

Readme WAM2-model in Python

In the first part of this readme, a step by step description is given of the process of converting the WAM2-model, originally created in MATLAB, to Python. Additionally, some new features in Python and installation guides have been added.

The MATLAB model is originally created by Ruud van der Ent for his PhD thesis [‘A new view on the hydrological cycle over continents’](#). His model describes the global atmospheric fluxes which show the evaporation and precipitation on the different continents resulting in the phenomenon known as moisture recycling.¹

The original MATLAB model consists of 23 different scripts. For convenience the same structure is used for the Python model. Therefore the Python model will consist of the same 23 scripts as the MATLAB model does. Also a new version of the model has been written in Python. In this version the full model is implemented in one large script. The benefit of this is that Python only needs to import everything ones which makes the model a lot faster. In this version the same type of code is used, therefore this will not be discussed in further detail in this read me.

“First solve the problem then, write the code”

- ***John Johnson***

June 2016

**Niek van de Koppel
Tolga Cömert**

¹ Ent, R.J. van der et al. (2009), *A new perspective on continental moisture recycling*, Department of Water Management, Faculty of Civil Engineering and Geosciences, Delft University of Technology, the Netherlands

How to use Python?

Python data and coding style is available in many different ways on the internet and therefore there is not a good or a wrong way of using Python. We have used Jupyter notebook to write our code and therefore we will give a short explanation of how to install the full package which we use as well. First, it should be noted that Jupyter itself runs many different programming languages, therefore Python is a requirement to be installed to be able to use the Jupyter notebook. To install both Python and Jupyter we have used the Anaconda distribution for Python 2.7, which can be downloaded [here](#).

After installation, to update to the newest version you can use your command prompt. Using Anaconda, this can be done using conda:

```
conda update jupyter
```

Also Python's own package manager pip can be used:


```
pip install -U jupyter
```

Use whatever suits you best to install everything. If any problems occur during installation try searching the internet for more information on this topic.

When the installation is done an additional package needs to be installed to be able to import and read netCDF files, since this filetype is used for all data from ERA-interim. To import these files we have used netCDF4. netCDF4 should be imported into your Python script as:

```
from netCDF4 import Dataset
```

The netCDF4 dataset can be downloaded [here](#) by  following the easy steps in the guide.

Other  packages which should be imported into Python for this script are in the Anaconda/Python by default so these do not have to be downloaded.

Using ERA-Interim data

As we have just explained above, we have used ERA-interim reanalysis data as input for the whole model. First, an ERA-interim account should be created. Next, to avoid downloading all files manually from the website ERA-interim have written a few very nice [example codes](#) to download the data. Take note that you use the script for netCDF files (Included in the zip file are the download scripts we have used)!

In these example codes, reference is made to parameters. These are standardized numbers which coincide with different parameters, for example precipitation or wind direction. More information on the parameters can be found [here](#) and the detailed database of all parameters is found [here](#). The parameters which should be imported for this model are given below:

Parameter name	Parameter abbreviation	Parameter number
Surface pressure	sp	134.128
Specific humidity	q	133.128
U-velocity	u	131.128
V-velocity	v	132.128
Total column water	tcw	136.128
Total column water vapour	tcwv	137.128
Eastern water vapour flux	ewvf	71.162
	eclwf	88.162
	ecfwf	90.162
Northern water vapour flux	nwvf	72.162
	nclwf	89.162
	ncfwf	91.162
Total precipitation	P	222.128
Evaporation	E	182.128
Land-sea-mask	lsm	172.128



The easiest way to download the lsm is from the website itself. The other data is easily downloaded with the above mentioned scripts. Take note that all data except P and E is downloaded every 6 hours (time: 00:00:00/06:00:00/12:00:00/18:00:00, step: 0) whereas P and E are downloaded for every 3 hours (time: 00:00:00/12:00:00, step: 3).

