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

Version 1

Alex West, October 2018

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 -. 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 2014-2016, 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 Python2.7 and is contained in 6 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

In addition, a text file called **mday\_flag.txt** contains a list of buoys for which date is in British 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.

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.

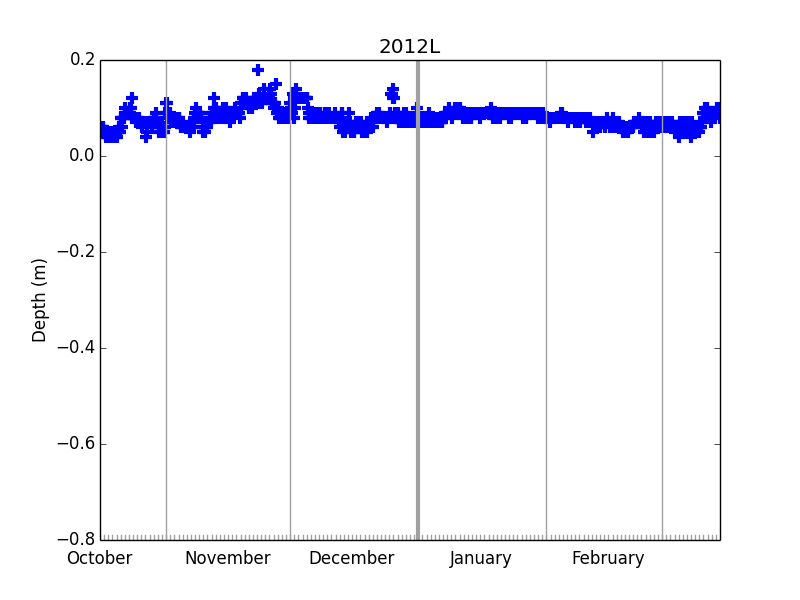
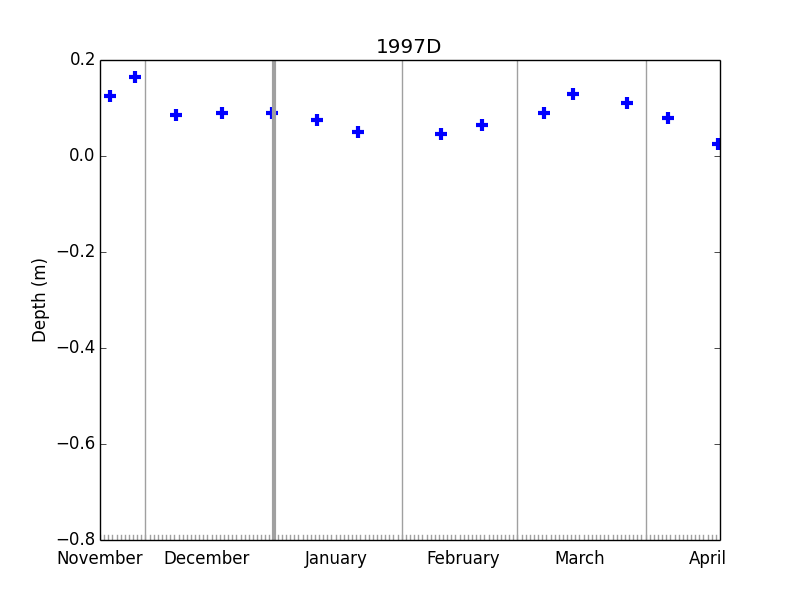
series.dates()

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() makes use of the estimate() method to produce a regularly-spaced daily timeseries (with times of observation at midnight every day) from a given raw timeseries. For example:

