PyAEZ: Python Package for Agro-Ecological Zonation

Collaborative work between GIC-AIT and FAO

April 14, 2021

Contents

Chapter 1

Introduction

1.1 Background

AEZ provides a standard framework for a land resources evaluations. The original Agro-ecological Zones Projects were started in 1976 by FAO. And after that several national, regional and global level AEZ projects have been implemented by various organizations. Brief historical background around AEZ is summarized below.

- Origins
  - The Framework of Land Evaluation (Brinkman and Smyth,1976)
  - The FAO/Unesco Soil Map of the World (R. Dudal etal, 1972 -1980)
  - Length of Growing Period (Cocheme and Franquin, 1976)
  - First Regional AEZ study for Africa (FAO - G. Higgins and A. Kassam, 1978)
  - Land Resources for populations of the future (FAO/IIASA, 1984)
- National AEZ studies (1980 – 1995)
  - Mozambique (Spiers and Voortman, 1980)
  - Tanzania (De Pauw, 1984)
  - Northern Algeria (Nachtergaele, 1985)
  - Bangladesh (BARC/FAO, Karim, Brammer and Antoine, 1988)
  - Kenya (Muchena, Fischer and Van Velthuyzen, 1993)
  - China (Chinese Academy of Sciences, Fischer and Van Velthuyzen, 1994)
  - Others: Ethiopia, Malaysia, Philippines, Ukraine, Macedonia, Bosnia-Herzegovina, Thailand, Pakistan, Afghanistan. . . .
- Global AEZ (1995-now)
  - Agriculture Towards the XXIst Century (G. Fischer, H. Van Velthuyzen, M. Shah, F. Nachtergaele, 2002)

- GAEZ v3.0 (G. Fischer, F. Nachtergaele etal., 2011)
- GAEZ v4.0 (in preparation 2020)

The main question that we want to answer with AEZ algorithm is, finding most suitable place to grow a particular crop. So first, we try to simulate all crop cycles and estimate, what will be the maximum yield under particular climate, soil, terrain conditions at all locations. With that estimation, we can decide areas that are well suited for a particular crop.

1.2 Introduction to PyAEZ

PyAEZ is a python package consisted of many algorithms related to AEZ framework. PyAEZ tries to encapsulate all complex calculations in AEZ and try to provide user friendly, and intuitive ways to input data and output results after calculations. PyAEZ includes 5 main modules as below. Additionally to that, *UtilityCalculations* module is also included in PyAEZ to perform additional related utility calculations.

- Module I: Climate Regime
- Module II: Crop Simulations
- Module III: Climate Constraints
- Module IV: Soil Constraints
- Module V: Terrain Constraints
- *UtilityCalculations* Module

Other than 5 main modules and utility module, following 3 major algorithms related to AEZ also are included in PyAEZ as separate modules. Those 3 major algorithms can be utilized individually without running whole PyAEZ. Details of those modules are in following 3 Appendices.

- Appendix I: Biomass Calculations
- Appendix II: Evapotranspiration Calculations
- Appendix III: CropWat Calculations

All those modules are connected to provide intuitive access to PyAEZ package. Overall module structure of PyAEZ is shown in Figure 1.1. Users can use whole PyAEZ package as well as individual components of PyAEZ package based on their requirements.

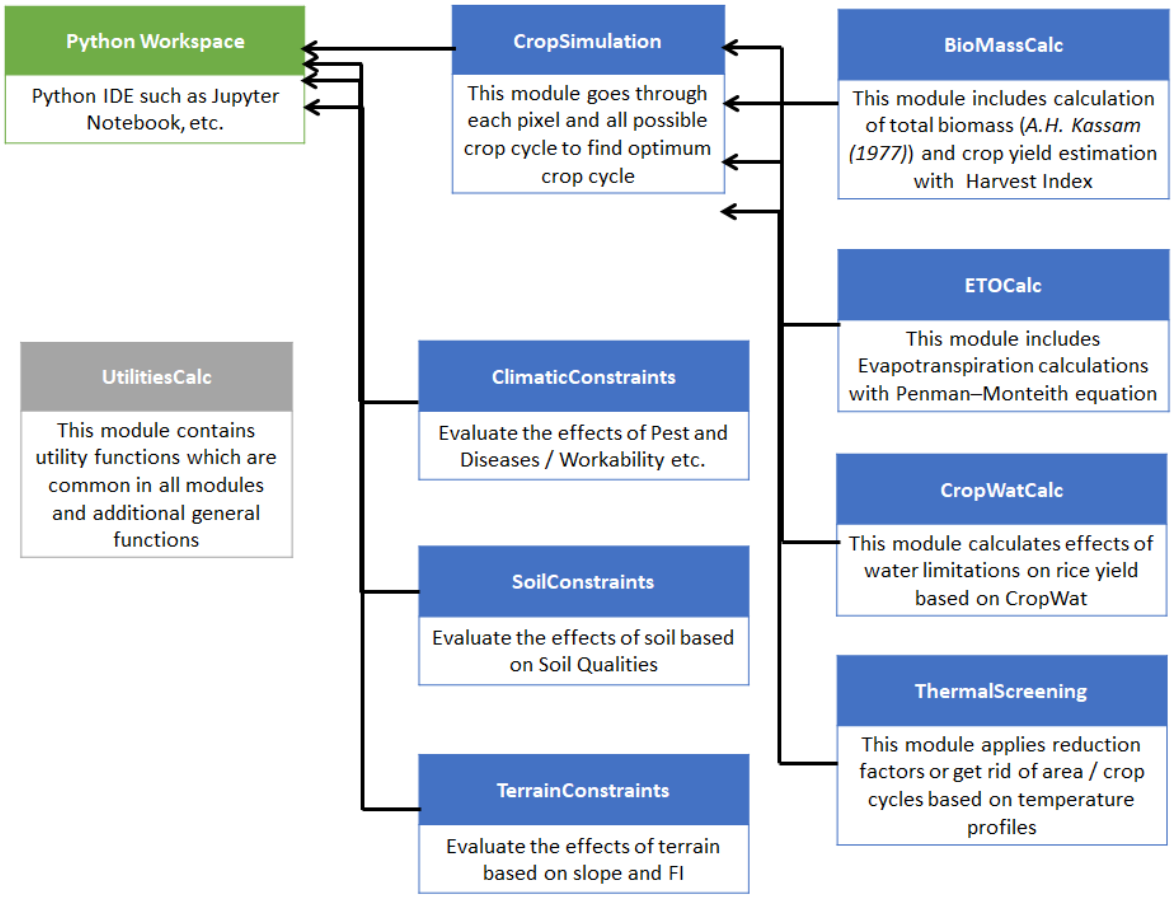

Figure 1.1: Module Structure (Class Diagram) of PyAEZ.

Chapter 2

Module I: Climate Regimes

2.1 Introduction

This is the first module of the AEZ framework. This module performs basic climate data analysis, compiling general agro-climatic indicators. These general agro-climatic indicators summarizes climatic profiles in the study area for each grid. Following agro-climatic indicators can be calculated in this module with temperature and precipitation data (either monthly or daily climate data). For more detailed calculations, refer to Global Agro-ecological Assessment for Agriculture in the 21st Century report (Fischer, van Velthuizen, Shah, & Nachtergaele, 2002a).

- Thermal Climate
- Thermal Zone
- Thermal Length of Growing Periods (LGPs)
- Temperature Sums
- Temperature Profiles
- Length of Growing Periods (LGPs)
- Multi Cropping Zones

First we have to import the Class and create an instance of that Class as below,

```
import ClimateRegime
clim_reg = ClimateRegime.ClimateRegime()
```

2.2 Setting-up Inputs

2.2.1 Climate Inputs

```
clim_reg.setMonthlyClimateData(min_temp, max_temp,
    precipitation, short_rad, wind_speed, rel_humidity)
```

This function allows setting of all climate data. And this is a mandatory function to set before executing calculations.

Arguments:

- min_temp: A 3D numpy array (height, width, and time are in 3D), corresponding to monthly minimum temperature. Units of minimum temperature must be in $Celcius$.
- max_temp: A 3D numpy array (height, width, and time are in 3D), corresponding to monthly maximum temperature. Units of minimum temperature must be in $Celcius$.
- precipitation: A 3D numpy array (height, width, and time are in 3D), corresponding to monthly precipitation in mm/day.
- short_rad: A 3D numpy array (height, width, and time are in 3D), corresponding to monthly short-wave radiation in W/m^2.
- wind_speed: A 3D numpy array (height, width, and time are in 3D), corresponding to wind speed corresponding to 2m elevation. Units of wind speed must be in m/s.
- rel_humidity: A 3D numpy array (height, width, and time are in 3D), corresponding to relative humidity as fractions (fraction values must be between 0 and 1)

Returns: None

If daily climate data are available, similarly, following function can be used to set daily climate data instead of monthly climate data.

```
clim_reg.setDailyClimateData(min_temp, max_temp,
    precipitation, short_rad, wind_speed, rel_humidity)
```

2.2.2 Location and Terrain Inputs

```
clim_reg.setLocationTerrainData(lat_min, lat_max, elevation
    )
```

This function allows setting of location specific and elevation related arguments. And this is a mandatory function to set before executing calculations.

Arguments:

- lat_min: A single value corresponding to minimum latitude in decimal degrees.

- lat_max: A single value corresponding to maximum latitude in decimal degrees.
- elevation: A 2D numpy array, corresponding to elevation of the study area. Units of elevation must be in *meters*.

Returns: None

2.2.3 Study Area Inputs

```
clim_reg.setStudyAreaMask(admin_mask, no_data_value)
```

This function allows setting up of the study area. And this is an optional function. This helps to reduce computational time avoiding calculations in outside of the study area.

Arguments:

- admin_mask: A 2D numpy array, corresponding to the study area.
- no_data_value: A single value, pixels equal to this value will be omitted during calculations.

