budget_process.py manual

Hello welcome to the first manual for my python budget processing script.

Introduction

Here are the key features of the script:
- It is highly generalisable for new data formats
- Data formats are in their raw format, no preprocessing/smoothing/ or regridding required
- It is currently capable of computing the daily and monthly budgets
- All regridding is done on the fly
- The code progresses through the input data, outputting monthly outputs as it goes
- It can cope with gaps in the data (say from winter only sea ice thickness)

I'm taking you through the files on the github page https://github.com/mimi1981/PIOMAS

This code is also running on the server on /home/hds/Python/Budget

If you wish to modify and run some budgets on the server, copy this directory - avoiding the Outputs folder (current results - big folder)

Running the code

- Put all the modules listed below in a directory somewhere.
- In that directory make another called 'grids' put all the grids in there.
- Make the edits suggested below to budget_process.py - particularly the input data paths.
- You may wish to try all the diagnostics and plotting options for first runs
- Then type the following from a command line whilst in the budget directory you just made
- `python ./budget_process.py`

If this give you too many annoying warnings about missing values (empty cells in your data grid) run this instead
- `python -W ignore ./budget_process.py`

Code description

The code consists of two custom modules and an executable script.
There is also my gird_set module for working with python geo grids
Also there are various grid files for pre-created grids.

Processing script

Runtime Options

Here I describe the options you can set in the budget_process.py script
You may well be able to do lots of interesting things with this code just by using these options.

The code starts with the usual python import scripts. This includes the modules I've written:
budget.py
budget_inputs.py
grid_set.py

Then comes the first part to edit.
Here is the directory we'll write the output data to.

```
## outputs
spath = './out_Pathfinder_AWI_monthly/'
```

This the path where your budget calculations will be saved
What next follows are two scripts that first: check the spath exists, and makes it if it doesn't, then another that will copy all your options to a text file in the spath.

```
##### OPTIONS #####
## dates

ts = dt.datetime(2011,1,1)
te = dt.datetime(2018,12,31)
tw = relativedelta(days = 1)
```

Here are the options for the time periods to consider.
ts is the start time, te is the end time, tw is the frequency of data.

ts and te are in python datetime format, this makes them very easy to work with and does all the days/hours/leap year faff for us.
tw is a relativedelta, that works well with the datetimes, and sets the period the model will step through. The code currently works for either

```
tw = relativedelta(days = 1)
tw = relativedelta(months = 1)
```

which considers data at daily or monthly time scales. If an input data set is on daily time scales, and we set tw to a month, the script will then average all the input data over each month. If the input data is given every month, and tw is a day, then the code linearly interpolates to give a daily data point.

Next are a bunch of options of things the code can do

```
## message option
load_verbos = False
hole_verbos = False
```

These two options get the code to write much more output, this tells you exactly what each forcing object is doing. Also we can tell what date it is considering when avoiding gaps in the data.

```
## diagnostic plots - for the first time point
print_inputs = True
print_budget = True
print_budget_square = True
print_dstats = True
print_history = True
print_history_square = True
```

These options will produce a bunch of plots to see what data we're working with and whether we've regridded and projected correctly. The first 4 all produce a single plot at the beginning of the run. This is in the form of a pdf saved in the spath directory. The two 'history' files will do the same every time the code saves a history file. The 'square' options produce plots that are on the velocity data gird, and are primarily diagnostics, as the data grid usually looks a bit odd. The other are projected due to the Basemap projection below.

These plots auto create themselves somewhat, I've endeavoured to make the velocity arrows work with any data, and also the colour scales, but no promises. They won't be publishable I doubt.

```
## deformation options
```

```
calc_dstats = True
```

This gets the code to run some diagnostics on the drift data, and save it to a separate file. This slows the code down a fair bit so turn it off if you're not interested.

```
## masking options
mask_poleV = False
mask_poleT = False
pole_hole_lat = 88.0
```

This is to add some extra masking options for input data. This can be needed for certain data that messes with the pole hole. It can however also cause some issues with smoothing on certain types. If you can leave this alone and get decent results then do so!

```
## Thickness/Con limit masking
Tlim = 0.2   ### meters below which we mask
Clim = 0.15  ### conc.  below which we mask
```

This removes all thickness and concentration data below a limit. Useful for data that has issues with land (NSIDC) and open ocean (PIOMAS)

```
### Time smoothing velocity option
V_time_smooth = False
```

This option we used for comparing model and observational drift data. This averages the velocity data over 3 time steps before performing the budget.

