Beenamics

Implementation of HoPoMo model

Jesse DINGLEY, Félix FROMENT, Charlie MEYER, Luca ORDONNEAU, Pierre VIRGAUX

For this project we worked on the HoPoMo model. Bees are essential to pollination and their presence guarantees us better yields for our farmers. We worked on the article that first appeared in Elsevier's newspaper and then in "ECOLOGICAL MODELLING" in 2007. We have implemented the equations in Python in order to reproduce the exposed model.

---

1. Setting up

Necessary imports

In [1]:

```
import math
import random
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
plt.style.use('fivethirtyeight')
```

Constants

The operation of a hive is particularly complex and requires many pre-calculated values. The operation of a hive is particularly complex and requires many pre-calculated values. Many of these were given explicitly in the article, others were read from the graphs given and some were not. For some of them we determined them experimentally by observing the results graphs. We have synthesized all these variables in an appex file so that the user can easily modify it.

In [2]:

```
from variables import *
```

Array Initializations

As you can see we use a lot of lists to try to control infinite recursion problems. In the article the equations are given but the handling of the initial conditions remains free to be interpreted with the chosen programming language. This method is a bit brutal but it allows us to stop the recursion problem and gives us the necessary lists to build the data frames.

In [3]:

```
global CELLSemptyArray, SUPcombArray, ELRArray, CELLSeggsArray, CELLSlarvaeArray, CELLSpupaeArray, CELLSb
CELLSemptyArray = []
SUPcombArray = []
ELRArray = []
CELLSeggsArray = []
CELLSlarvaeArray = []
CELLSpupaeArray = []
CELLSbroodArray = []
MORTALITYadultArray = []
BEESadultArray = []
INDEXflightArray = []
INDEXnectaroutsideArray = []
INDEXpollenoutsideArray = []
NEEDworkersArray = []
NEEDnursesArray = []
RATIOworkforceArray = []
NursesArray = []
INDEXnursingqualityArray = []
FORAGERSArray = []
FORAGERSactiveArray = []
NEEDpollenArray = []
NEEDpollen_larvaeArray = []
NEEDpollen_adultArray = []
NEEDpollenincomeArray = []
NEEDpollenforagerArray = []
FORAGERSpollenArray = []
FORAGERSpollenactiveArray = []
NEEDnectarArray = []
NEEDnectar_larvaeArray = []
NEEDnectar_adultArray = []
WORKFORCEnectarArray = []
FORAGERSnectarArray = []
FORAGERSnectaractiveArray = []
INCOMEpollenArray = []
INCOMEnectarArray = []
INDEXpollensituationArray = []
NEEDprocessorsArray = []
PROCESSORSArray = []
PROCESSEDnectarArray = []
USAGEpollenArray = []
STORESpollenArray = []
USAGEnectarArray = []
USAGEhoneyArray = []
STORESnectarArray = []
STOREShoneyArray = []
WEIGHTcolonyArray = []
BEESlazyArray = []
TEMPArray1 = np.linspace(15, 35, 183)
TEMPArray2 = np.linspace(35, 15, 183)
TEMPArray = np.concatenate([TEMPArray1,TEMPArray2])
```

```
#Swarm extension : Part 1

global ELRArray_swarm, BEESswarmArray, BEESadultArray_swarm, LOSShoney_swarmArray, USAGEhoneyArray_swarm,

ELRArray_swarm = []
BEESswarmArray = []
BEESadultArray_swarm = []
LOSShoney_swarmArray = []
USAGEhoneyArray_swarm = []
TURNSnectarforagerArray = []
WEIGHTcolonyArray_swarm = []

#Sawrm extension : Part 2
global FACTORqueenArray, NEEDnectarbuildingArray, NEEDnectar_adultArray_swarm, CELLShiveArray, CELLSempty

FACTORqueenArray = []
NEEDnectarbuildingArray = []
NEEDnectar_adultArray_swarm = []
CELLShiveArray = []
CELLSemptyArray_swarm = []
SUPcombArray_swarm = []
WEIGHTcolonyArray_swarm2 = []
TURNSnectarforagerArray_swarm2 = []
BEESadultArray_swarm2 = []
ELRArray_swarm2 = []
```

2. Model implementation

2.1. Modeling the queen's egg laying behavior

This seasonal factor influences the queen's daily egg-laying rate, it also affects the nectar and the availability of pollen in the environment.

In [4]:

```
def season(t):
    # (1)
    return max(1-(1/(1+x1*math.exp(-2*t/x2))),1/(1+x3*math.exp(-2*(t-x4)/x5)))
```

Number of empty cells at a day t.

In [5]:

```
def CELLSempty(t):
    # (2)
    global CELLSemptyArray
    if t>len(CELLSemptyArray):
        CELLSempty(t-1)
    if t==len(CELLSemptyArray):
        res = CELLShive0 - CELLSbrood(t) - STORESpollen(t) - STORESnectar(t) - STOREShoney(t)
        CELLSemptyArray = np.append(CELLSemptyArray, res)
    return CELLSemptyArray[t]
```

Describes the suppression in egg laying when the available empty space in the hive is below a threshold value : `SUPthreshold`

In [6]:

```
def SUPcomb(t):
    # (3)
    global SUPcombArray
    if t>len(SUPcombArray):
        SUPcomb(t-1)
    if t == len(SUPcombArray):
        if CELLSempty(t)/(CELLShive0 + 1) < SUPthreshold:
            res = CELLSempty(t)/((CELLShive0 + 1) * SUPthreshold)
        else:
            res = 1
        SUPcombArray = np.append(SUPcombArray, res)
    return SUPcombArray[t]
```

Factor around 0, so as not to be deterministic.

In [7]:

```
def ELRstoch(t):
    # (4)
    return random.uniform(-ERLstochrange,ERLstochrange)
```

Modeling of the whole queen's daily laying rate at day t.

In [8]:

```
def ELR(t):
    # (5)
    global ELRArray
    if t > len(ELRArray):
        ELR(t-1)
    if t == len(ELRArray):
        res = ELRbase0 * (1 + ELRstoch(t)) * (1 - season(t)) * SUPcomb(t)
        ELRArray = np.append(ELRArray, res)
    return ELRArray[t]
```

2.2. Modeling the immature stages

2.2.1 Eggs

In [9]:

```
def EGGS(i,t):
    # (6)
    if t==0:
        res = 0
    elif i == 1:
        res = ELR(t-1) * (1 - MORTALITYeggs)
    elif i in [2,3]:
        res = EGGS(i-1,t-1) * (1 - MORTALITYeggs)
    return res
```

Calculation of the daily total number of eggs.

In [10]:

```
def CELLSeggs(t):
    # (7)
    global CELLSeggsArray
    if t > len(CELLSeggsArray):
        CELLSeggs(t-1)
    if t == len(CELLSeggsArray):
        sum = 0
        for i in range(1,LIFESPANegg):
            sum += EGGS(i,t)
        CELLSeggsArray = np.append(CELLSeggsArray, sum)
    return CELLSeggsArray[t]
```

2.2.2. Larvae (unsealed)

We model the aging process of larvae with the notion of cannibalism depending on the age of the larvae.

In [11]:

```
def SURVIVALlarvae(i,t):
    # (9)
    return (1 - CANNIBALISMlarvae(i,t)) * (1 - MORTALITYlarvae)
```

In [12]:

```
def CANNIBALISMlarvae(i,t):
    # (10)
    if t == 0:
        res = 0
    else:
        res = CANNIBALISMhungerbase[i-1] * (1 - (INDEXpollensituation(t-1) * INDEXnursingquality(t-1)))
    return res
```

Calculation of the daily demographics of the age of the larvae.

In [13]:

```
def LARVAE(i,t):
    # (11)
    if t==0:
        res = 0
    elif i == 1:
        res = EGGS(LIFESPANegg,t-1)*SURVIVALlarvae(1,t)
    elif 1 < i and i <= LIFESPANlarva:
        res = LARVAE(i-1,t-1) * SURVIVALlarvae(i,t)
    return res
```

Calculation of daily number of cells containing any age of larvae.