Returns: None

2.3 Calculations and Extraction of Outputs

2.3.1 Thermal Climate and Thermal Zone

```
tclimate = clim_reg.getThermalClimate()
tzone = clim_reg.getThermalZone()
```

These functions calculate and return Thermal Climate and Thermal Zone respectively. Thermal Climate and Thermal Zone are recommended to produce with average climate data of 30 years rather than using climate data of a single year. For more detailed calculations and classification scheme of Thermal Climate and Thermal Zone, refer to Global Agro-ecological Assessment for Agriculture in the 21st Century report (Fischer et al., 2002a).

Arguments: None

Returns:

- tclimate: Thermal Climate as 2D numpy arrays.
- tzone: Thermal Zone as 2D numpy arrays.

Legends (pixel values vs. classes) of Thermal Climate and Thermal Zone are in Table 2.1 and 2.2 respectively,

Table 2.1: Legend of Thermal Climate.

Pixel values	Class
value 1	Tropical Lowland
value 2	Tropical Highland
value 3	Subtropics Low Rainfall
value 4	Subtropics Summer Rainfall
value 5	Subtropics Winter Rainfall
value 6	Oceanic Temperate
value 7	Sub-Continental Temperate
value 8	Continental Temperate
value 9	Oceanic Boreal
value 10	Sub-Continental Boreal
value 11	Continental Boreal
value 12	Arctic

2.3.2 Thermal Length of Growing Periods (LGPs)

```
lgp0 = clim_reg.getThermalLGP0()
lgp5 = clim_reg.getThermalLGP5()
lgp10 = clim_reg.getThermalLGP10()
```

These functions calculate and return $LGP_{t=0}$, $LGPt_{t=5}$, and $LGP_{t=10}$ respectively.

The length of the temperature growing periods (LGP_ts) are calculated as the number of days in the year when average daily temperature (T_a) is above a particular temperature threshold. AEZ framework defines following 3 LGP_ts.

- $LGP_{t=0}$ period in days when $T_a > 0^oC$
- $LGP_{t=5}$ period in days when $T_a > 5^oC$
- $LGP_{t=10}$ period in days when $T_a > 10^oC$

Arguments: None

Returns:

- lgp0: $LGP_{t=0}$ as 2D numpy arrays. Units are in days.
- lgp5: $LGP_{t=5}$ as 2D numpy arrays. Units are in days.
- lgp10: $LGP_{t=10}$ as 2D numpy arrays. Units are in days.

2.3.3 Temperature Sums

Table 2.2: Legend of Thermal Zone.

Pixel values	Class
value 1	Tropics - Warm
value 2	Tropics - Moderately Cool
value 3	Tropics - Cool
value 4	Tropics - Cold
value 5	Tropics - Very Cold
value 6	Sub Tropic - Warm
value 7	Sub Tropic - Moderately Cool
value 8	Sub Tropic - Cool
value 9	Sub Tropic - Cold
value 10	Sub Tropic - Very Cold
value 11	Temperate - Warm
value 12	Temperate - Moderately Cool
value 13	Temperate - Cool
value 14	Temperate - Cold
value 15	Temperate - Very Cold
value 16	Boreal - Warm
value 17	Boreal - Moderately Cool
value 18	Boreal - Cool
value 19	Boreal - Cold
value 20	Boreal - Very Cold
value 21	Arctic

```
tsum0 = clim_reg.getTemperatureSum0()
tsum5 = clim_reg.getTemperatureSum5()
tsum10 = clim_reg.getTemperatureSum10()
```

This functions calculate and return $Tsum_{t=0}$, $Tsum_{t=5}$, and $Tsum_{t=10}$ respectively.

Temperature Sums relate to heat requirements of crops. Temperature Sums are calculated by accumulating daily average temperatures (T_a) for days when T_a is above a particular threshold temperature. AEZ framework defines following 3 Temperature Sums.

- $Tsum_{t=0}$ sum of daily average temperatures $T_a > 0^oC$
- $Tsum_{t=5}$ sum of daily average temperatures $T_a > 5^oC$
- $Tsum_{t=10}$ sum of daily average temperatures $T_a > 10^oC$

Arguments: None

Returns:

- tsum0: $Tsum_{t=0}$ as 2D numpy arrays. Units are in $Celcius$.

Table 2.3: Temperature Profile Classes.

Average Temperature T_a	Temperature Trend	Class
> 30	Increasing	A1
$25 - 30$	Increasing	A2
$20 - 25$	Increasing	A3
$15 - 20$	Increasing	A4
$10 - 15$	Increasing	A5
$5 - 10$	Increasing	A6
$0 - 5$	Increasing	A7
$-5 - 0$	Increasing	A8
< -5	Increasing	A9
> 30	Decreasing	B1
$25 - 30$	Decreasing	B2
$20 - 25$	Decreasing	B3
$15 - 20$	Decreasing	B4
$10 - 15$	Decreasing	B5
$5 - 10$	Decreasing	B6
$0 - 5$	Decreasing	B7
$-5 - 0$	Decreasing	B8
< -5	Decreasing	B9

- tsum5: $Tsum_{t=5}$ as 2D numpy arrays. Units are in $Celcius$.
- tsum10: $Tsum_{t=10}$ as 2D numpy arrays. Units are in $Celcius$.

2.3.4 Temperature Profiles

```
tprofile = clim_reg.getTemperatureProfile()
```

This function calculates and returns Temperature Profiles.

Temperature profiles are defined in terms of 9 classes based on daily average temperatures ranges (from -5^oC to 30^oC at 5^oC intervals) and temperature trends (either temperature is increasing or decreasing). Temperature profile calculations produce 18, 2D numpy arrays with number of days that satisfied requirements of each of 18 classes. Precise definition of Temperature profile classes are summarized in Table 2.3.

Arguments: None

Returns:

- tprofile: List of 18, 2D numpy arrays corresponding to each of Temperature Profile class (A9, A8, A7, A6, A5, A4, A3, A2, A1, B1, B2, B3, B4, B5, B6, B7, B8, B9) respectively. Units are in days.

2.3.5 Length of Growing Periods (LGPs)

```
lgp = clim_reg.getLGP(Sa = 100, pc = 0.5, kc = 1, D = 1)
lgp_class = clim_reg.getLGPClassified(lgp)
lgp_equv = clim_reg.getLGPEquivalent()
```

These functions calculate and return LGP, Classified LGP as general moisture regime classes, and Equivalent LGP respectively.

Arguments (for *getLGP*):

- Sa: A single value or A 2D numpy array, corresponding to available soil moisture holding capacity (mm/m). Usually, this value varies with soil texture. Hence, Sa can be provided as single value for entire area or 2D numpy array that represent variation of soil moisture holding capacity depending on soil texture. Default value is 100 mm/m. This is an optional argument.
- pc: A single value between 0 and 1, corresponding to soil water depletion fraction below which $ETa < ETo$. Default value is 0.5. This is an optional argument.
- kc: A single value, corresponding to crop water requirements. Default value is 1. This is an optional argument.
- D: A single value, corresponding to corresponding rooting depth in meters. Default value is 1. This is an optional argument.

The agro-climatic potential productivity of a land depends largely on the number of days during the year moisture supply are adequate to crop growth and development. This period in terms of number of days, is known as length of the growing period (LGP). In the AEZ framework, LGP is the number of days where daily $ET_a >= 0.5ET_o$. The length of growing period (LGP) data is also used for the classification of general moisture regimes classes. Classification of general moisture regimes classes based on LGP is defined in Table 2.4 with legend (pixel values vs. classes).

Furthermore, the wetness conditions in different locations can be better compared by Equivalent LGP (LGP_{eq}, in days) which is calculated on the basis of regression analysis of the correlation between LGP and the humidity index P/ET_o. The equivalent LGP is used in the assessment of agro-climatic constraints (Module III) which relate environmental wetness with the occurrences of pest and diseases and workability constraints for harvesting conditions and for high moisture content of crop produce at harvest time. For more detailed calculations of Equivalent LGP, refer to Global Agro-ecological Assessment for Agriculture in the 21st Century report (Fischer et al., 2002a).

Arguments:

- lgp (in $getLGPClassified$ function): A 2D numpy array, corresponding to LGP calculated in the $getLGP$ function.

Returns:

Table 2.4: Moisture Regimes Classes.

Pixel values	Class	Length of growing period (days)
value 7	Per-humid	$>= 365$
value 6	Humid	$270 - 364$
value 5	Sub-humid	$180 - 269$
value 4	Moist semi-arid	$120 - 179$
value 3	Dry semi-arid	$60 - 119$
value 2	Arid	< 60
value 1	Hyper-arid	0

- lgp: LGP as 2D numpy arrays. Units are in days.
- lgp_class: Classified LGP as 2D numpy arrays after classifying LGP.
- lgp_equv: Equivalent LGP as 2D numpy arrays. Units are in days.

2.3.6 Multi Cropping Zones

```
multi_c_zone = clim_reg.getMultiCroppingZones(t_climate,
   lgp, lgp_t5, lgp_t10, ts_t0, ts_t10)
```

These function calculates and returns Multi Cropping Zones.

In the PyAEZ, all core modules perform calculations for single cropping systems. Additionally to this, number of potential multiple cropping zones have been defined through matching both growth cycle and temperature requirements based on Thermal Climate, Length of Growing Period, $LGP_{t=0}$, $LGP_{t=10}$, $Tsum_{t=0}$, and $Tsum_{t=10}$. For more detailed calculations of Multi Cropping Zones, refer to Global Agro-ecological Assessment for Agriculture in the 21st Century report (Fischer et al., 2002a).

Arguments: None

- t_climate: A 2D numpy array, corresponding to Thermal Climate (as calculated before)
- lgp: A 2D numpy array, corresponding to Length of Growing Period (as calculated before)
- lgp_t5: A 2D numpy array, corresponding to $LGP_{t=5}$ (as calculated before)
- lgp_t10: A 2D numpy array, corresponding to $LGP_{t=10}$ (as calculated before)
- ts_t0: A 2D numpy array, corresponding to $Tsum_{t=0}$ (as calculated before)
- ts_t10: A 2D numpy array, corresponding to $Tsum_{t=10}$ (as calculated before)

Returns:

Table 2.5: Legend of Multi Cropping Zone Classification.

