In [2]:

```
#import the required Python packages

import pandas as pd
import datetime   #check this
import pyeto
import numpy as np
import ast        #check this
import math
from ast import literal_eval  #check this
from pandas import DataFrame
from scipy.interpolate import interp1d

from pyeto import fao
from datetime import datetime
#%matplotlib inline     #check this

math.exp = np.exp
math.pow = np.power
math.sqrt = np.sqrt
```

- Make sure to import the necessary python packages and libraries

- Download pyeto source code and make sure to have it under the same directory (./pyeto), and then import it as shown

In [26]:

```
df=pd.read_csv('pilot_output2_clean.csv') #This is a sample dataset with 100 poin

#df=df.drop(df[df['harv_t']==0].index) #Deleting any point that has zero harvesti
#df=df.reset_index()    #reseting the index after deleting the zero value points
#del df['index']        #The previous step will generate a new column form the ol
```

In [27]:

```
#in the intital code, there are two more steps related to calibration and project
```

- I don’t have the input read (pilot_output2_clean.csv), however I assume it is the output of the GIS processing phase
- Tidy up a bit of the data, remove null values for now (maybe also check negative values?)

In [28]:

```
#define available water content  ##Not used in the ETO estimation
#where 0.9 rooting depth for maize and 50% maximum depletion factor
#def awc_class(row):
#    if (row['awc_class']==0):
#        return 0
#    elif (row['awc_class']==1):
#        return 150*0.9*0.5
#    elif (row['awc_class']==2):
#        return 125*0.9*0.5
#    elif (row['awc_class']==3):
#        return 100*0.9*0.5
#    elif (row['awc_class']==4):
#        return 75*0.9*0.5
#    elif (row['awc_class']==5):
#        return 50*0.9*0.5
#    elif (row['awc_class']==6):
#        return 15*0.9*0.5
#    elif (row['awc_class']==7):
#        return 0*0.9*0.5
#    else:
#        return 75*0.9*0.5

#df['awc'] = df.apply(awc_class,axis=1)
```

- Function returns available water content based on the layer's info on available water content class (awc_class)

Not included in Y's code; might consider to include it in our methodology?

In [3]:

```
#Estimating the DAILY ETo function:
#shf — Soil heat flux (G) [MJ m-2 day-1] (default is 0.0, which is reasonable for


def evap_i(lat,elev,wind,srad,tmin,tmax,tavg,month):
    if month ==1:
        J = 15
    else:
        J = 15 + (month-1)*30

    latitude = pyeto.deg2rad(lat)
    atmosphericVapourPressure = pyeto.avp_from_tmin(tmin)
    saturationVapourPressure = pyeto.svp_from_t(tavg)
    ird = pyeto.inv_rel_dist_earth_sun(J)
    solarDeclination = pyeto.sol_dec(J)
    sha = [pyeto.sunset_hour_angle(l, solarDeclination) for l in latitude]
    extraterrestrialRad = [pyeto.et_rad(x, solarDeclination,y,ird) for x, y in zi
    clearSkyRad = pyeto.cs_rad(elev,extraterrestrialRad)
    netInSolRadnet = pyeto.net_in_sol_rad(srad*0.001, albedo=0.23)
    netOutSolRadnet = pyeto.net_out_lw_rad(tmin, tmax, srad*0.001, clearSkyRad, a
    netRadiation = pyeto.net_rad(netInSolRadnet,netOutSolRadnet)
    tempKelvin = pyeto.celsius2kelvin(tavg)
    windSpeed2m = wind
    slopeSvp = pyeto.delta_svp(tavg)
    atmPressure = pyeto.atm_pressure(elev)
    psyConstant = pyeto.psy_const(atmPressure)

    return pyeto.fao56_penman_monteith(netRadiation, tempKelvin, windSpeed2m, sat
```

- Function returns daily ETo evapotranspiration (pyeto package employed - functions described in the pyeto library)

-Assumption 15th day of the month as a daily average - later on multiplied (*30) in order to get the monthly values

In [30]:

```
for i in range(1,13):
    df['ETo_{}'.format(i)]=0  ##To make sure the it is reset to zero
```