The next set of options set up the grid structures for the input data we are using.
These use my grid_set module. This example code has some pre built grid files that contain all the information needed. The module has all the code required to make these files, but that's for another manual.

The grids need to sit within a projection, this is defined first. Is should be possible to change this projection to anything, but be careful as it is used to handle the regridding

```
m = Basemap(projection='stere',
            lon_0=-45.0, lat_0=83, lat_ts=0,
            height = 3335000*1.8, width = 3335000*1.8)

### VELOCITY
GV = gs.grid_set(m)
GV.load_grid('grids/Pathfinder_gs.npz')
GV.load_mask('grids/Pathfinder_gs_mask.npz')
GV.reproject(m)

#### THICKNESSS
GT = gs.grid_set(m)
GT.load_grid('grids/AWI_gs.npz')
GT.load_mask('grids/AWI_gs_mask.npz')
GT.reproject(m)


#### CONCENTRATION
# GC = gs.grid_set(m)
# GC.load_grid('grids/AWI_gs.npz')
```

```
# GC.load_mask('grids/AWI_gs_mask.npz')
# GC.reproject(m)
```

Each input data needs to have a grid defined. In this case the Thickness and Concentration are on the same grid so we'll define one and then copy it.

Next we define the input data. These are all input data objects as described below in Module 1.

```
##### Set up budget input objects
### all the code is in the budget_inputs.py
### first we query the dates of interest
path = '/Volumes/BU_extra/BUDGET/Pathfinderv4/'
P = bi.Pathfinder(path)

P.get_dates(ts,te)

path = '/Volumes/BU_extra/CryoSat/AWI_thickness/monthly/'
A = bi.AWI_monthly(path)

A.get_dates(ts,te)
```

Each data object is initialised with the path where it exists. Then we query what data points are available in that path. In this case the AWI data contains both thickness and concentration so we'll be using this object for both cases.

Next we need to check the thickness data to see which time points we can actually perform the budget analysis on. The get loop list looks a list of input data points and returns a list of dates that a budget can be computed on.

```
### use data to set the dates to process
dlist = b.get_loop_list(A.dates,tw)
ndays = np.shape(dlist)[0]
```

Now we need to point the code to the the particular object that velocity, thickness and concentration is found in.

```
# input select
InputV = P # velocity forcing
InputT = A # thickness
InputC = A # concentration
```

In this case we point the code to Pathfinder for velocity and AWI monthly data for thickness and concentration.

The next option is for whether we need to create an effective thickness from the concentration and thickness - some options need this, some don't

```
# option for whether we're using effective thickness or not
make_eff_thk = True
```

Now we point the code to the correct grid objects for each input data

```
# grid select
Gvel = GV
Gthk = GT
Gcon = GT
```

Again the thickness and the concentration both live on the same grid so we pass the same object GT

To run the budget all data needs to sit on the same velocity grid. There are two options if we need to regrid the thickness and concentration. In this case yes we do.

```
rgd_thk = True
rgd_con = True
```

The regridding all uses the grid_set module and runs efficiently. Once the grid_set grid files have been run you need do nothing extra.

Next are options for spatial smoothing,

```
# smoothing options
smthvel = True
smththk = True
smthcon = True
```

These options will just turn the smoothing completely off, and leave the input data alone. Worth doing once just to compare what the smoothing is doing to your results

```
#### old uniform
velsmrad = 6
Vsmth = lambda x: b.smooth2a(x,velsmrad)
thksmrad = 4
Tsmth = lambda x: b.smooth2a(x,thksmrad)
consmrad = 1
Csmth = lambda x: b.smooth2a(x,consmrad)
```

This is the original Paul/Michel smoothing function that does a box kernel over a set number of grid cells. OK for near uniform grids that you already know the size of. It runs quite quickly

```
#### new vary smoothing
velsmthdist = 200e3
thksmthdist = 100e3
consmthdist = 60e3
```

This new smoothing script I developed allows you to set the dimensional distance of the box kernel, and the data will be smoothed over enough cells that represent that distance. This is designed to be more set and forget when changing data on different grids to make the smoothing consistent. Also the kernel will be larger in regions of uneven grids with finer resolution (like PIOMAS, or any lon/lat data near the pole). Distance is in meters.

```
# VSmthObj = gs.geo_vary_smooth(Gvel,velsmthdist,verbos=True)
# TSmthObj = gs.geo_vary_smooth(Gvel,thksmthdist,verbos=True)
# CSmthObj = gs.geo_vary_smooth(Gvel,consmthdist,verbos=True)

# Vsmth = VSmthObj.smooth
# Tsmth = TSmthObj.smooth
# Csmth = CSmthObj.smooth
```

This is where the smoothing weights are calculated. This script is slow. So comment it out if you don't use it.