In [14]:

```
def CELLSlarvae(t):
    # (12)
    global CELLSlarvaeArray
    if t > len(CELLSlarvaeArray):
```

```
        CELLSlarvae(t-1)
    if t == len(CELLSlarvaeArray):
        sum = 0
        for i in range(1, LIFESPANlarva):
            sum += LARVAE(i,t)
        CELLSlarvaeArray = np.append(CELLSlarvaeArray, sum)
    return CELLSlarvaeArray[t]
```

In [15]:

```
# def CELLSlarvae13(t):
#     # (13)
#     # alternative to (12)

#     def product(i):
#         prdct = 1
#         for k in range(i+1):
#             if k != 0:
#                 prdct *= SURVIVALlarvae(k,t-i+k)
#         return prdct

#     sum = 0
#     for i in range(LIFESPANlarva+1):
#         if i != 0:
#             sum += (ELR(t-i-LIFESPANegg)*math.pow(1-MORTALITYeggs,i)*product(i))

#     return sum
```

Calculation of age demographics of all sealed broods.

In [16]:

```
def PUPAE(i,t):
    # (14)
    if t==0:
        res =0
    elif i == 1:
        res = LARVAE(LIFESPANlarva,t-1) * (1 - MORTALITYpupae)
    elif 1 < i and i <= LIFESPANpupa:
        res = PUPAE(i-1,t-1) * (1 - MORTALITYpupae)
    return res
```

Calculation of the daily age demographics of the sealed brood.

In [17]:

```
def CELLSpupae(t):
    # (15)
    global CELLSpupaeArray
    if t > len(CELLSpupaeArray):
        CELLSpupae(t-1)
    if t == len(CELLSpupaeArray):
        sum = 0
        for i in range(1,LIFESPANpupa):
            sum += PUPAE(i,t)
        CELLSpupaeArray = np.append(CELLSpupaeArray, sum)
    return CELLSpupaeArray[t]
```

In [18]:

```
# def CELLSpupae16(t):
#     # (16)
#     # alternative to (15)

#     def product(i):
#         prdct = 1
#         for k in range(LIFESPANlarva+1):
#             if k != 0:
#                 prdct *= SURVIVALlarvae(k,t-i-LIFESPANlarva+k)
#         return prdct

#     sum = 0
#     for i in range(LIFESPANpupa+1):
#         if i != 0:
#             sum += (ELR(t-i-LIFESPANegg-LIFESPANlarva)*math.pow(1-MORTALITYeggs,LIFESPANegg)*product(i.

#     return sum
```

Calculation of the total number of cells filled with brood can now be calculated.

In [19]:

```
def CELLSbrood(t):
    # (17)
    global CELLSbroodArray
```

```
    if t > len(CELLSbroodArray):
        CELLSbrood(t-1)
    if t == len(CELLSbroodArray):
        if t==0:
            res = INITCELLSbrood
        else:
            res = CELLSeggs(t) + CELLSlarvae(t) + CELLSpupae(t)
        CELLSbroodArray = np.append(CELLSbroodArray, res)
    return CELLSbroodArray[t]
```

2.3 Modeling the population of adult bees

Modeling of the global daily mortality rate of adult bees.

In [20]:

```
def MORTALITYadult(t) :
    # (18)
    global MORTALITYadultArray
    if t > len(MORTALITYadultArray):
        MORTALITYadult(t-1)
    if t == len(MORTALITYadultArray):
        if t == 0:
            res = MORTALITYadultbase
        else:
            res = MORTALITYadultbase
            res += (MORTALITYnursing * NURSES(t - 1) / (BEESadult(t - 1) + 1))
            res += (MORTALITYprocessing * PROCESSORS(t - 1) / (BEESadult(t - 1) + 1))
            res += (MORTALITYforaging * FORAGERSactive(t - 1) / (BEESadult(t - 1) + 1))
        MORTALITYadultArray = np.append(MORTALITYadultArray, res)
    return MORTALITYadultArray[t]
```

We model using the global mortality rate and the daily number of newly emerging bees, the daily number of adult bees in the colony.

In [21]:

```
def BEESadult(t) :
    # (19)
    global BEESadultArray
    if t>len(BEESadultArray):
        BEESadult(t-1)
    if t==len(BEESadultArray):
        if t==0:
            res = INITBEESadult
        else:
            res = (BEESadult(t-1) + PUPAE(LIFESPANpupa, t-1))*(1 - MORTALITYadult(t))
        BEESadultArray = np.append(BEESadultArray, res)
    return BEESadultArray[t]
```

2.4 Modeling the influence of the environment

`INDEXrain` has a value between 0 and 1. The closer it is to one, the less rain has affected the foraging on that day. `RAIN` correpesponds the amount of rain on a given day.

In [22]:

```
def INDEXrain(t) :
    # (20)
 return (1 - RAIN(t))
```

In [23]:

```
def RAIN(t) :
    # (21)
    #res = HOURSraining_during_daylight(t) / HOURSdaylight(t)
    res = 0
    return res
```

Just like `INDEXrain` , `INDEXtemperature` also has a value between 0 and 1. The closer it is to one, the greater the flight activity for the temperature. `TEMP` correponds to the mean daily temperature.

In [24]:

```
def TEMP(t):
    global TEMPArray
    return TEMPArray[t]

def INDEXtemperature(t) :
    # (22)
 if ((TEMP(t) <= 14) or (TEMP(t) > 40)) :
  res = 0
 elif (TEMP(t) <= 22) :
```

```
    res = (TEMP(t) - 14) / 8
  elif (TEMP(t) <= 32) :
    res = 1
  else :
    res = (40 - TEMP(t)) / 8

  return res
```

By multiplying the two previous environmental factors we can determine how good the overall flight conditions are. The corresponding function is `INDEXflight` . It returns once again a value between 0 and 1. The closer it is to one, the better the conditions.

In [25]:

```
def INDEXflight(t):
    # (23)
    global INDEXflightArray
    if t > len(INDEXflightArray):
        INDEXflight(t-1)
    if t == len(INDEXflightArray):
        res = (INDEXrain(t) * INDEXtemperature(t))
        INDEXflightArray = np.append(INDEXflightArray, res)
    return INDEXflightArray[t]
```

We can also consider other factors such as the amount of nectar and pollen available:

In [26]:

```
def INDEXnectaroutside(t) :
    # (24)
    global INDEXnectaroutsideArray
    if t > len(INDEXnectaroutsideArray):
        INDEXnectaroutside(t-1)
    if t == len(INDEXnectaroutsideArray):
        res = min((1 - season(t)) * 1.5, 1)
        INDEXnectaroutsideArray = np.append(INDEXnectaroutsideArray, res)
    return INDEXnectaroutsideArray[t]
```

In [27]:

```
def INDEXpollenoutside(t) :
    # (25)
    global INDEXpollenoutsideArray
    if t > len(INDEXpollenoutsideArray):
        INDEXpollenoutside(t-1)
    if t == len(INDEXpollenoutsideArray):
        res = min((1 - season(t)) * 1.5, 1)
        INDEXpollenoutsideArray = np.append(INDEXpollenoutsideArray, res)
    return INDEXpollenoutsideArray[t]
```

2.5 Modeling task decisions

Here we model the workforce of the colony. `NEEDworkers` models the daily worker need in the two high priority tasks (nursing and pollen foraging). `RATIOworkforce` models the daily ratio of workforce to workload.

In [28]:

```
def NEEDworkers(t):
    # (26)
    global NEEDworkersArray
    if t > len(NEEDworkersArray):
        NEEDworkers(t-1)
    if t == len(NEEDworkersArray):
        res = NEEDnurses(t) + NEEDpollenforagers(t)
        NEEDworkersArray = np.append(NEEDworkersArray, res)
    return NEEDworkersArray[t]
```

In [29]:

```
def RATIOworkforce(t):
    # (27)
    global RATIOworkforceArray
    if t > len(RATIOworkforceArray):
        RATIOworkforce(t-1)
    if t == len(RATIOworkforceArray):
        res = min(BEESadult(t) * (1 - FACTORothertasks) / (NEEDworkers(t) + 1), 1)
        RATIOworkforceArray = np.append(RATIOworkforceArray, res)
    return RATIOworkforceArray[t]
```

2.6 Modeling the regulation of nursing

Theses functions models the need of nurses, the actual number of nurses for the day as well as the nursing quality.