In [31]:

[30,31] creating columns, resetting and calling ETo function with the appropriate input arguments of the function

```
%%time
#calculate ETo for each row for each month
## range(1,13) and .format(i): to generate monthly calculation of ETo
## df.iterrows() and use of .iloc[index]: To make sure the calculation will be re

for i in range(1,13):
    df['ETo_{}'.format(i)] = evap_i(df['lat'],df['elevation'],df['wind_{}'.format
```

```
CPU times: user 1.42 s, sys: 177 ms, total: 1.6 s
Wall time: 1.46 s
```

In [32]:

```
#Will save the ETO to save time and avoid computing it everytime
#Create a Pandas Excel writer using XlsxWriter as the engine.
writer = pd.ExcelWriter('Pilot20190124_ETO.xlsx', engine='xlsxwriter')

# Convert the dataframe to an XlsxWriter Excel object.
df.to_excel(writer, sheet_name='ETO_all')


# Close the Pandas Excel writer and output the Excel file.
writer.save()
```

In [33]:

```
#Effective rainfall function

def eff_rainfall(prec,eto):
    return (1.253*((prec**0.824)-2.935))*10**(0.001*eto)  #Find the source
```

In [34]:

```
%%time
#calculate eff rainfall for each row for each month
#This source: http://www.fao.org/docrep/S2022E/s2022e08.htm was initially used bu

for i in range(1,13):
    df['eff_{}'.format(i)]=0

for i in range(1,13):
    df.loc[df['prec_{}'.format(i)] < 12.5, 'eff_{}'.format(i)] = df['prec_{}'.for
    df.loc[df['prec_{}'.format(i)] >= 12.5, 'eff_{}'.format(i)] = eff_rainfall(df
```

[33,34] Defining and calling effective rainfall function

-Update source (USDA,USDS?)

```
CPU times: user 631 ms, sys: 396 ms, total: 1.03 s
Wall time: 519 ms
```

In [35]:

```
#Will save the ETO to save time and avoid computing it everytime
#Create a Pandas Excel writer using XlsxWriter as the engine.
writer = pd.ExcelWriter('Pilot20190124_ETO_RF.xlsx', engine='xlsxwriter')

# Convert the dataframe to an XlsxWriter Excel object.
df.to_excel(writer, sheet_name='RF_all')


# Close the Pandas Excel writer and output the Excel file.
writer.save()
```

In [36]:

```
#df=pd.read_excel('Results20180912_ETO_RF.xlsx')
```

In [37]:

```
#for the NWSAS we will assume all the region a unimodal area which means it has c
df['Mode']=('unimodal')
```

In [38]:

```
#calculate kc based on the growing stage (month - planting, growing, harvesting s
import math
import dateutil     #dateutil module provides powerful extensions to the standarc
from dateutil import parser  #This module offers reads the given date in string a

#introduce the kc function and its attributes

#def kc(plantation,Li,Ld,Lm,Le,kci,kcd,kcm,kce,isodate):  #initial code

def kc(plantation,Li1,Li2,Ld,Lm,Le,kci1,kci2,kcd,kcm,kce,isodate):  #new code: Li

    """
Each crop goes through four growing stages: initial - development - mid-season an

Inputs:
Plantation = plantation datetime
Li = length of the initial stage (in days)
Ld = length of the development stage (in days)
Lm = length of the mid-season stage (in days)
Le = length of the end-season stage (in days)

kci = crop coefficient 'kc' at the initial stage. In this stage the ckc value is
kcm = crop coefficient 'kc' at the mid-season stage.  In this stage the ckc value
kce = crop coefficient 'kc' at the end-season stage. In this stege the ckc value
isodate = current date (optional)

Outputs:
* ckc : current crop coefficient, which is constant in the initial and mid-season

Some Examples:
    Kc(plantation="2014-01-01",Li=25,Ld=25,Lm=30,Le=20,Kci=0.15,Kcm=1.19,Kce=0.3
        >>> 0.15
```