Explaining the model

Since this readme is included with the model, everything explained can be easily checked and used. Therefore not everything will be explained in full detail. Only the changes we made in Python with respect to the original MATLAB model and the additional features we have added will be mentioned.

As was explained in the introductory part, the model consists of 23 different scripts. Two of these scripts are referred to as 'masterscripts'. These scripts are used to import all data from the other scripts and will generate the wanted output data.

The first masterscript is the `Fluxes_and_States_masterscript_2layers_sigma` file. This file uses input from 9 different files. The step by step conversion of this script including the input files is given below:

Fluxes_and_States_masterscript_2layers_sigma

- In this file all functions from the previous files are imported and at the end a new file with the total fluxes in the atmosphere for your chosen coordinates is saved.
- The function type in MATLAB reads as `def 'name'` in Python.
- Important to notice is that Python starts with index **0**, not with **1** as MATLAB.
- The first step in this file is to import all constants and invariants from the `getconstants` file. Importing files in Python is done using the commands below:

```
import io
from IPython.nbformat import current
import matplotlib.pyplot as plt
```

```
def execute_notebook(nbfile):
```

```
    with io.open(nbfile) as f:
        nb = current.read(f, 'json')
```

```
    ip = get_ipython()
    for cell in nb.worksheets[0].cells:
        if cell.cell_type != 'code':
            continue
        ip.run_cell(cell.input)
```

```
    execute_notebook('your file')
```

Get_constants

- In MATLAB, `squeeze` is used to remove all singleton dimensions in an array. A singleton dimension is any dimension for which `size(A,dim) = 1`. Two-dimensional arrays are unaffected by `squeeze`. In Python, ***np.squeeze*** can be used giving the same result.
- For the `ism`, the lake mask needs to be added. In Python, the MATLAB arrays can be recreated by stacking the arrays and taking the transpose.

- In Python the index starts at 0 whereas in MATLAB, it starts at 1. Therefore all numbers used in the model should be subtracted with 1 to get the correct result. Good examples of this are the latitude and longitude datasets, which are 92 and 240 in length respectively. In Python this is written as [0:91] and [0:239], therefore 91 and 239 being the last values!
- Implement all defined constants in the code.
- Python defines arrays as one big row, whereas in MATLAB this is done in one big column. Therefore, the defined arrays need to be converted into columns, which is done using the command **np.vstack**.
- It is very important that all variables are returned at the end of a function. Next these variables should be defined to be able to 'call' these variables in the next scripts. This should be done for every function written. An example is given below:

```
latitude = np.vstack(getconstants())[0]
longitude = np.vstack(getconstants())[1]
lsm = getconstants()[2]
g = getconstants()[3]
density_water = getconstants()[4]
timestep = getconstants()[5]
A_gridcell = np.vstack(getconstants())[6]
L_N_gridcell = np.vstack(getconstants())[7]
L_S_gridcell = np.vstack(getconstants())[8]
L_EW_gridcell = getconstants()[9]
```

Begin of input

In the fluxes and states file you are able to choose your own input: for which years or year parts do you want the data? In how much time steps do you want the data to be represented? We recommend you to choose your own years and year parts but to keep the boundary, divt and count_time variables the same as defined. By changing these variables lots of changes in the model occur for which a thorough understanding of the model is needed.

- The isleap function in MATLAB is used to define whether a year is a leapyear or not. In Python we have written our own function for this doing exactly the same. This function is written below and is imported at the start of every notebook (**from isleap import isleap**):

```
def isleap(year):
    if year % 400 == 0:
        return 1
    elif year % 100 == 0:
        return 0
    elif year % 4 == 0:
        return 1
    else:
        return 0
```

- After the input is done the other files are imported in the same way as is done with the get constants file, however since you want to know the output for every year part and year defined separately, the other scripts should be put in a loop which runs through all days. Below are all scripts needed in the fluxes and states script described and elaborated:

getW_2layers_sigma

- The first step is to import the variables from the model_levels file:

Model_levels

- In Model_levels, different arrays are created (A, B, k) which are needed in other scripts.
 - In Python, this can be done using ***np.array*** and using ***np.vstack***, the array is vertically stacked.
-

- The 'not equal to' sign in MATLAB is ***~=*** whereas in Python this is ***!=***, for example, when referring to not the end of the year.
- ***DATA = NC_VARGET(NCFILE,VARNAME,START,COUNT)*** retrieves the contiguous portion of the variable specified by the index vectors ***START*** and ***COUNT*** in MATLAB. Remember that ***SNCTOOLS*** indexing is zero-based, not one-based. Specifying a -1 in ***COUNT*** means to retrieve everything along that dimension from the ***START*** coordinate. This is used in MATLAB to retrieve nc-files. In Python ***netCDF4*** is used as explained earlier. In this command, the same type of code is used specifying ***begin_time*** and ***count_time***, however this is not inside the function like with ***nc_varget*** in MATLAB.
- ***netCDF4*** is not able to read GRIB files, therefore all data should be downloaded and stored in nc-files. Additionally, we thought using one data type for all parameters was more convenient as well.
- When a variable is imported for multiple years, a ***+*** is used between every syntax in the function ***Dataset*** which originates from ***netCDF4***:

Dataset('C:/Users/TolgaC/MATLAB/Interim_data/' + str(yearnumber) + '-sp.nc'

- The variable named in the ***q_mod*** netCDF file is not 'Specific_humidity' but ***q*** and the variable named in the ***tcw_ERA*** netCDF file is not 'Total_column_water' but ***tcw***, referring to the parameters discussed above.
- The levels are flipped in the netCDF file, this can be solved by flipping the array indices with ***[:,::-1,:]***
- When it is the end of the year, the last observation is the same as the first observation in the following year. To ensure this works smoothly, we can make use of the ***np.insert*** function. We first extract the data for the 4 moments at a typical day and then we also extract the data for the first moment of a typical day in the next year. With ***np.insert***, the syntax is equal to the input array, index where new value needs to be put, input array and axis (which dimension in the array).
- It seems like **the values for different parameters are really close but not exactly similar** in MATLAB and Python. After speaking with Ruud van der Ent we acknowledged this problem as negligible.
- The levels are flipped again, so the line in 42 can be counteracted, which means just removing it. However, we still define ***q_f*** as ***q*** since this will be convenient.



- The parameter `p` can be created for columns in the first axis, however the MATLAB model has it swapped with the second axis. This needs to be changed, which is done with the **`np.swapaxes`** command.
 - The calculations are done in the swapped manner, so every time, the array swapped is needed for calculations.
 - The new array grid cells require an import function: `np.tile`.
 - The water volumes can be calculated in exactly the same way as is done with MATLAB, however one should keep in mind that MATLAB starts at 1, whereas Python starts at 0 index.
-

getwind_2layers_sigma

- In this file, we do exactly the same as in the file `getW_2layers_sigma` for importing the netCDF files of the wind in u-direction and in v-direction. Check the `getW_2layers_sigma` file or the explanation above if any error may occur.
-

getFa_2layers_sigma

- In this file we do exactly the same as in the file `getW_2layers_sigma` for importing the netCDF files of all vertically integrated fluxes. Check the `getW_2layers_sigma` file or the explanation above if any array may occur.
- To get back to what we have already described before, Python starts with index 0 where MATLAB starts at 1. Additionally, MATLAB has a parameter `end` which refers to the end whereas in Python this is just done in the following way:

<i>MATLAB</i>	<i>Python</i>
<i>1:end-1</i>	<i>:-1</i>
<i>2:end</i>	<i>1:</i>

As shown above, in Python `:` means everything, so `:-1` means everything except the last one and `1:` means everything except the first one. This will be used a lot in the different scripts and therefore this is explained here in detail ones more.

