} - \partial T/\partial \sigma|_{\sigma_{k-1/2}}}{1/2\,(\sigma_{k+1} - \sigma_{k-1})} \tag{3.42c}$$

Inserting (3.42a) and (3.42b) into (3.42c), we get:

$$= \frac{2(T_{k+1} - T_k)}{(\sigma_{k+1} - \sigma_k)(\sigma_{k+1} - \sigma_{k-1})} - \frac{2(T_k - T_{k-1})}{(\sigma_k - \sigma_{k-1})(\sigma_{k+1} - \sigma_{k-1})} \tag{3.42d}$$

Finally, the terms of equation (3.42d) are rearranged:

$$\left.\frac{\partial^2 T}{\partial \sigma^2}\right|_{\sigma_k} = \frac{2T_{k-1}}{(\sigma_k - \sigma_{k-1})(\sigma_{k+1} - \sigma_{k-1})} - \frac{2T_k}{(\sigma_{k+1} - \sigma_k)(\sigma_k - \sigma_{k-1})} + \frac{2T_{k+1}}{(\sigma_{k+1} - \sigma_k)(\sigma_{k+1} - \sigma_{k-1})} \tag{3.43}$$

3.2.2 Horizontal Advection

The horizontal advection term, $-\boldsymbol{u} \cdot \boldsymbol{\nabla} T$ is solved using an upwinding scheme. Let's start with the 1–dimensional case. The method discussed can be straightforwadly extented to 2D. As always, the temperature function is expressed as a Taylor series.

$$T(x + \Delta x) = T(x) + \Delta x T'(x) + \frac{\Delta x^2}{2} T''(x) + \ldots \tag{3.44a}$$

If we subsitute Δx with $2\Delta x$, Equation (3.44a)

$$T(x + 2\Delta x) = T(x) + 2\Delta x T'(x) + 2\Delta x^2 T''(x) + \ldots \tag{3.44b}$$

From (3.44a) and (3.44b) we can construct a difference formula where the $\mathcal{O}(\Delta x^2)$ error is cancelled, by multiplying (3.44a) with 4 and substracting the result from (3.44b):

$$T'_+(x) = \frac{4T(x+\Delta x) - T(x+2\Delta x) - 3T(x)}{2\Delta x} \tag{3.45a}$$

and similarly for the backward difference:

$$T'_-(x) = -\frac{4T(x-\Delta x) - T(x-2\Delta x) - 3T(x)}{2\Delta x} \tag{3.45b}$$

So the horizontal advection term in one dimensions becomes:

$$b_x = -u_x \frac{\partial T}{\partial x} = \frac{-u_x}{2\Delta x} \begin{cases} -(4T_{i-1} - T_{i-2} - 3T_i) & \text{when } u_x > 0 \\ 4T_{i+1} - T_{i+2} - 3T_i & \text{when } u_x < 0 \end{cases} \tag{3.46}$$

A similar expression is found for b_y by simply substituting y for x. Finally, the combined horizontal advection term, is simply

$$b = -\boldsymbol{u} \cdot \boldsymbol{\nabla} T = -\left(u_x \frac{\partial T}{\partial x} + u_y \frac{\partial T}{\partial y} \right) = b_x + b_y = b_1 + b_2 T_i \tag{3.47}$$

3.2.3 Heat Generation

Taking the derivative of (3.18) with respect to σ, we get

$$\frac{\partial u_x}{\partial \sigma} = -2(\rho g)^n H^{n+1} |\boldsymbol{\nabla} s|^{n-1} \frac{\partial s}{\partial x} A(T^*) \sigma^n \tag{3.48}$$

Thus,

$$\begin{aligned} \Phi(\sigma) &= \frac{\sigma g}{c} \frac{\partial \boldsymbol{u}}{\partial \sigma} \cdot \boldsymbol{\nabla} s = \frac{\sigma g}{c} \left(\frac{\partial u_x}{\partial \sigma} \frac{\partial s}{\partial x} + \frac{\partial u_y}{\partial \sigma} \frac{\partial s}{\partial y} \right) \\ &= -2(\rho g)^n H^{n+1} |\boldsymbol{\nabla} s|^{n-1} \frac{\sigma g}{c} A(T^*) \sigma^n \left(\left(\frac{\partial s}{\partial x} \right)^2 + \left(\frac{\partial s}{\partial y} \right)^2 \right) \\ &= -\frac{2}{c\rho} (g\sigma\rho)^{n+1} \left(H |\boldsymbol{\nabla} s| \right)^{n+1} A(T^*) \end{aligned} \tag{3.49}$$

The constant factor $\frac{2}{c\rho}(g\sigma\rho)^{n+1}$ is calculated during initialisation in the subroutine `init_temp`. This factor is assigned to array `c1(1:upn)`. `c1` also includes various scaling factors and the factor $1/16$ to normalise $\mathcal{A}$.

The next factor, $\left(H |\boldsymbol{\nabla} s| \right)^{n+1}$ is calculated in the subroutine `finddisp`:

$$c_{2i,j} = \left(\tilde{H}_{i,j} \sqrt{\tilde{S}_{x\,i,j}^{\,2} + \tilde{S}_{y\,i,j}^{\,2}} \right)^{n+1}, \tag{3.50}$$

The final factor is found by averaging over the neighbouring nodes:

$$\mathcal{A}_{i,j} = 4A_{i,j} + 2(A_{i-1,j} + A_{i+1,j} + A_{i,j-1} + A_{i,j+1}) + (A_{i-1,j-1} + A_{i+1,j-1} + A_{i+1,j+1} + A_{i-1,j+1}) \tag{3.51}$$

3.2.4 Vertical Advection

The vertical advection term, $\partial T / \partial \sigma$ is solved using the central difference formula for unevenly spaced nodes:

$$\frac{\partial T}{\partial \sigma} = \frac{T_{k+1} - T_{k-1}}{\sigma_{k+1} - \sigma_{k-1}} \tag{3.52}$$

3.2.5 Boundary Conditions

At the upper boundary, ice temperatures are set to the surface temperature, T_{surf}. The ice at the base is heated by the geothermal heat flux and sliding friction:

$$\left.\frac{\partial T}{\partial \sigma}\right|_{\sigma=1} = -\frac{GH}{k} - \frac{H\boldsymbol{\tau}_b \cdot \boldsymbol{u}(1)}{k}, \tag{3.53}$$

where $\boldsymbol{\tau}_b = -\rho g H \boldsymbol{\nabla} s$ is the basal shear stress and $\boldsymbol{u}(1)$ is the basal ice velocity. Ice temperatures are held constant if they reach the pressure melting point of ice, i.e.

$$T^* = T_{\text{pmp}} \quad \text{if } T \geq T_{\text{pmp}}. \tag{3.54}$$

Excess heat is then used to formulate a melt rate, S:

$$S = \frac{k}{\rho L}\left(\frac{\partial T^*}{\partial z} - \frac{\partial T}{\partial z}\right), \tag{3.55}$$

where L is the specific latent heat of fusion. Finally, basal temperatures are held constant, if the ice is floating:

$$\frac{\partial T(1)}{\partial t} = 0. \tag{3.56}$$

3.2.6 Putting it all together

Equation (3.39) is solved for each ice column. The horizontal dependency of the horizontal advection term, (3.41b), is resolved by iterating the vertical solution. Putting the individual terms together using a fully explicit finite differences scheme, Equation (3.40) becomes

$$\begin{aligned}\frac{T_{k,t+1} - T_{k,t}}{\Delta t} = \Bigg(&\frac{2aT_{k-1,t}}{(\sigma_k - \sigma_{k-1})(\sigma_{k+1} - \sigma_{k-1})} - \frac{2aT_{k,t}}{(\sigma_{k+1} - \sigma_k)(\sigma_k - \sigma_{k-1,t})} \\ &+ \frac{2aT_{k+1,t}}{(\sigma_{k+1} - \sigma_k)(\sigma_{k+1} - \sigma_{k-1})}\Bigg) + b_{1k,t} + b_{2k}T_{k,t} + \Phi_k + c_k\frac{T_{k+1,t} - T_{k-1,t}}{\sigma_{k+1} - \sigma_{k-1}}\end{aligned} \tag{3.57a}$$

and similarly the fully implicit scheme

$$\begin{aligned}\frac{T_{k,t+1} - T_{k,t}}{\Delta t} = \Bigg(&\frac{2aT_{k-1,t+1}}{(\sigma_k - \sigma_{k-1})(\sigma_{k+1} - \sigma_{k-1})} - \frac{2aT_{k,t+1}}{(\sigma_{k+1} - \sigma_k)(\sigma_k - \sigma_{k-1,t+1})} \\ &+ \frac{2aT_{k+1,t+1}}{(\sigma_{k+1} - \sigma_k)(\sigma_{k+1} - \sigma_{k-1})}\Bigg) + b_{1k,t+1} + b_{2k}T_{k,t+1} + \Phi_k + c_k\frac{T_{k+1,t+1} - T_{k-1,t+1}}{\sigma_{k+1} - \sigma_{k-1}}\end{aligned} \tag{3.57b}$$

Taking the average of Equations (3.57a) and (3.57b) gives the *Crank–Nicholson scheme*. The resulting equation is then rearranged and terms of $T_{k-1,t+1}$, $T_{k,t+1}$ and $T_{k+1,t+1}$ are combined to give the tri–diagonal system

$$\alpha_k T_{k-1,t+1} + \beta_k T_{k,t+1} + \gamma_k T_{k+1,t+1} = \delta_k \tag{3.58}$$

where, for $k = 2, N-1$

$$\alpha_k = -\frac{1}{2}\frac{2a\Delta t}{(\sigma_k - \sigma_{k-1})(\sigma_{k+1} - \sigma_{k-1})} + \frac{1}{2}\frac{c_k\Delta t}{\sigma_{k+1} - \sigma_{k-1}} \tag{3.59a}$$

$$\beta_k = 1 + \frac{1}{2}\frac{2a\Delta t}{(\sigma_{k+1} - \sigma_k)(\sigma_k - \sigma_{k-1})} - \frac{1}{2}b_{2k}\Delta t = 1 - \alpha_k - \gamma_k - \frac{1}{2}b_{2k}\Delta t \tag{3.59b}$$

$$\gamma_k = -\frac{1}{2}\frac{2a\Delta t}{(\sigma_{k+1} - \sigma_k)(\sigma_{k+1} - \sigma_{k-1})} - \frac{1}{2}\frac{c_k\Delta t}{\sigma_{k+1} - \sigma_{k-1}} \tag{3.59c}$$

$$\delta_k = -\alpha_k T_{k-1,t} + (2 - \beta_k)T_{k,t} - \gamma_k T_{k+1,t} + \frac{1}{2}(b_{1k,t} + b_{1k,t+1})\Delta t + \Phi_k\Delta t \tag{3.59d}$$

Boundary Conditions

At the upper boundary:

$$\alpha_1 = 0, \quad \beta_1 = 1, \quad \gamma_1 = 0, \quad \delta_1 = T_{\text{surf}} \tag{3.59e}$$

The lower boundary condition is somewhat more complicated. Here we only look at the case when the temperature is below the pressure melting point of ice. BC for floating ice and temperatures at the pressure melting point of ice are trivial. The geothermal heat flux is applied at the lower boundary, i.e. Equation (3.42b) becomes

$$\left.\frac{\partial T}{\partial \sigma}\right|_{\sigma_{k+1/2}} = -\frac{GH}{k} \tag{3.60}$$

Assuming that $\sigma_k - \sigma_{k-1} = \sigma_{k+1} - \sigma_k = \Delta\sigma$ and inserting (3.42a) and (3.60) into (3.42c), the second partial derivative becomes

$$\left.\frac{\partial^2 T}{\partial \sigma^2}\right|_{\sigma_N} = \left(-\frac{GH}{k} - \frac{T_N - T_{N-1}}{\Delta\sigma}\right)/\Delta\sigma = -\frac{GH}{k\Delta\sigma} - \frac{T_N - T_{N-1}}{\Delta\sigma^2} \tag{3.61}$$