In [30]:

```
def NEEDnurses(t):
    # (28)
    global NEEDnursesArray
    if t > len(NEEDnursesArray):
        NEEDnurses(t-1)
    if t == len(NEEDnursesArray):
        res = 0
        for i in range(1, LIFESPANlarva):
            res += LARVAE(i, t) * NEEDnurses_per_larva[i-1]
        res += CELLSeggs(t) * NEEDnurses_per_egg + CELLSpupae(t) * NEEDnurses_per_pupa
        NEEDnursesArray = np.append(NEEDnursesArray,res)
    return NEEDnursesArray[t]
```

In [31]:

```
def NURSES(t):
    # (29)
    global NursesArray
    if t > len(NursesArray):
        NURSES(t-1)
    if t==len(NursesArray):
        res = NEEDnurses(t) * RATIOworkforce(t)
        NursesArray = np.append(NursesArray,res)
    return NursesArray[t]
```

In [32]:

```
def INDEXnursingquality(t):
    # (30)
    global INDEXnursingqualityArray
    if t>len(INDEXnursingqualityArray):
        INDEXnursingquality(t-1)
    if t==len(INDEXnursingqualityArray):
        res = NURSES(t)/(NEEDnurses(t) + 1)
        INDEXnursingqualityArray = np.append(INDEXnursingqualityArray,res)
    return INDEXnursingqualityArray[t]
```

2.7 Modeling the regulation of foraging

We use these two functions in order to know the number of potentials forager bees and the actual numbers of foragers.

In [33]:

```
def FORAGERS(t):
    # (31) represents the available workforce for the foraging task.
    global FORAGERSArray
    if t>len(FORAGERSArray):
        FORAGERS(t-1)
    if t==len(FORAGERSArray):
        res = FORAGERSpollen(t)+FORAGERSnectar(t)
        FORAGERSArray = np.append(FORAGERSArray,res)
    return FORAGERSArray[t]
```

In [34]:

```
def FORAGERSactive(t):
    # (32) represents the actual number of foragers that fly out
    global FORAGERSactiveArray
    if t>len(FORAGERSactiveArray):
        FORAGERSactive(t-1)
    if t==len(FORAGERSactiveArray):
        res = FORAGERSpollenactive(t)+FORAGERSnectaractive(t)
        FORAGERSactiveArray = np.append(FORAGERSactiveArray,res)
    return FORAGERSactiveArray[t]
```

2.7.1 Recruitment of pollen foragers

Equations (33), (34), (35) and (36) calculate the necessary pollen for the day by calculating the sum of necessary pollen for the larvae and the necessary pollen for the adults.

In [35]:

```
def NEEDpollen(t):
    # (33) calculate the colony’s pollen demand
    global NEEDpollenArray
    if t>len(NEEDpollenArray):
        NEEDpollen(t-1)
    if t==len(NEEDpollenArray):
        res = NEEDpollen_larvae(t)+NEEDpollen_adult(t)
        NEEDpollenArray = np.append(NEEDpollenArray,res)
    return NEEDpollenArray[t]
```

In [36]:

```
def NEEDpollen_larvae(t):
    # (34) represents the pollen demand of larvae of all ages
    global NEEDpollen_larvaeArray
    if t>len(NEEDpollen_larvaeArray):
        NEEDpollen_larvae(t-1)
    if t==len(NEEDpollen_larvaeArray):
        result = 0
        for i in range(1,LIFESPANlarva):
            result+= (POLLENNEEDlarva[i-1]*(LARVAE(i,t)))
        NEEDpollen_larvaeArray = np.append(NEEDpollen_larvaeArray,result)
    return NEEDpollen_larvaeArray[t]
```

In [37]:

```
def NEEDpollen_adult(t):
    # (35) represents the adult’s pollen demand
    global NEEDpollen_adultArray
    if t>len(NEEDpollen_adultArray):
        NEEDpollen_adult(t-1)
    if t==len(NEEDpollen_adultArray):
        result = BEESadult(t) * POLLENNEEDadult + NURSES(t) * POLLENNEEDnurse
        NEEDpollen_adultArray = np.append(NEEDpollen_adultArray,result)
    return NEEDpollen_adultArray[t]
```

In [38]:

```
def NEEDpollenincome(t):
    # (36) represents the daily need for pollen income
    global NEEDpollenincomeArray
    if t>len(NEEDpollenincomeArray):
        NEEDpollenincome(t-1)
    if t==len(NEEDpollenincomeArray):
        S = 0
        if t==0:
            S = 0
        elif t==1:
            S += NEEDpollen(t)
            S += NEEDpollen(t-1)
        else:
            S += NEEDpollen(t)
            S += NEEDpollen(t-1)
            S += NEEDpollen(t-2)
        tmp = S/3 * FACTORpollenstorage - STORESpollen(t)
        result = max(0,tmp)
        NEEDpollenincomeArray = np.append(NEEDpollenincomeArray,result)
    return NEEDpollenincomeArray[t]
```

From the four equations above, we can now calculate the number of pollen forager bees needed today and the potential number of pollen foragers as well as the actual number.

In [39]:

```
def NEEDpollenforagers(t):
    # (37) models the number of pollen foragers needed according to the current need for additional poll
    global NEEDpollenforagerArray
    if t>len(NEEDpollenforagerArray):
        NEEDpollenforagers(t-1)
    if t==len(NEEDpollenforagerArray):
        if t==0:
            result = NEEDpollenincome(0) / (LOADpollenforager * TURNSpollenforager * FACTORforagingsucces
        else:
            result = NEEDpollenincome(t-1) / (LOADpollenforager * TURNSpollenforager * FACTORforagingsucc
        NEEDpollenforagerArray = np.append(NEEDpollenforagerArray,result)
    return NEEDpollenforagerArray[t]
```

In [40]:

```
def FORAGERSpollen(t):
    # (38) models the potential number of pollen foragers each day
    global FORAGERSpollenArray
    if t>len(FORAGERSpollenArray):
        FORAGERSpollen(t-1)
    if t==len(FORAGERSpollenArray):
        max1 = NEEDpollenforagers(t) * RATIOworkforce(t)
        max2 = (BEESadult(t)-NURSES(t))*FACTORminpollenforagers
        min1 = max(max1,max2)
        min2 = BEESadult(t) * FACTORforagingmax
        result = min(min1,min2)
        FORAGERSpollenArray = np.append(FORAGERSpollenArray,result)
    return FORAGERSpollenArray[t]
```

In [41]:

```
def FORAGERSpollenactive(t):
    # (39) models the number of foragers that actually leave the hive for foraging flights
    global FORAGERSpollenactiveArray
    if t>len(FORAGERSpollenactiveArray):
        FORAGERSpollenactive(t-1)
    if t==len(FORAGERSpollenactiveArray):
        result = FORAGERSpollen(t) * INDEXflight(t) * INDEXpollenoutside(t)
        FORAGERSpollenactiveArray = np.append(FORAGERSpollenactiveArray,result)
    return FORAGERSpollenactiveArray[t]
```

2.7.2 Recruitment of nectar foragers

Equations (40), (41) and (42) calculate the necessary nectar for the day by calculating the sum of necessary nectar for the larvae and the necessary nectar for the adults.