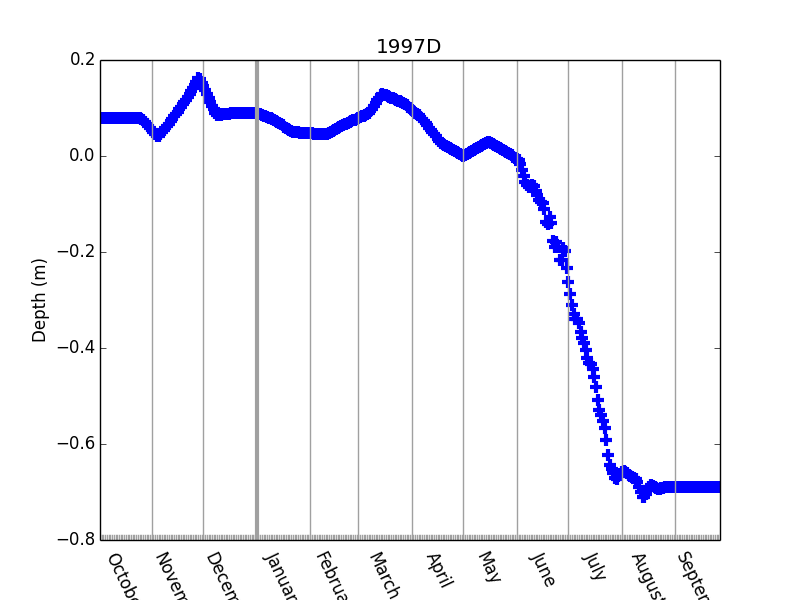
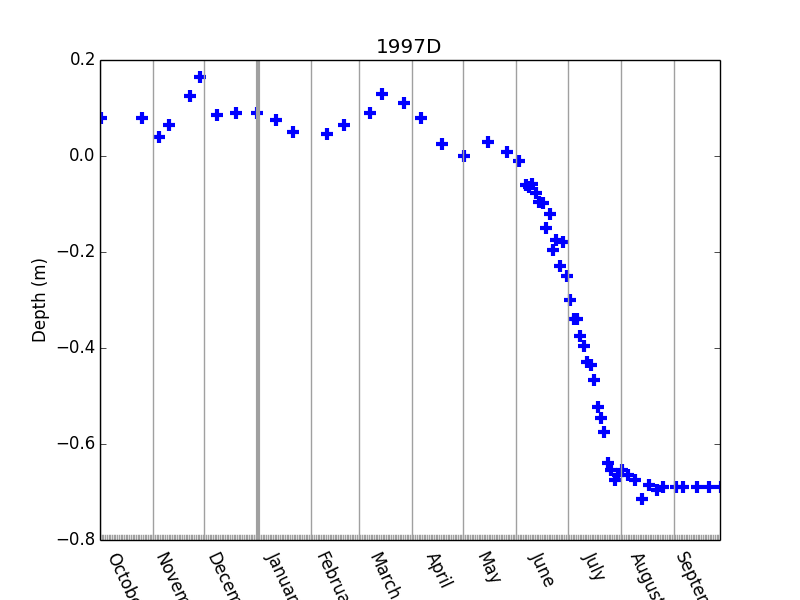
import buoys

buoy\_str = buoys.buoy(‘1997D’)

buoy\_str.extract\_data(‘surface’)

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

The resulting regular time series is shown in Figure 2:



*Figure 2. (left) Raw surface elevation data from buoy 1997D; (right) daily 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).

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.

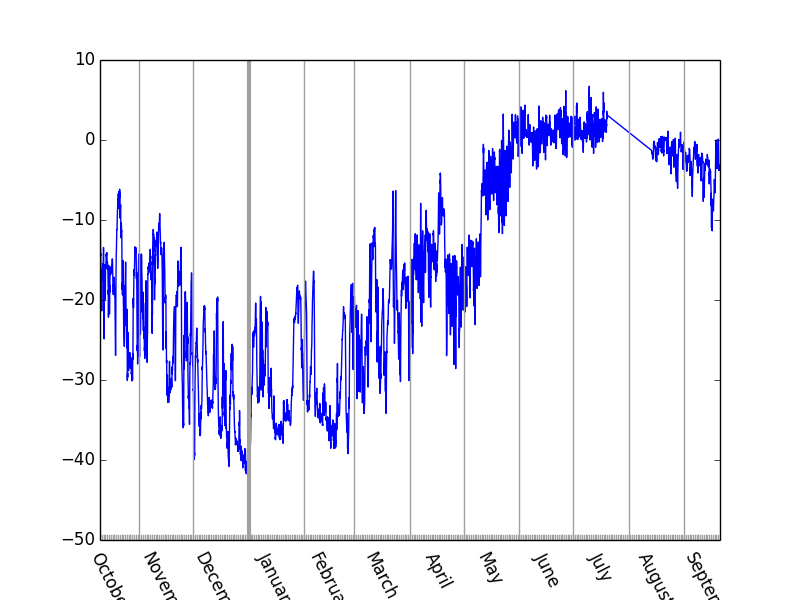
There are several methods provided to visualise the temperature data. The below code creates a timeseries plot of the temperature at level 10 (where level 0 is at the top):