Inserting the new conduction term and replacing the derivative of the vertical advection term with the Neuman boundary condition, Equation (3.57a) becomes

$$\frac{T_{N,t+1} - T_{N,t}}{\Delta t} = -a\left(\frac{GH}{k\Delta\sigma} + \frac{T_{N,t} - T_{N-1,t}}{\Delta\sigma^2}\right) + b_{1\,N,t} + b_{2\,N} T_{N,t} + \Phi_N - c_N \frac{GH}{k} \tag{3.62a}$$

and similarly for Equation (3.57b)

$$\frac{T_{N,t+1} - T_{N,t}}{\Delta t} = -a\left(\frac{GH}{k\Delta\sigma} + \frac{T_{N,t+1} - T_{N-1,t+1}}{\Delta\sigma^2}\right) + b_{1\,N,t+1} + b_{2\,N} T_{N,t+1} + \Phi_N - c_N \frac{GH}{k} \tag{3.62b}$$

The elements of the tri–diagonal system at the lower boundary are then

$$\alpha_N = -\frac{a\Delta t}{2(\sigma_N - \sigma_{N-1})^2} \tag{3.63a}$$

$$\beta_N = 1 - \alpha_N + \frac{1}{2} b_{2\,N} \Delta t \tag{3.63b}$$

$$\gamma_N = 0 \tag{3.63c}$$

$$\begin{aligned}\delta_N = &- \alpha_N T_{N-1,t} + (2 - \beta_N) T_{N,t} - a\frac{GH\Delta t}{k(\sigma_N - \sigma_{N-1})} \\ &+ \frac{1}{2}(b_{1\,N,t} + b_{1\,N,t+1})\Delta t + \Phi_N \Delta t - c_N \frac{GH\Delta t}{k}\end{aligned} \tag{3.63d}$$

3.3 Basal Boundary Condition

The Section describes the formulation of the basal boundary condition. An interface for the upper boundary condition (atmospheric BC) is easily defined by the surface temperature and mass balance. Similarly, the basal boundary consists of mechanical and thermal boundary conditions. The complications arise because the thermal and mechanical boundary conditions depend on each other. The interface of the basal boundary can be described with the following fields (see also Fig.3.8):

1. **basal traction:** this field specifies a parameter which is used to allow basal sliding.

2. **basal heat flux:** heat flux entering the ice sheet from below.

3. **basal water depth:** the presence of basal melt water affects the basal ice temperature

Additionally, the ice sheet model calculates a melt/freeze rate based on the temperature gradient and basal water depth. This is handled by GLIDE.

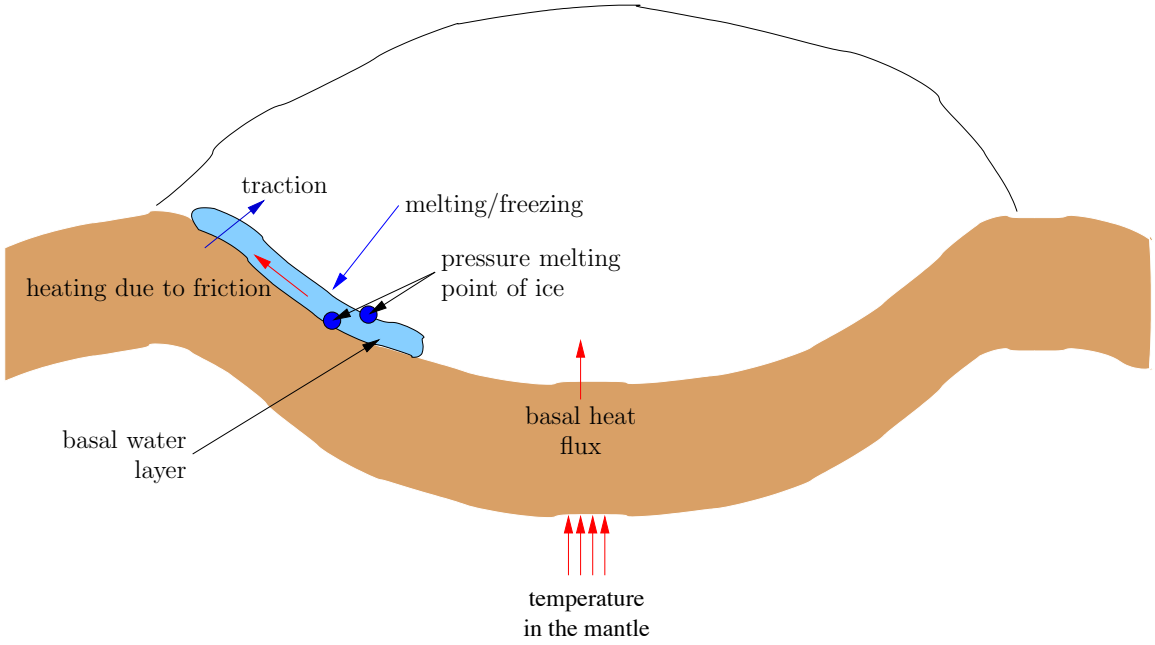

Figure 3.8: Basal boundary condition.

3.3.1 Mechanical Boundary Conditions

If the ice is not frozen to the bed, basal décollement may occur. This can be parameterised by a traction factor, t_b. Within the ice sheet model t_b is used to either calculate basal sliding velocities, $\boldsymbol{u}_b$, in the case of zeroth order physics, i.e.

$$\boldsymbol{u}_b = t_b \boldsymbol{\tau}_b \tag{3.64}$$

where τ_b is the basal shear stress. Alternatively, t_b can be used as part of the stress–balance calculations when the model is used with higher order physics. In simple models t_b may be uniform or prescribed as a spatial variable. More complex models may wish to make t_b dependant on other variables, e.g. basal melt rate. Typically t_b will depend on the presence of basal water.

The second mechanical boundary condition, basal melting/freeze–on $\dot{B}$, is handled within the ice sheet model. The details are described in Section 3.3.2.

3.3.2 Thermal Boundary Conditions

The thermal boundary condition at the ice base is more complicated than the mechanical BC. The ice is heated from below by the geothermal heat flux. Heat is generated by friction with the bed. Furthermore, the ice temperature is constrained to be smaller or equal to the pressure melting point of ice. The thermal boundary is set to the basal heat flux if there is no water present. If there is water, the thermal boundary condition is set to the pressure melting temperature[1].

[1] if it was lower there would be no water, if it was higher than there would be no ice

Basal Melting and Freezing

At the ice base, $z = h$, we can define outgoing and incoming heat fluxes, H_o and H_i:

$$H_o = -k_{\text{ice}} \left.\frac{\partial T}{\partial z}\right|_{z=h^+} \tag{3.65a}$$

and

$$H_i = -k_{\text{rock}} \left.\frac{\partial T}{\partial z}\right|_{z=h^-} + \boldsymbol{u}_b \cdot \boldsymbol{\tau}_b + \begin{cases} \rho_{\text{ice}} \dot{B}/L & \text{when } \dot{B} < 0 \\ 0 & \text{otherwise} \end{cases} \tag{3.65b}$$

where k_{ice} and k_{rock} are the thermal conductivities of ice and rock, $\boldsymbol{u}_b \cdot \boldsymbol{\tau}_b$ is the heat generated by friction with the bed and L is the latent heat of fusion of pure water. The basal melt/freeze–on rate, $\dot{B}$ can then be calculated from the difference between the incoming and outgoing heat fluxes:

$$\dot{B} = \frac{H_o - H_i}{\rho_{\text{ice}} L} \tag{3.66}$$

Freeze–on occurs if $\dot{B}$ is negative, basal melting occurs if $\dot{B}$ is positive.

Geothermal Heat Flux

The heat flux accross the basal boundary depends on past temperature variations since temperature perturbations penetrate the bed rock if the ice is frozen to the ground (Ritz, 1987). The heat equation for the bed rock layer is given by the diffusion equation

$$\frac{\partial T}{\partial t} = \frac{k_{\text{rock}}}{\rho_{\text{rock}} c_{\text{rock}}} \nabla^2 T = \frac{k_{\text{rock}}}{\rho_{\text{rock}} c_{\text{rock}}} \left(\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} + \frac{\partial^2 T}{\partial z^2} \right), \tag{3.67}$$

where k_{rock} is the thermal conductivity, ρ_{rock} the density and c_{rock} the specific heat capacity of the bed rock layer.

Initial conditions for the temperature field T are found by applying the geothermal heat flux, G to an arbitrary surface temperature T_0:

$$T(x, y, z) = T_0 + \frac{G}{k_{\text{rock}}} z. \tag{3.68}$$

This ensures that initially the geothermal heat flux experienced by the ice sheet is equal to the regional heat flux. The basal boundary condition of the bedrock layer is kept constant, i.e.

$$T(x, y, H_{\text{rock}}) = T_0 + \frac{G}{k_{\text{rock}}} H_{\text{rock}}. \tag{3.69}$$

Lateral boundary conditions are given by

$$\left.\frac{\partial T}{\partial x}\right|_{x=0} = \left.\frac{\partial T}{\partial x}\right|_{x=L_x} = \left.\frac{\partial T}{\partial y}\right|_{y=0} = \left.\frac{\partial T}{\partial y}\right|_{y=L_y} = 0. \tag{3.70}$$

At the upper boundary, the heat flux of the rock layer has to be matched with the heat flux in the basal ice layer when the ice is frozen to the bed, i.e.

$$k_{\text{rock}} \left.\frac{\partial T}{\partial z}\right|_{z=-0} = k_{\text{ice}} \left.\frac{\partial T}{\partial z}\right|_{z=+0}. \tag{3.71}$$

Otherwise the temperature of the top bedrock layer is set to the surface temperature (if the cell has been occupied by ice, but there is no ice present) or the basal ice temperature (if there is ice). Equation (3.71) is automatically fulfilled if we set the top bedrock temperature to the basal ice temperature *everywhere* and then calculate the geothermal heat flux to be used as boundary condition for Equation (3.38).

3.3.3 Numerical Solution

The horizontal grid is described in Section 3.1.1. The vertical grid is irregular like the vertical grid of the ice sheet model. However, it is not scaled. Also for now, I have ignored topography or isostatic adjustment, i.e. the bedrock layer is assumed to be flat and constant.

The horizontal second derivative in Equation (3.67) becomes using finite–differences

$$\left.\frac{\partial^2 T}{\partial x^2}\right|_{x_i,y_i,z_i} = T_{xx,i,j,k} = \frac{T_{i+1,j,k} - 2T_{i,j,k} + T_{i-1,j,k}}{\Delta x} \tag{3.72}$$

and similarly for $\partial^2 T/\partial y^2$. The vertical second derivative $\partial^2 T/\partial z^2$ is similar to Equation (3.43):

$$\left.\frac{\partial^2 T}{\partial z^2}\right|_{x_i,y_i,z_i} = T_{zz,i,j,k} = \frac{2T_{i,j,k-1}}{(z_k - z_{k-1})(z_{k+1} - z_{k-1})} - \frac{2T_{i,j,k}}{(z_{k+1} - z_k)(z_k - z_{k-1})} + \frac{2T_{i,j,k+1}}{(z_{k+1} - z_k)(z_{k+1} - z_{k-1})} \tag{3.73}$$

Using the Crank-Nicholson scheme, Equation (3.67) becomes

$$\frac{T^{t+1}_{i,j,k} - T^t_{i,j,k}}{\Delta t} = D\left\{\frac{T^{t+1}_{xx,i,j,k} + T^t_{xx,i,j,k}}{2} + \frac{T^{t+1}_{yy,i,j,k} + T^t_{yy,i,j,k}}{2} + \frac{T^{t+1}_{zz,i,j,k} + T^t_{zz,i,j,k}}{2}\right\}, \tag{3.74}$$

with $D = k_{\text{rock}}/(\rho_{\text{rock}} c_{\text{rock}})$. Equation (3.74) is solved by gathering all T^{t+1} terms on the LHS and all other terms on the RHS. The index (i, j, k) is linearised using $\iota = i+(j-1)N+(k-1)NM$. The resulting matrix system is solved using the same bi–conjugate gradient solver as for the ice thickness evolution.

3.3.4 Basal Hydrology

It is clear from the discussion above that the presence of basal water plays a crucial role in specifying both the mechanical and thermal boundary conditions. However, the treatment of basal water can vary greatly. Basal water is, therefore, left as an unspecified interface. GLIDE does provide a simple local water balance model which can be run in the absence of more complex models.