Pixel values	Class	Description
value 8	Zone H	zone of triple rice cropping (sequential cropping of three wetland rice crops possible)
value 7	Zone G	zone of triple cropping (sequential cropping of three short-cycle crops; two wetland rice crops possible)
value 6	Zone F	zone of limited triple cropping (partly relay cropping; no third crop possible in case of two wetland rice crops)
value 5	Zone E	zone of double cropping (sequential cropping; wetland rice crop possible)
value 4	Zone D	zone of double cropping (sequential cropping; double cropping with wetland rice not possible)
value 3	Zone C	zone of limited double cropping (relay cropping; single wetland rice may be possible)
value 2	Zone B	zone of single cropping
value 1	Zone A	zone of no cropping (too cold or too dry for rain-fed crops)

- multi_c_zone: Multi Cropping Zone as 2D numpy arrays.

Detailed classification scheme with legend (pixel values vs. classes) for Multi Cropping Zone is in Table 2.5,

Chapter 3

Module II: Crop Simulations

3.1 Introduction

This is the core module of the AEZ framework. This module simulates all possible crop cycles to find best crop cycle that produces maximum yield for a particular grid. During the simulation process for each grid, 365 crop cycle simulations are performed. Each simulation is corresponding to cycles that start from each day of the year (starting from Julian date of 0 to Julian date of 365). Similarly, this process is performed by the program for each grid in the study area. Schematic representation of this process is shown in Figure 3.1.

During each crop cycle attainable yields under irrigated and rain-fed conditions are calculated with the help of several deterministic and empirical models as follows.

- Calculation of total biomass (de Wit, 1965): This model calculates total biomass produced by Photosynthesis activities of plants under radiation condition of each grid. For more detailed calculations, refer to Appendix VI in Global Agro-ecological Assessment for Agriculture in the 21st Century report (Fischer et al., 2002a).
- Calculation of crop yield from total biomass: Crop yield is simply obtained as a portion of useful harvest from total biomass. This portion is defined by an index call Harvest Index (HI). Harvest index is defined as the amount of useful harvest divided by the total above ground biomass. For more detailed calculations, refer to Appendix VI in Global Agro-ecological Assessment for Agriculture in the 21st Century report (Fischer et al., 2002a).
- Calculation of the effects of water limitations on the yield: This component is carries out only for the rain-fed yield calculations. In case of irrigated conditions, this component is abandoned assuming water is not a limited factor for the crop growth. This component of assessing water limitations on the yield is consisted of following two major models.
  - Reference evapotranspiration calculations with Penman-Monteith algorithm (FAO, 1998) (Monteith, 1965) (Monteith, 1981). For more detailed calcula-

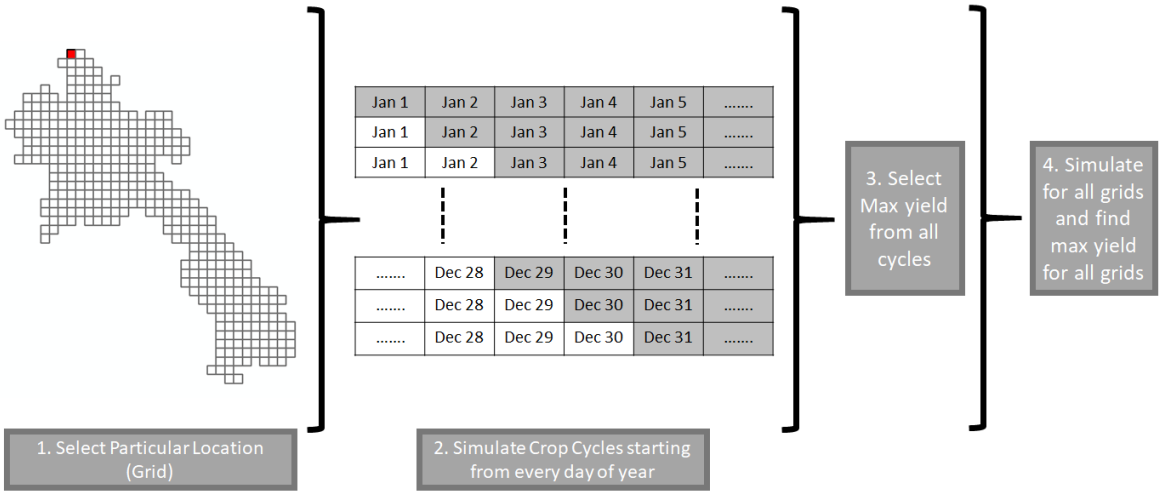

Figure 3.1: Overview of Crop Simulations.

tions, refer to Appendix V in Global Agro-ecological Assessment for Agriculture in the 21st Century report (Fischer et al., 2002a).

- Water balance calculations and applying of yield reduction factors based on water limitation, with FAO CropWat algorithm (FAO, 1992).

- Calculation of the effects of temperature during crop cycle and screening of crop cycles based on temperature requirements (Thermal Screening).

First we have to import the Class and create an instance of that Class as below,

```
import CropSimulation
aez = CropSimulation.CropSimulation()
```

3.2 Setting-up Inputs

3.2.1 Climate Inputs

```
aez.setMonthlyClimateData(min_temp, max_temp, precipitation
    , short_rad, wind_speed, rel_humidity)
```

This function allows setting up of all climate data. And this is a mandatory function to set before executing calculations.

Arguments:

- min_temp: A 3D numpy array (height, width, and time are in 3D), corresponding to monthly minimum temperature. Units of minimum temperature must be in *Celcius*.

- max_temp: A 3D numpy array (height, width, and time are in 3D), corresponding to monthly maximum temperature. Units of minimum temperature must be in $Celcius$.
- precipitation: A 3D numpy array (height, width, and time are in 3D), corresponding to monthly precipitation in mm/day.
- short_rad: A 3D numpy array (height, width, and time are in 3D), corresponding to monthly short-wave radiation in W/m^2.
- wind_speed: A 3D numpy array (height, width, and time are in 3D), corresponding to wind speed corresponding to 2m elevation. Units of wind speed must be in m/s.
- rel_humidity: A 3D numpy array (height, width, and time are in 3D), corresponding to relative humidity as fractions (fraction values must be between 0 and 1)

Returns: None

If daily climate data are available, similarly following function can be used to set daily climate data instead of monthly climate data.

```
aez.setDailyClimateData(min_temp, max_temp, precipitation,
   short_rad, wind_speed, rel_humidity)
```

3.2.2 Location and Terrain Inputs

```
aez.setLocationTerrainData(lat_min, lat_max, elevation)
```

This function allows setting up of location specific and elevation related arguments. And this is a mandatory function to set before executing calculations.

Arguments:

- lat_min: A single value corresponding to minimum latitude in decimal degrees.
- lat_max: A single value corresponding to maximum latitude in decimal degrees.
- elevation: A 2D numpy array, corresponding to elevation of the study area. Units of elevation must be in $meters$.

Returns: None

3.2.3 Crop Parameters Inputs

```
aez.setCropParameters(LAI, HI, legume, adaptability,
   cycle_len, D1, D2)
```

This function allows setting up of main crop parameters. And this is a mandatory function to set before executing calculations.

Arguments:

- LAI: A single value, corresponding to Leaf Area Index
- HI: A single value, corresponding to Harvest Index
- legume: A single binary value (either 0 or 1), corresponding to either the crop is legume or not
- adaptability: A single value, corresponding to adaptability class of the crop. Hence, value must be 1 or 2 or 3 or 4 corresponding to adaptability class of the crop.
- cycle_len: A single value, corresponding length of crop cycle in days
- D1: A single value, corresponding rooting depth in meters at the beginning of the crop cycle
- D2: A single value, corresponding rooting depth in meters after maturity (D1 and D2 can also be same value. In this case, interpolations will not be applied and same rooting depth will be applying during entire crop cycle)

Returns: None

3.2.4 Crop Cycle Parameter Inputs

```
aez.setCropCycleParameters(stage_per, kc, kc_all, yloss_f,
    yloss_f_all)
```

This function allows setting up of parameters related to crop cycles. And this is a mandatory function to set before executing calculations.

Arguments:

- stage_per: A 4 element numerical list, corresponding to percentage of each of 4 stages of a crop cycle, namely initial (d1), vegetative (d2), reproductive (d3), and maturation stage (d4). As an example: stage_per=[10, 30, 30, 30]
- kc: A 3 element numerical list, corresponding crop water requirements for initial, reproductive, the end of the maturation stage. As an example: kc=[1.1, 1.2, 1]
- kc_all: A single value, corresponding to crop water requirements for entire growth cycle.
- yloss_f: A 4 element numerical list, corresponding to yield loss factors of each of 4 stages of crop cycle, namely initial (d1), vegetative (d2), reproductive (d3), and maturation stage (d4). As an example: yloss_f=[1, 2, 2.5, 1]

- yloss_f_all: A single value corresponding to yield loss factor for entire growth cycle.

Returns: None

3.2.5 Soil Water Parameter Inputs

```
aez.setSoilWaterParameters(Sa, pc)
```

This function allows setting up of parameters related to soil water storage. And this is a mandatory function to set before executing calculations.

Arguments:

- Sa: A single value or A 2D numpy array, corresponding to available soil moisture holding capacity (mm/m). Usually, this value varies with soil texture. Hence, Sa can be provided as single value for entire area or 2D numpy array that represent variation of soil moisture holding capacity depending on soil texture.
- pc: A single value between 0 and 1, corresponding to soil water depletion fraction below which $ETa < ETo$.

