**Documentation of Met Office Hadley Centre code for reading and processing of data from CRREL ice mass balance buoys**

Version 2

Alex West, February 2020

1. **Background**

The CRREL ice mass balance buoys (IMBs) are a network of devices frozen into sea ice in the Arctic Ocean from 1993-2017 which measure ice surface and base elevation, temperature at 10cm intervals in the ice, latitude-longitude position, and sometimes other variables (e.g. surface air temperature and pressure). Data from the IMBs, of which there are over 100, is stored in a series of comma-delimited CSV files at http://imb-crrel-dartmouth.org. However, file format varies greatly, making processing of this data difficult. Examples of problems encountered include

* Identical variables named differently, e.g. ice base can be labelled ‘Bottom of Ice Position’ or ‘Ice Bottom Position(m)’ amongst other names
* Date format varies between British and American
* For some IMBs longitude is determined by a number between -180 and 180, for other IMBs by a number between 0-180 and an E/W marker
* Data units vary between buoys: sometimes elevation is reported in m, sometimes in cm

Over the period 2014-2020, code was developed at the Met Office Hadley Centre to enable easy reading and processing of data from these files. This is documented here.

1. **Overview**

The basic code for reading and processing of the IMB data is written in Python 3 and is contained in 2 repositories which process the IMB data in stages:

**2.1 Stage 1: reading\_interpolating\_qc**

For each IMB, this code performs the following tasks:

* takesthe raw IMB data as input, reading all data into a single Python structure
* performs basic quality control of the temperature data
* performs standard arithmetic operations allowing the production of a standardised set of elevation time series (surface height, interface height, base height, snow depth and ice thickness)
* produces elevation time series ‘regularised’ to temperature measurement times
* produces time series of temperature statistics estimated at specific points, or over specific depths, in the snow-ice column, for later use in calculating energy fluxes
* saves all regularised time series to a netCDF4 file.

A driver routine **buoys\_run.py** performs this process for all IMBs, using the modules:

* **buoys.py** – defines basic class for holding all data from an IMB
* **data\_series.py** – defines sub-class for holding a single timeseries from an IMB, and contains code for reading such a series from the IMB source files
* **temp\_series.py** – defines sub-class for holding the 2D temperature data from an IMB
* **linekey.py –** defines a class to describe the format of a single IMB file which is used to read data
* **dictionaries.py –** defines two dictionaries which help the code identify particular variables in IMB source files
* **functions.py –** contains various auxiliary functions used for data processing
* **tprof.py** – contains various functions used for manipulation of individual temperature profile data

In addition, a text file called **mday\_flag.txt** contains a list of buoys for which date is in DD-MM-YYYY format (as opposed to MM-DD-YYYY format), as it appeared very difficult to detect this automatically from an individual IMB source file in an efficient way; and a text file called **temp\_flag.txt** contains information about spurious temperature data.

**2.2 Stage 2: time\_mean\_variables**

This code takes as input the processed data produced by Stage 1, and uses this to produce monthly mean fluxes of top melt, basal melt, basal growth, top conduction, basal conduction and ocean heat flux. These variables are saved to a single netCDF4 file. Each variable has 2 dimensions: a record dimension (with each record corresponding to a single month of valid data from a single IMB) and a salinity dimension (as salinity is a fundamental unknown affecting all of the heat fluxes).

A driver routine **all\_variables\_monthly.py** controls the production of this dataset, using the modules

* **imb\_calc.py –** contains a series of functions each of which correspond to a flux to be calculated, two larger functions for calculating auxiliary variables such as conductivity and heat capacity, and a function to control error handling.
* **difference\_functions.py –** functions to calculate rates of change of data series
* **nc\_functions.py** – contains all code relating to setting up the output file
* **scientific\_constants.py –** defines all scientific constants used in imb\_calc.py
* **parameters.py –** defines all parameters that can be chosen by the user in calculating the monthly fluxes, e.g. conductivity scheme, layer over which to calculate conduction. These are written as attributes to the final netCDF file.

1. **Reading and analysing one-dimensional IMB data**

This section describes the reading of all IMB data except temperature, which as a 2D field has its own separate system, and is described in Section 4.