import buoys

buoy = buoys.buoy(‘1997D’)

buoy.process\_temp()

buoy.temp.show(10)

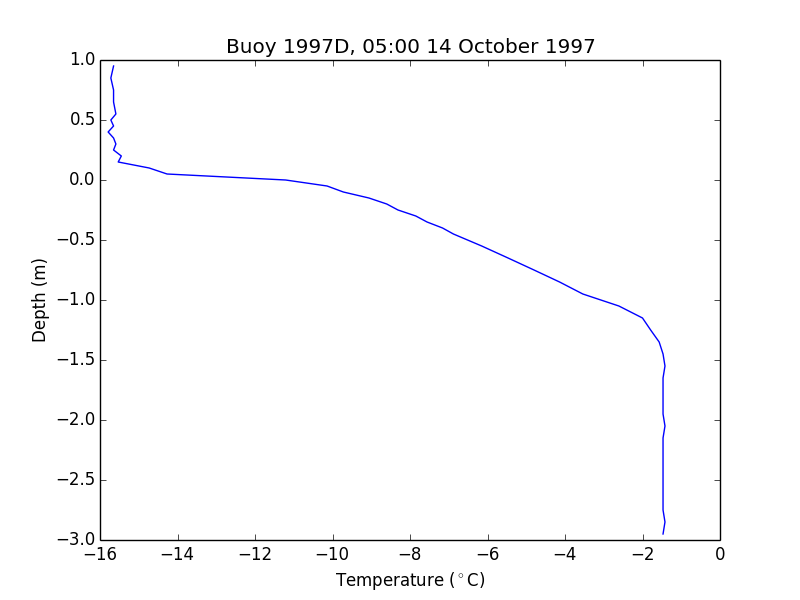


*Figure 3. Timeseries of temperature at level 10 in buoy 1997D.*

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

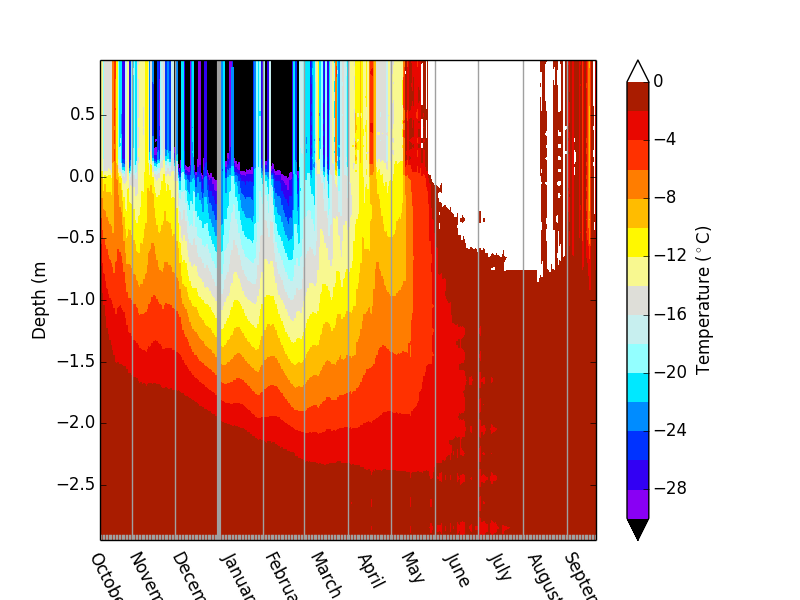
tdates = buoy.temp.dates()

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.*