Returns: None

3.2.6 Study Area Inputs

```
aez.setStudyAreaMask(admin_mask, no_data_value)
```

This function allows setting up of the study area. And this is an optional function. This helps to reduce computational time avoiding calculations in outside of the study area.

Arguments:

- admin_mask: A 2D numpy array, corresponding to study area.
- no_data_value: A single value, pixels equal to these values will be omitted during calculations.

Returns: None

3.2.7 Thermal Screening Inputs

```
aez.setThermalClimateScreening(t_climate, no_t_climate)
aez.setLGPTScreening(no_lgpt, optm_lgpt)
aez.setTSumScreening(no_Tsum, optm_Tsum)
aez.setTProfileScreening(no_Tprofile, optm_Tprofile)
```

These functions are optional functions. The Thermal Regime characteristics calculated in Module I, are matched with the temperature requirements of crops with this calculations. $Not-suitable$ and *Optimum* Thermal Regime characteristics can be provided as inputs. And crop cycles with Thermal Regime characteristics which are less than or equals to *NotSuitable* Thermal Regime characteristics are abundant. And yield reduction factors will not be applies for crop cycles with Thermal Regime characteristics which are higher than or equals to *Optimum* Thermal Regime characteristics. In case of sub-optimum conditions, the degree of sub-optimality is derived by interpolating Thermal Regime characteristics of *NotSuitable* and *Optimum* conditions with corresponding reduction factors 0 (for *Optimum* conditions) and 75% (for *NotSuitable* conditions). For more detailed calculations, refer to Global Agro-ecological Assessment for Agriculture in the 21st Century report (Fischer et al., 2002a).

Arguments:

- *ThermalClimate* Screening:
  - t_climate: A 2D numpy array, corresponding to thermal climate (a output of Module I).
  - no_t_climate: A numerical list, corresponding to pixel values of not suitable thermal climate zones.
- LGP_t Screening:
  - no_lgpt: A numerical list with 3 elements, corresponding to not suitable 3 LGP_t conditions (as in Module I).
  - optm_lgpt: A numerical list with 3 elements, corresponding to optimum 3 LGP_t conditions (as in Module I).
- T_{sum} Screening:
  - no_Tsum: A numerical list with 3 elements, corresponding to not suitable 3 T_{sum} conditions (as in Module I).
  - optm_Tsum: A numerical list with 3 elements, corresponding to optimum 3 T_{sum} conditions (as in Module I).
- $T_{profile}$ Screening:
  - no_Tprofile: A numerical list with 18 elements, corresponding to not suitable 18 $T_{profile}$ conditions (as in Module I).
  - optm_Tprofile: A numerical list with 18 elements, corresponding to optimum 18 $T_{profile}$ conditions (as in Module I).

Returns: None

3.2.8 Adjustment for Perennial Crop

```
aez.adjustForPerennialCrop(aLAI, bLAI, aHI, bHI)
```

This function allows setting up adjustments for perennial crops. These adjustments are performed on Leaf Area Index (LAI) and Harvest Index based on effective growing period. Two parameters are used to calculate the adjustments for each LAI and HI values. For more detailed calculations and tables of adjustment values, refer to Global Agro-ecological Assessment for Agriculture in the 21st Century report (Fischer et al., 2002a). This is an optional function. But these adjustments are essential with perennial crop simulations.

Arguments:

- aLAI: A single value, corresponding to α_{LAI}. As an example, this value for Coffee (arabica) is 0.
- bLAI: A single value, corresponding to β_{LAI}. As an example, this value for Coffee (arabica) is 270.
- aHI: A single value, corresponding to α_{HI}. As an example, this value for Coffee (arabica) is 120.
- bHI: A single value, corresponding to β_{HI}. As an example, this value for Coffee (arabica) is 120.

Returns: None

3.3 Calculations and Extraction of Outputs

3.3.1 Crop Cycle Simulations

```
aez.simulateCropCycle(start_doy=1, end_doy=365, step_doy=1,
    leap_year=False):
```

After setting up all input arguments, all simulations and calculations can be performed by executing this function.

Arguments:

- start_doy: A single value, corresponding to crop simulations starting Julian date. This is an optional argument. And by default, this value is 0..
- end_doy: A single value, corresponding to crop simulations ending Julian date (inclusive). This is an optional argument. And by default, this value is 365.
- step_doy: A single value, corresponding to spacing (in days) between 2 adjacent crop simulations. This is an optional argument. And by default, this value is 1.

- leap_year: $True$ or $False$, depending on weather year which is simulating is a leap year or not. This allows handing leap and non-leap year differently. This is only relevant for monthly climate data because this value will be used in interpolation processes. In case of daily climate data inputs, length of daily climate data vector will be taken as number of days in a year. This is an optional argument and the default value is $False$.

Returns: None

3.3.2 Estimated Maximum Yield

```
yield_map_rain = aez.getEstimatedYieldRainfed()
yield_map_irr = aez.getEstimatedYieldIrrigated()
```

These functions return maximum attainable yield under provided climate conditions in rain-fed and irrigated conditions respectively. Resulting units are Kilograms per hectare (Kg/ha).

Arguments: None

Returns:

- yield_map_rain: maximum attainable yield under provided climate conditions in rain-fed conditions as 2D numpy array. Resulting units are Kilograms per hectare (Kg / ha).
- yield_map_irr: maximum attainable yield under provided climate conditions in irrigated conditions as 2D numpy array. Resulting units are Kilograms per hectare (Kg/ha).

3.3.3 Optimum Crop Calendar

```
starting_date = aez.getOptimumCycleStartDate()
```

This function returns optimum crop cycle starting date that produces highest yield (estimated in above function).

Arguments: None

Returns:

- starting_date: optimum crop cycle starting date as 2D numpy arrays. Each pixel value is corresponding to the Julian date corresponding to optimum crop cycle starting date.

Chapter 4

Module III: Climate Constraints

4.1 Introduction

After estimating maximum attainable yield with core module (Module II) in PyAEZ, various reduction factors are applied to consider effects of constraints which are difficult to simulate. As example, climate-related effects can be pests and diseases and poor workability because of excess moisture which will be applied in this module. These effects are depend on different levels of inputs and Equivalent LGP.

All reduction factors in Module III, IV and V are located in the 2 parameter files corresponding to irrigated and rain-fed conditions. Before applying reduction factors, these parameter files must be edited with reduction factor values corresponding to the crop and input level. At national level, it is strongly suggested to use specific reduction factors based on national research.

Following 4 types of agro-climatic constraints are considered with this module. For more detailed calculations, refer to Global Agro-ecological Assessment for Agriculture in the 21st Century report (Fischer et al., 2002a).

- Long term limitation to crop performance due to year-to-year rainfall variability
- Pests, diseases and weeds damage on plant growth
- Pests, diseases and weeds damage on quality of produce
- Climatic factors affecting the efficiency of farming operations

First we have to import the Class and create an instance of that Class as below,

```
import ClimaticConstraints
obj_constraints = ClimaticConstraints.ClimaticConstraints()
```

4.2 Setting-up Parameter Files

First we have to insert all reduction factors in parameter files as shown in example code below. Reduction factors under irrigated and rain-fed conditions must be inserted in following two separate parameter files.

- ALL_REDUCTION_FACTORS_IRR.py: Parameter file with all for reduction factors in Module III, IV and V in irrigated conditions.
- ALL_REDUCTION_FACTORS_RAIN.py: Parameter file with all for reduction factors in Module III, IV and V in rain-fed conditions.

```
'''----------------------------------------'''
'''Reduction Factors for Climatic Constraints'''
'''----------------------------------------'''

#defining yield reduction factors based of LGP Equivalent
    class
lgp_eq_class = [[0,29], [30,59], [60,89], [90,119],
    [120,149], [150,179], [180,209], [210,239], [240,269],
    [270,299], [300,329], [330,366]]

lgp_eq_red_fr = [[25,25,25,25,25,25,25,50,50,50,75,75],
[100,100,100,100,100,100,100,100,100,100,100,100],
[50,50,50,50,50,75,75,100,100,100,100,75],
[100,100,100,100,100,100,100,100,100,100,100,75]]
```

Parameters:

- lgp_eq_class: A 2D List, corresponding to Equivalent LGP classes in days.
- lgp_eq_red_fr: A 2D List, corresponding to reduction factors. And rows are corresponding to 4 types of agro-climatic constraints which are mentioned in the above section and columns are corresponding to Equivalent LGP classes as in *lgp_eq_class*.

4.3 Calculations and Extraction of Outputs

4.3.1 Applying Climate Constraints

```
yield_out = obj_constraints.applyClimaticConstraints(lgp_eq
    , yield_in, irr_or_rain)
```

This function applies climate-related reduction factors. And it returns yield after applying climate-related reduction factors on inputted yield.

Arguments:

- lgp_eq: A 2D numpy array, corresponding to Equivalent LGP. Equivalent LGP is calculated in Module I.
- yield_in: A 2D numpy array, corresponding to yield before applying climate-related reduction factors. This can be either yield in rain-fed condition or yield in irrigated conditions which are outputs of Module II. Theoretically *yield_in* can be in any units and units of output will be same as unit of *yield_in*.
- irr_or_rain: single character String, indicating *yield_in* is in either rain-fed condition or irrigated condition. 'R' is for rain-fed condition, and 'I' is for irrigated condition.

Returns:

- yield_out: yield after applying climate-related reduction factors as a 2D numpy array. Unit of *yield_out* is same as *yield_in*.

Chapter 5

Module IV: Soil Constraints

5.1 Introduction

After applying agro-climatic constraints as in Module III, soil constraints can be applied in this module. Soil-related reduction factors are applies under following 7 soil qualities based on soil characteristics of each soil unit. And they are combined based on the input level to a single reduction factor (Soil Rating) which will be applies on the yield. For more detailed calculations, refer to Global Agro-ecological Assessment for Agriculture in the 21st Century report (Fischer et al., 2002a).