* 1. **Top level view**

The code can be used to read and plot a time series for an IMB in the following way (in this case, surface elevation from the buoy 2012L):

import buoys

buoy\_str = buoys.buoy(‘2012L’)

buoy\_str.extract\_data(‘surface’)

The surface elevation data is then stored as a dictionary

buoy\_str.data[‘surface’].data\_list

in which the keys are Python datetime objects.

buoy\_str.show([‘surface’])

will then plot the surface elevation data against time.

Other variable names that can be passed to extract\_data are ‘interface’, ’bottom’, ’snow depth’, ’ice thickness’, ‘latitude’, ‘longitude’, ‘air temperature’ and ‘air pressure’.

* 1. **How the data is read**

The line buoy\_str = buoys.buoy(‘2012L’)creates an instance buoy\_str of the ‘buoy’ class, with an empty dictionary where data will be held (buoy\_str.data = {}). However the main work of reading the data is done in the line buoy\_str.extract\_data(‘surface’).

In the extract\_data method, the file from which the data should be read is first determined by comparing the ‘file-variable’ dictionary in **dictionaries.py** against the actual files present in the source data directory for the buoy 2012L (in this case, the appropriate filename is found to be ‘2012L\_clean.csv’). In the case that no file is found, an empty data series is nevertheless created and appended to the data tag (buoy.data[‘surface’].data\_list = {}).

If a file is found, a new empty data series object is created, containing the information about data location:

import data\_series as ds

series = ds.data\_series(self.name, file\_ext\_name, varname)

(in this case, self.name = ‘2012L’, file\_ext\_name = ‘\_clean’, varname = ‘surface’.)

Then the data is read in series.read(full\_file,varname).

Firstly a ‘linekey’ object is created for the data file in question with key = linekey.get\_linekey(data\_file,[varname],self.name). The purpose of this object is to contain all necessary information about the file format for subsequent reading of data. Firstly, the object identifies the column in which the date and time of the measurement can be found (‘date\_index’). The object then contains two lists: ‘value\_index’, which holds a list of indices of the position within rows in which data can be found, and ‘phenomena\_names’, a list of the variables which correspond to the indices. The variable name dictionary in **dictionaries.py** is used to help identify these. In addition, the linekey object holds information about how longitude data should be processed. For example, if the longitude field title is ‘Longitude (W)’, a switch is turned on within the linekey object to ensure that all data read within that field is multiplied by -1. If a field title called ‘E/W’ is detected, meanwhile, a separate switch is activated which reverses data in the field marked ‘longitude’ only if the value of the E/W field is ‘W’. Finally, the object holds a data scaling factor which is set to 0.01 if it is judged that the data is in units of cm.

Once the linekey object is produced, the code iterates through the rows of the file. For each row, the field value of the column under ‘date\_index’ is extracted, and a set of functions in **data\_series.py** tests this value to determine if it is a genuine date and time. If this test is successful, the field value is converted to a Python datetime object, and the code proceeds to identify the field value under the index corresponding to ‘surface’. This field value is then subjected to the processing described above before being added to the data series:

self.data\_list[date] = value

Back at the top level, the newly-read data series is added to the main data tag of the buoy structure:

self.data[varname] = series

* 1. **IMB data analysis and visualisation**

Given a newly-read data series series,

series.period()

returns a list of 2 datetime objects corresponding to the earliest and latest time for which data is present.