- To make the `corr_E` and `corr_N` arrays, we need a **for loop** which will run through all longitudes and latitudes. This looks like it takes a lot of time, however, Python does it rather quickly.

getEP_2layers_sigma

- In MATLAB, for $x = 1:4:\text{count_time} \times 2$ is used to define values that apply for the amount fallen/evaporated during the previous three hours. In Python, this is done using the ***np.arange*** variable: for x in `np.arange(0, count_time*2, 4)`.
- To remove the negative values, make everything positive --> we do this by evaluating the maximum and minimum of the evaporation and precipitation. For this the matrix needs to be reshaped. This is explained in more detail in the `stablefluxes` script. Since this is a small matrix it can also be done with a loop but we chose to do it with `reshape` for consistency throughout the model scripts.
- To make a 3D shape to calculate the volumes, MATLAB uses `repmat` and `reshape`, in Python the function ***np.tile*** can be used in the same way `repmat` is used in MATLAB. For `reshape` Python has a MATLAB like variable ***np.matlib.reshape***.

getrefined_2layers_sigma

- In defining the `oddvector` and `partvector`, ***np.zeros*** is used. This is defined as `oddvector2 = np.zeros((1, np.int(count_time*2*divt2)))`.
- ***np.int*** is used to make sure `count_time*2*divt2` is an integer instead of a float in which case Python gives an error message.
- In the `getrefined` file, variables are 'cleared' in MATLAB on multiple occasions. This is only done to remove it from the workspace. In other scripts this is also done a couple of times. Since ***in Python no workspace exists*** apart from the workspace you write your code in, it is ***not needed to write a command to clear variables*** in Python.
- For the variable `nan` in MATLAB, `np.nan` can be used in Python which makes it an easy conversion. However, first a `np.zeros` matrix should be created which should be multiplied by ***np.nan*** to get a `nan` matrix.
- The reason that `np.nan` is chosen in favor of `np.zeros` in some cases is due to the fact that by using `NaN`, making a mistake quicker leads to an error message whereas `0` already is a number. So when a mistake is made with a zeros matrix, this error might be missed.

get_stablefluxes_2layers_sigma

- To convert `Fa_N_top` and `Fa_N_down` to m^3 , the same steps are done in Python as in MATLAB, however the only difference is the fact that the axis of the latitude and the `count_time*divt` should be swapped since Python always loops over the first column/axis which in this case should be the latitude. For this, the command `np.swapaxis` is used, as explained before. After the for loop has ran through all variables, the matrices are swapped back with the same command which results in the same matrix as obtained in MATLAB.
- A new code has to be written in Python to find where the negative fluxes are since this cannot be done in the same way as in MATLAB. In MATLAB, it is possible to just say `Fa_E_top < 0 = -1`, whereas Python sees `Fa_E_top` as a matrix and therefore a for loop should be

constructed in which Python loops through all variables in Fa_E_top and finds all negative values. With the if statement these values are then regarded to with -1 in the **np.ones** matrix.

- In the previous script we have changed the float divt into an integer since this is necessary when defining matrix dimensions. However, in multiplications/divisions the original float is needed for which we use the variable **np.float** to change it back.
 - To create a matrix of the same size as an already created matrix of a variable, the command size is used in MATLAB. In Python we use **np.shape**: e.g. np.shape(Fa_E_top))
 - For calculating Fa_E_top_stable, Fa_N_top_stable, Fa_E_down_stable, and Fa_N_down_stable we should **reshape the shape of the matrix, since Python is unable to check the minimum of every variable in a matrix without looping through it which takes a lot of time**. Therefore we use **np.reshape** twice. First we reshape it from 96,92,240 to 2119680 and calculate the minimum of the two arrays using **np.minimum**. Then we reshape it back to its original 96,92,240 shape by using np.reshape again.
-

getFa_Vert_2layers_sigma

- By defining all the horizontal fluxes over the boundaries, van der Ent uses a loop in his model to 'store' these fluxes. Since this is only done to be able to fold all the lines of code we skip this in our Python script.
 - An important aspect which is especially important in this script since it is a lot of code is the fact that **MATLAB uses .* and ./ whereas in Python this is just * and /**.
 - In this script it is needed to reshape the matrix twice the same way this is done in the stablefluxes script. This is explained above.
 - Code should be rewritten with caution. This does not just hold for this script but for all script, especially the ones with lots of text. These scripts might seem to be an easy conversion but a mistake is easily made.
-