In [42]:

```
def NEEDnectar(t):
    # (40) model the demand for nectar
    global NEEDnectarArray
    if t>len(NEEDnectarArray):
        NEEDnectar(t-1)
    if t==len(NEEDnectarArray):
        result = NEEDnectar_larvae(t) + NEEDnectar_adult(t)
        NEEDnectarArray = np.append(NEEDnectarArray,result)
    return NEEDnectarArray[t]
```

In [43]:

```
def NEEDnectar_larvae(t):
    # (41) model the demand for larvae nectar
    global NEEDnectar_larvaeArray
    if t>len(NEEDnectar_larvaeArray):
        NEEDnectar_larvae(t-1)
    if t==len(NEEDnectar_larvaeArray):
        result = 0
        for i in range(1,LIFESPANlarva):
            result += NECTARNEEDlarva[i-1] * LARVAE(i,t)
        NEEDnectar_larvaeArray = np.append(NEEDnectar_larvaeArray,result)
    return NEEDnectar_larvaeArray[t]
```

In [44]:

```
def NEEDnectar_adult(t):
    # (42)model the demand for adult nectar
    global NEEDnectar_adultArray
    if t>len(NEEDnectar_adultArray):
        NEEDnectar_adult(t-1)
    if t==len(NEEDnectar_adultArray):
        result = BEESadult(t) * NECTARNEEDadult + NURSES(t) * NECTARNEEDnurse + FORAGERSactive(t) * NECTAF
        NEEDnectar_adultArray = np.append(NEEDnectar_adultArray,result)
    return NEEDnectar_adultArray[t]
```

This equation take the number of adults bees that are not occupied and computes the number of bees that will work in nectar tasks.

In [45]:

```
def WORKFORCEnectar(t):
    # (43) calculates the number of adult bees that are not involved in other tasks and thus are still a
    global WORKFORCEnectarArray
    if t>len(WORKFORCEnectarArray):
        WORKFORCEnectar(t-1)
    if t==len(WORKFORCEnectarArray):
        result = 0
        if (RATIOworkforce(t) == 1):
            result = (BEESadult(t) * (1 - FACTORothertasks)) - NURSES(t) - FORAGERSpollen(t)
        WORKFORCEnectarArray = np.append(WORKFORCEnectarArray,result)
    return WORKFORCEnectarArray[t]
```

From the equations (42) to (45), we can now model the number of potential nectar foragers as well as the actual number.

In [46]:

```
def FORAGERSnectar(t):
    # (44) models the number of potential nectar foragers
    global FORAGERSnectarArray
    if t>len(FORAGERSnectarArray):
        FORAGERSnectar(t-1)
    if t==len(FORAGERSnectarArray):
        min1 = (BEESadult(t) * FACTORforagingmax) - FORAGERSpollen(t)
        min2 = WORKFORCEnectar(t) - PROCESSORS(t)
        result = min(min1,min2)
        FORAGERSnectarArray = np.append(FORAGERSnectarArray,result)
```

```
    return FORAGERSnectarArray[t]
```

In [47]:

```
def FORAGERSnectaractive(t):
    # (45) models the number of nectar foragers that actually leave the hive for foraging flights
    global FORAGERSnectaractiveArray
    if t>len(FORAGERSnectaractiveArray):
        FORAGERSnectaractive(t-1)
    if t==len(FORAGERSnectaractiveArray):
        result = FORAGERSnectar(t) * INDEXflight(t) * INDEXnectaroutside(t)
        FORAGERSnectaractiveArray = np.append(FORAGERSnectaractiveArray,result)
    return FORAGERSnectaractiveArray[t]
```

2.8 Modeling the resource influx into the colony

As the parts above calculates the number of active pollen and nectar foragers, we can now project the daily influx of pollen and nectar.

In [48]:

```
def INCOMEpollen(t):
    # (46) project the daily influx of pollen
    global INCOMEpollenArray
    if t>len(INCOMEpollenArray):
        INCOMEpollen(t-1)
    if t==len(INCOMEpollenArray):
        result = FORAGERSpollenactive(t) * LOADpollenforager * TURNSpollenforager * FACTORforagingstoch(t
        INCOMEpollenArray = np.append(INCOMEpollenArray,result)
    return INCOMEpollenArray[t]
```

In [49]:

```
def FACTORforagingstoch(t):
    # (47) used to vary the daily foraging success symmetrically around 1
    result = random.uniform(0.75, 1.25)
    return(result)
```

In [50]:

```
def INDEXpollensituation(t):
    # (48) describes the level of the pollen stores in relation to the demand situation of the colony
    global INDEXpollensituationArray
    if t>len(INDEXpollensituationArray):
        INDEXpollensituation(t-1)
    if t==len(INDEXpollensituationArray):
        min1 = STORESpollen(t) / (NEEDpollen(t) * FACTORpollenstorage + 1)
        result = min(1,min1)
        INDEXpollensituationArray = np.append(INDEXpollensituationArray,result)
    return INDEXpollensituationArray[t]
```

In [51]:

```
def INCOMEnectar(t):
    # (49) project the daily influx of nectar
    global INCOMEnectarArray
    if t>len(INCOMEnectarArray):
        INCOMEnectar(t-1)
    if t==len(INCOMEnectarArray):
        min1 = FORAGERSnectaractive(t) * LOADnectarforager * TURNSnectarforager * FACTORforagingstoch(t)
        if t == 0:
            min2 = CELLSempty(0)
        else:
            min2 = CELLSempty(t-1)
        result = min(min1,min2)
        INCOMEnectarArray = np.append(INCOMEnectarArray,result)
    return INCOMEnectarArray[t]
```

2.9 Regulation of food processing

The number of nectar processing bees and foraging bees should normally be in balance. However, nectar processing has priority over nectar treatment so that it is always available for foraging.

In [52]:

```
def NEEDprocessors(t):
    #(50)
    global NEEDprocessorsArray
    if t>len(NEEDprocessorsArray):
        NEEDprocessors(t-1)
    if t==len(NEEDprocessorsArray):
        if t==0:
            result = STORESnectar(0)*ProcessorsPerCell
```

```
        else:
            result = STORESnectar(t-1)*ProcessorsPerCell
        NEEDprocessorsArray = np.append(NEEDprocessorsArray,result)
    return NEEDprocessorsArray[t]
```

In [53]:

```
def PROCESSORS(t):
    #(51)
    global PROCESSORSArray
    if t>len(PROCESSORSArray):
        PROCESSORS(t-1)
    if t==len(PROCESSORSArray):
        result = min(NEEDprocessors(t), WORKFORCEnectar(t))
        PROCESSORSArray = np.append(PROCESSORSArray,result)
    return PROCESSORSArray[t]
```

In [54]:

```
def PROCESSEDnectar(t):
    #(52)
    global PROCESSEDnectarArray
    if t>len(PROCESSEDnectarArray):
        PROCESSEDnectar(t-1)
    if t==len(PROCESSEDnectarArray):
        if t==0:
            result = min(STORESnectar(0)-USAGEnectar(t), PROCESSORS(t)/ProcessorsPerCell)
        else:
            result = min(STORESnectar(t-1)-USAGEnectar(t), PROCESSORS(t)/ProcessorsPerCell)
        PROCESSEDnectarArray = np.append(PROCESSEDnectarArray,result)
    return PROCESSEDnectarArray[t]
```

The system proposed by the model is simplified compared to what exists in nature. Here we use only the nectar stored one day before.

2.10 Management of nutrient stores

The daily usage (consumption) of pollen is highly dependent on the amount of pollen stored. When it is too low the bees have the possibility to reduce their consumption and feed less of their brood.

In [55]:

```
def USAGEpollen(t):
    #(53)
    global USAGEpollenArray
    if t>len(USAGEpollenArray):
        USAGEpollen(t-1)
    if t==len(USAGEpollenArray):
        if t==0:
            result = min(STORESpollen(t), NEEDpollen(0)*(1-(FACTORpollensavingmax*(1-INDEXpollensituation
        else:
            result = min(STORESpollen(t-1), NEEDpollen(t-1)*(1-(FACTORpollensavingmax*(1-INDEXpollensitua
        USAGEpollenArray = np.append(USAGEpollenArray,result)
    return USAGEpollenArray[t]
```

The daily amount of stored pollen is expressed as below.