- SQ1: Nutrient availability
- SQ2: Nutrient retention capacity
- SQ3: Rooting conditions
- SQ4: Oxygen availability to roots
- SQ5: Excess salts
- SQ6: Toxicity
- SQ7: Workability (constraining field management)

Soil characteristics of each soil unit must be prepared in CSV tabular format. Soil characteristics of top and sub soil are prepared in 2 separate CVS files.

- soil_characteristics_topsoil.csv: Soil characteristics of top soil (located in *./sample_data/input* folder).
- soil_characteristics_subsoil.csv: Soil characteristics of sub soil (located in *./sample_data/input* folder).

Column definitions for both CVS files are as follows,

- CODE: soil unit code (numerical values)
- TXT: soil texture (Strings)
- OC: soil organic carbon (numerical values)
- pH: soil pH (numerical values)
- TEB: total exchangeable bases (numerical values)
- BS: base saturation (numerical values)
- CECsoil: cation exchange capacity of soil (numerical values)
- CECclay: cation exchange capacity of clay (numerical values)
- RSD: effective soil depth (numerical values)
- GRC: soil coarse material (Gravel) as Percentage
- DRG: drainage class (VP: very poor, P: Poor, I: Imperfectly, MW: Moderately well, W: Well, SE: Somewhat Excessive, E: Excessive)
- ESP: exchangeable sodium percentage
- EC: electric conductivity (dS/m)
- SPH: soil phase rating (either 0 or 1)
- SPR: soil property rating (either 0 or 1)
- OSD: other soil depth/volume related characteristics rating
- CCB: calcium carbonate content as Percentage
- GYP: gypsum content as Percentage
- VSP: vertical properties (either 0 or 1)

All reduction factors in Module III, IV and V are located in the 2 parameter files corresponding to irrigated and rain-fed conditions. Before applying reduction factors, these parameter files must be edited with reduction factor values corresponding to the crop and input level. At national level, it is strongly suggested to use specific reduction factors based on national research.

First we have to import the Class and create an instance of that Class as below,

```
import SoilConstraints
soil_constraints = SoilConstraints.SoilConstraints()
```

5.2 Setting-up Parameter Files

First we have to insert all reduction factors in parameter files as shown in example code below. Reduction factors under irrigated and rain-fed conditions must be inserted in following two separate parameter files.

- ALL_REDUCTION_FACTORS_IRR.py: Parameter file with all for reduction factors in Module III, IV and V in irrigated conditions.
- ALL_REDUCTION_FACTORS_RAIN.py: Parameter file with all for reduction factors in Module III, IV and V in rain-fed conditions.

```
1
2  '''-----------------------------------------'''
3  '''Reduction Factors for Soil Constraints'''
4  '''-----------------------------------------'''
5
6  # value - values of soil characteristics (must be ascending
       order)
7  # factor - yield reduction factors corresponding to each
       value
8
9  # soil texture for SQ1
10 TXT1_value = ['Fine', 'Medium', 'Coarse']
11 TXT1_factor = [90, 70, 30]
12
13 # soil texture for SQ2
14 TXT2_value = ['Fine', 'Medium', 'Coarse']
15 TXT2_factor = [90, 70, 30]
16
17 # soil texture for SQ7
18 TXT7_value = ['Fine', 'Medium', 'Coarse']
19 TXT7_factor = [90, 70, 30]
20
21 # soil organic carbon
22 OC_value = [0, 0.8, 1.5, 2]
23 OC_factor = [50, 70, 90, 100]
24
25 # soil pH
26 pH_value = [3.6, 4.1, 4.5, 5, 5.5, 6]
27 pH_factor = [10, 30, 50, 70, 90, 100]
28
29 # total exchangeable bases
30 TEB_value = [0, 1.6, 2.8, 4, 6.5]
31 TEB_factor = [30, 50, 70, 90, 100]
32
33 # base saturation
34 BS_value = [0, 35, 50, 80]
```

```
BS_factor = [50, 70, 90, 100]

# cation exchange capacity of soil
CECsoil_value = [0, 2, 4, 8, 10]
CECsoil_factor = [30, 50, 70, 90, 100]

# cation exchange capacity of clay
CECclay_value = [0, 16, 24]
CECclay_factor = [70, 90, 100]

# effective soil depth
RSD_value = [35, 70, 85]
RSD_factor = [50, 90, 100]

# soil coarse material (Gravel)
GRC_value = [10, 30, 90] # \%
GRC_factor = [100, 35, 10]

# drainage
# VP: very poor, P: Poor, I: Imperfectly, MW: Moderately
   well, W: Well, SE: Somewhat Excessive, E: Excessive
DRG_value = ['VP', 'P', 'I', 'MW', 'W', 'SE', 'E']
DRG_factor = [50, 90, 100, 100, 100, 100, 100]

# exchangeable sodium percentage
ESP_value = [10, 20, 30, 40, 100] # \%
ESP_factor = [100, 90, 70, 50, 10]

# electric conductivity
EC_value = [1, 2, 4, 6, 12, 100] # dS/m
EC_factor = [100, 90, 70, 50, 30, 10]

# soil phase rating for SQ3
SPH3_value = ['Lithic', 'skeletic', 'hyperskeletic']
SPH3_factor = [100, 50, 30]

# soil phase rating for SQ4
SPH4_value = ['Lithic', 'skeletic', 'hyperskeletic']
SPH4_factor = [100, 50, 30]

# soil phase rating for SQ5
SPH5_value = ['Lithic', 'skeletic', 'hyperskeletic']
SPH5_factor = [100, 50, 30]

# soil phase rating for SQ6
SPH6_value = ['Lithic', 'skeletic', 'hyperskeletic']
SPH6_factor = [100, 50, 30]

```

```
# soil phase rating for SQ7
SPH7_value = ['Lithic', 'skeletic', 'hyperskeletic']
SPH7_factor = [100, 50, 30]

# other soil depth/volume related characteristics rating
OSD_value = [0]
OSD_factor = [100]

# soil property rating - vertic or not
SPR_value = [0, 1]
SPR_factor = [100, 90]

# calcium carbonate
CCB_value = [3, 6, 15, 25, 100] # \%
CCB_factor = [100, 90, 70, 50, 10]

# gypsum
GYP_value = [1, 3, 10, 15, 100] # \%
GYP_factor = [100, 90, 70, 50, 10]

# vertical properties
VSP_value = [0, 1]
VSP_factor = [100, 90]
```

Parameters:

- soil texture:
  - TXT1_value: List of Strings, corresponding to soil texture types for SQ1.
  - TXT1_factor: List of numerical values, corresponding to respective reduction factors to $TXT1\_value$.
  - TXT2_value: List of Strings, corresponding to soil texture types for SQ2.
  - TXT2_factor: List of numerical values, corresponding to respective reduction factors to $TXT2\_value$.
  - TXT7_value: List of Strings, corresponding to soil texture types for SQ7.
  - TXT7_factor: List of numerical values, corresponding to respective reduction factors to $TXT7\_value$.
- soil organic carbon:
  - OC_value: List of numerical values, corresponding to soil organic carbon. Values must me in ascending order.
  - OC_factor: List of numerical values, corresponding to respective reduction factors to OC_value.
- soil pH:

  - pH_value: List of numerical values, corresponding to soil pH. Values must me in ascending order.
  - pH_factor: List of numerical values, corresponding to respective reduction factors to pH_value.

- total exchangeable bases:
  - TEB_value: List of numerical values, corresponding to total exchangeable bases. Values must me in ascending order.
  - TEB_factor: List of numerical values, corresponding to respective reduction factors to TEB_value.

- base saturation:
  - BS_value: List of numerical values, corresponding to base saturation. Values must me in ascending order.
  - BS_factor: List of numerical values, corresponding to respective reduction factors to BS_value.

- cation exchange capacity of soil:
  - CECsoil_value: List of numerical values, corresponding to cation exchange capacity of soil. Values must me in ascending order.
  - CECsoil_factor: List of numerical values, corresponding to respective reduction factors to $CECsoil\_value$.

- cation exchange capacity of clay:
  - CECclay_value: List of numerical values, corresponding to cation exchange capacity of clay. Values must me in ascending order.
  - CECclay_factor: List of numerical values, corresponding to respective reduction factors to $CECclay\_value$.

- effective soil depth:
  - RSD_value: List of numerical values, corresponding to effective soil depth. Values must me in ascending order.
  - RSD_factor: List of numerical values, corresponding to respective reduction factors to RSD_value.

- soil coarse material (Gravel):
  - GRC_value: List of numerical values, corresponding to soil coarse material (Gravel) content as percentage. Values must me in ascending order.
  - GRC_factor: List of numerical values, corresponding to respective reduction factors to GRC_value.

- drainage class:

  - DRG_value: List of Strings, corresponding to drainage class (VP: very poor, P: Poor, I: Imperfectly, MW: Moderately well, W: Well, SE: Somewhat Excessive, E: Excessive).
  - DRG_factor: List of numerical values, corresponding to respective reduction factors to DRG_value.

- exchangeable sodium percentage:
  - ESP_value: List of numerical values, corresponding to exchangeable sodium percentage. Values must me in ascending order.
  - ESP_factor: List of numerical values, corresponding to respective reduction factors to ESP_value.

- electric conductivity:
  - EC_value: List of numerical values, corresponding to electric conductivity. Values must me in ascending order.
  - EC_factor: List of numerical values, corresponding to respective reduction factors to EC_value.

