
Delft University of Technology

# MATLAB to Python

Use of precipitation sheds to explain the increasing rainfall trend in Northern Australia

Niek van de Koppel

TU Delft

# MATLAB to Python

## Use of precipitation sheds to explain the increasing rainfall trend in Northern Australia

by

Niek van de Koppel

to obtain the degree of Bachelor of Science
in Civil Engineering

at the Delft University of Technology.


| | | |
|---|---|---|
| Student number: | 4312473 | |
| Project duration: | April 18, 2016 – June 13, 2016 | |
| Thesis committee: | Prof. dr. ir. H. H. G. Savenije, | TU Delft, supervisor |
| | Dr. Ir. R. J. van der Ent, | TU Delft |


keywords: MATLAB, moisture recycling, Python, water accounting, North Australia, precipitation sheds

An electronic version of this thesis is available at `http://repository.tudelft.nl/`.

Front picture:
`http://www.inspired-tours.com/2015/03/10-beautiful-places-in-australia.html`

TU Delft

*"Learn how to see. Realize that everything connects to everything else."*

*Leonardo da Vinci*

# Preface

This thesis is written as requirement for the Bachelor of Science, Civil Engineering at the Delft University of Technology. Moisture recycling plays an important role in the hydrological world and this thesis focusses on this topic.

The base of this thesis is converting R.J. van der Ent's WAM2-model from MATLAB to Python. After this conversion, a precipitation shed is made for North Australia. This is done since there is an increasing rainfall trend and no explanation for this phenomenon has been found yet.

Reason for choosing moisture recycling as a topic for this thesis is mainly due to the fact that I already have affinity with this subject. In my minor 'De Delta Denker' at the Delft University of Technology, I already did a study on moisture recycling together with T.Cömert. This study can be found at `http://repository.tudelft.nl/view/ir/uuid%3Afbd08671-4551-40ec-9837-cf5dfc99f4d8/`.

I would like to thank my colleague T.Cömert for working together on this subject. I also want to acknowledge prof.dr.ir. H.H.G. Savenije and dr.ir. R.J. van der Ent for their tremendous help and support. Also, their trust and confidence in my capacity will definitely help me in the rest of my academic career.

Despite the fact that I had little to no knowledge about MATLAB, the same applying to my colleague T.Cömert, we eventually learned the MATLAB syntax and ultimately converted the WAM2-model.

I have done this thesis together with T.Cömert. Additional research and results on moisture recycling can be found in T.Cömert's Bachelor thesis.

*Niek van de Koppel*
*Delft, June 2016*

# Abstract

In this thesis, focus lays on converting a MATLAB model to Python. Models are used a lot by students and researchers at universities and they provide an unique way of calculating and reviewing data and results. However, at the Delft University of Technology, a lot of these models are written in MATLAB. MATLAB is a programming language which requires a expensive license. Many researchers in- and outside universities can't afford this license which means that models are not available for everyone although lots of people would like to work hands on with these type of models.

By making the first step in solving this problem, the WAM2-model originally created by R.J. van der Ent, is converted from MATLAB to Python. This conversion is done due to the fact that Python is open-source software which makes the model free to use for everyone. Also, MATLAB and Python syntax is generally the same which makes conversion even easier.

Ultimately, the WAM2-model is converted to Python and ready to use by everyone. There are two versions in Python available, namely 1.0 and 2.0. 1.0 is the version with the same structure as the model in MATLAB. However, it was found that the overall running time for version 1.0 was much larger than the original model in MATLAB. This is due to Python executing notebooks inside of one masterscript notebook, which is not very fast with Python's data structure.

To solve this problem, Python WAM2-model 2.0 is created. Version 2.0 makes more effective use of Python's data structure and this has led to running times almost twice as fast as the original model in MATLAB.

Another benefit of conversion is that both model versions in Python are more user-friendly. Headers are possible in Python and by defining the data paths in the beginning of every masterscript, users don't have to go through every script to change directories. In the tracking of moisture, sometimes the first day requires data which is not available. To make the calculation possible, empty arrays are needed which are now automatically created in Python whereas in MATLAB, the user has to do this at his own expense.

Ultimately, the plots created in Python are the same as in MATLAB. However, one additional plot is added, called the contourplot. With agreeing statements from the supervisors, these contourplots are more pleasing to the human eye and therefore they are included in the Python model.

Next to converting the model, a readme has been made which can be found in Appendix B. This readme is a step by step guide, helping users by explaining lots of the Python syntax. This guide is therefore also included with the Python model.

Using this converted model a precipitation shed is created for North Australia since previous research has shown an increasing rainfall pattern in this region. Different studies have tried to find the cause of this phenomenon. Most important seems to be the rise in sea surface temperature causing stronger monsoons, however also an increasing evaporation trend in Asia is mentioned. On the other hand different research has also shown some interesting effects caused by El Niño. To find answers to link these different studies an analysis is done covering two different periods. First the period 1980 – 1985 has been observed and afterwards this result is compared with the period 2010 – 2015. The sheds are created by tracking the atmospheric moisture backward from North Australia to its original source for both periods.

The precipitation sheds provide answers for the increasing precipitation showing an increase in evaporation in the different source areas in the last decades. Also, an increasing precipitation trend in January is observed, which is in line with the stronger monsoons which might have been caused by the increase in sea surface temperature. Therefore from this research it can be stated that both local sea surface temperature and the atmospheric changes in Asia have a positive correlation with the increasing rainfall pattern in North Australia as was stated in earlier research. Also, it is found that El Niño is negatively correlated with the increase in precipitation.

To conclude, North Australia is indeed dependent on both the atmospheric changes above the Asian continent and the changes of the temperature of the surrounding water bodies. This dependency seems definitely important, since research shows a correlation between rising sea surface temperature and precipitation in Australia of almost one. Also the increase in evaporation in Asia follows an almost linear trend which is in line with the increasing precipitation in North Australia. Therefore it can be stated that following this research it is believed that this might be the cause of the increasing rainfall trend above Northern Australia.

# Contents

# 1 Introduction

*MATLAB to Python*

## 1.1. Models

In the study of Civil Engineering at the Delft University of Technology, lots of models are used by students. For example, Monte Carlo algorithms are applied in designing beams relative to strength. Another example is the use of wave models, enabling students into calculating significant wave heights and wavelengths. These models can be published in different languages, but the two most commonly used ones at Civil Engineering are MATLAB and Python. Both software packages have their advantages and disadvantages but nevertheless, students are urged to choose Python over MATLAB. This preference for Python is particularly evident in the second year of the Bachelor of Science in Civil Engineering. During the programming courses at Bouwplaats, Python is being taught to students as the go-to programming language.

## 1.2. Water Accounting Model-2layers

A very interesting hydrological model is the Water Accounting Model-2layers (in short WAM2-model) created by Ruud van der Ent [R.J. van der Ent, 2014]. This offline Eulerian atmospheric moisture tracking model describes the moisture fluxes through the atmosphere, ultimately describing the phenomenon known as moisture recycling. Moisture recycling consists of two parts, namely evaporation and precipitation recycling. The WAM2-model can track both evaporation and precipitation in time. The only difference between these two recycling parameters are time and space. For forward tracking, precipitation is shown which evaporates from predefined regions. For backward tracking, evaporation is shown which precipitates in predefined regions.