3.3.5 Putting It All Together

The basal boundary consists of the individual components described in the previous sections. All components are tightly linked with each other. Figure 3.9 illustrates how the modules are linked and in what order they are resolved. The order of executions is then:

1. Find the basal heat flux by either solving the equation describing the thermal evolution of the lithosphere, Eq. (3.67), or by using the geothermal heat flux directly. The upper boundary condition of (3.67) is the same as the lower boundary condition of the thermal evolution of the ice sheet.

2. The lower boundary condition for the thermal evolution of the ice sheet is either given by the basal heat flux from *Step 1*; or if melt water is present the basal temperature is set to the pressure melting point of ice.

3. Calculate the temperature distribution within the ice sheet given the boundary condition found during *Step 2* and the atmospheric BC.

4. Calculate a melt/freeze–on rate using Equation (3.66) given the outgoing heat flux calculated during *Step 3*, friction with the bed (calculated during the previous *Step 7*) and the incoming heat flux from *Step 1*. Freezing only occurs when there is basal water.

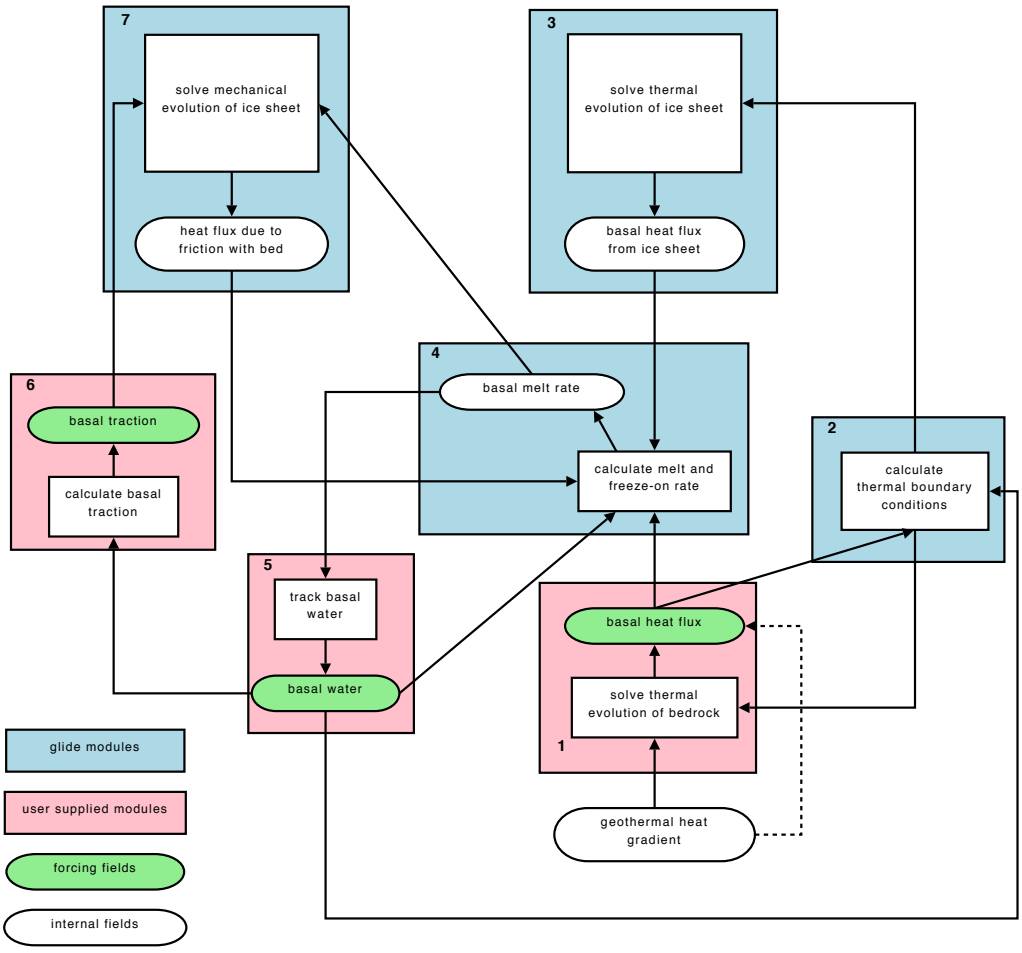

Figure 3.9: Flow diagram illustrating how the various modules communicate with each other by exchanging data fields.

5. Track basal water. This is a user supplied module which can take any complexity. Inputs will typically be the melt/freeze–on rate determined during *Step 4*.

6. Calculate the basal traction parameter. Again, this is a user supplied module which typically will involve the presence of basal melt water (calculated during *Step 5*).

7. Solve the mechanical ice equations given basal traction parameter from *Step 6*.

Clearly, this scheme has the problem that heat is lost if the basal heat flux is such that more water could be frozen than is available. This might be avoided by iterating the process. On the other hand if time steps are fairly small this might no matter to much.

3.4 Isostatic Adjustment

The ice sheet model includes simple approximations for calculating isostatic adjustment. These approximations depend on how the lithosphere and the mantle are treated. For each subsystem there are two models. The lithosphere can be described as a

local lithosphere: the flexural rigidity of the lithosphere is ignored, i.e. this is equivalent to ice floating directly on the asthenosphere;

elastic lithosphere: the flexural rigidity is taken into account;

while the mantle is treated as a

fluid mantle: the mantle behaves like a non-viscous fluid, isostatic equilibrium is reached instantaneously;

relaxing mantle: the flow within the mantle is approximated by an exponentially decaying hydrostatic response function, i.e. the mantle is treated as a viscous half space.

3.4.1 Calculation of ice-water load

At each isostasy time-step, the load of ice and water is calculated, as an equivalent mantle-depth (L). If the basal elevation is above sea-level, then the load is simply due to the ice:

$$L = \frac{\rho_i}{\rho_m} H, \tag{3.75}$$

where H is the ice thickness, with ρ_i and ρ_m being the densities of the ice and mantle respectively. In the case where the bedrock is below sea-level, the load is calculated is that due to a change in sea-level rise and/or the presence of non-floating ice. When the ice is floating ($\rho_i H < \rho_o(z_0 - h)$), the load is only due to sea-level changes

$$L = \frac{\rho_o}{\rho_m} z_0, \tag{3.76}$$

whereas when the ice is grounded, it displaces the water, and adds an additional load:

$$L = \frac{\rho_i H + \rho_o h}{\rho_m}. \tag{3.77}$$

here, ρ_o is the density of sea water, z_0 is the change in sea-level relative to a reference level and h is the bedrock elevation relative to the same reference level. The value of h will be negative for submerged bedrock, hence the plus sign in (3.77).

3.4.2 Elastic lithosphere model

This is model is selected by setting `lithosphere = 1` in the configuration file. By simulatuing the deformation of the lithosphere, the deformation seen by the aesthenosphere beneath is calculated. In the absence of this model, the deformation is that due to Archimedes' Principle, as though the load were floating on the aesthenosphere.

The elastic lithosphere model is based on work by Lambeck and Nakiboglu (1980), and its implementation is fully described in Hagdorn (2003). The lithosphere model only affects the geometry of the deformation — the timescale for isostatic adjustment is controlled by the aesthenosphere model.

The load due to a single (rectangular) grid point is approximated as being applied to a disc of the same area. The deformation due to a disc of ice of radius A and thickness H is given by these expressions. For $r < A$:

$$w(r) = \frac{\rho_i H}{\rho_m}\left[1 + C_1 \,\mathrm{Ber}\left(\frac{r}{L_r}\right) + C_2 \,\mathrm{Bei}\left(\frac{r}{L_r}\right)\right], \tag{3.78}$$

and for $r \geq A$:

$$w(r) = \frac{\rho_i H}{\rho_m}\left[D_1 \,\mathrm{Ber}\left(\frac{r}{L_r}\right) + D_2 \,\mathrm{Bei}\left(\frac{r}{L_r}\right) + D_3 \,\mathrm{Ker}\left(\frac{r}{L_r}\right) + D_4 \,\mathrm{Kei}\left(\frac{r}{L_r}\right)\right], \tag{3.79}$$

where $\mathrm{Ber}(x)$, $\mathrm{Bei}(x)$, $\mathrm{Ker}(x)$ and $\mathrm{Kei}(x)$ are Kelvin functions of zero order, $L_r = (D/\rho_m g))^{1/4}$ is the radius of relative stiffness, and D is the flexural rigidity. The constants C_i and D_i are given by

$$\begin{aligned} C_1 &= a\,\mathrm{Ker}'(a) \\ C_2 &= -a\,\mathrm{Ker}'(a) \\ D_1 &= 0 \\ D_2 &= 0 \\ D_3 &= a\,\mathrm{Ber}'(a) \\ D_4 &= -a\,\mathrm{Ber}'(a). \end{aligned} \tag{3.80}$$

Here, the prime indicates the first spatial derivative of the Kelvin functions.

3.4.3 Relaxing aesthenosphere model

If a fluid mantle is selected, it adjusts instantly to changes in lithospheric loading. However, a relaxing mantle is also available.

Chapter 4

Developer Guide

4.1 Introduction

Everyone with an interest in ice sheet modelling is encouraged to contribute to GLIMMER. However, the structure of the model is complex, and the coding style is more object-oriented than is generally common in geosciences models. The aim of this chapter is to introduce these characteristics of the model and to suggest some approaches to GLIMMER development. Detail is then provided on some of the modules present within the GLIMMER code base.

In developing new code, several important principles should be borne in mind. These are useful ideas when developing any code, but are especially important when contributing to an established collaborative project like GLIMMER:

- **Modularise/objectify.** Divide the code up into logically self-contained tasks. In a numerical model of a set of physical processes, this usually means taking each process separately, though of course the numerical techniques used will have a bearing on how the division is done — implicit solutions of systems of equations require a more integrated approach. Objectification is covered in more detail below.

- **Don't duplicate.** At all costs, avoid duplicating code within a model, as it makes bug-fixing and other maintenance nightmarish. Put it in a subroutine and call it from different places. Consider doing this even if the duplicated code is slightly different — write the subroutine so it will do these different things.

- **Don't reinvent the wheel.** If an existing piece of code in the model does what you need already, use it. If it nearly does what you need, extend it.

- **Respect the hierarchy of the code.** Some parts of the model are 'core', some are shared utility code, some are extensions/drivers/models of boundary conditions. Avoid creating a birds-nest of dependencies. If a piece of code from an extension acquires general usefulness, consider moving it to one of the shared utility modules.

- **Write for flexibility.** The is almost never any justification for using static arrays (i.e. fixed sizes at compile-time), and certainly the size of an array should be defined at only one point in the code. Likewise, physical and other numerical parameters should be organised in a central place, and made configurable at runtime if there is a chance someone might want to change them. Think beyond what you're doing now to what you might want to do in the future, and write code accordingly.

- **Be conservative.** Although radical restructuring or rewriting can sometimes be necessary, it should only be done after careful consideration of the potential consequences.

Usually, incremental, gradual change is best. Also, when making changes, it's very important to retain the existing functionality and the form of the user interfaces (code interfaces and configuration options) if at all possible. Extend rather than alter.

- **Write comments.** It's almost impossible to write too many comments in your code. It's especially important to document subroutine arguments, including the units of quantities, temporal validity, etc. Remember, in a few weeks' time, you'll probably have forgotten half of it, and will be glad of the comments...
- **Lay out your code neatly.** Indentation of code blocks and leaving blank lines between different sections of code are important ways to make code more readable. Most good editors have an automatic indentation facility — Emacs is a good editor in this respect[1].
- **Use long, meaningful names.** There's not much virtue these days in making variable and subroutine names short and opaque — we don't have issues with memory, and Fortran 90/95 allows names to be up to 31 characters. It may involve a bit more typing to use long names, and care must be taken to ensure they don't get too long, but the extra time spent typing is worth it if the code actually means something when you look at it again a few weeks later.
- **Use version control.** Although it takes a little time to learn to use a version control system, it really does make life easier. If you've ever got confused about which version of the code you or your collaborators have, or wished you could revert the changes you've just made, then you need version control. We use CVS for GLIMMER, but other systems are available. Of these, Subversion is generally seen as a promising alternative and successor to CVS.