- soil phase rating - stagnic or gleyic, present or not:
  - SPH3_value: List of Strings, corresponding to soil phase class for SQ3.
  - SPH3_factor: List of numerical values, corresponding to respective reduction factors to $SPH3\_value$.
  - SPH4_value: List of Strings, corresponding to soil phase class for SQ4.
  - SPH4_factor: List of numerical values, corresponding to respective reduction factors to $SPH4\_value$.
  - SPH5_value: List of Strings, corresponding to soil phase class for SQ5.
  - SPH5_factor: List of numerical values, corresponding to respective reduction factors to $SPH5\_value$.
  - SPH6_value: List of Strings, corresponding to soil phase class for SQ6.
  - SPH6_factor: List of numerical values, corresponding to respective reduction factors to $SPH6\_value$.
  - SPH7_value: List of Strings, corresponding to soil phase class for SQ7.
  - SPH7_factor: List of numerical values, corresponding to respective reduction factors to $SPH7\_value$.
  - OSD_value: List of numerical values, corresponding to other soil depth/volume related characteristics rating.
  - OSD_factor: List of numerical values, corresponding to respective reduction factors to OSD_value.

- soil property rating - vertic or not:

  - SPR_value: List of numerical values, corresponding to soil property rating. Values in the list can be either 0 or 1 depending on availability of particular soil phases. Values must me in ascending order.
  - SPR_factor: List of numerical values, corresponding to respective reduction factors to SPR_value.

- calcium carbonate:
  - CCB_value: List of numerical values, corresponding to calcium carbonate content as percentage. Values must me in ascending order.
  - CCB_factor: List of numerical values, corresponding to respective reduction factors to CCB_value.

- gypsum:
  - GYP_value: List of numerical values, corresponding to gypsum content as percentage. Values must me in ascending order.
  - GYP_factor: List of numerical values, corresponding to respective reduction factors to GYP_value.

- vertical properties:
  - VSP_value: List of numerical values, corresponding to vertical properties. Values in the list can be either 0 or 1 depending on availability of vertical properties. Values must me in ascending order.
  - VSP_factor: List of numerical values, corresponding to respective reduction factors to VSP_value.

5.3 Calculations and Extraction of Outputs

5.3.1 Soil Qualities Calculations

```
soil_constraints.calculateSoilQualities(irr_or_rain)
```

This function calculates 7 soil qualities for each soil unit based on inputted soil characteristics.

Arguments:

- irr_or_rain: single character String, indicating calculations are considered under either rain-fed condition or irrigated condition. 'R' is for rain-fed condition, and 'I' is for irrigated condition.

Returns: None

5.3.2 Soil Ratings Calculations

```
soil_constraints.calculateSoilRatings(input_level)
```

This function calculates soil ratings for each soil unit combining 7 soil qualities based on input level.

Arguments:

- input_level: single character String, corresponding to input level. 'L' is for Low input level, 'I' is for Intermediate input level, and 'H' is for High input level.

Returns: None

5.3.3 Extracting Soil Qualities

```
soil_qualities = soil_constraints.getSoilQualities()
```

This function returns 7 soil qualities calculated for each soil unit based on inputted soil characteristics. This is not a mandatory function. But 7 soil qualities can be extracted with this function if required.

Arguments: None

Returns:

- soil_qualities: A 2D numpy array, each row is corresponding to soil units. Each column except first column (column number 2 to 8) corresponding to 7 soil qualities. And first column is corresponding to soil unit code.

5.3.4 Extracting Soil Ratings

```
soil_ratings = soil_constraints.getSoilRatings()
```

This function returns soil ratings for each soil unit combining 7 soil qualities based on input level. This is not a mandatory function. But soil ratings can be extracted with this function if required.

Arguments:

- soil_ratings: A 2D numpy array, each row is corresponding to soil units. First column is corresponding to soil unit code. And second column is corresponding to soil rating of each soil unit.

Returns: None

5.3.5 Applying Climate Constraints

```
yield_out = soil_constraints.applySoilConstraints(soil_map,
    yield_in)
```

This function applies all soil-related reduction factors. And it returns yield after applying all soil-related reduction factors on inputted yield.

Arguments:

- soil_map: A 2D numpy array, corresponding to soil. Each pixel value must be soil unit code. This code is used to link soil rating that will be applied on the inputted yield.
- yield_in: A 2D numpy array, corresponding to yield before applying soil-related reduction factors. This can be either yield in rain-fed conditions or yield in irrigated conditions which are outputs of Module III. Theoretically *yield_in* can be in any units and units of output will be same as unit of *yield_in*.

Returns:

- yield_out: yield after applying all soil-related reduction factors as a 2D numpy array. Unit of *yield_out* is same as *yield_in*.

Chapter 6

Module V: Terrain Constraints

6.1 Introduction

After applying soil constraints in the Module IV, terrain constraints can be applied in this module. Slope and soil erosion related constraints with Fournier index (FI) which is based on monthly precipitation are applied in this section. For more detailed calculations, refer to Global Agro-ecological Assessment for Agriculture in the 21st Century report (Fischer et al., 2002a).

All reduction factors in Module III, IV and V are located in the 2 parameter files corresponding to irrigated and rain-fed conditions. Before applying reduction factors, these parameter files must be edited with reduction factor values corresponding to the crop and input level. At national level it is strongly suggested to use specific reduction factors based on national research.

First we have to import the Class and create an instance of that Class as below,

```
import TerrainConstraints
terrain_constraints = TerrainConstraints.TerrainConstraints
    ()
```

6.2 Setting-up Parameter Files

First we have to insert all reduction factors in parameter files as shown in example code below. Reduction factors under irrigated and rain-fed conditions must be inserted in following two separate parameter files.

- ALL_REDUCTION_FACTORS_IRR.py: Parameter file with all for reduction factors in Module III, IV and V in irrigated conditions.
- ALL_REDUCTION_FACTORS_RAIN.py: Parameter file with all for reduction factors in Module III, IV and V in rain-fed conditions.

```
'''----------------------------------------'''
'''Reduction Factors for Terrain Constraints'''
'''----------------------------------------'''

Slope_class = [[0,0.5], [0.5,2], [2,5], [5,8], [8,16],
   [16,30], [30,45], [45,100]] # classes of slopes (
   Percentage Slope)
FI_class = [[0,1300], [1300,1800], [1800,2200],
   [2200,2500], [2500,2700], [1700,100000]] # classes of
   Fournier index

# sample data are for irrigated-intermediate input-wetland
   rice
# rows corresponding to FI classed and columns
   corresponding to slope classes
Terrain_factor = [[100, 100,  75, 50, 25, 0,  0,  0],
[100, 100,  100,  100,  100,  75, 0,  0],
[100, 100,  100,  100,  75, 25, 0,  0],
[100, 100,  100,  100,  50, 0,  0,  0],
[100, 100,  100,  100,  25, 0,  0,  0],
[100, 100,  100,  100,  25, 0,  0,  0]]
```

Parameters:

- Slope_class: A 2D List, corresponding to slope classes. Slope unit must me Percentage Slope.
- FI_class: A 2D List, corresponding to Fournier index (FI) classes.
- Terrain_factor: A 2D List, corresponding to reduction factors. And rows are corresponding to FI classes and columns are corresponding to slope classes.

6.3 Setting-up Inputs

6.3.1 Climate and Terrain Inputs

```
terrain_constraints.setClimateTerrainData(precipitation,
   slope)
```

This function allows setting up of monthly precipitation data and slope data. And this is a mandatory function to set before executing calculations.

Arguments:

- precipitation: A 3D numpy array (height, width, and time are in 3D), corresponding to monthly precipitation. Unit of monthly precipitation can be any unit. Since Fournier index (FI) is a ratio, unit conversion factors will be cancelled out.

- slope: A 2D numpy array (height, and width are in 2D), corresponding to slope. Unit of slope must be Percentage Slope.

Returns: None

6.4 Calculations and Extraction of Outputs

6.4.1 Fournier Index Calculation

```
terrain_constraints.calculateFI()
```

This function calculates Fournier Index (FI) based on inputted monthly precipitation. FI is a simple index that indicates potential of soil erosion based on monthly precipitation.

Arguments: None

Returns: None

6.4.2 Extracting Fournier Index

```
fi = terrain_constraints.getFI()
```

This function returns Fournier Index (FI) based on inputted monthly precipitation. This is not a mandatory function. But Fournier Index (FI) can be extracted with this function if required.

Arguments: None

Returns:

- fi: A 2D numpy array, corresponding to Fournier Index (FI) based on inputted monthly precipitation.

6.4.3 Applying Terrain Constraints

```
yield_out = terrain_constraints.applyTerrainConstraints(
   yield_in, irr_or_rain)
```

This function applies terrain-related reduction factors. And it will return yield after applying terrain-related reduction factors on inputted yield.

Arguments:

- yield_in: A 2D numpy array, corresponding to yield before applying terrain-related reduction factors. This can be either yield in rain-fed condition or yield

in irrigated conditions which are outputs of Module IV. Theoretically *yield_in* can be in any units and units of output will be same as unit of *yield_in*.

- irr_or_rain: single character String, indicating *yield_in* is in either rain-fed condition or irrigated condition. 'R' is for rain-fed condition, and 'I' is for irrigated condition.

Returns:

- yield_out: yield after applying terrain-related reduction factors as a 2D numpy array. Unit of *yield_out* is same as *yield_in*.

Chapter 7

Module VI: Economic Suitability Analysis

7.1 Introduction

Economical Suitability Analysis Module is the most recent addition to AEZ framework. This module converts AEZ's final crop suitability which is produced with previous 5 modules into economic suitability. Addition to that, all interested crops are then compared to the umbrella crop (which has highest economical potential) in order to indicate and map out its comparative advantage in terms of attainable net revenue relative to the best available option. For more detailed calculations, refer to Module 6 chapter in National Agro-Economic Zoning for Major Crops in Thailand (NAEZ) report (FAO & IIASA, 2017). Following steps of calculations were included in this module.

- Modeling of cost of production with respect to yield and calculation of break-even yield.
- Calculation of attainable net revenue with classification of economical suitability.
- Comparative economic performance analysis with respect to umbrella crop.