The output of the model is very important for the hydrology. An interesting topic to be considered is the link between moisture recycling and land use. Since land use changes the characteristics of the land, like for example the vegetation and evaporative capacity, there is a fundamental link between these two. Savenije [H.H.G. Savenije, 1996] has linked the recycling ratio to the run-off, showing the importance of this phenomenon for understanding the behaviour of the water cycle and water resources management. Finally, the model can also be used to show a pattern in rainfall and therefore be able to predict this [T. Cömert et al., 2016]. This could prove useful for efficient irrigation, fighting drought etc.

## 1.3. Problem analysis

The WAM2-model by van der Ent works very fast and shows results in which lots of researchers are interested. These researchers would also like to tweak and debug in the code, thus understanding the model and potentially improving it. However, the WAM2-model is only available in MATLAB which is problematic for researchers in- and outside universities. The issue with MATLAB is the high license price required to make use of MATLAB. Lots of universities are renunciating or even switching from MATLAB to other languages due to this high cost. Models made by researchers in MATLAB at universities aren't used a lot outside of universities. This is for the reason that independent or corporate researchers make use of free programming languages.

Another danger occurring with the aforementioned trend is the fact that lots of models like the WAM2-model will not be available any more for students or researchers at universities.

## 1.4. Aim of thesis

The aim of this thesis is to find a solution for the WAM2-model not being available for people without MATLAB and/or budget to purchase the MATLAB license. The problem stated above will be solved by converting the WAM2-model from MATLAB to Python. This is done since the Python programming language is free and therefore available for everyone with a computer or laptop.

## 1.5. Research question

The main research question in this thesis is:

**"How can the WAM2-model be converted and improved from MATLAB to Python?"**

In section 2.1, a theoretical framework will be made for converting models from MATLAB to Python. Why and how should the model be converted to Python is a reoccurring question. Section 2.2 will show the method used in converting the WAM2-model. These are some guidelines generally used by translators. In section 2.3, the results and discussion will be available. Ultimately, the model will be converted, however, some inconveniences are mentioned in the discussion and how these are dealt with. Section 2.4 will provide a conclusion.

# 2

# MATLAB to Python

## 2.1. Theoretical Framework

There are quite some people for and against MATLAB. The same holds for Python. This comes down to the fact that not one programming language is ideal for everyone and people tend to use the language that suits them the best. Nevertheless, some programming languages can prove to be more sustainable over time. As previously mentioned, in the last few years more people are making the transition from MATLAB to Python. There are even compilers written of which OMPC might be the most well-known.

OMPC generates Python compatible syntax and changes MATLAB files to Python files [P. Jurica et al., 2009]. Nonetheless, these compilers tend to fail for more difficult code. Therefore it would be best to learn the program and manually convert MATLAB code to Python. The questions however are: why and how should this be done?

Python is a better option relative to MATLAB when looked upon the benefits of Python against MATLAB. An easy one is the fact that MATLAB is a very expensive application for which licenses are needed whereas Python is a free, open-source program. Additionally, MATLAB tends to make changes in the environment of newer version which leads to regular maintenance of your code [J.C. Bretz, 2015], whereas in Python this is not the case. However, there have been some problems with the new Python 3, but these problems seem minor in comparison with the changes MATLAB makes in different versions. Also, since MATLAB is not open-source, difficulties occur when a code is used by multiple users on different platforms [J.C. Bretz, 2015].

Python being open-source also adds additional benefits. The code can be easily shared online, enhancing wider visibility and usage [P. Jurica et al., 2009]. This has also led to the fact that Python is not used for scientific computing only. Websites, like YouTube, and even the gaming and movie industries use Python for several purposes [J. Hoekstra, 2012].

The last statement answering the why question is the fact that Python is also multi-platform. This makes it even easier to share Python code since Mac OS or Linux users are also able to use Python without needing difficult additional software.

Continuing with the second question, how: this should be done with focus on the similarities between the two languages. Being an experienced MATLAB user, one should not face many difficulties using Python since its structure and syntax is nearly the same as in MATLAB [J.C. Chaves et al., 2006].

Secondly, Python is really designed to be well understood by new users. It has a very clear syntax and when errors occur these are easily filtered. Also the core of Python only exists of a few necessary commands for beginners [H. Fagohr, 2004]. Extending this core with packages like Numpy and Scipy [M. Scott, 2014], leads to a MATLAB-like environment which is easy to use, especially for users transitioning from MATLAB to Python.

Finally, a lot of very clearly written manuals are available introducing Python and sharing beginner guides. There are even websites comparing MATLAB and Python syntax, which makes conversion even easier.

## 2.2. Method

As mentioned in section 2.1, manual conversion is preferred to prevent the occurrence of bugs in the model, which could lead to a different outcome. Also manually converting the model leads to a better understanding of its principles, which is very beneficial when the model will be used in the second part of this thesis.

The model rewritten is the WAM2-model as originally created by Ruud van der Ent. Since the method of rewriting the code is explained here, a short explanation of the model itself is given as well. The model uses reanalysis data taken from ERA-Interim [Berrisford et al., 2009; Dee et al., 2011] as input. The most important data used are the precipitation and evaporation data in 3 hourly intervals. Other data used is specific humidity, the surface pressure, the pressure levels at different levels and the meridional and zonal wind speeds and directions. This data is needed to be able to calculate the moisture fluxes and the total precipitable water in the atmosphere. Using this data, the model calculates its final output in several steps finally leading to a plot of the precipitation and/or evaporation recycling ratio. These plots show the different source and sink regions which are largely dependent on moisture recycling from its precipitation and evaporation and areas which are not. This global overview gives a very clear view how moisture is transported around the world and in which way this moisture recycling phenomenon is important. For more detail on the model's limitations, assumptions and outcome, the papers by van der Ent et al. [2010] and van der Ent and Savenije [2011] should be read through.

The model consists of 23 different scripts which are all linked to each other and lead to the outcome as explained above. When converting the model into Python, the layout of the scripts is kept in the same manner as the original model. The different scripts, including the names and the way in which they are linked to each other, are given in the flowchart, which can be found in Appendix A.

The model is converted script by script to make the conversion process as clear as possible and to avoid any miscalculations. Also, by keeping in mind that the model should be available for everyone, this step by step conversion makes it easy to explain what is done during rewriting of the model. This whole process is described in a readme file which is included with the model (online zip) and in Appendix B.

The difficulty in translating a model from one coding language into another is the fact that some operations are completely different. In most cases, for example calculation syntax, this doesn't form a barrier and therefore a lot of code can be easily rewritten. However, for some pieces of code a whole new type of code needs to be written. These difficulties are all included in the readme file. The big fundamental difference between MATLAB and Python is the fact that MATLAB uses 1 based indexing whereas Python uses 0 based indexing.

Another thing to remember while converting is making the model more user-friendly. The original WAM2-model in MATLAB is rather complex and without the help of others, it's impossible to run it. To avoid this problem, more definitions should be given and a manual should be made. This is solved by including the readme file.