The sections that follow cover some of these topics in more detail, but much that passes for good-practice in code development is best learnt from experience, and by looking at existing code.

4.2 Introduction to GLIMMER programming techniques

Although GLIMMER is written in Fortran, a very widely used language in the geosciences, many of the techniques employed in the model will be unfamiliar to even seasoned Fortran programmers. What follows is a brief description of some of these more advanced techniques. For further information about the topics covered, the reader is directed to Metcalf and Reid (1999) and Decyk et al. (1997).

4.2.1 Fortran Modules

Module definition

The basic building block of the GLIMMER structure is the Fortran 90/95 module, which is a way of collecting together subroutines, functions type definitions and variables into a single scope[2]. The module may then be used within another piece of code, so that the names in the modules scope are available in the other piece of code. A module block is defined like this:

`module` *module-name*

[1] Hint: if emacs doesn't enter the F90 edit mode immediately upon opening a file, you need to type *Alt*-`x`, followed by `f90-mode`, and press <*enter*>.

[2] A *scope* or *namespace* is the term for the program unit where a given *name*, such as a variable or subroutine name, is valid. For instance, if a variable is declared at the beginning of a subroutine, the body of the subroutine is that variable's scope; outside the scope, the name is undefined, or may be defined differently.

```
    module variable declarations
    derived-type definitions

contains

    function and subroutine declarations

end module
```

The `contains` is omitted if no subroutines or functions are present. It is good practice to put each module in a separate file, and give the filename the same name as the module.

Using modules

A module is accessed by another piece of code with the `use` statement. Thus, if the module `foo` contains the subroutine `bar`, another piece of code may make use of it in this way:

```
program program-name

    use foo

    implicit none

    variable declarations

    call bar

end program program-name
```

Of course, subroutines, functions and modules can all contain `use` statements as well.

Privacy in modules

By default, all the names in a module's scope become available to any program element that references it with a `use` statement. There are circumstances where this is undesirable, and so Fortran 90/95 provides a way to define names as public or private. The `public` and `private` statements may both be present in the first block of a module (i.e. before the `contains`, if present). The list of variable and/or subroutine names follows. For example:

```
private :: foo, bar, blah
```

It is good practice to be clear about what parts of a module form a public interface, and define this formally. A good way of doing this is to set the default to private, and then set specific names as public:

```
private
public :: foo, bar
```

This technique is useful for avoiding name conflicts when two modules might define internal variables or subroutines with the same name (e.g. `pi` for the value of π). Also, public subroutines are best named in an un-generic way, by prefixing their names with the name of the module. For instance, the module `glimmer_daily_pdd` contains the public subroutines `glimmer_daily_pdd_init` and `glimmer_daily_pdd_mbal`.

4.2.2 Derived types

A derived type is a way of collecting together an arbitrary set of variables and arrays into a composite variable type, instances of which can be addressed with a single name. The concept of derived types takes us some way towards so-called object-oriented programming (OOP) techniques, though there are some important OOP techniques that are not implemented in Fortran 90/95.

A derived type is defined in this kind of way:

```
type typename
    real :: realvar
    integer :: intvar
    real,pointer,dimension(:,:) :: realarr => null()
end type typename
```

So, a derived type can contain any scalar variables, and also pointers (either scalar pointers or array pointers). Fortran 90/95 does not permit derived type elements to be allocatable arrays, but since the behaviour of pointer arrays is very similar, this isn't a serious problem.

The type definition doesn't actually create any variables though. To do that, you need to use the `type` declaration to create an instance of the derived type (known as a *structure*), just as with any other type of variable:

```
type(typename) :: fred, jim, bob
```

This creates three structures of type `typename`, called `fred`, `jim` and `bob`. Structures can be handled like ordinary variables, and passed to subroutines in argument lists, etc. The individual elements may be addressed using the `%` operator:

```
fred%realvar = 4.5
jim%intvar = fred%intvar + 7
allocate(bob%realarr(nx,ny))
```

Note that by default, mathematical operators have no meaning when applied to derived types; writing `bob = fred + jim` isn't allowed. If you want to add the elements of `fred` to the elements of `jim`, it's necessary to write a function to do so, so that you would, for example, write `bob = typename_add(fred,jim)` It is possible to define the meaning of the `+` operator so that it uses the function `typename_add` to perform the addition (a process known as *operator overloading*), but describing how is beyond the scope of this document. In any case, derived types in GLIMMER are not generally used in a way such that arithmetic operations would make sense.

Usually, derived type definitions are put into modules (before `contains`), so that they can easily be used in different pieces of code. The power that this combination of type definitions and modules gives is described in the next section.

4.2.3 Object-orientation with modules and derived types

As the name implies, the *object* is the central concept of object-oriented programming. In traditional OOP terminology, a *class* defines a type of object. The class has particular *attributes*, which might be numerical values, strings or whatever (i.e. they're variables that are part of the class), and *methods*, which are things you can ask an object to *do* to itself (i.e. methods are functions or subroutines which are part of a class).

Explaining OOP with examples is notoriously tricky because the easiest examples of classes are those that mirror classes of objects in the real world, but this inevitably seems a bit contrived. So, although it seems a bit ridiculous, here's one such example... Imagine a class which describes a domestic oven. What are the attributes of an oven, that differentiate one oven from another? These could be the size of the oven, it's type (electric or gas), as well those things that describe

its present state: the thermostat setting, the actual oven temperature, whether the heating element is on or off, and whether the door is open or not

Secondly, what kind of actions can we perform on the oven? Most likely, we want to be able to open and close the oven door, check the present temperature, and set the thermostat. These actions are the methods.

In terms of Fortran 90, we can use the mechanism of modules and derived types to implement a somewhat limited form of OOP:

```
module domestic_oven

  type oven
    real :: width,height,depth
    real :: thermostat
    real :: temperature
    logical :: element      ! .true. for no, .false. for off
    logical :: door         ! .true. for open, .false. for closed
    character(10) :: oven_type
  end type oven

contains

  subroutine oven_set_thermostat(self,temp)

    type(oven) :: self
    real :: temp

    self%temperature = temp

  end subroutine oven_set_thermostat

  ... plus other functions/subroutines

end module domestic_oven
```

A module like this will usually include a subroutine to initialise objects; in this case, you would expect to be able to specify the size of the oven, its type, initial temperature, etc.

So much for computer models of domestic ovens; what use is OOP to ice modellers? Well, using the technique described above is what allows GLIMMER to be used to run several regional ice models simultaneously. It also makes adding new mass-balance models to GLINT much easier. In fact, the principles of OOP are used all over the GLIMMER code, so it is worth getting to grips with them.

4.2.4 Example of OOP in Glimmer

A good, self-contained example of GLIMMER programming style, and the use of OOP is the way that the daily PDD mass-balance scheme (in `glimmer_daily_pdd.F90`) has been implemented. Here, the parameters of the scheme are stored in the derived type `glimmer_daily_pdd_params`, and the public interface of the module is limited to this derived type, and two subroutines (`glimmer_daily_pdd_init` and `glimmer_daily_pdd_mbal`). Everything else is kept private within the module. The initialisation subroutine uses the standard GLIMMER configuration file reader to get input parameters, and the GLIMMER logging mechanism to output data to the screen and/or file. The object-like structure means that it is easy to use the daily PDD model within some other piece of code: all one needs to do is declare an instance of `glimmer_daily_pdd_params`, initialise it appropriately, and then call the `glimmer_daily_pdd_mbal` subroutine. Also, although

the PDD model is currently initialised from file, it would be easy to write an alternative initialisation routine within the module to get the parameters from a subroutine argument list, extending the capabilities of the module rather than changing the present interface.

4.2.5 Pointers

The final more advanced topic covered here is the Fortran pointer. Thankfully, the name is self-explanatory — a pointer is a thing that points to another thing. This might not sound like much use, but since pointers can be made to point at different things as the program runs, and can have memory allocated for them to point to, they can be used to create flexible, dynamic data structures. The thing that a pointer points to is called a *target*.

Pointer basics

Pointers are declared much like an ordinary variable, but with the `pointer` attribute. For example, to declare an integer pointer, one would write this:

```
integer, pointer :: foo => null()
```

The second part of this statement (`=>null()`) initialises the pointer to null (i.e. pointing to nothing). This is only available in Fortran 95, but it is highly desirable to use it, as otherwise the pointer's target is undefined[3].

Note that although we have declared a pointer with this statement, we can't use it as a variable yet, as it isn't pointing at an integer-sized chunk of memory. There are two ways of rectifying this: either the pointer can be made to point to an existing variable, or a new block of memory can be allocated to hold the target. These two methods are known as *pointer assignment* and *pointer allocation*, respectively.

Pointer assignment

Pointing a pointer at something is very simple. The pointer assignment statement uses the `=>` operator:

```
a => b
```

This statement sets pointer `a` to point to target `b`. For this to be valid Fortran, `a` must have the `pointer` attribute, while `b` must be either a variable with the `target` attribute, or another pointer. If the target is a pointer, then the first pointer is set to point at the second pointer's target. This means that

```
a => b
c => a
```

is equivalent to

```
a => b
c => b
```

Pointer allocation

A pointer can be made to point to a newly-allocated chunk of memory using the `allocate` statement, in the same way as an allocatable array is handled:

```
allocate(a)
```

[3] *Undefined* is not the same thing as pointing at nothing. Undefined pointers can cause a program to crash very easily; the lack of null initialisation in Fortran 90 is one of the more serious omissions that Fortran 95 sought to address.

By using the pointer assignment described above, other pointers may be made to point at the same piece of memory:

```
b => a
```

Un-pointing pointers, and avoiding memory leaks.

A pointer can be nullified (made to point to nothing) in two ways: using the `null` function, or the nullify statement:

```
p => null()
nullify(p)
```

Care has to be taken with this, however. Recall the memory allocation example given above:

```
allocate(a)
b => a
```

It is significant that the memory location that `a` and `b` now point to doesn't have a name of its own, which means that if `a` and `b` are subsequently both made to point to something else, or to nothing, the target variable becomes impossible to recover. Consider this sequence of two statements:

```
allocate(a)
a => null()
```

Here, a block of memory is allocated, and `a` is made to point to it. Then, `a` is made to point to nothing. However, the block of memory allocated in the first statement hasn't been deallocated, and is now unrecoverable — and unusable — since we don't have a name to refer to it by. This is known as a *memory leak*, and it is a Bad Thing. It's not disastrous if a few integer-sized blocks of memory suffer this fate, but if large amounts of memory are lost like this, it is quite possible for the program to crash or become very slow as a result.

The proper way to avoid memory leaks is to deallocate memory before it becomes dereferenced, using the `deallocate` statement[4]:

```
allocate(a)
deallocate(a)
a => null()
```

The reason that this isn't done automatically when a pointer is nullified is because there may be other pointers still pointing to that memory location. The process of working out which chunks of memory have been 'orphaned' and so need to be deallocated (known as *garbage collection*) is complex, and compiled languages like Fortran don't usually do it automatically. Avoiding memory leaks therefore depends on careful program design.

Linked lists

A major use for pointers is the linked list, a flexible data structure whose size and structure can be changed dynamically, without having to reallocate it from scratch (which is the case with allocatable arrays).

The principle of the linked list is that each element of the list points to the next element. Extra elements can be added to the end of the list by allocating more memory, and making the final element point to it. Other actions can be done by manipulating the pointers of the various elements.

A typical linked list might use the following derived type:

[4] It's not clear to me whether the assignment of `a` to `null` is necessary in this example. Some compilers take it as read that if a block of memory that a pointer points to is deallocated, that pointer is also nullified, but I am not sure if this is universal.

```
type list
    type(list) :: next => null()
    type(list) :: previous => null()
    integer :: value
end type list
```

So, the type contains pointers that point to the next and previous elements of the list. Describing how to implement subroutines and functions to construct, read and modify the list is beyond the scope of this document, but a full example is provided in Metcalf and Reid (1999).

GLIMMER makes several uses of linked lists and pointer techniques, most notably in the handling of configuration and output files.