[38] Definition of kc function - returns kc value based on the crop calendar stage (planting, growing, harvesting - aggregated from 4 stages)

```
     Kc(plantation="2014-01-01",Li=25,Ld=25,Lm=30,Le=20,Kci=0.15,Kcm=1.19,Kce=0.3
        >>> 0.774

     Kc(plantation="2014-01-01",Li=25,Ld=25,Lm=30,Le=20,Kci=0.15,Kcm=1.19,Kce=0.3
        >>> 1.19

     Kc(plantation="2014-01-01",Li=25,Ld=25,Lm=30,Le=20,Kci=0.15,Kcm=1.19,Kce=0.3
        >>> 0.559

    """
#step 1:

    plantation = pd.to_datetime(plantation, format='%d/%m') #converting the plant
    isodate = pd.to_datetime(isodate , format='%d/%m')  #converting the current c
    test = ((isodate-plantation).days)%365   #The difference in days between the

    # Setting the plantation date and the current date (this is not used)
    Jc = test
    Jp = 0
    J = (Jc - Jp)%365  # %365 means the remaing days of the year

#Step 2: Calculating the day of the year when each crop stage ends placing the da
    JLi1 = Jp + Li1     #end of initial stage = plantation date + lenght of initia
    JLi2 = JLi1 + Li2
    JLd = JLi2 + Ld   #end of development stage = end of initial stage + length c
    JLm = JLd + Lm   #end of mid-season stage = end of development stage + length
    JLe = JLm + Le   #end of end-season stage = end of mid-season stage + length

#step 3: calculating ckc based on the end of each stage date

    if Jc > Jp and Jc < JLe:   #if the current date is greater than the plantatic
        if J <= JLi1:
            ckc = kci1  #if the current date is before the end of initial stage t
        elif Jc > JLi1 and Jc <=JLi2: #New: to account for two init stages
            ckc = kci2
        elif Jc > JLi2 and Jc <=JLd:  #if the current date is betweeen the end of
            ckc = kci2 + ((Jc-JLi2)/Ld * (kcm-kci2))
        elif Jc > JLd and Jc <= JLm:
            ckc = kcm
        elif Jc > JLm and Jc <= JLe:
            ckc = kcm + ((Jc-JLm)/Le * (kce-kcm))

    else:
        ckc = 0

    return ckc
```

In [39]:

```
%%time

#calculate kc based on the growing stage (month - planting, growing, harvesting s
import math
import dateutil     #dateutil module provides powerful extensions to the standarc
from dateutil import parser  #This module offers reads the given date in string a
```

```
mode = pd.read_excel('NWSAS_DATES_CC2_201809.xlsx')
```

Feed the model with the crop calendar dates (planting, growing, harvesting, initial and end dates for each stage respectively)

```
#Note: The code here is adjusted to avoid the end of year issue. In other cases,
#pay attention to all changes, you may need to change this if the crop calendar c


#Planting season: Initial Stage 1 (plant = init1+ init2 )
init1_start = pd.to_datetime(mode['init1_start'], format='%d/%m') #defining the p
init1_end = pd.to_datetime(mode['init1_end'], format='%d/%m')
mode['init1_start_month'] = init1_start.dt.month
mode['init1_end_month'] = init1_end.dt.month
mode['init1_days'] = abs(init1_end - init1_start).dt.days #Calculating the length
Li1 = abs(init1_end - init1_start).dt.days

#Planting season: Initial Stage 2 (plant = init1+ init2 )
init2_start = pd.to_datetime(mode['init2_start'], format='%d/%m') #defining the p
init2_end = pd.to_datetime(mode['init2_end'], format='%d/%m')
mode['init2_start_month'] = init2_start.dt.month
mode['init2_end_month'] = init2_end.dt.month
mode['init2_days'] = abs(init2_end - init2_start).dt.days #Calculating the length
Li2 = abs(init2_end - init2_start).dt.days


#growing 1: Development Stage (grow = dev)
dev_start = pd.to_datetime(mode['dev_start'], format='%d/%m')
dev_end = pd.to_datetime(mode['dev_end'], format='%d/%m')
mode['dev_start_month'] = dev_start.dt.month
mode['dev_end_month'] = dev_end.dt.month
mode['dev_days'] = abs(dev_end - dev_start).dt.days
Ld = abs(dev_end - dev_start).dt.days


#growing 2: Mid stage ( add : mid)
mid_start = pd.to_datetime(mode['mid_start'], format='%d/%m')
mid_end = pd.to_datetime(mode['mid_end'], format='%d/%m')
mode['mid_start_month'] = mid_start.dt.month
mode['mid_end_month'] = mid_end.dt.month
mode['mid_days'] = abs(mid_end - mid_start).dt.days
Lm = abs(mid_end - mid_start).dt.days



#Harvesting: Late stage (harv = late)
late_start = pd.to_datetime(mode['late_start'], format='%d/%m') #defining the pla
late_end = pd.to_datetime(mode['late_end'], format='%d/%m')
mode['late_start_month'] = late_start.dt.month
mode['late_end_month'] = late_end.dt.month
mode['late_days'] = abs(late_end - late_start).dt.days #Calculating the length of
Le = abs(late_end - late_start).dt.days
```