## 2.3. Results & Discussion

The WAM2-model has been successfully converted from MATLAB to Python. The total size of all Python files together amounts to 0.5 MB whereas in MATLAB, this is equal to 0.2 MB. This difference is of course neglectable since modern computers have storage spaces well above 100 GB.

Some parts of the MATLAB code were not possible in Python. For example, when the model contains arrays with dimensions 96 x 92 x 240, MATLAB can easily compare the values at the first axis while a statement is given.

```
Fa_N_down_stable = min( Fa_N_down_abs, (Fa_N_down_abs
./ (Fa_E_down_abs + Fa_N_down_abs)) .* stab .* W_down(1:end-1,:,:) );
```

In MATLAB, the minimum is evaluated for above expressions in the first axis, thus returning an array of the same size (in this case 96 x 92 x 240). In Python, this is not possible since Python can't evaluate values in seperate axes. One way to solve this problem would be looping through every axis, which means you would get a nested loop in three dimensions. As can be predicted, runtime for this code was very long so another solution was needed.

With some manipulation and smart tricks, the following syntax is created in Python to get the same results as in the MATLAB syntax:

```
Fa_N_down_stable = np.reshape(np.minimum(np.reshape(Fa_N_down_abs, (np.size(Fa_N_down_abs))),
(np.reshape(Fa_N_down_abs,
(np.size(Fa_N_down_abs))) / (np.reshape(Fa_E_down_abs, (np.size(Fa_E_down_abs))) +
np.reshape(Fa_N_down_abs, (np.size(Fa_N_down_abs)))) * stab * np.reshape(W_down[:-1,:,:],
(np.size(W_down[:-1,:,:]))),(np.int(count_time*np.float(divt)),len(latitude),len(longitude)))
```

The above syntax reshapes the input arrays from their original shape to an one-dimensional array. Python can indeed evaluate values in one-dimensional arrays, so after the evaluation has ended, the array is reshaped back into its original shape. With this method, the model improved with respect to the nested loop by almost halving the runtime. Other examples can be found in the readme, Appendix B.

Something else Python will not do on its own is saving files compressed. For example, when running the Fluxes_and_States_Masterscript in MATLAB, the output file for 1 day will be around 50 MB whereas in Python, the outputfile will be around 150 MB. This can be solved in Python by stating decompression equal to True but the runtime almost quadruples, giving Python a big disadvantage in runtime.

To benchmark the model performance, the runtime for several masterscripts in both MATLAB as well in Python is shown in the following table. The first 5 days of both 2007 and 2008 are used in the scripts:

| Script | MATLAB (s) | Python (s) |
|---|---|---|
| Fluxes_and_States_Masterscript | 115.0 | 227.14 |
| Con_P_Recyc_Masterscript | 83.3 | 148.9 |
| Con_P_Recyc_Masterscript_time | 99.9 | 273.0 |
| Con_P_Output | 72.2 | 11.5 |
| Hor_Fluxes_Output | 40.4 | 9.8 |

Table 2.1: Runtime for scripts in MATLAB and Python

As can be seen in the table above, the first three masterscripts are more than 100% faster in MATLAB than Python. However, the last two scripts are much faster in Python than in MATLAB.

Due to these slow runtimes for the masterscripts, a new version of the Python model has been developed. In this version of the WAM2-model, the method as mentioned in 2.2 is not followed. Rather than having a MATLAB structure, in this version the coding is much more fixated for Python syntax. The runtimes for this new version can be seen in the following table:

| Script | MATLAB (s) | Python 1.0 (s) | Python 2.0 (s) |
|---|---|---|---|
| Fluxes_and_States_Masterscript | 115.0 | 227.14 | 71.5 |
| Con_P_Recyc_Masterscript | 83.3 | 148.9 | 51.8 |
| Con_P_Recyc_Masterscript_time | 99.9 | 273.0 | 80.0 |
| Con_P_Output | 72.2 | 11.5 | 11.2 |
| Hor_Fluxes_Output | 40.4 | 9.8 | 8.0 |

Table 2.2: Runtime for scripts in MATLAB and Python

As can be seen in the table above, the new Python version is much faster than MATLAB and the older version. This is explainable: in the old Python model, scripts were called from the current directory into the respective masterscript. However, Pythons power lays in the fact that functions can be defined inside on masterscript. This leads to a speed improvements of 300% relative to the old Python model and 62% relative to MATLAB.

Next to the speed improvement, the WAM2-model is much more user-friendly in Python than MATLAB. For example, when the fluxes and storages need to be calculated, ERA-Interim data is used. To use this data in coding, the data path needs to be specified. In the MATLAB version, these data paths need to be changed in every script used by the masterscript. However, in the old and new Python version, data paths are defined in the beginning, costing less time for the user to change all these paths.

Another example of user-friendliness is Python automatically creating empty arrays for Sa_track and Sa_time. For calculation of Sa_track and Sa_time, the first day defined needs to make use of data on the previous day. However, when data is not present, an empty array for both Sa_track and Sa_time is needed. In the past, the user had to make his/her own empty arrays and save them in the directory needed. But now, the option is giving in the beginning and users don't need to perform such actions.

Since Python allows its editors to make use of headers, this also makes the scripts of the WAM2-model very clear. To keep some similarities with the original WAM2-model in MATLAB, the WAM2-model in Python is published in two versions. Version 1.0 has the same structure as MATLAB and is very handy in understanding the Python syntax. Version 2.0 however is the faster version and it is definitely recommended to use this version instead of 1.0 for actual calculations.

**Datapaths (FILL THIS IN)**

```
constants = 'E:/WAM2_Data/invariants.nc'#constants
def data_path_ea(years,yearpart):
    save_empty_arrays_ly_track = 'E:/WAM2/interdata/2layers/sigma_levels/continental/' + str(years[0]-1) + '-' + str(364+isleap(years[0]-1)) + 'Sa_track.mat'
    save_empty_arrays_ly_time = 'E:/WAM2/interdata/2layers/sigma_levels/continental/' + str(years[0]-1) + '-' + str(364+isleap(years[0]-1)) + 'Sa_time.mat'

    save_empty_arrays_track = 'E:/WAM2/interdata/2layers/sigma_levels/continental/' + str(years[0]) + '-' + str(yearpart[0]-1) + 'Sa_track.mat'
    save_empty_arrays_time = 'E:/WAM2/interdata/2layers/sigma_levels/continental/' + str(years[0]) + '-' + str(yearpart[0]-1) + 'Sa_time.mat'


    return save_empty_arrays_ly_track,save_empty_arrays_ly_time,save_empty_arrays_track,save_empty_arrays_time

def data_path(previous_data_to_load,yearnumber,a):
    load_Sa_track = 'E:/WAM2/interdata/2layers/sigma_levels/continental/' + previous_data_to_load + 'Sa_track.mat'

    load_fluxes_and_storages = 'E:/WAM2/interdata/2layers/sigma_levels/' + str(yearnumber) + '-' + str(a) + 'fluxes_storages.mat'

    load_Sa_time = 'E:/WAM2/interdata/2layers/sigma_levels/continental/' + previous_data_to_load + 'Sa_time.mat'

    save_path_track = 'E:/WAM2/interdata/2layers/sigma_levels/continental/' + str(yearnumber) + '-' + str(a) + 'Sa_track.mat'
    save_path_time = 'E:/WAM2/interdata/2layers/sigma_levels/continental/' + str(yearnumber) + '-' + str(a) + 'Sa_time.mat'
    return load_Sa_track,load_fluxes_and_storages,load_Sa_time,save_path_track,save_path_time
```