4.3 GLIMMER structure and design

4.3.1 Overview

The 'design' of GLIMMER is a consequence of the way it has been developed. Initially, as a stand-alone model with a single domain, module variables were used to hold all model fields and parameters. With the move to use GLIMMER as the ice model component within GENIE, and the desire to enable several active regions to be run simultaneously, the module variables were converted into components of derived types, and an extra layer added on top of the existing structure to deal with global fields and parameters, and deal with the downscaling/interpolation of input fields. A subsequent major reorganisation then allowed the use of the ice model in stand-alone mode again, and the present GLINT-GLIDE structure was born. Doubtlessly, the resulting structure is more complex than necessary, but hopefully it is still reasonably logical.

Currently (July 2006), the `fortran` directory in the GLIMMER source tree contains 83 `.F90` files, including automatically generated NetCDF IO files (more on this below). Figure 1.1 indicates the general relationship between the various parts of GLIMMER; a more comprehensive overview, with filename patterns, is shown in figure 4.1. The various parts of GLIMMER shown

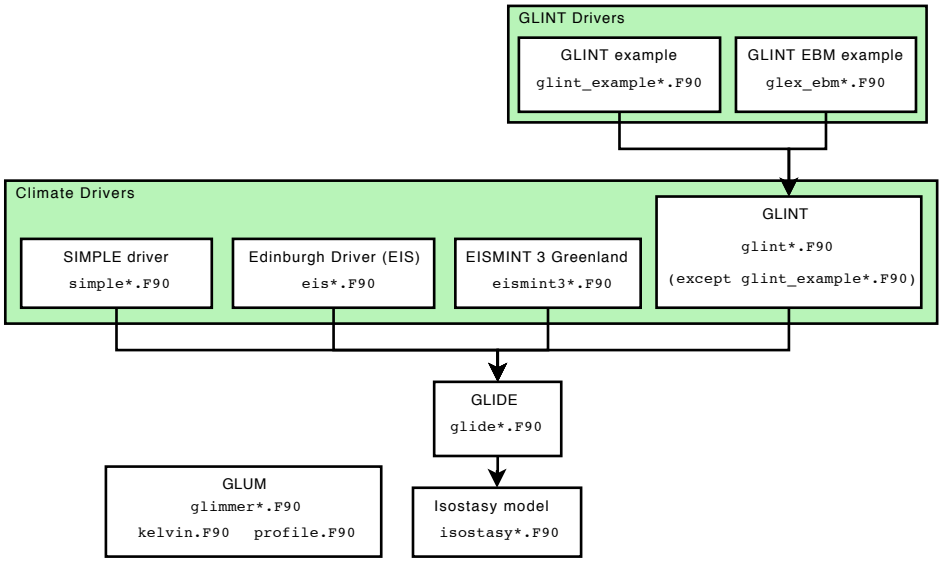

Figure 4.1: Relationship between the various GLIMMER components, giving filenames and dependencies. The modules in GLUM are used by all other elements of the diagram.

in the diagram are as follows:

- **GLIDE** (**GL**IMMER **I**ce **D**ynamics **E**lement). This is the core ice model, whose core interface consists of a derived type definition, and a small number of subroutines to initialise the model, perform time steps and tidy up at the end. In the most basic set up, all that needs to be supplied to GLIDE is the surface mass-balance and surface air temperature fields.
- **GLUM** (**GL**IMMER **U**seful **M**odules). These are modules that contain code which is used by all the other parts of GLIMMER. Some of these (for instance, the configuration parser, NetCDF IO and logging system) are of general use to any model built on top of GLIMMER.
- **Isostasy model.** Although currently only used by GLIDE, the isostasy model is written in such a way as to be easily coupled to another ice model if necessary. Probably this should be moved to within GLUM, though?
- **Climate drivers.** The flexibility of GLIMMER comes from the ease with which GLIDE may be coupled to a custom climate driver. A number of these are provided as part of the GLIMMER code base:
  - **Simple driver.** This is essentially an example driver, almost as simple as possible in design. It is used to implement the idealised EISMINT 1 tests.
  - **Edinburgh Ice Sheet driver (EIS).** This is a driver based on the Edinburgh Model (see Hagdorn (2003), for example).
  - **EISMINT 3 Greenland.** This driver implements the Greenland test cases from the second phase of the EISMINT project (termed EISMINT 3 to distinguish it from the second part of the first EISMINT project...)
  - **GLINT** (**GL**IMMER **Int**erface). This complex driver is designed to form an interface between GLIDE and a global climate model. Originally developed for the GENIE project, it can be used in conjunction with any global model defined on a lat-lon grid. GLINT has its own complex structure of derived types, described below, and is itself supplied with two example climate drivers, `glint_example` and `glex_ebm`.

4.3.2 GLIDE structure

GLIDE is the heart of GLIMMER, the ice dynamics model. The top-level class or derived type is `glide_global_type`, and this contains instances of other derived types, some defined within GLIDE, others within GLUM. The situation is summarised in figure 4.2. All GLIDE derived types are defined in `glide_types.F90`; the others are defined within various GLUM files as shown.

In addition to the relationships illustrated in figure 4.2, there is a web of dependencies based on module `use` statements. This structure is too complex to be illustrated in a dependency diagram; figure 4.3 shows the direct module dependencies of module `glide`.

A brief description of the modules and files that comprise GLIDE is given here:

Module name	Filename	Description
`glide`	`glide.F90`	Top-level GLIDE module. Contains subroutines to read a config file, initialise the model, and perform time-steps.
`glide_io`	`glide_io.F90`	NetCDF IO routines for GLIDE. This file is auto-generated (see below).

continued on next page

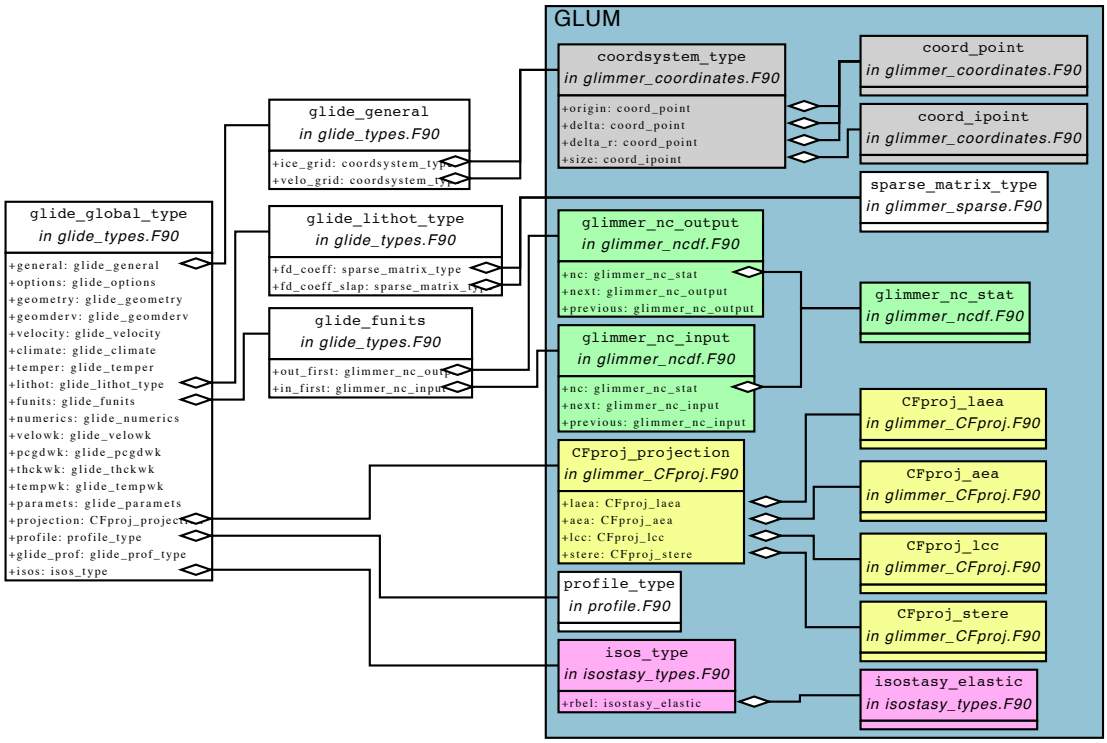

Figure 4.2: Main 'Class Diagram' for GLIDE, in quasi-UML notation. Only derived type components which are themselves derived types are shown; intrinsic type components are omitted. Also omitted are derived types which are defined within GLIDE but which contain no derived type components. Derived types in GLUM are coloured to show which file they are defined in.

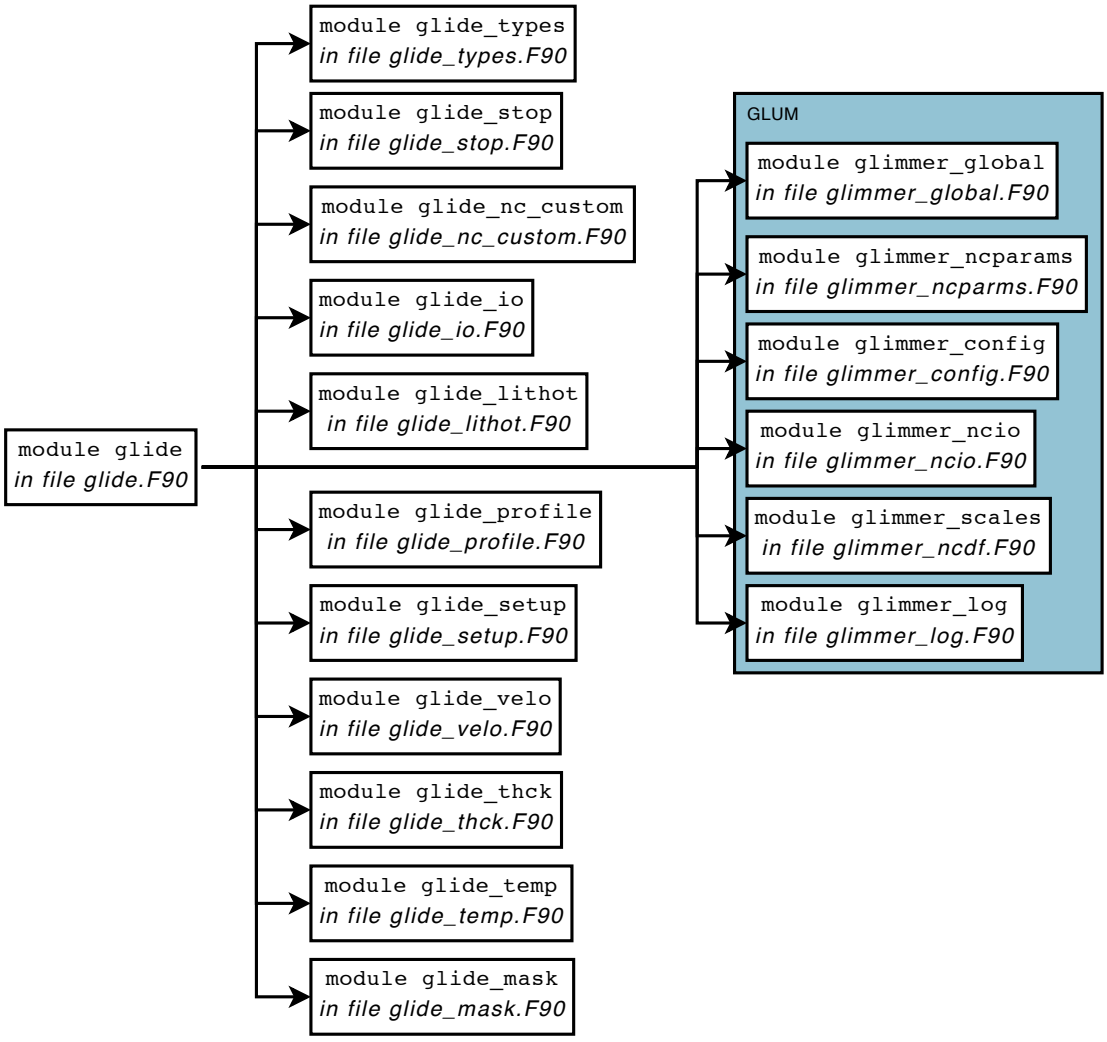

Figure 4.3: Direct module dependency diagram for module `glide` only, illustrating the complexity of module dependencies.

continued from previous page		
glide_lithot	glide_lithot.F90	Top-level module for lithosphere temperature/heat-flux model.
glide_lithot1d	glide_lithot1d.F90	One-dimensional model for lithosphere temperature calculations. Used by glide_lithot.
glide_lithot3d	glide_lithot3d.F90	Three-dimensional model for lithosphere temperature calculations. Used by glide_lithot.
glide_mask	glide_mask.F90	Contains code to determine the properties of each grid-box — whether ice is present, whether it is floating, is a marine margin cell, etc. The output for each cell is a number whose bits each represent a different property.
glide_nc_custom	glide_nc_custom.F90	Code to write contents of dimension variables (x1, x0, etc.) to output NetCDF file.
glide_profile	glide_profile.F90	Wrapper for profile.F90 (determines how much processor time is devoted to each task) with GLIDE-specific configuration. Useful for debugging/analysis.
glide_setup	glide_setup.F90	Subroutines used in GLIDE initialisation (reading configuration files, etc). Also contains a few bits of code used by the model each time step, that should properly be located somewhere else (glide_maskthck, glide_marinlim and glide_calclsrf).
glide_stop	glide_stop.F90	Module for tidying up at the end of a model run (contains only subroutine glide_finalise).
glide_temp	glide_temp.F90	GLIDE ice thermodynamics code.
glide_thck	glide_thck.F90	Thickness evolution code.
glide_types	glide_types.F90	Definitions of all GLIDE derived types (see figure 4.2).
glide_velo	glide_velo.F90	Code for GLIDE ice velocity calculations.