First we have to import the Class and create an instance of that Class as below,

```
import EconomicSuitability
econ_su = EconomicSuitability.EconomicSuitability()
```

7.2 Setting-up Inputs

7.2.1 Crop Parameters Inputs

```
econ_su.addACrop(crop_name, crop_cost, crop_yield,
   farm_price, yield_map)
```

This function allows setting of all crop parameters for economic analysis. Crop yield information generated from 5 main modules in PyAEZ as well as prices, costs of the crop are inputted here. Since this function performs comparative economic analysis, we can call this function multiple times to add multiple crops for comparative economic analysis. And this is a mandatory function to set before executing calculations.

Arguments:

- crop_name: A single string value, corresponding to the crop name that you are adding. This name will be used later to extract output for each crop.
- crop_cost: A 1D numpy array, corresponding to cost of production for each yield values in *crop_yield* variable. Values of *crop_cost* and *crop_yield* must be corresponding to each other and they must be in ascending order. Units of this variable must be in cost per hectare. And all costs and prices in this module must be in same currency.
- crop_yield: A 1D numpy array, corresponding to yield values. Values of *crop_cost* and *crop_yield* must be corresponding to each other and they must be in ascending order. Units of this variable must be in tons per hectare.
- farm_price: A 1D numpy array, historical price that famers sell. All the price values can be given as unordered 1D numpy array. This 1D numpy array is used to calculate distribution (mean) of prices. Units of this variable must be price per ton. And all costs and prices in this module must be in same currency.
- yield_map: A 2D numpy array, corresponding to yield map of the crop. This is typically the output after performing all calculations in first five modules of PyAEZ. Units of this variable must be in tons per hectare.

Returns: None

7.3 Calculations and Extraction of Outputs

7.3.1 Getting Net Revenue

```
crop_rev = econ_su.getNetRevenue(crop_name)
```

This function returns net revenue for the crop passed in to the function under *crop_name* variable.

Arguments:

- crop_name: A single string value, corresponding to the crop name that you are getting the output.

Returns:

Table 7.1: Legend of Net Revenue Classification.

Pixel values	Class	Description
value 7	very high	net revenue are equivalent to 75% or more of the overall maximum
value 6	high	net revenue between 63% and 75%
value 5	good	net revenue between 50% and 63%
value 4	medium	net revenue between 35% and 50%
value 3	moderate	net revenue between 20% and 35%
value 2	marginal	net revenue between 10% and 20%
value 1	very marginal	net revenue between 0% and 10%
value 0	not suitable	net revenue less than 0%

- crop_rev: net revenue for the crop passed in to the function under *crop_name* variable as 2D numpy array. Resulting units are revenue per hectare (revenue / ha).

7.3.2 Getting Classified Net Revenue

```
crop_rev_class = econ_su.getClassifiedNetRevenue(crop_name)
```

This function returns classified net revenue for the crop passed in to the function under *crop_name* variable. Classification scheme of net revenue is in Table 7.1.

Arguments:

- crop_name: A single string value, corresponding to the crop name that you are getting the output.

Returns:

- crop_rev_class: classified net revenue according to above classification scheme for the crop passed in to the function under *crop_name* variable as 2D numpy array.

7.3.3 Getting Normalized Net Revenue

```
crop_rev_norm = econ_su.getNormalizedNetRevenue(crop_name)
```

This function returns normalized net revenue for the crop passed in to the function under *crop_name* variable. Normalization is performed by comparing all crops passed to the module, assigning highest possible net revenue to 1 which is known as umbrella crop. And net revenue values of other crops are normalized as portion of the umbrella

crop, normalizing all crops from 0 to 1 scale. This normalization is performed for each pixel separately.

Arguments:

- crop_name: A single string value, corresponding to the crop name that you are getting the output.

Returns:

- crop_rev_norm: normalized net revenue for the crop passed in to the function under *crop_name* variable as 2D numpy array. Output values are between 0 and 1 scale.

Chapter 8

Utility Calculations

8.1 Introduction

This module contains general utility functions which are common for all modules. And also additional general functions related to agro ecological zoning and data processing are also included in this module. Functions and their brief descriptions are as follows,

- interpMonthlyToDaily: This function performs interpolation of monthly climate data in to daily climate data.
- averageDailyToMonthly: This function aggregates daily climate data to monthly climate data.
- generateLatitudeMap: This function generates Latitude Map as 2D numpy array. Latitude Map is generated by linearly interpolating bottom and top latitude values of the study area.
- classifyFinalYield: This function classifies yield estimations and produces suitability maps according to classification scheme defined in AEZ framework.
- saveRaster: This function allows saving 2D numpy arrays as GeoTIFF raster files.
- averageRasters: This function averages list of rasters in time dimension.
- windSpeedAt2m: This function converts wind speed from a particular altitude to wind speed at 2m altitude.

First we have to import the Class and create an instance of that Class as below,

```
1 import UtilitiesCalc
2 obj_utilities = UtilitiesCalc.UtilitiesCalc()
```

8.2 Utility Function Details

8.2.1 Monthly to Daily Interpolation

```
daily_vector = obj_utilities.interpMonthlyToDaily(
   monthly_vector, cycle_begin, cycle_end, no_minus_values=
   False):
```

This function performs interpolation of monthly climate data in to daily climate data with quadratic spline interpolation as recommended in AEZ framework. Interpolation is performed between *cycle_begin* and *cycle_end* Julian dates.

Arguments:

- monthly_vector: A 1D numpy array (time dimension is in 1D) with 12 elements, corresponding to any monthly climate parameter.
- cycle_begin: A single value corresponding to beginning Julian date of the crop cycle.
- cycle_end: A single value corresponding to ending Julian date of the crop cycle.
- no_minus_values: $True$ or $False$, if this argument is $True$, negative resulting values is forced to be zero. This helps to get rid of unrealistic negative interpolated values in climate parameters such as precipitation data. If this argument is $False$, negative resulting values are allowed. By default, this argument is set as $False$ and it's not a mandatory argument to pass.

Returns:

- daily_vector: A 1D numpy array (time dimension is in 1D), corresponding to interpolated daily climate data between *cycle_begin* and *cycle_end* Julian dates.

8.2.2 Daily to Monthly Aggregation

```
monthly_vector = obj_utilities.averageDailyToMonthly(
   daily_vector):
```

This function aggregates daily climate data in to monthly climate data. Aggregationis performed by averaging the data in each month.

Arguments:

- daily_vector: A 1D numpy array (time dimension is in 1D) with 365 elements, corresponding to any daily climate parameter.

Returns:

- monthly_vector: A 1D numpy array (time dimension is in 1D) with 12 elements, corresponding to aggregated monthly climate data.

8.2.3 Generation of Latitude Map

```
lat_map = obj_utilities.generateLatitudeMap(lat_min,
   lat_max, im_height, im_width)
```

This function generates Latitude Map as 2D numpy array. Latitude Map is generated by linearly interpolating bottom and top latitude values of the study area.

Arguments:

- lat_min: A single value corresponding to minimum (bottom) latitude as decimal degrees.
- lat_max: A single value corresponding to maximum (top) latitude as decimal degrees.
- im_height: A single value corresponding to height of resulting latitude map as number of pixels.
- im_width: A single value corresponding to width of resulting latitude map as number of pixels.

Returns:

- lat_map: A 2D numpy array, corresponding to latitude map. Height and width of resulting latitute map will be *im_height* and *im_width* respectively.

8.2.4 Classification of Yield

```
est_yield_class = obj_utilities.classifyFinalYield(
   est_yield)
```

This function classifies yield estimations and produces suitability maps according to classification scheme defined in AEZ framework. Classification scheme consists of 5 classes (very suitable, suitable, moderately suitable, marginally suitable, and not suitable). And detailed classification scheme with legend (pixel values vs. classes) is in Table 8.1,

Arguments:

- est_yield: A 2D numpy array, corresponding to estimated yield.

Returns:

- est_yield_class: Suitability map after classifying inputted yield as a 2D numpy array.

Table 8.1: Legend of Suitability Classification.

Pixel values	Class	Description
value 5	very suitable	yields are equivalent to 80% or more of the overall maximum yield
value 4	suitable	yields between 60% and 80% of the overall maximum yield
value 3	moderately suitable	yields between 40% and 60% of the overall maximum yield
value 2	marginally suitable	yields between 20% and 40% of the overall maximum yield
value 1	not suitable	yields between 0% and 20% of the overall maximum yield

8.2.5 Saving Raster

```
obj_utilities.saveRaster(ref_raster_path, out_path,
   numpy_raster)
```

This function allows saving 2D numpy array as GeoTIFF raster file. This function can be used to save any output of this PyAEZ package as a GeoTIFF raster file.

Arguments:

- ref_raster_path: String, locating reference raster. This must be GeoTIFF raster file. Projection information is copied from this raster to final raster. Any input GeoTIFF raster to PyAEZ package with Projection information can be passed for this argument.
- out_path: String, locating output raster with tif extension. Output must be a GeoTIFF raster file.
- numpy_raster: A 2D numpy array, corresponding raster that need to be saved.

Returns: None

8.2.6 Averaging Raster Files

```
avg_raster = obj_utilities.averageRasters(raster_3d)
```

This function averages list of raster files in time dimension. Some calculations in AEZ framework are recommended to perform with averaged climate data for 30 years. This function can be used for such calculations.

Arguments:

- raster_3d: A 3D numpy array (height, width, and time are in 3D), corresponding to any climate data that required averaging. Averaging will be done through last dimension (usually corresponding to years).

Returns:

- avg_raster: A 2D numpy array, after averaging inputted climate data through last dimension (time dimension, usually corresponding to years).

8.2.7 Calculating Wind Speed at 2m Altitude

```
wind_speed_2m = obj_utilities.windSpeedAt2m(wind_speed,
   altitude)
```

This function converts wind speed from a particular altitude to wind speed at 2m altitude. All wind speed related calculations in PyAEZ is performed with wind speed at 2m altitude. If your wind speed data is from different altitude, this function can be used to convert wind speed from a particular altitude to wind speed at 2m altitude, before performing calculation in PyAEZ.