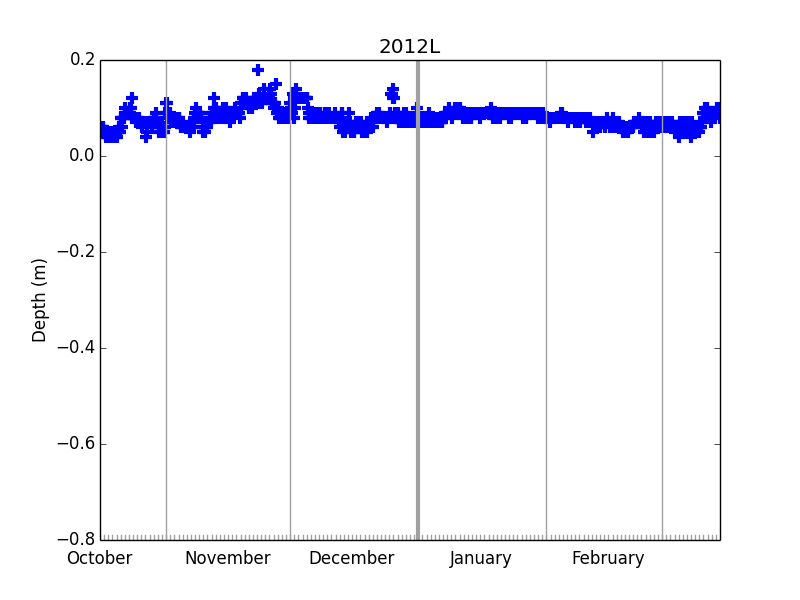
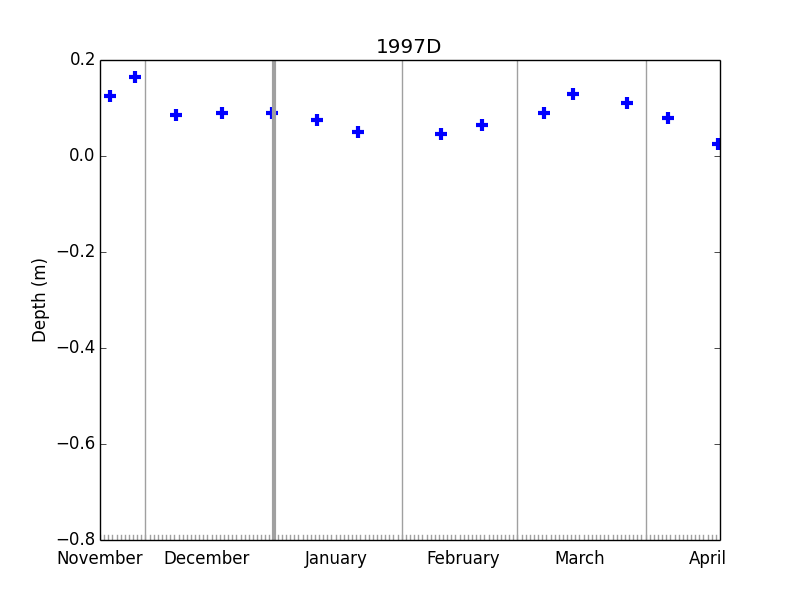
series.dates()

returns a full list of datetime objects for which data is present;

series.values()

returns a full list of data corresponding to the datetime objects.

Because the IMB data is often measured at irregular time points, methods are provided to convert the data to a more regular time series (‘regularise’). These methods rely on the ability to estimate the value of a data series at arbitrary points in time. This estimation is performed either by interpolation or by binomial mean, depending on whether 3 or more data points are present in the time window of length 2 days, centred on the time point in question (Figure 1).



*Figure 1. Raw surface data series for two IMBs. For the left series, values will be estimated by interpolation; for the right series, by binomial mean.*

For example, surface data for the buoy 1997D is provided at quite sparse, irregular time points, on average 2-3 per month. series.estimate(datetime.datetime(1998,3,10,0,0)) returns 0.104m, interpolated between the values measured on 7th and 14th March. On the other hand, surface data for the buoy 2012L is provided much more frequently, and can be noisy. series.data\_list[datetime.datetime(2013,3,10,0,0) is equal to 0.05m in the raw data series, but series.data\_list.estimate( datetime.datetime(2013,3,10,0,0)) returns 0.066m instead as it considers nearby values also.

The data series method regularise\_temp() makes use of the estimate() method to produce a time series whose times of measurement are equal to those of the buoy temperature data from a given raw timeseries. For example:

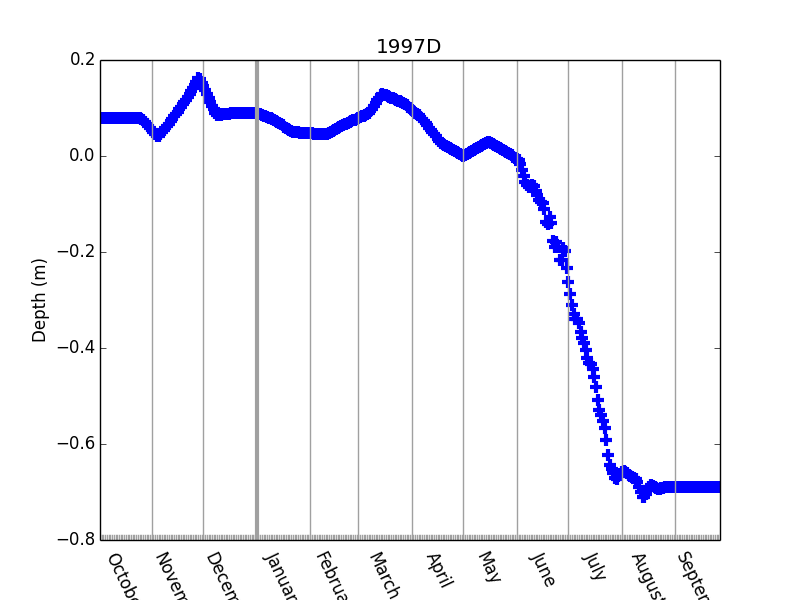
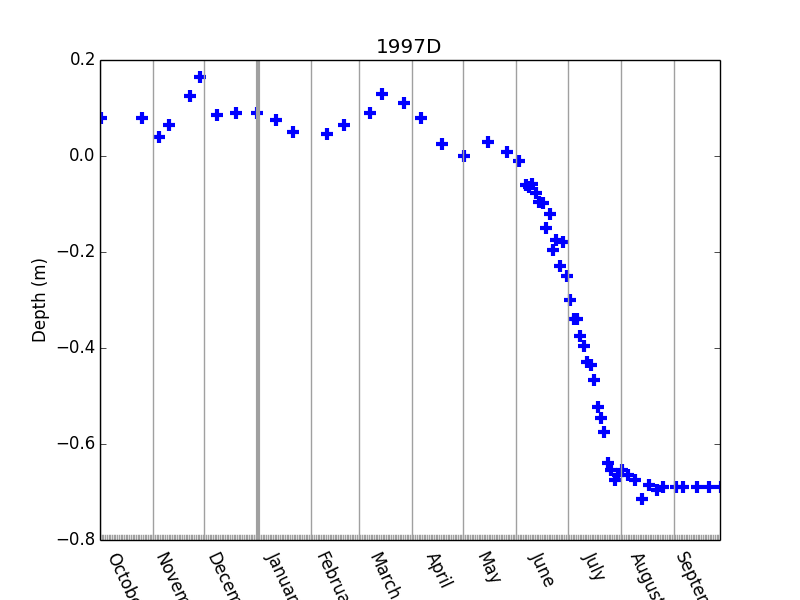
import buoys

buoy\_str = buoys.buoy(‘1997D’)

buoy\_str.extract\_data(‘surface’)

surface\_regular = buoy\_str.data[‘surface’].regularise\_temp()

The resulting regular time series is shown in Figure 2:



*Figure 2. (left) Raw surface elevation data from buoy 1997D; (right) regularised surface elevation data for the same buoy*

The plots displayed make use of the data series method show(), which produces a time series plot of any data series. Show() takes amongst its arguments start\_date and end\_date (Python datetime objects that set the x-axis limits) and ylim (y-axis limits).

Throughout the code it is the convention that all regularised temperature series are given a name ending ‘\_rt’ to identify them as having time points equal to those of the temperature data.

* 1. **Rate of change of a data series**

Given a data series, the method rate\_of\_change estimates the time derivative of that data series by performing a series of linear fits at each data point. The default time interval over which to take the linear fit is the period from one day before to one day after the time point in question. This method is used for calculation of sensible heat uptake.

1. **Reading and analysing IMB temperature data**

**4.1 Top level view**

Because of its 2-dimensional nature, IMB temperature data requires a separate class for reading and processing. This class and associated functions are stored in temp\_series.py.

The following code reads IMB temperature data for the buoy 1997D:

import buoys

buoy\_str = buoys.buoy(‘1997D’)

buoy\_str.extract\_temp()

The temperature data is then stored as a ‘temp\_series’ (temperature series) object in the tag ‘.temp’. This object stores the data in a tag ‘.profile\_set’, a dictionary in which the keys are Python datetime objects and the values are lists of length equal to the number of vertical temperature measurement points.