Figure 2.1: Example of data path input

```
#### create empty array for track and time

def create_empty_array(count_time,divt,latitude,longitude,yearpart,years):
    Sa_time_top = np.zeros((np.int(count_time*divt)+1,len(latitude),len(longitude)))
    Sa_time_down = np.zeros((np.int(count_time*divt)+1,len(latitude),len(longitude)))
    Sa_track_top = np.zeros((np.int(count_time*divt)+1,len(latitude),len(longitude)))
    Sa_track_down = np.zeros((np.int(count_time*divt)+1,len(latitude),len(longitude)))
    if yearpart[0] == 0:
        sio.savemat(datapathea[0],{'Sa_track_top':Sa_track_top,'Sa_track_down':Sa_track_down,},do_compression=True)
        sio.savemat(datapathea[1],{'Sa_time_top':Sa_time_top,'Sa_time_down':Sa_time_down},do_compression=True)
    else:
        sio.savemat(datapathea[2], {'Sa_track_top':Sa_track_top,'Sa_track_down':Sa_track_down},do_compression=True)
        sio.savemat(datapathea[3], {'Sa_time_top':Sa_time_top,'Sa_time_down':Sa_time_down},do_compression=True)

    return
```

Figure 2.2: Example of empty arrays input

Ultimately, MATLAB will create plots of the evaporation and precipitation recycling ratio's. This is also done in Python, but 1 more variant is added called the contour plot. The contour plot is much smoother and easy on the eyes. The plot does lose some information relative to the meshgrid plot, however, the meshgrid plot is also added. For comparison, look at the three figures below. These are plots for the precipitation recycling ratio $\rho_c$ for the years 2001 and 2002.

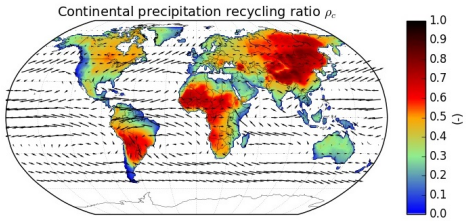


Figure 2.3: Python contour plot

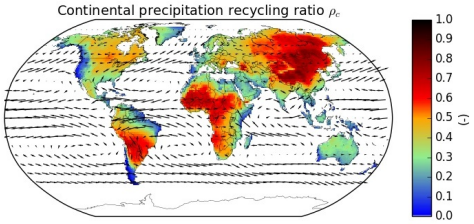


Figure 2.4: Python meshgrid plot

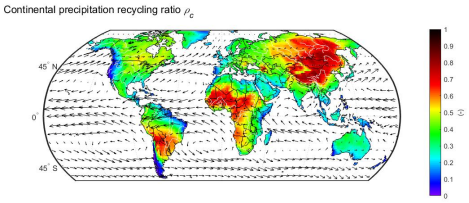


Figure 2.5: MATLAB plot

## 2.4. Conclusion

As can be seen in the previous section, the WAM2-model has been successfully converted from MATLAB to Python. There are even two versions of the model now, one which has the same structure and one making use of Pythons more advantageous data structure. This resulted in Python version 1.0 and version 2.0 models, with the Python 2.0 model being faster than the model in MATLAB. Next to the speed improvement, Python is also much more user-friendly. There are headers between the codes and data paths make life much easer for the users. Also, the model in Python removes some needed things in order for the model to work (for example empty arrays for Sa_track and Sa_time). The simple conclusion is that converting models from MATLAB to Python is very much possible. However, a neat approach is needed since debugging takes a lot of time and big models like WAM2 can be very confusing at times. The best way to handle this is making a readme, in which every made step is clarified. This is useful for the translators as well as the users later on. Furthermore, improvements are definitely possible and like the WAM2-model in Python, it is much improved relative to the MATLAB version.

# 3

# Use of precipitation sheds to explain the increasing rainfall trend in Northern Australia

## 3.1. Introduction

In the past years different researches have been done regarding the origin of precipitation. An important aspect in defining the origin of the precipitation for different regions are the spatial boundaries as defined by Dirmeyer [Dirmeyer and Brubaker, 2007; Dirmeyer et al., 2009a,b]. Additionally, previous research [van der Ent et al., 2010] has shown that this precipitation originates not only at sea but also on land. This terrestrial evaporation and the atmospheric transport of this evaporated moisture is known as moisture recycling and seems to tribute to a large amount of the total precipitation for some areas globally. This knowledge is used to create precipitation sheds. A precipitation shed is defined as an atmospheric watershed of a region regarding the origin of its rainfall. This origin can be both oceanic as terrestrial. In this research focus is on both the oceanic and terrestrial part since the increase in sea surface temperature in the water bodies surrounding Australia might be of influence to the precipitation as well [Ma and Xie, 2012; Van der Ent and Savenije, 2013]. Since the sheds do not have fixed boundaries different periods might give different results probably referring to changes in the atmosphere. Therefore, using precipitation sheds changes can provide important answers to the research questions. So using a backtracking of moisture, this research will focus on the origin of the precipitation in Northern Australia using the WAM2-model in Python as was described in the earlier sections. The most important questions for this research arise from earlier research due to the fact that the conclusions made in these studies are yet not convincing as will be explained in the next section. So the research questions are whether or not the precipitation in Australia is dependent on the atmospheric changes above Asia, the Indian Ocean and the Coral Sea and if so, if this is the cause of the increasing rainfall trend above Northern Australia.

## 3.2. Theoretical Background

Previous research [e.g. Ge Shi et al., 2007; Rotstayn et al., 2007] has shown an increasing rainfall pattern over Northern Australia. This increase seems to mainly occur during the Australian summer season, especially during the monsoon period. This increasing rainfall is shown in the picture below from the Australian Bureau of Meteorology. In the picture the trend in total rainfall is shown from the 1960's until now. The colors in the map indicate an increase or decrease in precipitation of millimeters per decade. For parts of Northern Australia this increase is as high as 50 mm precipitation per decade.