- **When Python is finished running through all these scripts the output should be saved. This is done using the scipy.io package:**

```
sio.savemat(('your target folder' + str(yearnumber) + '-' + str(a) + 'fluxes_storages.mat'),  
{ 'Fa_E_top':Fa_E_top, 'Fa_N_top':Fa_N_top, 'Fa_E_down':Fa_E_down, 'Fa_N_down':Fa_N_down,  
  'Fa_Vert':Fa_Vert, 'E':E, 'P':P, 'W_top':W_top, 'W_down':W_down})
```

- This is saved as a .mat file and can be loaded again in another file with sio.loadmat.
-

After running the fluxes and states master script file we should run the **Con_P_Recyc_Masterscript** or **Con_E_Recyc_Masterscript**, depending on the fact if you want a **forward tracking or backward tracking** respectively. These scripts use the saved data from the Fluxes and States master script which can be loaded into these files. Additionally, the different files for tracking and timing of the moisture are needed. **Below a description shall be given for the Con_P_Recyc_Masterscript and the different tracking and timing files needed. These files contain forward tracked moisture. The**

reason only these files will be considered is that the backward tracking and the Con_E file are built up in the exact same way.

Con_P_Recyc_Masterscript_2layers_sigma

- In the same way as is done in the Fluxes and States masterscript, we execute the get_constants notebook since its parameters are needed in the script.
- In this file, we should also start with some general input which is needed for the script to run. Most important difference in comparison to the Fluxes and States script are the definition of the Region and the Kv_f.
- In this (default) case the Region is just stated as the lsm (land-sea mask) created in the getconstants file, which is a pre-defined region of the whole world (without the poles and arctic).
- Kv_f is the vertical dispersion factor. For advection only this is 0, when dispersion is the same size as the advective flux this is 1. For stability Kv_f shouldn't be larger than 3.
- ***In this file a loop is created which runs through all the years, whereas in MATLAB you can just state yearnumber = years.***
- In MATLAB, the if and elseif statement are used. In Python the ***if statement*** is the same, however in contrast to the elseif statement ***else or elif*** is used.
- As already referred to earlier in the Readme, when data is imported for multiple years a + should be used, like:

previous_data_to_load = (str(yearnumber) + '-' + str(a-1))

- In this script ***sio.loadmat*** is used to import the fluxes and states outcome files and the track and time files. The parameters introduced from the MATLAB files should be defined in Python like:

Sa_track_top = sio.loadmat('C:/Users/TolgaC/Documents/MATLAB/sigma_levels/continental/' + previous_data_to_load + 'Sa_track.mat')['Sa_track_top']

- The tracking should always be used since the moisture in the atmosphere should be checked. The time scripts however, can be manually controlled choosing time tracking = 0 (no time tracking) or time tracking = 1 (time tracking). The time tracking is used to get an overview of the amount of time (days) a moisture particle is in the atmosphere and the length the particle travels (length- and timescales). This is not necessary for the tracking of the moisture itself!
- Below the tracking and timing files shall be explained shortly since these are needed for this masterscript and other scripts later on.

get_Sa_track_forward_2layers_sigma

- To start this script, a 3D region should be defined. For this we use the np.tile and np.matlib.reshape functions which were already used in previous scripts