4.3.3 GLINT structure

As the most complex of the driver programs supplied as part of GLIMMER, the structure of GLINT is described next. As with GLIDE, on which it is built, the GLINT structure uses a hierarchy of derived types, shown in figure 4.4. The most significant aspect of this diagram is that the top-level type (glint_params) contains the parameters that are relevant to the coupling on a global level, including an array (instances) of type glint_instance. Each element of this array contains a single GLIDE model instance. The reason for using a wrapper type, rather than having an array of type glide_global_type, is that there is a lot of instance-specific information (the mass-balance model, downscaling/upscaling, etc.) that isn't contained in the GLIDE derived type.

There is some redundancy in this data structure, which should be corrected at some point in the future. Most significant is the presence in both GLIDE and GLINT of data structures describing the map projection of the model. (types projection and CFproj_projection). GLINT uses the former to work out where the model instance is on the globe, and obtains the information from the configuration file; GLIDE reads and writes the latter from the NetCDF files, but doesn't use the information for calculation. Currently, there is no coordination between the two sets of information.

The modules comprising GLINT are as follows:

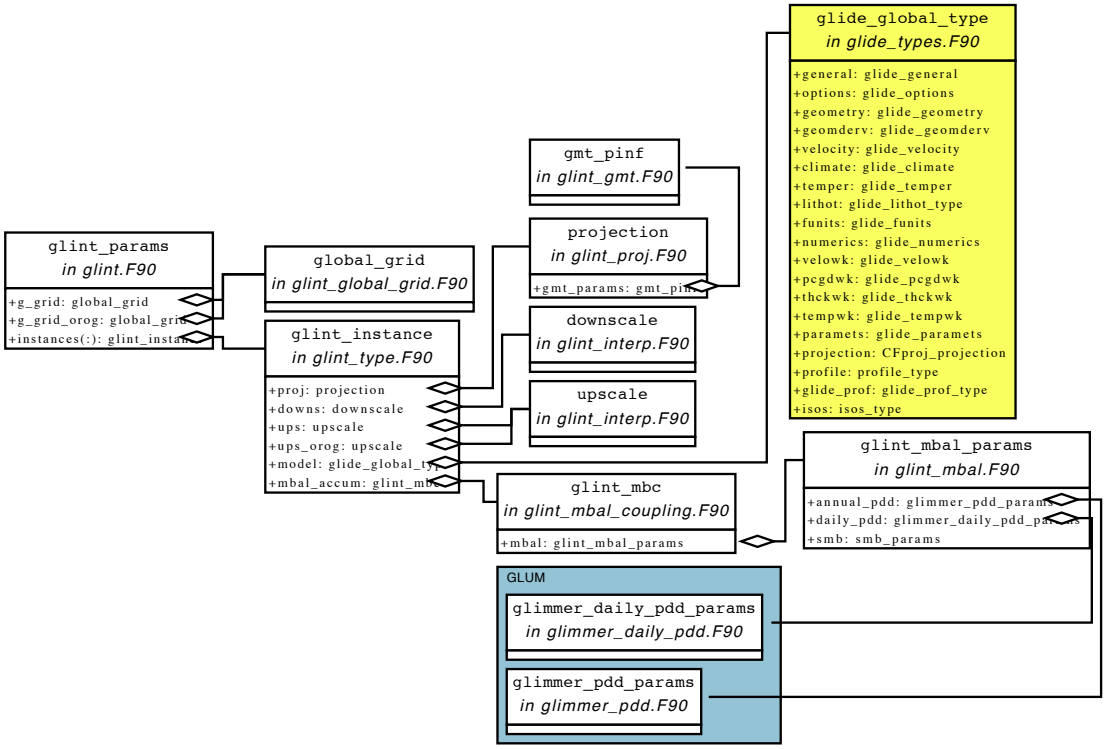

Figure 4.4: GLINT 'class diagram', in quasi-UML notation. Only derived type components which are themselves derived types are shown; intrinsic type components are omitted. The top-level GLIDE type is shown in yellow.

Module name	Filename	Description
glint_main	glint.F90	Top-level GLINT module - public subroutines initialise model, perform a time step, and finalise the model at a global level. Other modules deal with individual GLINT instances. Temporal averaging of global input fields and collation of upscaled output takes place in this module.
glint_climate	glint_climate.F90	Various subroutines used to process the GLINT climate. Includes climate field downscaling and lapse-rate correction.
glint_constants	glint_constants.F90	Various constants used by GLINT but not GLIDE. Mostly relate to sub-year timescales.
glint_global_grid	glint_global_grid.F90	Derived type that defines a global lat-lon grid, and subroutines to handle it.
glint_global_interp	glint_global_interp.F90	Utility code to regrid data from one global grid to another, taking into account spherical geometry.
glint_gmt	glint_gmt.F90	Code adapted from the Generic Mapping Tools (GMT), used for map projection handling.
glint_initialise	glint_initialise.F90	Module used to initialise a GLINT instance (type glint_instance).
glint_interp	glint_interp.F90	Code to upscale and downscale between global and local grids.
glint_io	glint_io.F90	GLINT NetCDF IO routines. Automatically-generated (see below).
glint_mbal_coupling	glint_mbal_coupling.F90	This module coordinates the temporal accumulation of mass-balance quantities.
glint_mbal	glint_mbal.F90	A wrapper for different mass-balance schemes — gives them a common interface.
glint_mbal_io	glint_mbal_io.F90	Auto-generated NetCDF IO routines for GLINT mass-balance calculations.
glint_precip_param	glint_precip_param.F90	Implementation of the precipitation downscaling parameterization of Roe & Lindzen.
glint_proj	glint_proj.F90	Derived type definition and handling code for map projections.
smb_dummy	glint_smb.F90	Dummy module in place of externally-connected RAPID energy-balance mass-balance model.
glint_timestep	glint_timestep.F90	GLINT instance timestep code.
glint_type	glint_type.F90	Definition of GLINT instance type.

4.4 Physics documentation

4.4.1 Ice temperature evolution routines

Summary

Call structure (filenames in brackets).

- subroutine testinisthk [glimmer_setup] and
- subroutine glimmer_i_tstep [glimmer_object] call

- subroutine timeevoltemp [glimmer_temp] calls
- subroutine calcartm [glimmer_temp] and
- subroutine timeders [glimmer_thck] and
- subroutine gridwvel [glimmer_velo] and
- subroutine wvelintg [glimmer_velo] and
- subroutine chckwvel [glimmer_velo] and
- subroutine finddisp [glimmer_temp] and
- subroutine hadvall [glimmer_temp] and
- subroutine hadvpnt [glimmer_temp] and
- subroutine findvtri [glimmer_temp] and
- subroutine tridag [glimmer_temp] and
- subroutine corrpmpt [glimmer_temp] and
- subroutine swapbndt [glimmer_temp] and
- subroutine calcbmlt [glimmer_temp] and
- subroutine calcflwa [glimmer_temp]

Modules used.

-

Introduction

The section describes the routines that are concerned with calculating the three-dimensional distribution of temperature within the ice mass. They can be broken down into five groups.

- determining air temperature (upper boundary condition) [`calcartm`];
- determining vertical velocity field from existing horizontal velocity fields (normally only needed if temperature is being calculated) [`wvelintg`, chckwvel];
- routines associated with vertical grid coordinate system [`gridwvel`, `timeders`];
- the main temperature solver [`finddisp, hadvall, hadvpnt, findvtri, tridag, corrpmpt, swapbndt`];
- ancillary calculations that only make sense if temperature is being calculated [`calcbmlt, calcflwa`].

The basic quantity returned is a three-dimensional grid of temperature in $\circ^{-1}$C (uncorrected for variations in pressure melting point and unscaled). Temperature is held in the array `temp` and will be referred to here using the symbol T.

In addition to temperature a number of other quantities are calculated by these routines. They include: basal melt rate (m `bmlt` m yr^{-1} scaled using `thk0/tim0`); basal water depth (W `bwat` m scaled using `thk0`); vertical velocity (w `wvel` m yr^{-1} scaled using `thk0/tim0`); vertical velocity of numerical grid (w_0 `wgrd` m yr^{-1} scaled using `thk0/tim0`); Glen's A (A `flwa` Pa^{-3} yr^{-1} scaled using `vis0`); air temperature (T_a $\circ^{-1}$C unscaled). All scales are held in the module `paramets` in `glimmer_paramets`.

Three options are currently available for calculating T. The particular option chosen is controlled by the input parameter `whichtemp` (`gln` file).

0 Set whole column to the appropriate surface air temperature (T_a).

1 This option is the main solver that determines temperature at the new time step from the appropriate three-dimensional advection-diffusion equation.

2 Set the upper surface temperature to T_a and do a linear interpolation from this value to 0 °C at the lower surface. Check for pressure melting and adjust any temperatures that are above melting point.

The subroutine `timeevoltemp` controls calculation of the T etc. It is called in the main time loop in `glimmer_object` and resides in `glimmer_temp`.

4.5 Configuration File Parser

The run–time behaviour of the ice sheet model is controlled by configuration files. The old file format is based on Fortran namelists. The new configuration file format is loosely based on the format of Windows `.ini` files with sections containing name/value pairs. The new format is more flexible and can be easily understood by reading the configuration files. This section contains a description of the configuration file parser API.

4.5.1 File Format

The parser assumes a maximum number of 250 characters per line. Leading and trailing white space is ignored. Names are case sensitive.

Comments: Empty lines and lines starting with `!`, `;` or `#` are ignored.

Sections: Section names are enclosed with square prackets, `[]` and can be 20 character long.

Parameters: Parameter names are separated from their associated values with either `:` or `=`. The names can be 20 characters long. Values can be 200 characters long.

An example configuration file:

```
;a comment
[a section]
an_int  : 1
a_float = 2.0
a_char  = hey, this is rather cool
an_array = 10. 20. -10. 40. 100.

[another section]
! more comments
foo : bar
```

4.5.2 Architecture Overview

The configuration data is stored as linked list. Each section is described by the following list element:

```
type ConfigSection
   character(len=namelen) :: name
   type(ConfigValue), pointer :: values=>NULL()
   type(ConfigSection), pointer :: next=>NULL()
end type ConfigSection
```

The parameter name/value pairs defined in each section are stored in another linked list:

```
type ConfigValue
   character(len=namelen) :: name
   character(len=valuelen) :: value
   type(ConfigValue), pointer :: next=>NULL()
end type ConfigValue
```

These linked lists are setup and read using subroutines.