The rest of the script is a little messy but works, the user 'should' be able to leave it all alone and just play with the above options. There several additional developments to consider but I'll discuss them at the end.

Module 1, budget inputs

This module contains all the code the script requires to access the input data.

Each type of input data (example PIOMAS) has its own class. A python class is a bundle of code and data that can passed around a larger piece of code. I have used classes so that I can let the script decide which data it needs to access and when to access it. The input data class is then used to access the code.

For example Pathfinder drift is initialised as

```
import budget_inputs as bi

path = '/Volumes/BU_extra/BUDGET/Pathfinderv4/'
P = bi.Pathfinder(path)
```

Each data class has the method

```
    def get_dates(self,time_start,time_end):
        """
        returns the all encompassing date list for use with the forcing
object
        """
        dates =[]
```

This method looks through the directory you supplied and works out what data is in the directory at what time point.

Also each class has a at least one of

```
def get_vels(self,dates_u):
def get_hi(self,dates_u):
def get_aice(self,dates_u):
```

that return a list of data arrays for a given list of dates for either velocity, thickness, concentration. The inner workings of these methods are specific for the data format. However they all return the same numpy arrays of data.

If we want to add a new input to the budget, the first thing to do is write a new class for it. See below in the **Add new input** section.

Module 2, budget.py

This module contains all the custom functions used in the code. I'll explain them one by one

```
def smooth2a(matrixIn,N):
```

This is Paul/Michels function for smoothing an array over N data points.

```
def select_dates(dates,time_u,time_w,diag = False):
```

This function decides which dates of data from a budget_input class lie within the period of interest. The period is defined but it's start point time_u - which is a python datetime, and a length time_w a python relativedelta. Currently the code will work with a window of 1 day or 1 month only, for all input types. If the input data is all daily data then any multiples of days might work.
This returns all the dates in the list of dates inputted plus the previous date and the following date. This means that if no dates are found, you will still be given the previous and following data point. This allows us to interpolate monthly data to daily data.
Also if a large amount of data is within the period the code will be able to average them.

```
def get_load_points(datesQ,time_u,time_w,diag=False):
```
This function takes he output of the previous function and analysses it to see which dates from the list to actually load, and what to do with them.
If the list of dates datesQ is at least three dates long, we ignore the endpoints and just return the rest as the points to load - d_load. All the data from the list will be averaged later.
If the list is only two points long then we need to work out what to do - maybe interpolate, there are several cases.
We compare then compare two dates in datesQ and compare them with time_u and time_w
If the if any of them align perfectly, then we're trying to access a single data point so only one will be accessed and no interpolation.
If time time_u and time_w combine to lie with datesQ then the interpolation weights are calculated and an interpolation flag is passed.

```
def get_hi_array(bfhi,time_u,time_w,diag=False):
def get_aice_array(bfhi,time_u,time_w,diag=False):
def get_vels_array(bfhi,time_u,time_w,diag=False):
```
These functions call the previous two functions and apply them to the list of dates incorporated in the bfhi input data object (for example set bfhi = PIOMAS to access piomas data)
Dates are queried, data is read,
then it maybe averaged in time
or maybe has interpolation weights applied to it
or simply return as is
depending on what the previous code finds out when looking at the list of input data dates.
get_vels_array returns two arrays for the x/y components.

```
def get_loop_list(dsearch,tw,include_last = False):
```

This function decides which dates we want to try and compute a budget for. This was written for computing a budget over seasonal data (CryoSat winter thickness for example) The returned list of dates are what we loop over when doing the budget. Here dsearch is the date list from an input data object.

Tutorial: adding a new input data type

Here I'll talk you through adding a different input data. The example will be for PIOMAS data, whos data object is included in the budget_inputs module.