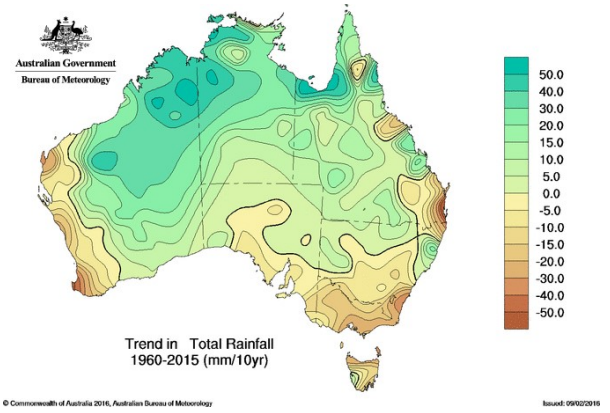


Figure 3.1: Trend in total rainfall Australia 1960-2015

Other studies have stated several causes for this phenomenon, all giving a somewhat similar conclusion. As assumed earlier, research by Wardle and Smith [2004] confirms the statement that the increase of precipitation is indeed caused by stronger monsoons. These monsoons strengthen due to the fact that the sea surface temperature and the surface temperature of the continent change due to climate change, increasing the land-sea temperature contrast. This increase results in a lower surface pressure. Bin Wang et al. [2003] suggest the same cause. However, Wang et al. describe a stronger East Asian winter monsoon as the origin of the stronger Australian monsoon. As such, this stronger East Asian monsoon could indeed lead to the already stated increase in the land-sea temperature contrast. In line with this view of stronger East Asian monsoon is the research done by Rotstayn et al. [2007] which refers to the strong impact of aerosols above the Asian continent. The aerosols lead to an Asian haze which changes the temperature and pressure gradients in the Indian Ocean. This change increases the monsoon wind pattern towards Australia. Furthermore there are several contradicting studies referring to 'the El Nino effect' in Australia [e.g. Wenju Cai et al., 2014] leading to more droughts in parts of the country. Due to these contradicting studies and not fully convincing conclusion this study focuses on solving these unsolved questions.

## 3.3. Method

For initiating the tracking of moisture and creating a precipitation shed a sink region in which the precipitation precipitates should be defined. For this research only Northern Australia is of interest, therefore a land-sea mask of Northern Australia is created bounded by 10.5° – 22.5° south and 103.5° - 163.5° east only taking the land mass into account (0 for sea, 1 for land). Using the van der Ent [2010] method a backward tracking of moisture is initiated using the Python WAM2-model. For a detailed description of the WAM2 model the readme of the conversion into Python should be read or for the original MATLAB model, refer to van der Ent et al. [2010, 2011, 2013, 2014].

The analysis using a precipitation shed requires a boundary as just referred to above. This boundary is based on evaporation contribution. With a precipitation shed the fraction of the total evaporation contribution to a certain region is found, North Australia in this case.

The model uses reanalysis data taken from ERA-Interim [Berrisford et al., 2009, Dee et al. 2011] as input. For a detailed description of how to use ERA-Interim data, refer to the readme of the WAM2 Python model in which the use of ERA- Interim data is explained step by step.

Since we want to make a shed of the origin of the rainfall in Northern Australia also the ocean should be included therefore the total precipitation in the region will be compared to the total evaporation contributing to this region (not only from land).

All the data is used on a 1.5° x 1.5° grid. The analysis is covered for two separate periods for which a precipitation shed will be displayed. The first period investigated is the period stretching from 1980-1985. The second period is the period from 2010-2015 which is used to investigate if any changes to the precipitation shed have occurred in the 30 year period. If the comparison between these two periods indeed show significant difference this could be (one of) the reason(s) for the increase of precipitation in the last decades in Northern Australia. The reason for choosing 1980-1985 and not an earlier period is due to the fact that ERA-Interim data is only available from 1979 onwards.

As stated earlier the backward tracking method is used which holds that the precipitation in the sink

region will be followed back to its original source. This is done by calculating the total amount of evaporation for each grid cell, which eventually travels through the atmosphere and precipitates in the sink region.

Important to notice is the fact that the original model plots the data on a grid plot, showing the data on the 1.5°x 1.5° gridcell plot. In Python this is an option as well. However, Python also has the option to show the data on a contour plot, which gives a much prettier and clearer overview of the data. Therefore the results will be shown as contourplots. The difference between the plot in MATLAB, the same style plot in Python and the contourplot in Python are shown below as example.

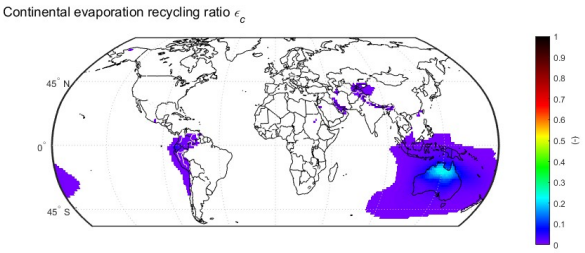


Figure 3.2: Map of evaporation recycling in MATLAB

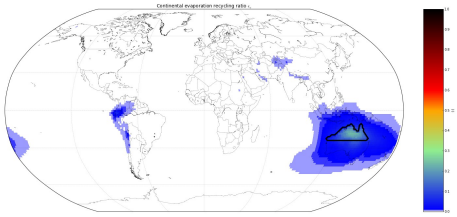

Figure 3.3: Map of evaporation recycling in Python, grid cell

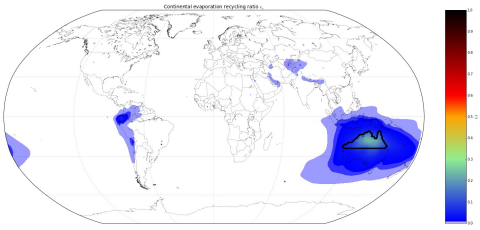

Figure 3.4: Map of evaporation recycling in Python, contour

## 3.4. Results & Discussion

The tracking of moisture shows some interesting results for North Australia in the precipitation sheds which will be presented in the following section, beginning with plots of the precipitation sheds for the different periods. The first plots shown below are plots of the evaporation recycling ratio for the period 1985 – 1980 and 2015 – 2010 and plots of the evaporation recycling ratio for January 1985 and 2015.