In [56]:

```
def STORESpollen(t):
    #(54)
    global STORESpollenArray
    if t>len(STORESpollenArray):
        STORESpollen(t-1)
    if t==len(STORESpollenArray):
        if t==0:
            result = INITpollen
        else:
            result = STORESpollen(t-1) + INCOMEpollen(t) - USAGEpollen(t)
        STORESpollenArray = np.append(STORESpollenArray,result)
    return STORESpollenArray[t]
```

The use of pollen and nectar also depends on the storage situation. The bees first favour the consumption of fresh nectar and then pick up the stored nectar. In winter the bees are more likely to draw from stored nectar.

In [57]:

```
def USAGEnectar(t):
    #(55)
    global USAGEnectarArray
    if t>len(USAGEnectarArray):
        USAGEnectar(t-1)
    if t==len(USAGEnectarArray):
        if t==0:
            result = min(STORESnectar(0), NEEDnectar(0))
```

```
        else:
            result = min(STORESnectar(t-1), NEEDnectar(t))
        USAGEnectarArray = np.append(USAGEnectarArray,result)
    return USAGEnectarArray[t]
```

In [58]:

```
def USAGEhoney(t):
    #(56)
    global USAGEhoneyArray
    if t>len(USAGEhoneyArray):
        USAGEhoney(t-1)
    if t==len(USAGEhoneyArray):
        if t==0:
            result = min(STOREShoney(0), (NEEDnectar(t)-USAGEnectar(t))*RATIOnectar_to_honey)
        else:
            result = min(STOREShoney(t-1), (NEEDnectar(t)-USAGEnectar(t))*RATIOnectar_to_honey)
        USAGEhoneyArray = np.append(USAGEhoneyArray,result)
    return USAGEhoneyArray[t]
```

From the equations (49), (52), (55) and (56) we can deduce the 2 following functions.

In [59]:

```
def STORESnectar(t):
    #(57)
    global STORESnectarArray
    if t>len(STORESnectarArray):
        STORESnectar(t-1)
    if t==len(STORESnectarArray):
        if t==0:
            result = INITnectar
        else:
            result = STORESnectar(t-1)+INCOMEnectar(t) - USAGEnectar(t)-PROCESSEDnectar(t)
        STORESnectarArray = np.append(STORESnectarArray,result)
    return STORESnectarArray[t]
```

In [60]:

```
def STOREShoney(t):
    #(58)
    global STOREShoneyArray
    if t>len(STOREShoneyArray):
        STOREShoneyArray(t-1)
    if t==len(STOREShoneyArray):
        if t==0:
            result = INIThoney
        else:
            result = STOREShoney(t-1)-USAGEhoney(t)+(PROCESSEDnectar(t)*RATIOnectar_to_honey)
        STOREShoneyArray = np.append(STOREShoneyArray,result)
    return STOREShoneyArray[t]
```

The weight of the hive is given in kg. To obtain it, the weight of each of the elements of the hive must be added together.

In [61]:

```
def WEIGHTcolony(t):
    #(59)
    global WEIGHTcolonyArray
    if t>len(WEIGHTcolonyArray):
        WEIGHTcolony(t-1)
    if t==len(WEIGHTcolonyArray):
        if t==0:
            result = INITWEIGHTcolony
        else:
            res = 0
            for i in range(1, LIFESPANlarva):
                res = res + (w_larva[i-1]*LARVAE(i,t))

            result = (1/1000)*(w_hivebase+w_cellsbase*CELLShive0+w_pollen*STORESpollen(t)
                    +w_nectar*STORESnectar(t)
                    +w_honey*STOREShoney(t)+w_egg*CELLSeggs(t)
                    +w_pupa*CELLSpupae(t)
                    +res
                    +w_adult*BEESadult(t))
        WEIGHTcolonyArray = np.append(WEIGHTcolonyArray,result)
    return WEIGHTcolonyArray[t]
```

A number of bees are sometimes not necessary because of the weather or the fact that the cells of the hive are already filled.

In [62]:

```
def BEESlazy(t):
    #(60)
    global BEESlazyArray
```

```
    if t>len(BEESlazyArray):
        BEESlazy(t-1)
    if t==len(BEESlazyArray):
        result = (BEESadult(t)*(1-FACTORothertasks))-FORAGERSactive(t)-NURSES(t)-PROCESSORS(t)
        BEESlazyArray = np.append(BEESlazyArray,result)
    return BEESlazyArray[t]
```

3. Running the model

3.1. Execution

In [63]:

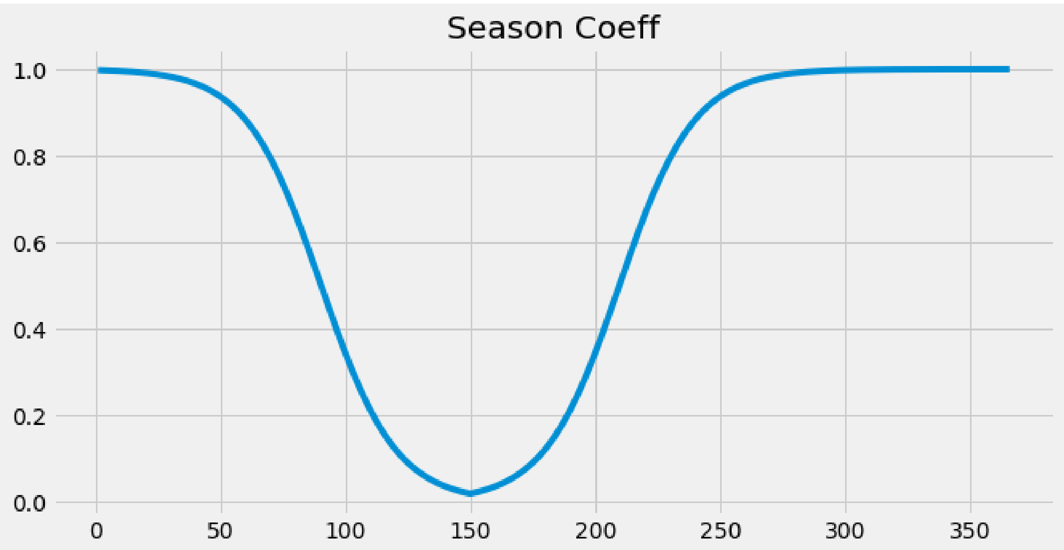
```
def Main(daysEnd, swarm = 0):
    t = np.arange(daysEnd+1)
    for i in range(len(t)):
        CELLSeggs(t[i])
        CELLSlarvae(t[i])
        CELLSpupae(t[i])
        CELLSbrood(t[i])
        BEESadult(t[i])
        STORESpollen(t[i])
        STORESnectar(t[i])
        STOREShoney(t[i])
        WEIGHTcolony(t[i])
    dfResult = pd.DataFrame({'STORESpollen' : STORESpollenArray, 'STOREShoney' : STOREShoneyArray, 'STORE

    if swarm==1 :
        dfResult = pd.DataFrame({'STORESpollen' : STORESpollenArray, 'STOREShoney' : STOREShoneyArray, 'S
    if swarm==2 :
        dfResult = pd.DataFrame({'STORESpollen' : STORESpollenArray, 'STOREShoney' : STOREShoneyArray, 'S

    return dfResult
```

In [64]:

```
import matplotlib.pyplot as plt
import numpy as np

graph = []

for i in range(366):
    graph.append(season(i))

plt.figure(figsize=(10,5))
plt.plot(np.linspace(1,366, 366), graph)
plt.title("Season Coeff")
```

Out[64]:

```
Text(0.5, 1.0, 'Season Coeff')
```



In [65]:

```
res = Main(365)
res.to_csv("out.csv")
```

In [66]:

```
res.describe()
```

Out[66]:

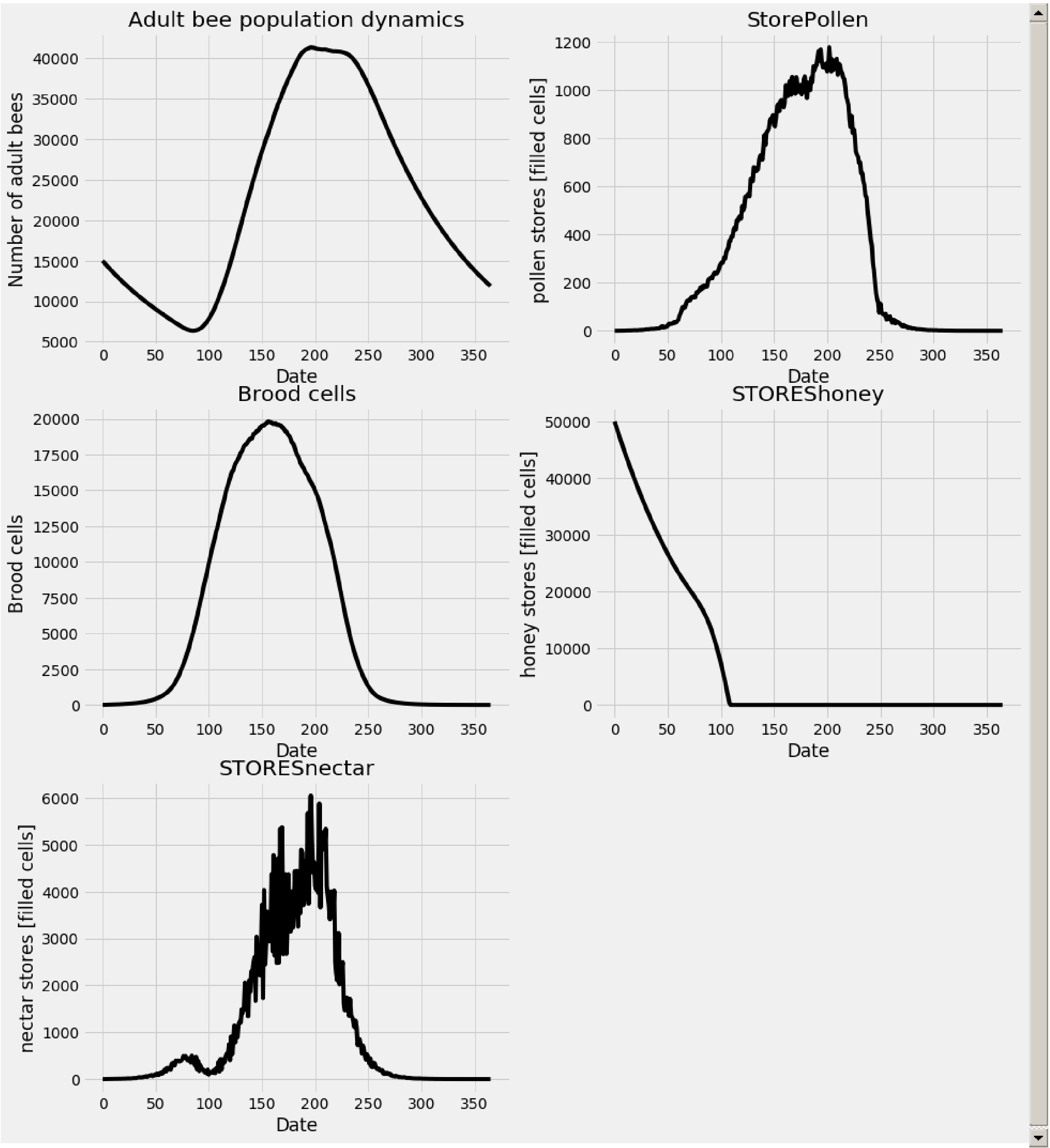
	STORESpollen	STOREShoney	STORESnectar	BroodCells	BEESadult	WEIGHTcolony
count	366.000000	366.000000	366.000000	366.000000	366.000000	366.000000
mean	344.433933	7783.004222	1082.233303	6278.864438	22493.639183	30.554842
std	413.517771	13800.481374	1578.367246	7491.510127	12099.437471	6.003641
min	0.000000	0.000000	0.000000	0.000000	6356.997058	24.437292
25%	2.744307	0.000000	11.274622	54.380124	11919.524709	26.040071
50%	103.775434	0.000000	204.379832	1354.190675	19800.311741	28.976764
75%	719.513685	13141.725848	1700.443950	13934.682892	34514.744677	31.658624
max	1177.391908	50000.000000	6046.838197	19811.580365	41343.212158	50.000000

In [67]:

```
plt.figure(figsize=(14,17))

t = np.arange(366)

plt.subplot(3,2,1)
#plt.legend(loc = 'upper right')
plt.plot(t, BEESadultArray,color = 'black')
plt.title('Adult bee population dynamics')
plt.xlabel('Date')
plt.ylabel('Number of adult bees')


plt.subplot(3,2,2)
plt.plot(t, STORESpollenArray,color = 'black')
plt.title('StorePollen')
plt.xlabel('Date')
plt.ylabel('pollen stores [filled cells]')

plt.subplot(3,2,3)
plt.plot(t,CELLSbroodArray,color  = 'black')
plt.title('Brood cells')
plt.xlabel('Date')
plt.ylabel('Brood cells')

plt.subplot(3,2,4)
plt.plot(t, STOREShoneyArray,color = 'black')
plt.title('STOREShoney')
plt.xlabel('Date')
plt.ylabel('honey stores [filled cells]')

plt.subplot(3,2,5)
plt.plot(t, STORESnectarArray,color = 'black')
plt.title('STORESnectar')
plt.xlabel('Date')
plt.ylabel('nectar stores [filled cells]')
```

Out[67]:

```
Text(0, 0.5, 'nectar stores [filled cells]')
```



3.2 Interpretation

We decided to interpret the 5 graphs above because they show the primary materials (brood, adults, nectar, honey and pollen).

The graph **"Adult bee population"** shows the number of adult bees in the colony. There is a peak on the 200th day due to the `x4` parameter which represents the day when the queen lays the most eggs. The egg cells hatch a few days later (`x4 = 155`).

At the same time, the graph **"StorePollen"** shows the amount of pollen stored by the bees in the hive. The curve is similar to that of the "Adult bee population" which is normal, the two variables are correlated.

Similarly, the graph **"STORESnectar"** shows that the bees will collect nectar to store it in the cells of the hive.

The nectar and pollen will be used to feed the hive (the larvae, the adult bees) but also 40% of this nectar will be transformed into honey (`RATIOnectar_to_honey`).

The graph **"BROODcells"** represents the children of the colony (eggs -> larvae -> pupae), the peak represents the moment when the queen lays the maximum number of eggs.

"STOREShoney" is supposed to represent the honey stored in the cells of the hive however the graph does not show it, our function is broken.

4. Swarming extension

Modeling the 'mother colony'

In [68]:

```
def FACTORqueen(t):
    #(61)

    if ((swd - 30) <= t and t < (swd - 3)):
        res = 1.2
    elif ((swd - 3) <= t and t < (swd + 14)):
        res = 0
    else:
        res = 1
    return res
```

In [69]:

```
def ELRbase(t):
  # (62)
    if (t<swd):
        res = 2000
    else:
        res = 1200
    return res
```

In [70]:

```
def ELR(t):
    # (5a)
    global ELRArray_swarm
    if t > len(ELRArray_swarm):
        ELR(t-1)
    if t == len(ELRArray_swarm):
        res = ELRbase(t) * (1 + ELRstoch(t)) * (1 - season(t)) * SUPcomb(t) * FACTORqueen(t)
        ELRArray_swarm = np.append(ELRArray_swarm, res)
    return ELRArray_swarm[t]
```

In [71]:

```
def BEESswarm(t):
  # (63)
    global BEESswarmArray

    if t > len(BEESswarmArray):
        BEESswarm(t - 1)
    if t == len(BEESswarmArray):
        if (t == swd):
            res = BEESadult(t-1) * 0.6
        else:
            res = 0

        BEESswarmArray = np.append(BEESswarmArray, res)

    return BEESswarmArray[t]
```