- In the script you define $K_{vf} = 0$ in which case nothing should be done. The command for this in Python is pass.
 - All other code can almost identically be converted from the MATLAB model or commands are used which were already explained earlier on.
 - Important in this script is the amount of code which needs to be rewritten. Take caution when rewriting large pieces of code!
-

get_Sa_track_forward_2layers_sigma_TIME

- In this file, the exact same thing as in `get_Sa_track_forward_2layers_sigma` is done, however now the moisture is also timed. This is done at the end where the timetracking code is added.
- This file computes tracking together with time.
- Since the moisture is timed, Python should work with numbers therefore all parameters being NaN in a certain array should be converted to 0. We have done this by constructing a variable `where_are_NaNs` which seeks the NaNs using the `np.isnan` command:

```
where_are_NaNs = np.isnan(Sa_time_after_Fa_down)
```

```
Sa_time_after_Fa_down[where_are_NaNs] = 0
```

- ***This file is imported in the `get_Sa_track_forward_2layers_sigma_TIME` file.***
 - Finally the output files should be saved with `sio.savemat`. Depending on whether the data is tracked only or also timed, the script saves `Sa_track.mat` only or both `Sa_track` and `Sa_time`.
-

Con_P_Recyc_Output_2layers_sigma

- This file continues with the output gained from the fluxes and states and the `Con_P` masterscripts.
- Also in this file the `getconstants` script needs to be imported.
- Again, some input should be defined at the start of this script. Only the years and yearpart are important in this case, as well as whether you want the moisture to be timed or not.
- In this file, we need every day of the year, therefore ***the variable `monthstruct`*** is created. This variable consists of all months including every day for every month. In MATLAB this is defined as:

```
for y = years
```

```
    ly = leapyear(y);
```

```
    Jan = 1:31;
```

```
    Feb = Jan(end)+1:Jan(end)+(28+ly);
```

```
    Mar = Feb(end)+1:Feb(end)+31;
```

```
    Apr = Mar(end)+1:Mar(end)+30;
```

```
May = Apr(end)+1:Apr(end)+31;
```

```
Jun = May(end)+1:May(end)+30;
```

```
Jul = Jun(end)+1:Jun(end)+31;
```

```
Aug = Jul(end)+1:Jul(end)+31;
```

```
Sep = Aug(end)+1:Aug(end)+30;
```

```
Oct = Sep(end)+1:Sep(end)+31;
```

```
Nov = Oct(end)+1:Oct(end)+30;
```

```
Dec = Nov(end)+1:Nov(end)+31;
```

```
monthstruct = struct ('Month', {'Jan','Feb','Mar','Apr','May','Jun','Jul','Aug','Sep','Oct',  
'Nov','Dec'}, 'monthdays', {Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec});
```

- This generates 2 columns, the first column with the month's names and the second column with the amount of days. However in the second column the value is also an array (e.g. for January 1:31). **Since this is not possible with Python we have chosen to save this monthstruct as a .mat file for both a normal and a leap year.**
- This monthstruct.mat and monthstruct_ly.mat files are important with sio.loadmat into python depending on the fact if the chosen year is a leap year or not.
- As explained in previous scripts, NaN's should be converted to zeros. This is done using:

```
where_are_NaNs = np.isnan(P_time_per_day)
```

```
P_time_per_day[where_are_NaNs] = 0
```

- At the end the file is saved as:

```
P_track_2layers_sigma_continental' + str(years[0]) + '-' + str(years[-1]) + '.mat'
```

or:

```
P_track_2layers_sigma_continental' + str(years[0]) + '-' + str(years[-1]) + '-timetracking' +  
str(timetracking) + '.mat',
```

in the case the moisture is tracked in time as well as in space.

Hor_Fluxes_Output_2layers_sigma

- In this script, the output from the fluxes and storages files is squeezed into data per month per year instead of data from each day. This is done to keep the output data manageable when used to plot it in the last script.
- In this file, the constants are needed again so getconstants is imported.
- Input is needed in this file as well. The years and yearpart should be defined.