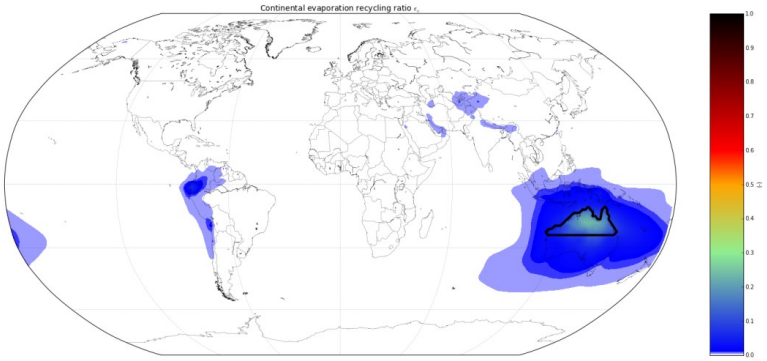

Figure 3.5: Australia evaporation recycling ratio 1980 – 1985

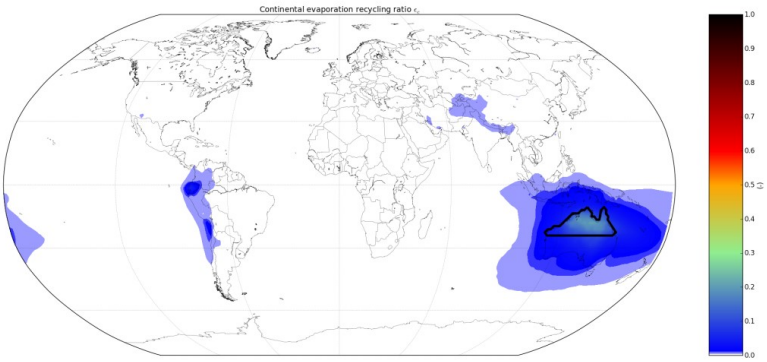

Figure 3.6: Australia evaporation recycling ratio 2010 – 2015

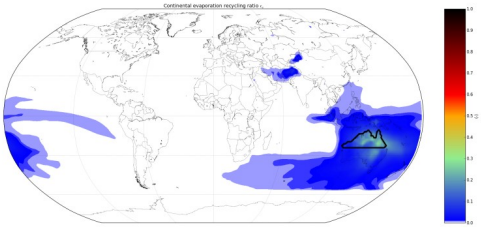

Figure 3.7: Australia January evaporation recycling ratio 1985

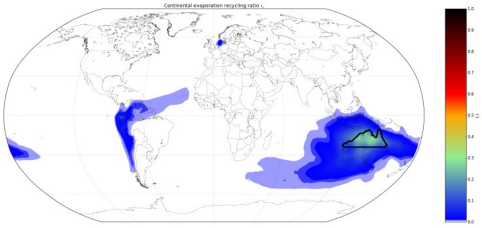

Figure 3.8: Australia January evaporation recycling ratio 2015

Comparing these plots, some interesting conclusions can be drawn. First of all it is very surprising to observe a lack in difference between the two periods as shown in figure 3. However the plots for January 1985 and 2015 in figure 4 do show some interesting differences. Assuming the contribution of the light blue areas negligible, it can be observed that the contribution of North Australia itself is a lot higher as well as a interesting contribution from South America. January is chosen as an example here because it is seen as a key month for precipitation in Australia, since it is in the middle of the Southern Hemisphere summer and the Australian monsoon period.

Additionally, it is evident that there is a substantial increase in evaporation from the source area in the last three decades as is shown in the graphs below. Also, a pattern in the rainfall in Northern Australia can be observed as well for this period, however only for part of the summer and monsoon months.

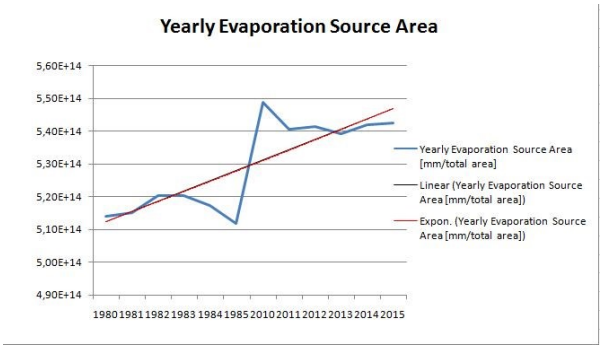


Figure 3.9: Increase in yearly evaporation in the source area

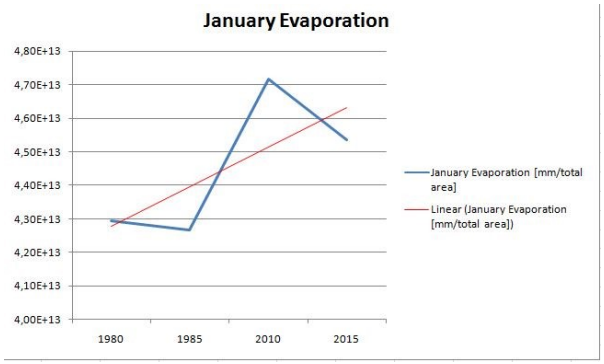


Figure 3.10: Increase in evaporation for January

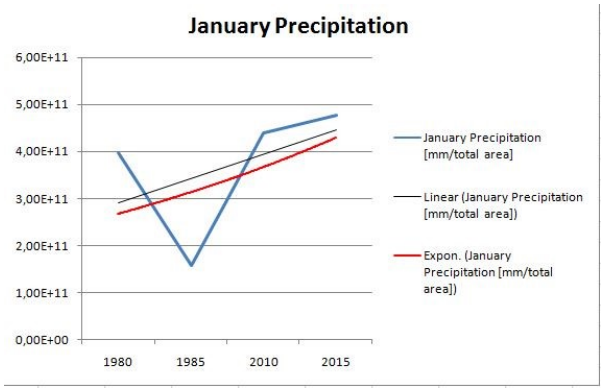


Figure 3.11: Increase in precipitation for January

However, the total yearly precipitation seems to remain almost constant for the region. The absence of evaporation data from the ocean towards the Australian continent could have caused this absence of clear increase in yearly rainfall. Nevertheless the increase in evaporation from the source areas and the clear increase in rainfall in the monsoon period as shown above lead to a conclusion that this may have caused the increasing precipitation in Northern Australia. Supporting this statement is the fact that this increasing evaporation also precipitates in the ocean and other source areas which could then again precipitate (multiple recycling). Also the increase in the SST over the investigated period supports this increase in evaporation. The rise in sea surface temperature is shown below in detail:

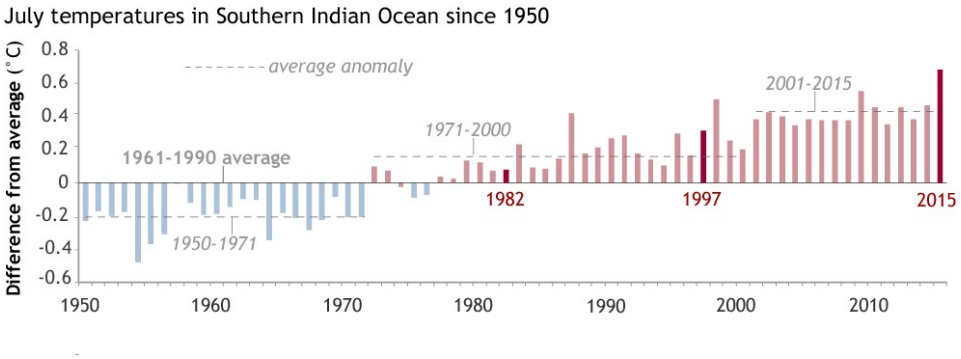


Figure 3.12: Trend in SST for the Southern Indian Ocean, July 1950 - 2015

A legitimate justification can be made regarding this research compared with previous research as explained in the earlier sections. The justification is in fact twofold. On the one hand the claim of increasing aerosols and temperature in Asia is indeed correct and leads to the found increase in evaporation in these regions. Second, the rise in sea surface temperature also gives rise to an increase in evaporation and therefore

increasing precipitation in Northern Australia. In previous research [van der Ent and Savenije, 2013] a positive correlation between the precipitation in Australia and the sea surface temperature for both the Indian Ocean and the Coral Sea is found.

Additionally van der Ent's study also describes a negative correlation in precipitation and sea surface temperature for the same regions during El Niño leading to drought periods. Research by the Australian bureau of meteorology (2014) agrees with this statement on the El Nino phenomenon. The bureau of meteorology describes a decrease in rainfall for Australia, especially in the eastern regions. However, based on this research and the previous researches described, it is thought that the twofold cause of the increase in rainfall as described above will overrule the El Nino effect for the Northern part of Australia which has led to an increase in rainfall in the last decades and will lead to even more precipitation in the future.