4.5.3 API

Reading configuration files Configuration files are read using `ConfigRead`. This subroutine parses the configuration file and populates the linked lists.

```
subroutine ConfigRead(fname,config)
  character(len=*), intent(in) :: fname
  type(ConfigSection), pointer :: config
end subroutine ConfigRead
```

The pointer `config` contains the first section of the configuration file.

Dumping configuration The subroutine `PrintConfig` traverses the linked lists and prints them to standard output.

```
subroutine PrintConfig(config)
  type(ConfigSection), pointer :: config
end subroutine PrintConfig(config)
```

Searching for a Section The subroutine `GetSection` can be used to find a specific section.

```
subroutine GetSection(config,found,name)
  type(ConfigSection), pointer :: config
  type(ConfigSection), pointer :: found
  character(len=*),intent(in) :: name
end subroutine GetSection
```

On exit the pointer `found` will point to the first section called `name`. `found` points to `NULL()` if the section `name` is not found.

Reading parameters Paramter name/value pairs are found using the `GetValue` family of subroutines. `GetValue` provides an interface to the individual subroutines `GetValueChar`, `GetValueInt`, `GetValueReal`, `GetValueIntArray` and `GetValueRealArray`.

```
subroutine GetValue(section,name,val)
  type(ConfigSection), pointer :: section
  character(len=*),intent(in) :: name
  sometype :: val
  integer,intent(in), optional :: numval
end subroutine GetValue
```

`section` is the section that should be searched for the parameter `name`. On exit `val` contains the parameter value if it is found, otherwise it is unchanged.

The array versions of `GetValue` expect value to be a pointer to a one–dimensional array. `val` is deallocated if it was allocated on entry. The array versions of `GetValue` also accept an optional value, `numval`, with which the maximum number of array elements can be set. The default is 100. Array elements are separated by white space.

4.6 netCDF I/O

The netCDF[5] library is used for platform independent, binary file I/O. GLIMMER makes use of the f90 netCDF interface. The majority of the source files are automatically generated from template files and a variable definition file using a python script. The netCDF files adhere to the CF[6] convention for naming climatic variables. The netCDF files also store parameters used to define the geographic projection.

The netCDF related functionality is split up so that other subsystems of the model can easily define their own variable sets without the need to recompile the main model. These subsystems can also define their own dimensions and access the dimensions defined by other subsystems. The only restriction is that names should not clash. Have a look at the implementation of the `eis` climate driver.

4.6.1 Data Structures

Information associated with each dataset is stored in the `glimmer_nc_stat` type. Variable and dimension ids are retrived from the data set by using the relevant netCDF library calls. Meta data (such as title, institution and comments) is stored in the derived type `glimmer_nc_meta`.

Input and output files are managed by two separate linked lists. Elements of the input file list contain the number of available time slices and information describing which time slice(s) should be read. Output file elements describe how often data should be written and the current time.

4.6.2 The Code Generator

Much of the code needed to do netCDF I/O is very repetative and can therefore be automatically generated. The code generator, `generate_ncvars.py`, is written in python and produces source files from a template `ncdf_template.in` and the variable definition file, see Section 4.6.3. The templates are valid source files, all the generator does is replace special comments with the code generated from the variable file. For further information check the documentation of `generate_ncvars.py`[7].

4.6.3 Variable Definition File

All netCDF variables are defined in control files, `MOD_vars.def`, where `MOD` is the name of the model subsystem. Variables can be modified/added by editing these files. The file is read using the python `ConfigParser` module. The format of the file is similar to Windows `.ini` files, lines beginning with `#` or `;` or empty lines are ignored. These files must have a definition section `[VARSET]` (see Table 4.1).A new variable definition block starts with the variable name in square brackets []. Variables are further specified by parameter name/value pairs which are separated by `:` or `=`. Parameter names and their meanings are summarised in Table 4.2. All parameter names not recognised by the code generator (i.e. not in Table 4.2) are added as variable attributes.

[5] `http://www.unidata.ucar.edu/packages/netcdf/`
[6] `http://www.cgd.ucar.edu/cms/eaton/cf-metadata/index.html`
[7] run `pydoc generate_ncvars.py`

name	description
`name`	Name of the model subsystem, e.g. `glide`. The f90 file is renamed based on this name. The f90 module and module procedures are prefixed with this name.
`datatype`	The name of the f90 type on which the netCDF variables depend.
`datamod`	The name of f90 module in which the f90 type, `datatype`, is defined.

Table 4.1: Each variable definition file must have a section, called `[VARSET]`, containing the parameters described above.

name	description
`dimensions`	List of comma separated dimension names of the variable. C notation is used here, i.e. the slowest varying dimension is listed first.
`data`	The variable to be stored/loaded. The f90 variable is assumed to have one dimension smaller than the netCDF variable, i.e. f90 variables are always snapshots of the present state of the model. Variables which do not depend on time are not handled automatically. Typically, these variables are filled when the netCDF file is created.
`factor`	Variables are multiplied with this factor on output and divided by this factor on input. Default: 1.
`load`	Set to 1 if the variable can be loaded from file. Default: 0.
`hot`	Set to 1 if the variable should be saved for hot–starting the model (implies `load`=1)
`average`	Set to 1 if the variable should also be available as a mean over the write–out interval. Averages are only calculated if they are required. To store the average in a netCDF output file append `_tavg` to the variable.
`units`	UDUNITS compatible unit string describing the variable units.
`long_name`	A more descriptive name of the variable.
`standard_name`	The corresponding standard name defined by the CF standard.

Table 4.2: List of accepted variable definition parameters.

Appendix A

netCDF Variables

The following list shows all the variable names used by GLIMMER. Only variables marked with * are loaded by the input routines.

A.1 Glide Variables

Name	Description	Units
level	sigma layers CF name: land_ice_sigma_coordinate	1
lithoz	vertical coordinate of lithosphere layer	meter
x0	Cartisian x-coordinate, velocity grid	meter
x1*	Cartisian x-coordinate	meter
y0	Cartisian y-coordinate, velocity grid	meter
y1*	Cartisian y-coordinate	meter
acab	accumulation, ablation rate CF name: land_ice_surface_specific_mass_balance	meter/year
acab_tavg	accumulation, ablation rate (time average) CF name: land_ice_surface_specific_mass_balance	meter/year
age*	ice age CF name: land_ice_age	year
artm	annual mean air temperature CF name: surface_temperature	degree_Celsius
beta*†	higher-order bed stress coefficient	unknown
bheatflx*	basal heat flux	watt/meter2
bmlt*	basal melt rate CF name: land_ice_basal_melt_rate	meter/year
bmlt_tavg	basal melt rate (time average) CF name: land_ice_basal_melt_rate	meter/year
btemp	basal ice temperature CF name: land_ice_temperature	degree_Celsius
btrc	basal slip coefficient	meter/pascal/year
bwat*	basal water depth	meter
calving	ice margin calving	meter
diffu	apparent diffusivity	meter2/year

continued on next page

continued from previous page

Name	Description	Units
dusrfdtm	rate of upper ice surface elevation change	meter/year
eus	global average sea level CF name: global_average_sea_level_change	meter
flwa*	Pre-exponential flow law parameter	pascal/year
iarea	area covered by ice	km2
ivol	ice volume	km3
kinbcmask*†	Mask of locations where uvelhom, vvelhom value should be held as Dirichlet boundaries	1
lat*	latitude CF name: latitude	degreeN
litho_temp*	lithosphere temperature	degree_Celsius
lon*	longitude CF name: longitude	degreeE
lsurf	ice lower surface elevation	meter
relx*	relaxed bedrock topography	meter
slc	isostatic adjustment CF name: bedrock_altitude_change_due_to_isostatic_adjustment	meter
soft*	bed softness parameter	meter/pascal/year
tau_xz†	X component vertical shear stress	kPa
tau_yz †	Y component vertical shear stress	kPa
taux	basal shear stress in x direction	kilopascal
tauy	basal shear stress in y direction	kilopascal
temp*	ice temperature CF name: land_ice_temperature	degree_Celsius
thk*	ice thickness CF name: land_ice_thickness	meter
thkmask	mask	1
topg*	bedrock topography CF name: bedrock_altitude	meter
ubas*	basal slip velocity in x direction CF name: land_ice_basal_x_velocity	meter/year
ubas_tavg	basal slip velocity in x direction (time average) CF name: land_ice_basal_x_velocity	meter/year
uflx	flux in x direction	meter2/year
usurf*	ice upper surface elevation CF name: surface_altitude	meter
uvel	ice velocity in x direction CF name: land_ice_x_velocity	meter/year
uvelhom*†	ice velocity in x direction according to higher order model CF name: land_ice__x_velocity	meter/year
vbas*	basal slip velocity in y direction CF name: land_ice_basal_y_velocity	meter/year
vbas_tavg	basal slip velocity in y direction (time average) CF name: land_ice_basal_y_velocity	meter/year
velnormhom †	Ice velocity magnitude according to higher-order model	meter/year

continued on next page

continued from previous page		
Name	Description	Units
vflx	flux in x direction	meter2/year
vvel	ice velocity in y direction CF name: land_ice_y_velocity	meter/year
vvelhom*†	ice velocity in y direction according to higher order model CF name: land_ice_y_velocity	meter/year
wgrd	Vertical grid velocity	meter/year
wvel	vertical ice velocity	meter/year

A.2 EIS Variables

Name	Description	Units
cony*	continentality	1
ela*	equilibrium line altitude CF name: equilibrium_line_altitude	meter

A.3 GLINT Variables

Name	Description	Units
ablt	ablation	meter (water)/year
arng	air temperature half-range	degreeC
global_orog	orographic elevation provided by global model CF name: surface_altitude	meter
local_orog	orographic elevation provided by local model CF name: surface_altitude	meter
mask*	upscaling and downscaling mask	1
prcp	precipitation CF name: lwe_precipitation_rate	meter (water)/year
siced*	superimposed ice depth	meter
snowd*	snow depth CF name: surface_snow_thickness	meter

Appendix B

The GLIMMER API

B.1 GLUM

GLUM provides some utility subroutines which are shared by all components of GLIMMER.

B.1.1 Subroutine `open_log`

Purpose open and initialise log file

Name and mandatory arguments

```
subroutine open_log
```

Arguments

Mandatory			
Optional			
`unit`	`integer`	`intent(in)`	file unit to use (defualt: 6)
`fname`	`character(len=*)`	`intent(in)`	name of log file (default: `glide.log`)

B.1.2 Subroutine `ConfigRead`

Purpose Read configuration file and store config options.

Name and mandatory arguments

```
subroutine ConfigRead(fname,config)
```

Arguments

Mandatory			
`fname`	`character(len=*)`	`intent(in)`	name of configuration file to be read
`config`	`type(ConfigSection), pointer`		pointer to first element of linked list containing configuration

Additional Notes Each section within the configuration file is stored as an element of a linked list. These elements contain another linked list storing the key–value pairs.

B.1.3 Subroutine `CheckSections`

Purpose To check if all sections within a configuration file were subsequently used. Report unused sections to the log.

Name and mandatory arguments

```
subroutine CheckSections(config)
```

Arguments

Mandatory		
`config`	`type(ConfigSection), pointer`	pointer to first element of linked list containing configuration

B.2 GLIDE

B.2.1 Subroutine `glide_config`

Purpose To parse configuration file and print configuration to log

Name and mandatory arguments

```
subroutine glide_config(model,config)
```

Arguments

Mandatory			
`model`	`type(glide_global_type)`	`intent(inout)`	f95 type containing all variables associated with an instance of the model.
`config`	`type(ConfigSection), pointer`		pointer to first element of linked list containing configuration

B.2.2 Subroutine `glide_initialise`

Purpose To initialise the basic ice sheet model

Name and mandatory arguments

```
subroutine glide_initialise(model)
```

Arguments

Mandatory			
`model`	`type(glide_global_type)`	`intent(inout)`	f95 type containing all variables associated with an instance of the model.

Additional Notes This subroutine initialises the model. Memory for all variables is allocated. Input files are opend and read. Output files are created. Variables are scaled.

B.2.3 Subroutine `glide_nc_fillall`

Purpose fill netCDF coordinate variables.

Name and mandatory arguments

```
subroutine glide_nc_fillall(model)
```

Arguments

Mandatory			
`model`	`type(glide_global_type)`	`intent(inout)`	f95 type containing all variables associated with an instance of the model.