Youssef changed the code implementation - need to go through it again if we stick to a similar crop calendar implementation (unimodal area - and only temporal variation, not spatial)

- Don't know however how the spatial character of the model/approach is incorporated in this case

```
CPU times: user 39.6 ms, sys: 2.52 ms, total: 42.1 ms
Wall time: 40.6 ms
```

In [40]:

```
%%time
#mode = pd.read_excel('NWSAS_CC.xls')
for i in range(1,13):
    mode['kc_{}'.format(i)]=0

for index,row in mode.iterrows():
    for i in range(0,12):
        init1_start = pd.to_datetime(mode['init1_start'].iloc[index], format='%d/
        day_start= (init1_start.day+1-31)%31   #what does this represent??

        if (init1_start.day-1==30):
            month_start = (init1_start.month+1-12)%12  #next month
        else:
            month_start = (init1_start.month-12)%12  #the current month

        month_start = (month_start+i)%12
        if (month_start==0):
            month_start = 12
        mode.loc[index,'kc_{}'.format(month_start)] = kc(mode['init1_start'].iloc
        #print (kc)

        # reacall that def kc(plantation,Li,Ld,Lm,Le,kci,kcd,kcm,kce,isodate):
        #Assuming that :
        #Li = plant_days
        #Ld = dev_days
        #lm = mid_days.
        #le = late_days
        #kci = 0.8 tabulated values FAO
        #kcd = 0.9 tablated values FAO
        #kcm = 1 tablated values FAO
        #kce = 0.8 tabulated values FAO
        #isodate = '{}/{}'.format(day_start,month_start)
```

```
CPU times: user 30.1 ms, sys: 3.62 ms, total: 33.7 ms
Wall time: 31.1 ms
```

In [41]:

```
#so far we worked with (df) dataframe which contains GIS outputs, then we createc

data = pd.merge(df,mode,on='Mode')  #merging the two dataframes on 'Mode' column
```

In [42]:

```
# Calculating the annual precipitation: which is the sum of precipitation values

data['precipitation_annual']=data.filter(like='prec_').sum(axis=1)  #Filter is us
```

- Used later on to calculate a MONTHLY weighted average for the recharge amount (might not be necessary)

In [43]:

```
#Create a Pandas Excel writer using XlsxWriter as the engine.
writer = pd.ExcelWriter('Pilot20190124_Part1.xlsx', engine='xlsxwriter')

# Convert the dataframe to an XlsxWriter Excel object.
data.to_excel(writer, sheet_name='Total_area_dateCC')


# Close the Pandas Excel writer and output the Excel file.
writer.save()
```

In [91]:

```
#Not used in NWSAS calculation since we are dealing with ground water only

%%time
for index,row in data.iterrows():
    for i in range(1,13):
        data['rech_{}'.format(i)].iloc[index]=row['prec_{}'.format(i)]/row['preci
```

Wall time: 501 µs

Aquifer recharge and abstraction limit? Should we consider reviewing it and potentially including it? Or a similar approach

In [92]:

```
%%time
for index,row in data.iterrows():
    for i in range(1,13):
        data['max_rech_{}'.format(i)].iloc[index]=row['rech_{}'.format(i)]*10*row
```

Wall time: 499 µs