## 3.5. Conclusion

In an earlier study on precipitation sheds by Keys et al. [2014] was found that a precipitation shed seems to be very stable throughout time. Also Keys et al. found that a core precipitation shed exists. For North Australia it can be concluded that these statements are indeed true. The shed of 1980-1985 and 2010-2015 show a similar core precipitation shed. The change in this shed seems to be very small over this thirty year span. Most of the variance inside the shed is explained by an increase in evaporation in existing source areas.

It can be concluded that both local sea surface temperature and the atmospheric changes in Asia have a positive correlation with the increasing rainfall pattern in North Australia. Also, it is found that El Niño is negatively correlated with the increase in precipitation.

Referring back to the research questions it is found that the precipitation in Australia is indeed dependent on the atmospheric changes above Asia and the changes in the surrounding water bodies. This dependency seems even that important that the correlation between rising SST and precipitation is almost one. Also the increase in evaporation in Asia follows an almost linear trend which is in line with the increasing precipitation in North Australia. Therefore it can be stated that following this research it is believed that this might be the cause of the increasing rainfall trend above Northern Australia.

Regarding future change in this pattern the strength of El Niño and the changes in magnitude should be studied in further research. Also the change in SST should be monitored and Asian environmental measures should be followed closely.

# References


(1) Australian bureau of meteorology: What is El Niño and what might it mean for Australia?, http://www.bom.gov.au/climate/updates/articles/a008-el-nino-and-australia.shtml, 2014.

(2) Berrisford, P. et al.: Atmospheric conservation properties in ERA-Interim, 2009.

(3) Bretz, J.C., Can We Migrate Our Analysis Routines To Python?, 412 TW-PA-15520, 2015.

(4) Cai, W. et al.: Increasing frequency of extreme El Niño events due to greenhouse warming, 2014.

(5) Chaves, J.C. et al., Octave and Python: High-Level Scripting Languages Productivity and Performance Evaluation, IEEE Computer Society, 2006.

(6) Cömert, T., van de Koppel, N.T.D., The importance of moisture recycling, http://repository.tudelft.nl/view/ir/uuid%3Afbd08671-4551-40ec-9837-cf5dfc99f4d8/, 2016.

(7) Dee, D.P. et al.: The ERA-Interim reanalysis: configuration and performance of the data assimilation system, 2011.

(8) Dirmeyer, P.A. et al.: Import and export of atmospheric water vapor between nations, Journal of Hydrology, 365, 11 – 22, 2009a.

(9) Dirmeyer P.A. et al.: Precipitation, recycling, and land memory: An integrated analysis, Journal of Hydrometeorology, 10, 278 – 288, 2009b.

(10) Dirmeyer, P. A. and Brubaker, K. L.: Characterization of the global hydrologic cycle from a back-trajectory analysis of atmospheric water vapor, Journal of Hydrometeorology, 8, 20–37, 2007.

(11) Ent, R.J. van der et al.: Origin and fate of atmospheric moisture over continents, 2010.

(12) Ent, R.J. van der et al.: Should we use a simple or complex model for moisture recycling and atmospheric moisture tracking?, 2013.

(13) Ent, R.J.van der, A new view on the hydrological cycle over continents, Ph.D. thesis, 2014.

(14) Ent, R.J. van der and Savenije, H.H.G.: Length and time scales of atmospheric moisture recycling, 2011.

(15) Ent, R.J. van der and Savenije, H.H.G.: Oceanic sources of continental precipitation and the correlation with sea surface temperature, 2013.

(16) Fagohr, H., A comparison of C, MATLAB, and Python as teaching languages in engineering, Conference Paper, University of Southampton, 2014.

(17) Hoekstra, J., GOODBYE MATLAB! HELLO PYTHON!, changing the programming language in our curriculum, Leonardo Times, 2012.

(18) Jurica, P., Leeuwen, C. van, OMPC: an open-source MATLAB-to-Python compiler, Frontiers in Neuroinformatics, Volume 3 No. 5, 2009.

(19) Keys, P.W. et al.: Variability of moisture recycling using a precipitationshed framework, 2014.

(20) Rotstayn, L.D. et al: Have Australian rainfall and cloudiness increased due to the remote effects of Asian anthropogenic aerosols?, journal of geophysical research, vol. 112, 2007.

(21) Savenije, H.H.G., the runoff coefficient as the key to moisture recycling, Journal of hydrology, 03/1996.

(22) Scott, M., An introduction to Numpy and Scipy, http://www.engr.ucsb.edu/~shell/che210d/numpy.pdf, referenced on 08-05-2016, 2014.

(23) Shi, G. et al.: Variability and Trend of North West Australia Rainfall: Observations and Coupled Climate Modeling, journal of climate, vol. 21, 2007.

(24) Wang, B. et al.: Contrasting the Indian and East Asian monsoons: implications on geologic timescales, 2003.

(25) Wardle R. and Smith I.: Modeled response of the Australian monsoon to changes in land surface temperatures, 2004.

**Figures**

(1) **Figure 3.1:** Australian Bureau of Meteorology, Total rainfall trend maps 1960 – 2015, `http://www.bom.gov.au/climate/change/index.shtml#tabs=Tracker&tracker=trend-maps`, accessed at 21-05-2016.

(2) **Figure 3.12** National Oceanic and Atmospheric Administration, July temperatures in Southern Indian Ocean since 1950, `https://www.climate.gov/news-features/blogs/enso/el-ni%C3%B1o-other-side-pacific-pond`, accessed at 08-06-2016.