B.2.4 Subroutine `glide_tstep_p1`

Purpose Performs first part of time-step of an ice model instance: calculate vertical velocity and temperature field. Set model time.

Name and mandatory arguments

```
subroutine glide_tstep_p1(model,time)
```

Arguments

Mandatory			
`model`	`type(glide_global_type)`	`intent(inout)`	f95 type containing all variables associated with an instance of the model.
`time`	`real(rk)`	`intent(in)`	Current time in years

B.2.5 Subroutine `glide_tstep_p2`

Purpose Performs second part of time-step of an ice model instance: write data and move ice and update horizontal velocities.

Name and mandatory arguments

```
subroutine glide_tstep_p2(model)
```

Arguments

Mandatory			
`model`	`type(glide_global_type)`	`intent(inout)`	f95 type containing all variables associated with an instance of the model.

B.2.6 Subroutine `glide_tstep_p3`

Purpose Performs third part of time-step of an ice model instance: calculate isostatic adjustment and upper and lower ice surface.

Name and mandatory arguments

```
subroutine glide_tstep_p3(model)
```

Arguments

Mandatory			
`model`	`type(glide_global_type)`	`intent(inout)`	f95 type containing all variables associated with an instance of the model.

B.2.7 Subroutine `glide_finalise`

Purpose To shut–down model, close all open files and deallocate memory.

Name and mandatory arguments

```
subroutine glide_finalise(model)
```

Arguments

Mandatory			
`model`	`type(glide_global_type)`	`intent(inout)`	f95 type containing all variables associated with an instance of the model.
Optional			
`crash`	`logical`	`intent(in)`	set to true if the model died unexpectedly

B.3 GLINT

This appendix details the subroutine calls provided by GLINT, and their arguments. Note that where a type is given as `real(rk)`, this indicates that the kind of the real type is specified by the value of the parameter `rk`, which may be altered at compile-time (see appropriate other documentation for details).

B.3.1 Subroutine `initialise_glint`

Purpose To initialise the ice model, and load in all relevant parameter files.

Name and mandatory arguments

```
subroutine initialise_glint(params,lats,longs,paramfile,time_step)
```

Arguments

Mandatory			
`params`	`type(glint_params)`	`intent(inout)`	Ice model to be configured
`lats(:)`	`real(rk)`	`intent(in)`	latitudinal location of grid-points in global data (given in °N)
`longs(:)`	`real(rk)`	`intent(in)`	longitudinal location of grid-points in global data (given in °E)
`paramfile`	`character(*)`	`intent(in)`	name of top-level parameter file
`time_step`	`integer`	`intent(in)`	Intended calling time-step (hours)
Optional			
`latb(:)`	`real(rk)`	`intent(in)`	Latiudinal locations of grid-box boundaries (degrees). This array has one more element than `lats`.

continued on next page

continued from previous page			
lonb(:)	real(rk)	intent(in)	Longitudinal locations of grid-box boundaries (degrees). This array has one more element than longs.
orog(:,:)	real(rk)	intent(out)	The initial orography (m).
albedo	real(rk)	intent(out)	The initial ice albedo field
ice_frac	real(rk)	intent(out)	The initial ice fraction
orog_lats	real(rk)	intent(in)	Latitudinal location of gridpoints for global orography output
orog_longs	real(rk)	intent(in)	Longitudinal location of gridpoints for global orography output
orog_latb	real(rk)	intent(in)	Locations of the latitudinal boundaries of the grid-boxes (orography)
orog_lonb	real(rk)	intent(in)	Locations of the longitudinal boundaries of the grid-boxes (orography)
output_flag	logical	intent(out)	Set to show outputs have been updated (provided for consistency with main glint subroutine).

Additional notes

- The ice model determines the size of the global domain from the sizes of the arrays lats and longs.
- The latitudes contained in lats must be in descending order, so that lats(i) > lats(i + 1) for $1 \leq$ i $\leq$ size(lats).
- The optional arguments orog_lats, orog_longs, orog_latb, and orog_lonb may be used to define the frid on which the orography is output from GLINT. This is useful if the global model has spectral dynamics, and thus a higher-resolution orography is needed for greater accuracy when transforming to spectral space. These arguments may not be present in arbitrary combinations - only orog_lats+orog_longs, orog_lats+orog_longs+orog_latb, orog_lats+orog_longs+orog_lonb, and orog_lats+orog_longs+orog_latb+orog_lonb are permitted. Other combinations will generate a fatal error.

B.3.2 Subroutine glint

Purpose To perform temporal averaging of input fields, and, if necessary, down-scale those fields onto local projections and perform an ice model time-step. Output files may be appended to, and if optional arguments used, fields made available for feedback.

Name and mandatory arguments

```
subroutine glint(params,time,temp,precip,orog)
```

Arguments

Mandatory			
params	type(glint_params)	intent(inout)	parameters for this run
time	integer	intent(in)	Current model time (hours)
temp(:,:)	real(rk)	intent(in)	Surface temperature field (°C)
precip(:,:)	real(rk)	intent(in)	Precipitation field (mm/s)

continued on next page

continued from previous page			
`orog(:,:)`	`real(rk)`	`intent(in)`	Global orography (m)
Optional			
`zonwind(:,:)`	`real(rk)`	`intent(in)`	Zonal component of the wind field (ms^{-1})
`merwind(:,:)`	`real(rk)`	`intent(in)`	Meridional component of the wind field (ms^{-1})
`output_flag`	`logical`	`intent(out)`	Set to show new output fields have been calculated after an ice-model time-step. If this flag is not set, the output fields retain their values at input.
`orog_out(:,:)`	`real(rk)`	`intent(inout)`	Output orography (m)
`albedo(:,:)`	`real(rk)`	`intent(inout)`	Surface albedo
`ice_frac(:,:)`	`real(rk)`	`intent(inout)`	Fractional ice coverage
`water_in(:,:)`	`real(rk)`	`intent(inout)`	The input fresh-water flux (mm, over ice time-step). Essentially precip, but provided for consistency.
`water_out(:,:)`	`real(rk)`	`intent(inout)`	The output fresh-water flux (mm, over ice time-step). This is simply the ablation calculated by the model, scaled up to the global grid. It is up to the global model to then deal with it (route it to the oceans, land scheme, etc.) Note that the precipitation fed to the model but which doesn't get incorporated into the ice sheet because it falls over the sea is returned in this field.
`total_water_in`	`real(rk)`	`intent(inout)`	Area-integrated water flux in (kg)
`total_water_out`	`real(rk)`	`intent(inout)`	Area-integrated water flux out (kg)
`ice_volume`	`real(rk)`	`intent(inout)`	Total ice volume (m^3)

Additional notes

- The sizes of all two-dimensional fields passed as arguments must be the same as that implied by the sizes of the arrays used to pass latitude and longitude information when the model was initialised using `initialise_glint`. There is currently no checking mechanism in place for this, so using fields of the wrong size will lead to unpredictable results.
- Zonal and meridional components of the wind are only required if the small-scale precipitation parameterization is being used (with `whichprecip` set to 2).
- The output field arguments only return data relevant to the parts of the globe covered by the ice model instances. The fraction of each global grid-box covered by ice model instances may be obtained using the `glint_coverage_map` subroutine below.
- The output orography field is given as a mean calculated over the part of the grid-box covered by ice model instances. Thus, to calculate the grid-box mean, the output fields should be multiplied point-wise by the coverage fraction.
- Albedo is currently fixed at 0.4 for ice-covered ground, and set to zero elsewhere. The albedo is given for the part of the global grid box covered by ice, not as an average of the part covered by the ice model. No attempt is made to guess the albedo of the parts of the ice model domain *not* covered by ice.

Example interpretation of output fields Consider a particular point, (i, j) in the global domain. Suppose value returned by `glint_coverage_map` for this point is 0.7, and the output fields have these values:

```
orog_out(i,j)  = 200.0
albedo(i,j)    =   0.4
ice_frac(i,j)  =   0.5
```

What does this mean? Well, the ice model covers 70% of the grid-box, and in that part the mean surface elevation is 200 m. Of the part covered by the ice model, half is actually covered by ice. Thus, 35% (0.5×0.7) of the global grid-box is covered by ice, and the ice has an mean albedo of 40%. The model makes no suggestion for the albedo or elevation of the other 65% of the grid-box. Currently, ice albedo is a constant that may be changed in the appropriate configuration file, but this output field is provided against the possibility that the model may be extended at some point to include a model of ice albedo.

B.3.3 Subroutine `end_glint`

Purpose To perform general tidying-up operations, close files, etc.

Name and mandatory arguments

```
subroutine end_glint(params)
```

Arguments

`params`	`type(glint_params)`	`intent(inout)`	Ice model paramters

B.3.4 Function `glint_coverage_map`

Purpose To obtain a map of fractional coverage of global grid-boxes by the ice model instances. The function returns a value indicating success, or giving error information.

Type, name and mandatory arguments

```
integer function glint_coverage_map(params,coverage,cov_orog)
```

Arguments

`params`	`type(glint_params)`	`intent(in)`	Ice model parameters
`coverage(:,:)`	`real(rk)`	`intent(out)`	Coverage map (all fields except orography)
`cov_orog(:,:)`	`real(rk)`	`intent(out)`	Coverage map (orography)

Returned value

Value	Meaning
0	Coverage maps have been returned successfully
1	Coverage maps not yet calculated; must call `initialise_glint` first
2	Arrays `coverage` or `cov_orog` are the wrong size

Bibliography

V. Decyk, C. Norton, and B. K. Szymanski. How to express C++ concepts in Fortran 90. *Scientific Programming*, 6(4):363–390, 1997.

M. K. M. Hagdorn. *Reconstruction of the Past and Forecast of the Future European and British Ice Sheets and Associated Sea Level Change*. PhD thesis, University of Edinburgh, 2003.

J. R. Holton. *An Introduction to Dynamic Meteorology*, volume 48 of *International Geophysics Series*. Academic Press, New York, 3rd edition, 1992.

K. Hutter. *Theoretical Glaciology*. Mathematical Approaches to Geophysics. D. Reidel Publishing Company, Dordrecht, Boston, Lancaster, 1983.

P. Huybrechts. A three–dimensional time–dependent numerical model for polar ice sheets; some basic testing with a stable and efficient finite difference scheme. Technical report, Geografisch Instituut, Vrije Universiteit Brussel, Belgium, 1986.

K. Lambeck and S. M. Nakiboglu. Seamount loading and stress in the ocean lithosphere. *Journal of Geophysical Research*, 85:6403–6418, 1980.

M. Metcalf and J. Reid. *Fortran 90/95 explained*. Oxford University Press, Oxford, 2nd edition, 1999.

W. S. B. Paterson. *The Physics of Glaciers*. Butterworth–Heinemann, Oxford, 3rd edition, 1994.

A. J. Payne. A thermomechanical model of ice flow in West Antarctica. *Climate Dynamics*, 15: 115–125, 1999.

A. J. Payne and P. W. Dongelmans. Self–organisation in the thermomechanical flow of ice sheets. *Journal of Geophysical Research*, 102(B6):12219–12233, 1997.

W. R. Peltier, D. L. Goldsby, D. L. Kohlstedt, and L. Tarasov. Ice–age ice–sheet rheology: constraints from the Last Glacial Maximum form of the Laurentide ice sheet. *Annals of Glaciology*, 30:163–176, 2000.

W.H. Press, S.A. Teukolsky, W.T. Vetterling, and B.P. Flannery. *Numerical Recipes in FORTRAN*. Cambridge University Press, 2nd edition, 1992.

C. Ritz. Time dependent boundary conditions for calculation of temperature fields in ice sheets. In *The Physical Basis of Ice Sheet Modelling*, number 170 in IAHS Publications, pages 207–216, 1987.

L. Tarasov and W. R. Peltier. Impact of thermomechanical ice sheet coupling on a model of the 100kyr ice age cycle. *Journal of Geophysical Research*, 104(D8):9517–9545, 1999.

L. Tarasov and W. R. Peltier. Laurentide ice sheet aspect ratio in models based on Glen's flow law. *Annals of Glaciology*, 30:177–186, 2000.