The two things required when using a new data input are:
it's grid, contained in a grid_set object saved in a file. Making these is best save for another tutorial.
it's input_data object, written in a module somewhere, in this case in budget_inputs. I'll describe the PIOMAS object

```
class PIOMAS():
    """
    forcing class for the budget
    lets the forcing load efficiently

    """
    def __init__(self,ppath):
        self.name = 'PIOMAS'
        self.path = ppath
        self.vyear_load = 0
        self.hyear_load = 0
        self.ayear_load = 0
        self.vels_loaded = False
```

```
        self.hi_loaded = False
        self.aice_loaded = False
```

The __init__ mehtod is crucial part of object programming in python. It is how we bring the object into another piece of code. Here we start our data object by telling it where the code lives - path. Also specific for PIOMAS, as all the data is in large yearly arrays I want to record which data has been read, so we don't have to keep rereading data from the same year that's already been loaded, but not used yet.

Next is the get dates method. As PIOMAS has big yearly arrays with no missing values within it (it's a model) we can simply take the queried time range and make a list to reflect that. This method also prints a report when it's called at the beginning of the code.

```
    def get_dates(self,time_start,time_end):
        """
        returns the all encompassing date list for use with the forcing
object
        PIOMAS is a standardised list so we can just build a list
        """
        dates =[]
        n_yrs = (time_end.year - time_start.year)-1
        if n_yrs>-1:
            y0 = dt.datetime(time_start.year,1,1)
            ye = dt.datetime(time_start.year,12,31)
            for d in range(time_start.timetuple().tm_yday-1,
                           ye.timetuple().tm_yday):
                dates.append(y0 +  relativedelta(days = d))
            for y in range(n_yrs):
                y0 += relativedelta(years=1)
                ye += relativedelta(years=1)
                for d in range(ye.timetuple().tm_yday):
                    dates.append(y0 +  relativedelta(days = d))
            y0 += relativedelta(years=1)
            ye  = time_end
            for d in range(ye.timetuple().tm_yday):
                    dates.append(y0 +  relativedelta(days = d))
        else:
            y0 = dt.datetime(time_start.year,1,1)
            for d in range(time_start.timetuple().tm_yday-1,
                           time_end.timetuple().tm_yday):
                dates.append(y0 +  relativedelta(days = d))

        self.dates= dates
        print(self.name+' Found '+str(np.shape(dates)[0])+' dates')

    # daily points in yearly files
```

The next three methods are very similar, they just access a different form of data: velocity, thickness or concentration. I'll talk us through the code. Most of this is specific to PIOMAS, look through budget inputs to see how I've dealt with the other data.

```
    # next function will take a list of dates and return an appropriately
orientated arrays
    # give a
    def get_vels(self,dates_u):
```

The way the code wants to access data is from a list of dates that are needed.

Firs this method works out which year the dates are from, and sees whether that year of data has been loaded yet. If it has fine if not then load it.

```
        # does dates_u cover one year or more
        if (dates_u[-1].year -dates_u[0].year) == 0:
            # one year, one file
            yu = dates_u[0].year
            if ((self.vyear_load != yu) or (not self.vels_loaded)):
                print('loading new year of data: ‘+str(yu))
```

Here we load the data. This is where the power of the datetime format is. it’s very easy to build the correct file name from the date supplied.

```
                data_f = self.path+'uiday.H'+str(yu)
                with open(data_f, mode='rb') as file:

                    fileContent = file.read()
                    data = struct.unpack("f" * (len(fileContent)// 4),
fileContent)

                self.vels=np.asarray(data).reshape(365,2,120,360)
                self.vyear_load = yu
                self.vels_loaded= True
```

The current method then looks at the list of dates given and returns the data associated with these dates. The current limitations to the code:
The dates in the list need to be consecutive, the dates need to be all from the same year.
This is fine for the current uses of the script.
The use of object programming is that if this is no longer fine, all we need to do is adjust these methods to deal with it.

```
            p0 = dates_u[ 0].timetuple().tm_yday -1
            p1 = dates_u[-1].timetuple().tm_yday
    #         print(p0,p1)
            datau = self.vels[p0:p1,0,:,:].transpose((0,2,1))
            datav = self.vels[p0:p1,1,:,:].transpose((0,2,1))
            return datau,datav
```