# A

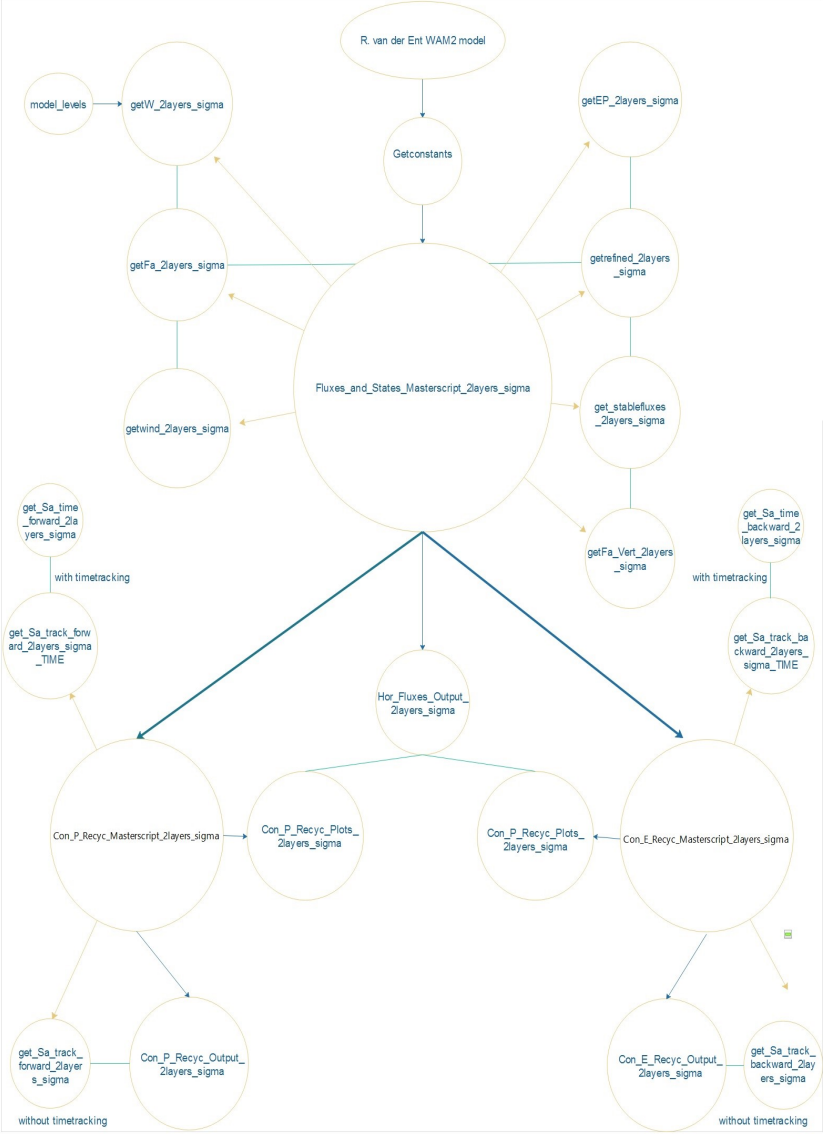


Figure A.1: Flowchart

# B

# Readme

In this appendix, 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.

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.

*Tolga Cömert*
*Niek van de Koppel*
*Delft, June 2016*

## B.1. 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** 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.

## B.2. 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 |

Table B.1: Runtime for scripts in MATLAB and Python

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).

## B.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.

### B.3.1. Fluxes_and_States_masterscript_2layers_sigma

(1) 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.

(2) The function type in MATLAB reads as def 'name' in Python.

(3) Important to notice is that Python starts with index **0**, not with **1** as MATLAB.

(4) 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’))
```

### B.3.2. Get_constants

(1) 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.

(2) For the lsm, the lake mask needs to be added. In Python, the MATLAB arrays can be recreated by stacking the arrays and taking the transpose.

(3) 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!

(4) Implement all defined constants in the code.

(5) 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**.

(6) 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]
```

### B.3.3. 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.

(1) 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
```

(2) 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.

### B.3.4. getW_2layers_sigma

(1) The first step is to import the variables from the model_levels file:

**Model levels**

(2) In Model_levels, different arrays are created (A, B, k) which are needed in other scripts.

(3) In Python, this can be done using **np.array** and using **np.vstack**, the array is vertically stacked.

**getW_2layers_sigma**

(4) The 'not equal to' sign in MATLAB is = whereas in Python this is !=, for example, when referring to not the end of the year.

(5) 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.

(6) 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.

(7) 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'**

(8) 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.

(9) The levels are flipped in the netCDF file, this can be solved by flipping the array indices with [:,:-1,:,:]

(10) 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).

(11) 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.

(12) 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.

(13) 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.

### B.3.5. getwind_2layers_sigma

(1) 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.

### B.3.6. getFa_2layers_sigma

(1) 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.

(2) 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:

| MATLAB | Python |
|---|---|
| 1:end-1 | :-1 |
| 2:end | 1: |

Table B.2: Difference syntax MATLAB and Python

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.

(3) 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.

### B.3.7. getEP_2layers_sigma

(1) In MATLAB, for x = 1:4:count_time*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) 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.

(3) 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**.

### B.3.8. getrefined_2layers_sigma

(1) In defining the oddvector and partvector, **np.zeros** is used. This is defined as oddvector2 = np.zeros((1,np.int(count_tir

(2) **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.

(3) 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.

(4) 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.

(5) 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.

### B.3.9. get_stablefluxes_2layers_sigma

(1) To convert Fa_N_top and Fa_N_down to m3, 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.

(2) 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.

(3) 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.

(4) 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))

(5) 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.

### B.3.10. getFa_Vert_2layers_sigma

(1) 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.

(2) 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 /**.

(3) 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.

(4) 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.

### B.3.11. Con_P_Recyc_Masterscript_2layers_sigma

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.**

(1) 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.

(2) 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 Kvf.

(3) 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).

(4) Kvf 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 Kvf shouldn't be larger than 3.

(5) **In this file a loop is created which runs through all the years, whereas in MATLAB you can just state yearnumber = years.**

(6) 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.

(7) 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))**

(8) 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']**

(9) 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!

(10) Below the tracking and timing files shall be explained shortly since these are needed for this master-script and other scripts later on.

### B.3.12. get_Sa_track_forward_2layers_sigma

(1) 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

(2) In the script you define Kvf = 0 in which case nothing should be done. The command for this in Python is pass.

(3) All other code can almost identically be converted from the MATLAB model or commands are used which were already explained earlier on.

(4) Important in this script is the amount of code which needs to be rewritten. Take caution when rewriting large pieces of code!

### B.3.13. get_Sa_track_forward_2layers_sigma_TIME

(1) 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.

(2) This file computes tracking together with time.

(3) 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

(4) This file is imported in the get_Sa_track_forward_2layers_sigma_TIME file.

(5) 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

### B.3.14. Con_P_Recyc_Output_2layers_sigma

(1) This file continues with the output gained from the fluxes and states and the Con_P masterscripts.

(2) Also in this file the getconstants script needs to be imported.

(3) 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.

(4) 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);
```

(5) 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.**

(6) 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.

(7) 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
```

(8) 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.

### B.3.15. Hor_Fluxes_Output_2layers_sigma

(1) 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.

(2) In this file, the constants are needed again so getconstants is imported.

(3) Input is needed in this file as well. The years and yearpart should be defined.

(4) 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.

(5) At the end the output is saved so it can be used in the Con_P_Recyc_Plot and Con_E_Recyc_Plot scripts.

### B.3.16. 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.

(1) 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.

(2) 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.

(3) 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 **LinearSegmenteColormap 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')])
```

(4) We use the command **%matplotlib** qt which closes any loaded/active figures when running the code again.

(5) For creating a plot of the earth we use **Basemap from mpl_toolkits.basemap**, using a **Robinson projection**.

(6) To plot the data on the correct latitude and longitude we create a meshgrid:

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

(7) We use contourf to create a filled contourplot of the data over the longitude and latitude grid. In contourf clevs is used to define the ranges:

**cs = map.contourf(x,y,rho_c_landtotal,clevs2,linewidths=1.5,latlon=True, cmap=cmap)**

(8) Colorbar is used to create a colorbar with our own inputdata:

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

(9) 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

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

### B.3.17. Additional extensions

(1) 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.

(2) 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 eachother.

(3) 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

(4) 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.

(5) 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.

## B.4. Making sheds with the model

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.

### B.4.1. 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.

(1) 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.

(2) 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

(3) **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).

(4) 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.

### B.4.2. 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.

### B.4.3. Psheds_plots / Esheds_plots

(1) 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).

(2) For the colorbar in this plot we have chosen to use the **LinearSegmentedColormap from matplotlib.colors.**

(3) 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)

(4) 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]**

(5) 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 % imput for pshed**
**Region = Continent * lsm**

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

(6) 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.