Due to various issues discussed below, the raw temperature data is not easy to use. A wrapper method process\_temp() reads the temperature data and performs some additional basic processing. ‘Cleaner’ temperature data is then stored in the tag ‘.mprofile\_set’, as a dictionary of masked numpy arrays.

**4.2 How the data is read**

The data file (full\_file) containing the temperature is identified by a similar process to that of the 1D series (section 3.2). A ‘temperature linekey’ is then created for that file, with information about the measurement elevation points:

import linekey

key = linekey.get\_temp\_linekey(full\_file)

An empty temperature series object ts is then created using this information:

import temp\_series as tss

ts = tss.temp\_series(self.name,file\_ext\_name,key.phenomena\_names)

Finally the method ts.read(full\_file,key) is called, to use the linekey information to extract the temperature data from the file.

The processing of information on the elevation of the points of measurement is complex and requires some discussion. This is because the format of the temperature labels varies in a significant way: while for the early buoys, the measurement points are labelled with elevation co-ordinates (e.g. 60, 55,… 5, 0, -10, -20,…), for later buoys, the measurements points are simply labelled with integers (e.g. T1, T2,…), usually with no explicit indication given as to the points of measurement.

Code in linekey.py examines the raw elevation labels in the source file and decides whether each label is ‘objective’ (it refers directly to the elevation) or ‘subjective’ (it does not so refer). If labels of different type were detected for the same buoy the reading code is designed to fail, but in practice no such buoys exist. Hence the temperature series for each buoy can be classified as ‘objective’ or ‘subjective’ depending on the type of its elevation labels (and buoy\_str.temp.classify()) will return this type).

The processing and visualisation that can be performed on a subjective temperature series object is very limited. Hence, the method subjective\_to\_objective() is provided to convert an object, given a dictionary whose keys are the subjective labels and whose values are the corresponding elevation points. The function standard\_ztemp\_subj\_obj\_dic() provides a standard set of rules to carry this out, based on inspecting a large sample of the subjectively-labelled IMBs, which assumes that the top measurement begins at elevation 60cm, and that subsequent elevations descend at an interval of 10cm.

Methods are also provided to carry out rudimentary quality control on the temperature data. Instances of temperature values that are obviously wrong occur very frequently in the IMBs, usually over lengthy time periods and more than one elevation at once. The source file temp\_mask.txt provides a format to document and mark these areas as they are discovered. The method buoy\_str.temp.mask() then creates a quality-controlled version of the temperature data in the tag mprofile\_set by

* For each time of observation, creating 1D numpy.ma masked arrays of the source temperature data of length equal to the source data
* Masking out each data point for which the value is equal to the preset missing data index (-999. for all buoys)
* Masking out all data points which are within regions marked in temp\_mask.txt

The above processing is packaged in the single buoy method process\_temp, which reads, objectifies and QCs temperature data as described above, returning in the tag .temp a temperature series structure that is ready for analysis. (In addition, this method applies a check to see if temperature for any period needs to be translated in elevation. This was required due to a particular problem with buoy 2006B).

**4.3 Analysis and visualisation of the temperature data**

Given a temperature series object temp\_series, the methods dates()and period() perform similar functions to those of the respective data series methods. The method values() takes an integer argument corresponding to a vertical level number to produce a 1D numpy array, a timeseries of temperature measured at that level. In addition, a method zpoints() returns a numpy array of the measurement point elevations, and values\_2D() returns a 2D numpy array of the temperature values.