The correct data is selected and returned. We also do a transposition of the data to make it consistent with the grid_set object for PIOMAS.

```
            # path finder needs transposition on the final 2 dims
        # else:
            # get no. of years/files
```

Similar code for thickness:

```
    # next function will take a list of dates and return an appropriately
orientated arrays
    # give a
    def get_hi(self,dates_u):
        # does dates_u cover one year or more
        if (dates_u[-1].year -dates_u[0].year) == 0:
            # one year, one file
            yu = dates_u[0].year
            if ((self.hyear_load != yu) or (not self.hi_loaded)):
                print('loading new year of data: '+str(yu))
                data_f = self.path+'hiday.H'+str(yu)
                with open(data_f, mode='rb') as file:

                    fileContent = file.read()
                    data = struct.unpack("f" * (len(fileContent)// 4),
fileContent)
```

```
                self.hi=np.asarray(data).reshape(365,120,360)
                self.hyear_load = yu
                self.hi_loaded  = True

            p0 = dates_u[ 0].timetuple().tm_yday -1
            p1 = dates_u[-1].timetuple().tm_yday
    #         print(p0,p1)
            data = self.hi[p0:p1,:,:].transpose((0,2,1))
            return data
            # path finder needs transposition on the final 2 dims
        # else:
            # get no. of years/files
```

Similar code for concentration:

```
  # next function will take a list of dates and return an appropriately
orientated arrays
    # give a
    def get_aice(self,dates_u):
        # does dates_u cover one year or more
        if (dates_u[-1].year -dates_u[0].year) == 0:
            # one year, one file
            yu = dates_u[0].year
            if ((self.ayear_load != yu) or (not self.aice_loaded)):
                print('loading new year of data: '+str(yu))
                data_f = self.path+'aiday.H'+str(yu)
                with open(data_f, mode='rb') as file:

                    fileContent = file.read()
                    data = struct.unpack("f" * (len(fileContent)// 4),
fileContent)

                self.aice=np.asarray(data).reshape(365,120,360)
                self.ayear_load = yu
                self.aice_loaded= True

            p0 = dates_u[ 0].timetuple().tm_yday -1
            p1 = dates_u[-1].timetuple().tm_yday
    #         print(p0,p1)
            data = self.aice[p0:p1,:,:].transpose((0,2,1))
            return data
            # path finder needs transposition on the final 2 dims
        # else:
            # get no. of years/files
```

So now how do we introduce this input object to the script? I'll show the lines that need to be modified.

These two lines need to be replaced for PIOMAS

```
GV.load_grid('grids/Pathfinder_gs.npz')
GV.load_mask('grids/Pathfinder_gs_mask.npz')
```

The PIOMAS files are in the grids directory

If you want to use PIOMAS for all inputs just define one grid and copy it to all cases on the lines

```
# grid select
Gvel = GV
Gthk = GT
Gcon = GT
```

So replace GT with GV in this case.

The same for the inputs objects.

```
path = '/Volumes/BU_extra/BUDGET/Pathfinderv4/'
P = bi.Pathfinder(path)

P.get_dates(ts,te)
```

needs to be modified to use the PIOMAS object with

```
P = bi.PIOMAS(path)
```

Changing the path as necessary.
Again to use PIOMAS for everything set all of the below to PIOMAS

```
# input select
InputV = P # velocity forcing
InputT = A # thickness
InputC = A # concentration
```

Also we don't need to regrid so change these options

```
rgd_thk = True
rgd_con = True
```

Also you need to consider the effective thickness option too

Then chose an appropriate location to put the output fields.

Then chose the time range and window options.

Finally you need to generate the date list to loop over

```
### use data to set the dates to process
dlist = b.get_loop_list(A.dates,tw)
ndays = np.shape(dlist)[0]
```

Changing the A.dates to the PIOMAS dates: P.dates

Extra developments

the current restrictions for the script then can be dealt with if needed.

the script can only consistently deal with time steps of a single day or single month. This is due to how a multiple day time step will interact with monthly data. The code will crash as the time step period bridges a time point. Also I'm not sure how it will cope with accumulating history files.

Currently the history files are saved every month. If we want different periods then I need to deal with this. The history will save at the end of the run, or at the beginning of a hole. So if you just run 5 days of interest, you'll get a history file.