Arguments:

- wind_speed: A numpy array (can be 1D, 2D or 3D), corresponding to wind speed. Theoretically *wind_speed* can be in any units and units of output will be same as unit of *wind_speed*.
- altitude: A single value corresponding to the altitude (above ground) of *wind_speed* data. Units of altitude must be in *meters*.

Returns:

- wind_speed_2m: Converted wind speed at 2m altitude as a numpy array. Units will be same as unit of *wind_speed*.

Appendix 1

Appendix I: Biomass Calculations

1.1 Introduction

This module calculates total biomass produced by Photosynthesis activities of plants under given radiation condition (de Wit, 1965). With that, crop yield is simply obtained as a portion of useful harvest from total biomass. This portion is defined by an index call Harvest Index (HI). Harvest index is defined as the amount of useful harvest divided by the total above ground biomass. For more detailed calculations, refer to Appendix VI in Global Agro-ecological Assessment for Agriculture in the 21st Century report (Fischer et al., 2002a).

First we have to import the Class and create an instance of that Class as below,

```
import BioMassCalc
obj_maxyield = BioMassCalc.BioMassCalc(cycle_begin,
   cycle_end, latitude)
```

Arguments:

- cycle_begin: A single value corresponding to beginning Julian date of the crop cycle.
- cycle_end: A single value corresponding to end Julian date of the crop cycle.
- latitude: A single value corresponding to latitude of the location in decimal degrees.

1.2 Setting-up Inputs

1.2.1 Climate Inputs

```
obj_maxyield.setClimateData(min_temp, max_temp, short_rad)
```

This function allows setting of all climate data. And this is a mandatory function to set before executing calculations.

Arguments:

- min_temp: A 1D numpy array (time dimension is in 1D), corresponding to daily minimum temperature. Units of minimum temperature must be in *Celcius*.
- max_temp: A 1D numpy array (time dimension is in 1D), corresponding to daily maximum temperature. Units of maximum temperature must be in *Celcius*.
- short_rad: A 1D numpy array (time dimension is in 1D), corresponding to daily short-wave radiation in W/m^2.

Returns: None

1.2.2 Crop Parameters Inputs

```
obj_maxyield.setCropParameters(LAI, HI, legume,
   adaptability)
```

This function allows setting up of main crop parameters. And this is a mandatory function to set before executing calculations.

Arguments:

- LAI: A single value, corresponding to Leaf Area Index
- HI: A single value, corresponding to Harvest Index
- legume: A single binary value (either 0 or 1), corresponding to either the crop is legume or not
- adaptability: A single value, corresponding to adaptability class of the crop. Hence, value must be 1 or 2 or 3 or 4 corresponding to adaptability class of the crop.

Returns: None

1.3 Calculations and Extraction of Outputs

1.3.1 Estimating Maximum Yield

```
est_biomass = obj_maxyield.calculateBioMass()
```

This function returns biomass under provided climate conditions.

Arguments: None

Returns:

- est_biomass: biomass under provided climate conditions as 1D numpy arrays. Resulting units are Kilograms per hectare (Kg / ha).

1.3.2 Optimum Crop Calendar

```
est_yield = obj_maxyield.calculateYield()
```

This function returns yield ($Biomass \times HI$) under provided climate conditions.

Arguments: None

Returns:

- est_biomass: yield under provided climate conditions as 1D numpy arrays. Resulting units are Kilograms per hectare (Kg / ha).

Appendix 2

Appendix II: Evapotranspiration Calculations

2.1 Introduction

This module calculats reference evapotranspiration with Penman-Monteith algorithm (FAO, 1998) (Monteith, 1965) (Monteith, 1981). For more detailed calculations, refer to Appendix V in Global Agro-ecological Assessment for Agriculture in the 21st Century report (Fischer et al., 2002a).

First we have to import the Class and create an instance of that Class as below,

```
import ETOCalc
obj_eto = ETOCalc.ETOCalc(cycle_begin, cycle_end, latitude,
    altitude)
```

Arguments:

- cycle_begin: A single value, corresponding to beginning Julian date of the crop cycle.
- cycle_end: A single value, corresponding to end Julian date of the crop cycle.
- latitude: A single value, corresponding to latitude of the location in decimal degrees.
- altitude: A single value, corresponding to altitude of the location in meters.

2.2 Setting-up Inputs

2.2.1 Climate Inputs

```
obj_eto.setClimateData(min_temp, max_temp,wind_speed,
   short_rad, rel_humidity)
```

This function allows setting up of all climate data. And this is a mandatory function to set before executing calculations.

Arguments:

- min_temp: A 1D numpy array (time dimension is in 1D), corresponding to daily minimum temperature. Units of minimum temperature must be in $Celcius$.
- max_temp: A 1D numpy array (time dimension is in 1D), corresponding to daily maximum temperature. Units of maximum temperature must be in $Celcius$.
- short_rad: A 1D numpy array (time dimension is in 1D), corresponding to daily short-wave radiation in $MJ/m2/day$.
- wind_speed: A 1D numpy array (time dimension is in 1D), corresponding to daily wind speed at 2m elevation. Units of wind speed must be in m/s.
- rel_humidity: A 1D numpy array (time dimension is in 1D), corresponding to relative humidity as fractions (fraction values must be between 0 and 1)

Returns: None

2.3 Calculations and Extraction of Outputs

2.3.1 Estimating ETo

```
pet = obj_eto.calculateETO()
```

This function returns reference evapotranspiration under provided climate conditions.

Arguments: None

Returns:

- pet: reference evapotranspiration under provided climate conditions as 1D numpy array. Resulting units are in millimeters per day (mm / day).

Appendix 3

Appendix III: CropWat Calculations

3.1 Introduction

This module performs water balance calculations and applies of yield reduction factors based on water limitation, with FAO CropWat algorithm (FAO, 1992).

First we have to import the Class and create an instance of that Class as below,

```
import CropWatCalc
obj_cropwat = CropWatCalc.CropWatCalc(cycle_begin,
   cycle_end)
```

Arguments:

- cycle_begin: A single value, corresponding to beginning Julian date of the crop cycle.
- cycle_end: A single value, corresponding to end Julian date of the crop cycle.

3.2 Setting-up Inputs

3.2.1 Climate Inputs

```
obj_cropwat.setClimateData(pet, precipitation)
```

This function allows setting-up of climate data. And this is a mandatory function to set before executing calculations.

Arguments:

- pet: A 1D numpy array (time dimension is in 1D), corresponding to daily reference evapotranspiration.

- precipitation: A 1D numpy array (time dimension is in 1D), corresponding to daily precipitation.

Returns: None

3.2.2 Crop Parameters Inputs

```
obj_cropwat.setCropParameters(stage_per, kc, kc_all,
   yloss_f, yloss_f_all, est_yield, D1, D2, Sa, pc)
```

This function allows setting up of main crop parameters related to CropWat. And this is a mandatory function to set before executing calculations.

Arguments:

- stage_per: A 4 elements numerical list, corresponding to percentage of each of 4 stages of crop cycle, namely initial (d1), vegetative (d2), reproductive (d3), and maturation stage (d4). As an example: stage_per=[10, 30, 30, 30]
- kc: A 3 elements numerical list, corresponding crop water requirements for initial, reproductive, the end of the maturation stage. As an example: kc=[1.1, 1.2, 1]
- kc_all: A single value, corresponding to crop water requirements for entire growth cycle.
- yloss_f: A 4 elements numerical list, corresponding to yield loss factors of each of 4 stages of crop cycle, namely initial (d1), vegetative (d2), reproductive (d3), and maturation stage (d4). As an example: yloss_f=[1, 2, 2.5, 1]
- yloss_f_all: A single value, corresponding to yield loss factor for entire growth cycle.
- est_yield: A single value corresponding to yield before applying of yield reduction factors based on water limitation. Yield reduction factors based on water limitation is applied on this value. Theoretically *est_yield* can be in any units and units of output will be same as unit of *est_yield*.
- D1: A single value, corresponding rooting depth in meters at the beginning of the crop cycle.
- D2: A single value, corresponding rooting depth in meters after maturity ($D1$ and $D2$ can also be same value. In this case, interpolations will not be applied and same rooting depth will be applying during entire crop cycle).
- Sa: A single value, corresponding to available soil moisture holding capacity (mm/m). Usually, this value depends on soil texture.
- pc: A single value between 0 and 1, corresponding to soil water depletion fraction below which $ETa < ETo$.

Returns: None

3.3 Calculations and Extraction of Outputs

3.3.1 Water Limited Yield Estimation

```
water_limited_est_yield = obj_cropwat.
   calculateMoistureLimitedYield()
```

This function returns yield after applying of yield reduction factors based on water limitation.

Arguments: None

Returns:

- water_limited_est_yield: yield after applying of yield reduction factors based on water limitation as a single value. Units of *water_limited_est_yield* will be same as unit of *est_yield*, passed with *obj_cropwat.setCropParameters* function.

References

de Wit, C. T. (1965). Photosynthesis of leaf canopies. *Agricultural Research Report No. 663. PUDOC, Wageningen*, *57*.

FAO. (1992). Cropwat: A computer program for irrigation planning and management. *Land and Water Development Division, Rome, Italy*, *FAO Irrigation and Drainage Paper no 46*.

FAO. (1998). Crop evapotranspiration. *FAO Irrigation and Drainage Paper no.56 Rome, Italy*.

FAO, & IIASA. (2017). National agro-economic zoning for major crops in thailand (naez). *Project TCP/THA/3403*.

Fischer, G., van Velthuizen, H., Shah, M., & Nachtergaele, F. (2002a). Global agro-ecological assessment for agriculture in the 21st century: Methodology and results. *IIASA RR-02-02, IIASA, Laxenburg, Austria*.

Monteith, J. L. (1965). Evapotranspiration and the environment. *In The State and Movement of Water in Living Organisms*, 205–234.

Monteith, J. L. (1981). Evapotranspiration and surface temperature. *Quarterly Journal Royal Meteorological Society*, *107*, 1–27.