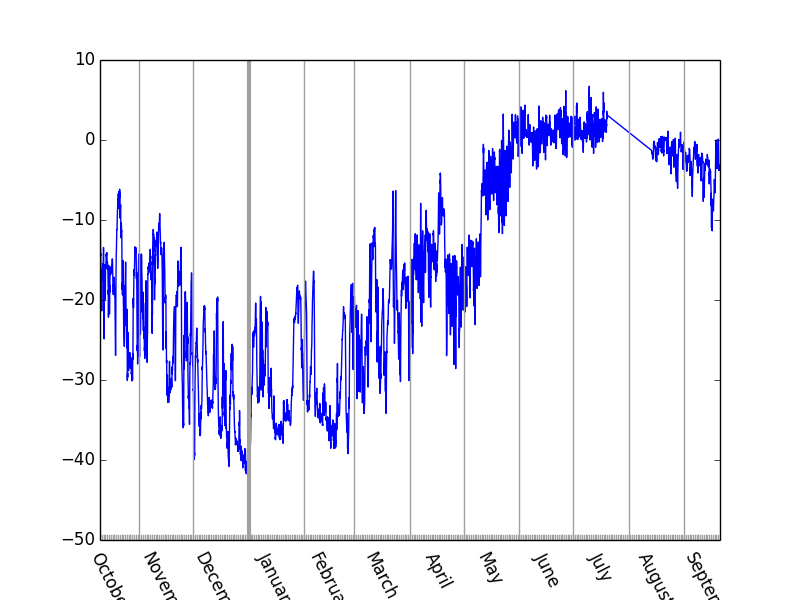
There are several methods provided to visualise the temperature data. The below code creates a timeseries plot of the temperature at level 10 (where level 0 is at the top):

import buoys

buoy = buoys.buoy(‘1997D’)

buoy.process\_temp()

buoy.temp.show(10)

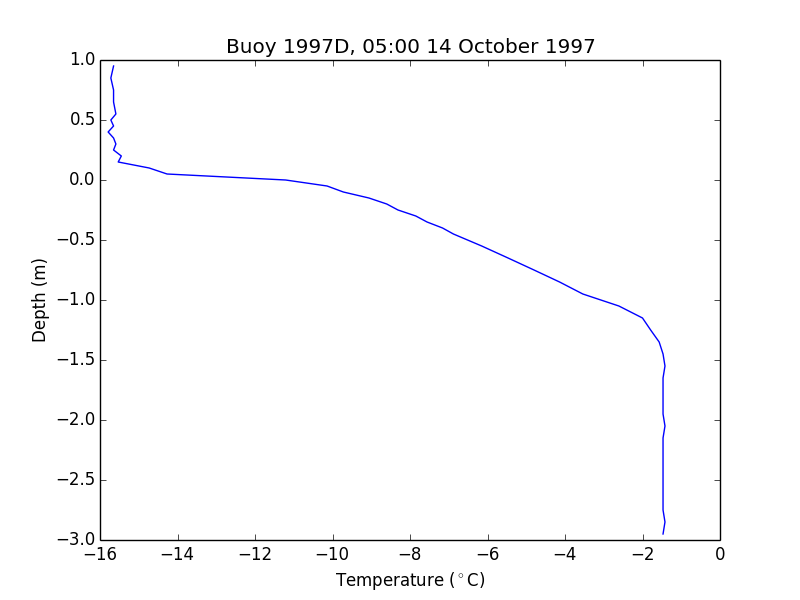


*Figure 3. Timeseries of temperature at level 10 in buoy 1997D.*

The method zshow creates a temperature profile plot for a particular date:

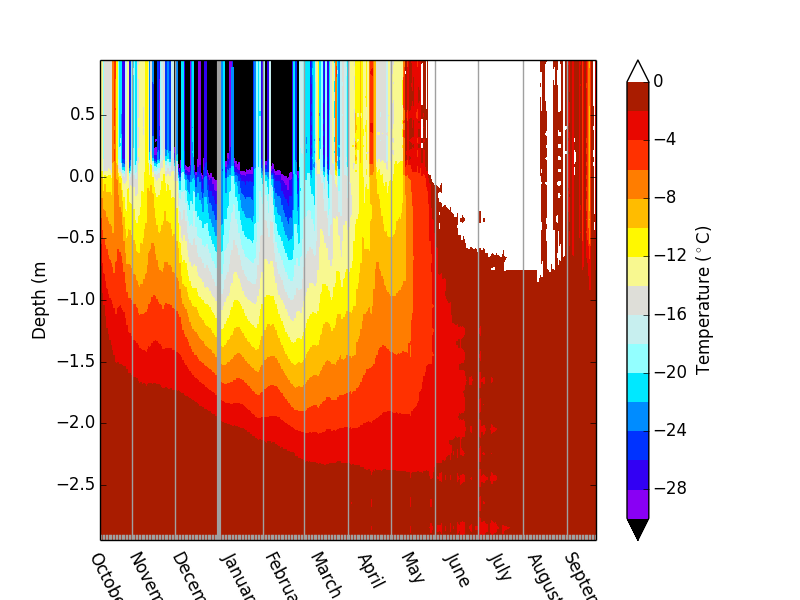
tdates = buoy.temp.dates()

buoy.temp.zshow(tdates[100])