- In Con_P_Recyc_Output we have explained that the file monthstruct.mat was created since this is not possible in Python. This file is needed in this script as well and is loaded here.
- At the end the output is saved so it can be used in the Con_P_Recyc_Plot and Con_E_Recyc_Plot scripts.

Con_P_Recyc_Plots_2layers_sigma

All output needed is now finished so we can start plotting something! This script changes the code into visible results.

- All data which should be loaded and defined is done in the exact same way as is done in earlier scripts converting it from MATLAB to Python.
- However plotting the data is done completely different since the plots cannot be converted from MATLAB to Python. Additionally, plotting data depends on anyone's own taste.
- First we have defined **vmax = 1.0**, meaning the maximum in the plot is 1 (which is the maximum precipitation or evaporation recycling ratio). Next we have defined the colormap cmap using **LinearSegmentedColormap** from **matplotlib**:

```
cmap = LinearSegmentedColormap.from_list('mycmap', [(0 / vmax, 'white'),
                                                    (0.01 / vmax, 'blue'),
                                                    (0.3 / vmax, 'lightgreen'),
                                                    (0.5 / vmax, 'orange'),
                                                    (0.6 / vmax, 'red'),
                                                    (1.0 / vmax, 'black')])
```

- We use the command **%matplotlib qt** which closes any loaded/active figures when running the code again.
- For creating a plot of the earth we use **Basemap** from **mpl_toolkits.basemap**, using a **Robinson projection**. 
- To plot the data on the correct latitude and longitude we create a meshgrid:

```
x,y = np.meshgrid(longitude,latitude)
```

- We use **contourf** to create a filled contourplot of the data over the longitude and latitude grid. In **contourf** **clefs** is used to define the ranges:

```
cs = map.contourf(x,y,rho_c_landtotal,clefs2,linewidths=1.5,latlon=True, cmap=cmap)
```

- Colorbar is used to create a colorbar with our own inputdata:

```
map.colorbar(cs,location='right',pad="5%", ticks = clefs, label = '(-)')
```

- A windplot is created using a quiver plot:

```
windplot = map.quiver(x,y,Fa_E_total_m2a_part,Fa_N_total_m2a_part,color = 'black',scale = 199999999,latlon=True,width = 0.002, headaxislength = 5, headwidth = 2)
```

```
#### GRIDCELL PLOT
```

- Additionally we have created the plot using the 1.5° gridcells as well for convenience with the matlab plot, however we prefer the contour plots ourselves. For the gridcell plot we make use of **pcolormesh**.

Additional extensions

- Not shown in the readme is the conversion of the scripts for the continental evaporation recycling ratio. This is not done since the similarities with the precipitation recycling ratio scripts are so big, there is no need to explain this too.
- MATLAB shows in its command center the elapsed time for every day. To also add this feature in Python, we make use of the timer module. Starting a timer in the loop after a day is defined and a timer after the saving is done, will give the running time when both timers are subtracted from each other.
- MATLABs saving command will save the output files in a compressed manner whereas Python will save output files uncompressed. This leads to files being three times as big as the files saved by MATLAB. To solve this, additional code is needed in the save command, which is **decompression = True**. The only disadvantage is that running takes a bit longer (sometimes around 10%).
- In MATLAB, `inter(data)` is called within a certain directory in every script. This is not practical though since every directory needs to be changed in every script when these are not the same as defined in the original code. Therefore a new function is added, called `data_path`. This function makes use of the same strings as in the original code, however years and days are defined as parameters. When a is defined as a day, the function will be called which in return will give the correct data paths.
- Another extension is the addition of the model making its own empty arrays whereas in MATLAB, this has to be done by the user in the directory. When there is no data for the first day, a function called `create_empty_arrays` will make the needed arrays. If this is not needed, the lines can be commented.