In [72]:

```
def BEESadult(t) :
    # (19a)
```

```
    global BEESadultArray_swarm

    if t>len(BEESadultArray_swarm):
        BEESadult(t-1)
    if t==len(BEESadultArray_swarm):
        if t==0:
            res = INITBEESadult
        else:
            res = (BEESadult(t-1) + PUPAE(LIFESPANpupa, t-1))*(1 - MORTALITYadult(t)) - BEESswarm(t)
        BEESadultArray_swarm = np.append(BEESadultArray_swarm, res)
    return BEESadultArray_swarm[t]
```

In [73]:

```
def TURNSnectarforager(t):
    # (64)
    global TURNSnectarforagerArray

    if t > len(TURNSnectarforagerArray):
        TURNSnectarforager(t - 1)
    if t == len(TURNSnectarforagerArray):
        if t == 0:
            TURNSnectarforagerArray = np.append(TURNSnectarforagerArray, 15 + 7 * ((CELLSempty(0))/(CELLS
        else:
            TURNSnectarforagerArray = np.append(TURNSnectarforagerArray, 15 + 7 * ((CELLSempty(t-1))/(CEL

    return TURNSnectarforagerArray[t]
```

In [74]:

```
def LOSShoney_swarm(t):
    # (65)
    global LOSShoney_swarmArray

    if t > len(LOSShoney_swarmArray):
        LOSShoney_swarm(t - 1)

    if t == len(LOSShoney_swarmArray):
        if t == swd:
            res = min(STOREShoney(t-1),BEESswarm(t)*LOADnectarforager)
        else:
            res = 0
        LOSShoney_swarmArray = np.append(LOSShoney_swarmArray, res)

    return LOSShoney_swarm[t]
```

In [75]:

```
def USAGEhoney(t):
    #(56a)
    global USAGEhoneyArray_swarm

    if t>len(USAGEhoneyArray_swarm):
        USAGEhoney(t-1)
    if t==len(USAGEhoneyArray_swarm):
        if t==0:
            result = min(STOREShoney(0), ((NEEDnectar(t)-USAGEnectar(t))*RATIOnectar_to_honey) + LOSShone
        else:
            result = min(STOREShoney(t-1), ((NEEDnectar(t)-USAGEnectar(t))*RATIOnectar_to_honey) + LOSSho
        USAGEhoneyArray_swarm = np.append(USAGEhoneyArray_swarm,result)
    return USAGEhoneyArray_swarm[t]
```

In [76]:

```
def WEIGHTcolony(t):
    #(59)
    global WEIGHTcolonyArray_swarm
    if t>len(WEIGHTcolonyArray_swarm):
        WEIGHTcolony(t-1)
    if t==len(WEIGHTcolonyArray_swarm):
        if t==0:
            result = INITWEIGHTcolony
        else:
            res = 0
            for i in range(1, LIFESPANlarva):
                res = res + (w_larva[i-1]*LARVAE(i,t))

            result = (1/1000)*(w_hivebase+w_cellsbase*CELLShive0+w_pollen*STORESpollen(t)
                    +w_nectar*STORESnectar(t)
                    +w_honey*STOREShoney(t)+w_egg*CELLSeggs(t)
                    +w_pupa*CELLSpupae(t)
                    +res
```

```
                   +w_adult*BEESadult(t))
        WEIGHTcolonyArray_swarm = np.append(WEIGHTcolonyArray_swarm,result)
    return WEIGHTcolonyArray_swarm[t]
```

In [77]:

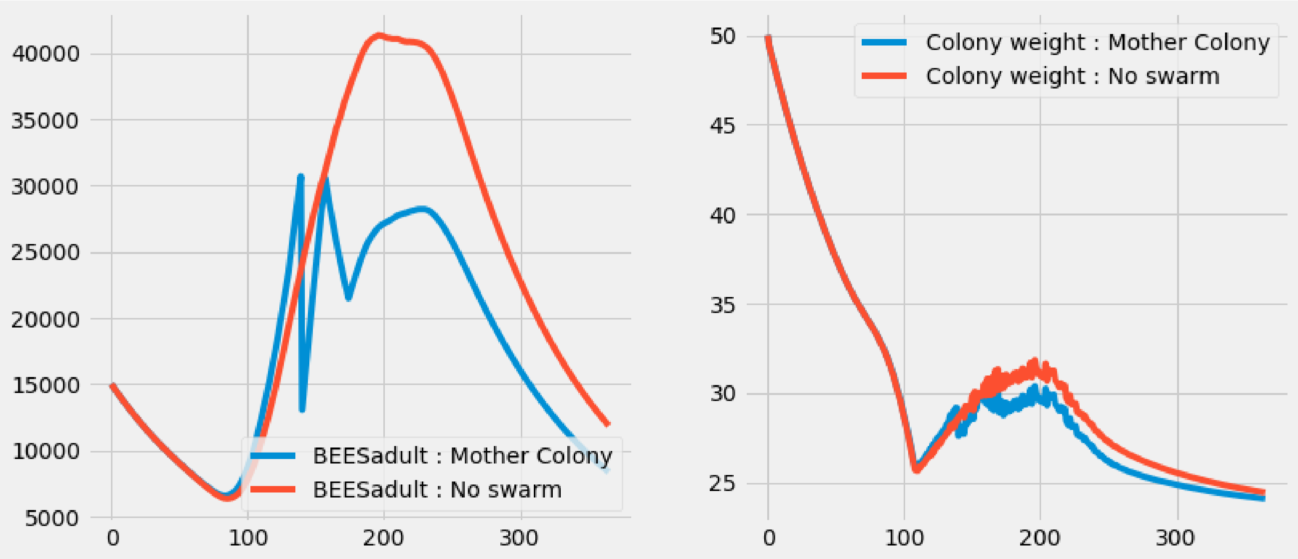
```
res_swarming = Main(365, swarm= 1)
```

In [78]:

```
plt.figure(figsize=(12,12))

plt.subplot(2,2,1)
res_swarming['BEESadult'].plot(label = "BEESadult : Mother Colony")
res['BEESadult'].plot(label = "BEESadult : No swarm")
plt.legend()

plt.subplot(2,2,2)
res_swarming['WEIGHTcolony'].plot(label = "Colony weight : Mother Colony")
res['WEIGHTcolony'].plot(label = "Colony weight : No swarm")
plt.legend()
```

Out[78]:

```
<matplotlib.legend.Legend at 0x7f64857a6450>
```



In red, we have a colony that has high congestion (high laying rate, little comb space) ans does swarm on day t=140

Modeling the 'daughter colony'

In [79]:

```
def FACTORqueen(t):
    #(61a)
    global FACTORqueenArray
    if t > len(FACTORqueenArray):
        FACTORqueen(t-1)
    if t == len(FACTORqueenArray):
        if (t <= swd):
            res = 0
        elif (swd) < t and t <= (swd + 5):
            res = FACTORqueen(t-1) +0.2
        elif t > (swd+5):
            res = 1
        FACTORqueenArray = np.append(FACTORqueenArray, res)
    return FACTORqueenArray[t]
```

In [80]:

```
def NEEDnectarbuilding(t):
    # (66)
    global NEEDnectarbuildingArray

    if t > len(NEEDnectarbuildingArray):
        NEEDnectarbuilding(t-1)
    if t == len(NEEDnectarbuildingArray):
        if t == 0:
            res = 0
        else:
            res = (CELLShive(t) - CELLShive(t-1)) * RATIOnectar_to_wax
        NEEDnectarbuildingArray = np.append(NEEDnectarbuildingArray, res)
    return NEEDnectarbuildingArray[t]
```

In [81]:

```
def NEEDnectar_adult(t):
    # (42a)model the demand for adult nectar
    global NEEDnectar_adultArray_swarm
    if t>len(NEEDnectar_adultArray_swarm):
        NEEDnectar_adult(t-1)
    if t==len(NEEDnectar_adultArray_swarm):
        result = BEESadult(t) * NECTARNEEDadult + NURSES(t) * NECTARNEEDnurse + FORAGERSactive(t) * NECTAF
        NEEDnectar_adultArray_swarm = np.append(NEEDnectar_adultArray_swarm,result)
    return NEEDnectar_adultArray_swarm[t]
```