*Figure 4. Temperature profile from the buoy 1997D.*

Finally, the method contour\_obj creates a Hovmuller contour plot of the temperatures measured by the buoy:



*Figure 5. Contour plot of temperatures measured by buoy 1997D.*

1. **Calculating temperature statistics at specified depths and layers in the ice and snow profile**

For the calculation of monthly mean fluxes, it is necessary to calculate a large number of additional time series. This is because

* Conductive fluxes are derived, in Stage 2, from a particular processed time series: the vertical gradient of temperature, measured by a linear fit, across a specific layer in the ice, e.g. 40cm-70cm above the ice base for basal conduction. This layer is obviously not always going to be coincident with the same set of temperature measurement points. To avoid sudden jumps in the gradient as measurement depths enter and leave the layer in question, it’s necessary to estimate the temperatures at the upper and lower surface of this layer to use in the fit also.
* In calculating ocean heat flux, it’s also necessary to estimate sensible heat uptake in the lowest layer of the ice (0-40cm above the base in the standard formulation). To allow this quantity to be calculated in Stage 2, an additional data series must be calculated: the average temperature across this layer. As above, the temperature at the ice base, and 40cm above, must be estimated and used as boundary conditions for this calculation to avoid sudden jumps in this statistic.
* In Stage 2, conductivity and heat capacity of the ice are also important in calculating the fluxes. In the Maykut and Untersteiner formulation, these variables depend upon and respectively, where is ice temperature. Hence and are also calculated over the layers of interest.

1. **Saving regularised time series to netCDF**

To produce a standard format dataset to work from more easily, a buoy method save\_rt\_nc() is provided to save a buoy structure, together with its temperature data, and all other available time series regularised to temperature measurement times, to netCDF format.

In full, the method creates a new netCDF4 dataset with two co-ordinates, time and depth. Time and depth variables are created corresponding to these co-ordinates, with values equal to the buoy temperature measurement points. A temperature variable is created, and the buoy temperature data assigned. Finally, all buoy data series with names ending ‘\_rt’, indicating regularisation to temperature measurement times, are also assigned to variables in the file object.

1. **Calculating monthly mean fluxes from the processed data**

Having described the first stage of the IMB calculation (production of processed, cleaned data at standardised time points), the second stage is now described (production of monthly mean vertical fluxes).

The driving routine, all\_variables\_monthly.py, opens a new netCDF file for the output, with an empty record dimension and a salinity dimension (with associated salinity values set in parameters.py). For each desired output variable, a netCDF variable object is created, as well as an associated error variable object, to store information about why data might not be returned for a particular month, buoy and salinity. The routine then loops over all buoys. For each buoy, all valid months during which data is present, and the time points corresponding to these months are identified.

Next, auxiliary variables not dependent on ice salinity are calculated (e.g. changes in elevation), using the function imb\_calc.aux\_vars\_non\_salin(). This makes use of many of the rate of change functions in difference\_functions.py.

A loop over salinity values then begins, and auxiliary variables dependent on salinity (e.g. conductivity, heat capacity) are calculated using imb\_calc.aux\_vars\_salin(). For each desired output variable, the corresponding function in imb\_calc is then called, and an estimate of the variable for that month and salinity returned. Both auxiliary functions, and all of the individual variable functions, make use of scientific constants defined in scientific\_constants.py, and user-defined parameters in parameters.py (e.g. conductivity scheme, layers over which to define conductive flux and sensible heat uptake).

Error handling follows. This is controlled by imb\_calc.handle\_errors(), which looks for common sources of error such as:

* not enough data being present in a key variable for the month in question
* the ice not being of sufficient thickness for a basal conduction calculated 40-70cm above the ice base to be meaningful
* the ice temperature being too high for a given salinity to be physically possible

For each variable, the associated buoy error variable is assigned an error code according to which error is detected (or 0 if no error is detected).

Finally, if the error code for the error variable is 0 following the error handling, the variable estimate is read into the main netCDF variable object for the record and salinity value in question.