This readme is made for the Python model version 1.0. However, this version of the model is rather slow relative to the MATLAB model. To solve this, a Python model version 2.0 has been created in which the same syntax is used. The only difference with version 1.0 is the fact that `execute_notebook` is not used anymore and functions are all defined in the masterscript itself. This leads to running times faster than version 1.0 and even the original MATLAB model. Since no additional code is used in version 2.0, an explanatory text will be left out.

To conclude, this is the full description of the conversion of the Water Accounting Model – 2layers (WAM2) from MATLAB into Python.

In the second part of this readme, sheds created with the Python model are explained.

In the second part of this readme, a description is given of the conversion of the scripts, for creating precipitation or evaporation sheds, from MATLAB to Python. This document is created as an addition to the original readme describing to conversion of the complete WAM2-model from MATLAB to Python.

Psheds_Masterscript / Esheds_Masterscript

This masterscript is in fact nothing different than the original masterscript which is described in the Readme WAM2 model in Python. However there are a few vital differences which should be explained to make it easy for you to make a precipitation shed of your data in Python.

- A new mask should be created for your region of interest. We have done this by creating an empty 92 x 240 sheet with all zeros. In this sheet the regions of interest should be defined with other numbers. The latitude and longitude coordinates of your region can be found in any mapping program or on the internet.
- Next you should explain to Python what to do with the information in this sheet (this sheet was named pshed_mask):

```
Continentnumber = 1 (any number which you have given to your region(s) of interest)
Continent = np.zeros(np.shape(lsm))
Continent[pshed_mask == Continentnumber] = 1
Region = Continent * lsm
```

- ***Creating a precipitation shed is in fact a backward tracking of moisture whereas an evaporation shed is forward tracking of moisture*** (likewise to con_E and con_P respectively).
- The rest of the script is exactly the same as the con_E_masterscript or con_P_masterscript the only difference being that ***an if statement*** should be created which makes Python choose between different continent numbers which you have defined yourself.

Psheds_Output / Esheds_Output

- In the output file exactly the same is done as in the original Con_E or Con_P output scripts the only difference being that the Continentnumber should be defined and you should make an if statement to let Python choose between different continent numbers for loading good data.

Psheds_plots / Esheds_plots

- The first part of these scripts are exactly the same as the original plotting scripts which creates a plot of the whole world, however it only displays moisture concerning your defined region (continentnumber).
- For the colorbar in this plot we have chosen to use the ***LinearSegmentedColormap from matplotlib.colors.***
- For both the absolute and relative sheds we have used a Mercator projection zoomed in on the defined area.

```
map = Basemap(projection='merc', lat_0=-30, lon_0=125,
resolution = 'l', area_thresh = 1000.0,
llcrnrlon=100, llcrnrlat=-50,
urcrnrlon=210, urcrnrlat=25)
```

- To define the steps on the colorbar we define ***'ticks'*** as shown below:


```
aa = [0,0.2,0.4,0.6,0.8,1,2,3,4,5]  
bb = [1,2,3,4,5,5.2,5.4,5.6,5.8,6]
```

```
ticks1 = [x*mult_month for x in aa]  
ticks2 = [x*mult_month for x in bb]  
Amounts = np.zeros((len(ticks1),1)) # cumulative amounts per tick [m3]
```

- By defining the Region again, we can create a contour around our area to show which area is the sink area in the plot:

```
pshed_mask = sio.loadmat('... /pshed_mask.mat')['pshed_mask']  
Continentnumber = 2 # 1 = China, 2 = North-Australia  
Continent = np.zeros(np.shape(lsm))  
Continent[pshed_mask == Continentnumber] = 1 # input for pshed  
Region = Continent * lsm
```

```
map.contour(x,y,Region, latlon=True, levels = [0], colors = 'black', linewidths = 4)
```

- The relative plot is done in the exact same way, however in the absolute plot the absolute amount of evaporation/precipitation is plotted, whereas in the relative plot the percentage of the recycling ration contributing to the evaporation is shown.