In [82]:

```
def CELLShive(t):
    # (67)
    global CELLShiveArray
    if t > len(CELLShiveArray):
        CELLShive(t-1)
    if t == len(CELLShiveArray):
        if t <= swd:
            res = 0
        elif swd < t and t <= (swd + 7):
            res = CELLShive(t-1)+(2000/7)
        elif (swd + 7) < t and t <= (swd + 97):
            res = CELLShive(t-1)+(8000/90)
        elif t > (swd + 97):
            res = CELLShive(t-1)
        CELLShiveArray = np.append(CELLShiveArray, res)
    return CELLShiveArray[t]
```

In [83]:

```
def CELLSempty(t):
    # (2a)
    global CELLSemptyArray_swarm
    if t>len(CELLSemptyArray_swarm):
        CELLSempty(t-1)
    if t==len(CELLSemptyArray_swarm):
        res = CELLShive(t) - CELLSbrood(t) - STORESpollen(t) - STORESnectar(t) - STOREShoney(t)
        CELLSemptyArray_swarm = np.append(CELLSemptyArray_swarm, res)
    return CELLSemptyArray_swarm[t]
```

In [84]:

```
def SUPcomb(t):
    # (3a)
    global SUPcombArray_swarm
    if t>len(SUPcombArray_swarm):
        SUPcomb(t-1)
    if t == len(SUPcombArray_swarm):
        if CELLSempty(t)/(CELLShive(t) + 1) < SUPthreshold:
            res = CELLSempty(t)/((CELLShive(t) + 1) * SUPthreshold)
        else:
            res = 1
        SUPcombArray_swarm = np.append(SUPcombArray_swarm, res)
    return SUPcombArray_swarm[t]
```

In [85]:

```
def WEIGHTcolony(t):
    #(59a)
    global WEIGHTcolonyArray_swarm2
    if t>len(WEIGHTcolonyArray_swarm2):
        WEIGHTcolony(t-1)
    if t==len(WEIGHTcolonyArray_swarm2):
        if t==0:
            result = INITWEIGHTcolony
        else:
            res = 0
            for i in range(1, LIFESPANlarva):
                res = res + (w_larva[i-1]*LARVAE(i,t))

            result = (1/1000)*(w_hivebase+w_cellsbase*CELLShive(t)+w_pollen*STORESpollen(t)
                     +w_nectar*STORESnectar(t)
                     +w_honey*STOREShoney(t)+w_egg*CELLSeggs(t)
                     +w_pupa*CELLSpupae(t)
                     +res
                     +w_adult*BEESadult(t))
        WEIGHTcolonyArray_swarm2 = np.append(WEIGHTcolonyArray_swarm2,result)
    return WEIGHTcolonyArray_swarm2[t]
```

In [86]:

```
def TURNSnectarforager(t):
    # (64a)
    global TURNSnectarforagerArray_swarm2

    if t > len(TURNSnectarforagerArray_swarm2):
        TURNSnectarforager(t - 1)
    if t == len(TURNSnectarforagerArray_swarm2):
        if t == 0:
            TURNSnectarforagerArray_swarm2 = np.append(TURNSnectarforagerArray_swarm2, 15 + 7 * (CELLSemp
        else:
            TURNSnectarforagerArray_swarm2 = np.append(TURNSnectarforagerArray_swarm2, 15 + 7 * ((CELLSem
    return TURNSnectarforagerArray_swarm2[t]
```

In [87]:

```
def BEESadult(t):
    # (19b)
    global BEESadultArray_swarm2

    if t>len(BEESadultArray_swarm2):
        BEESadult(t-1)
    if t==len(BEESadultArray_swarm2):
        if t==0:
            res = INITBEESadult
        else:
            res = (BEESadult(t-1) + PUPAE(LIFESPANpupa, t-1))*(1 - MORTALITYadult(t)) - BEESswarm(t)
        BEESadultArray_swarm2 = np.append(BEESadultArray_swarm2, res)
    return BEESadultArray_swarm2[t]
```

In [88]:

```
def ELR(t):
    # (5b)
    global ELRArray_swarm2
    if t > len(ELRArray_swarm2):
        ELR(t-1)
    if t == len(ELRArray_swarm2):
        res = ELRbase(t) * (1 + ELRstoch(t)) * (1 - season(t)) * SUPcomb(t) * FACTORqueen(t)
        ELRArray_swarm2 = np.append(ELRArray_swarm2, res)
    return ELRArray_swarm2[t]
```

In [89]:

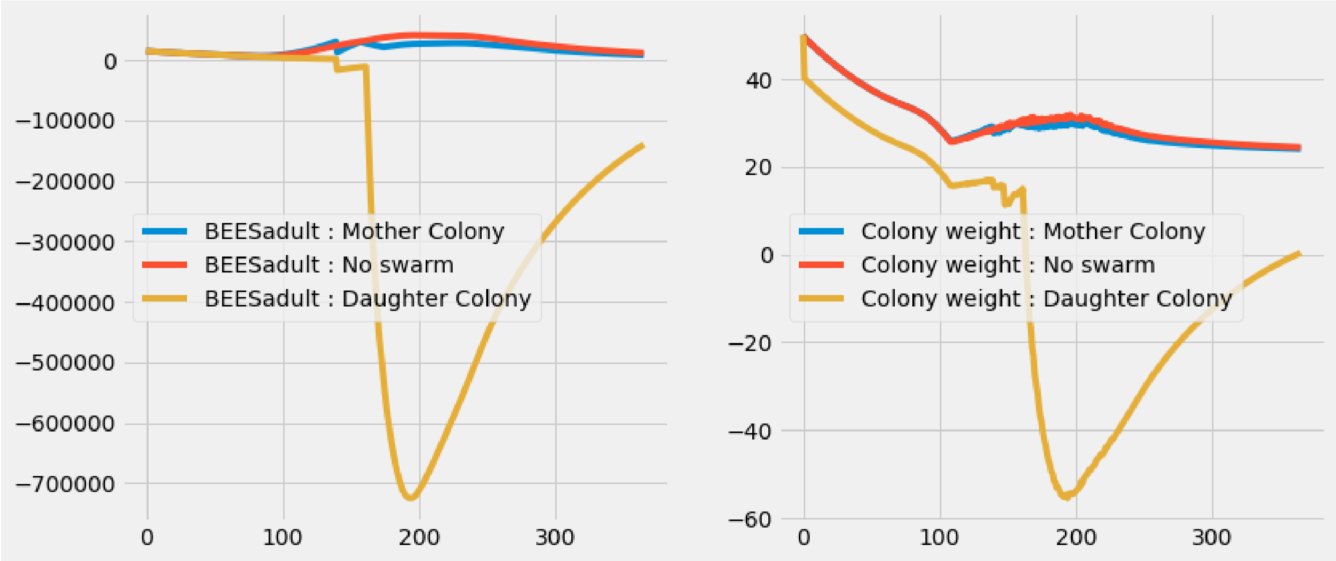
```
res_swarming_2 = Main(365, swarm= 2)
```

In [90]:

```
plt.figure(figsize=(12,12))

plt.subplot(2,2,1)
res_swarming['BEESadult'].plot(label = "BEESadult : Mother Colony")
res['BEESadult'].plot(label = "BEESadult : No swarm")
res_swarming_2['BEESadult'].plot(label = "BEESadult : Daughter Colony")
plt.legend()

plt.subplot(2,2,2)
res_swarming['WEIGHTcolony'].plot(label = "Colony weight : Mother Colony")
res['WEIGHTcolony'].plot(label = "Colony weight : No swarm")
res_swarming_2['WEIGHTcolony'].plot(label = "Colony weight : Daughter Colony")
plt.legend()
```

Out[90]:

```
<matplotlib.legend.Legend at 0x7f64856c6090>
```



We have a problem in this example: Daughter colony