Population Dynamics

2.1 Learning outcomes

Mathematics and Physics

- Iterative systems.
- Units and dimensions.

Computing and Python

- Using classes to organise code and re-use it.
- Making plots and saving them.

Preparation for assignment

- Write programs which generate specific output.
- Debug computer programs.

2.2 Introduction

In biology, an important question is how population numbers of species change over time. There are clearly many factors which play a role in this: both birth and death are influenced by the interaction with other species, by the presence or absence of food, by genetic and by other environmental factors. The branch of science that studies the composition of populations is called *population dynamics*. The mathematical models which it produces are useful for ecological or agricultural purposes. Related models can also be used to determine how diseases develop or how bacteria grow, which is useful for e.g. the pharmaceutical industry.

There are many different mathematical models available in the literature, which each try to capture different aspects of population dynamics. Some models focus on the interaction between predators (the animals that eat others) and preys (the ones that are being eaten). Other models focus on the influence of environmental factors. Some models look at population numbers from one year to the next, others try to follow them in a more fine-grained way. Because of the fact that there are typically many factors at play, predicting how a population evolves over time is a difficult task. Population dynamics forms a classic example of mathematical modelling, where understanding modelling assumptions and their consequences are as important as writing down equations and solving them.

In the present chapter we will start by looking at some very simple systems involving just a single species. Mathematically the models are just iteration models, described by finite difference equations. In later chapters we will return to this problem and study progressively more complicated and more realistic models, involving more than one species which evolve continuously in time (chapter 4), leading to (systems of) ordinary differential equations, or species which migrate, leading to partial differential equations. While all these models are very simple, they do have non-trivial properties, sometimes exhibiting chaos.

From a programming point of view, we will start with very simple programs, but we will then introduce the concept of classes which allow one to collect variables and functions together and to reuse code easily.

2.3 A simple rabbit population model

The simplest population dynamics model consists in stating that the number of rabbits in year t, N_t, is proportional to the number of rabbits the previous year, N_{t-1}:

$$N_t = RN_{t-1}, \tag{2.1}$$

where R is a constant. The rabbit population will increase when new rabbits are born. If the average number of bunnies born annually per rabbit is r^+, the total number of rabbits after 1 year will be R^+N with $R^+ = 1 + r^+$. The rabbit population will decrease when some rabbits die. If the average life time of rabbits is λ, the total number of rabbits still alive after 1 year will be R^-N with $R^- = 1 - \lambda^{-1}$. The two effects are simultaneous and so we must take $R = 1 + r^+ - \lambda^{-1}$. Notice that for $R > 1$, the rabbit population increases (the birth rate exceeds the dead rate) while if $R < 1$ the rabbits population decreases (the death rate exceeds the birth rate).

This equation can be easily solved by hand (see extra questions 2.9). However, to prepare for more complicated models, where you typically cannot solve for the population N_t at an arbitrary t, we would like to solve this equation numerically. This is done in the following code.

file: pop1.py

```python
N = 2 # initial population
R = 3 # each couple of rabbits has 4 youngs every year on average

for t in range(51): # Compute population for 50 years
   print(t, N)
   N = R*N          # Population the next year
```

The variable `t` counts the years, `N` keeps track of the current rabbit population, and `R` is the average rate of population increase: the number by which the population will increase every year. In our program, we assume that each rabbit couple has 4 youngs every year (*i.e.* 2 each), so that $R = 3$ because the parents remain alive and no rabbits die.

Problem 2.1:

Run the program `pop1.py`. Unsurprisingly, the population increases very quickly.

Printing values on the screen can be useful, but this is not always easy to interpret. Instead we usually prefer to generate a figure of N_t. This can be done as follows.

file: pop2.py

```python
import matplotlib.pyplot as plt

t_list = []                      # List of t values
N_list = []                      # List of N values
R = 3                            # Each couple of rabbits has 4 youngs on
     average
N = 2                            # The initial population.

for t in range(51):              # Compute population for 50 years
  t_list.append(t)
  N_list.append(N)
  N = R*N                        # Population the next year

plt.plot(t_list, N_list, "b-") # Graph of the population
plt.ylabel('$N_{t}$')            # Add axis labels
plt.xlabel('$t$')
plt.show()                       # Diplay the figure on screen
```

The program is very much the same as the original one, except that we use 2 lists, `t_list` and `N_list` to store all the values of t and N. We can then use the `plot` function from the `matplotlib.pyplot` module to generate a figure.

The figure generated by `pop2.py` is not easy to read because it grows very quickly. When this happens, it is usually more useful to plot the population N_t using a logarithmic scale. This can fortunately be done very simply in Python:

Problem 2.2:

Copy the program `pop2.py` into a file called `pop2b.py` and replace the function `plot` by the function `semilogy`, then run `pop2b.py`. Do you understand the resulting figure?

semilogy

For information, `pyplot` also has a function called `semilogx` to use a logarithmic scale on the x axis and `loglog` to use a logarithmic scale for both axes.

semilogx, loglog

Insert the line `plt.savefig("pop2.pdf")` just before `plt.show()` and run the program again. You will now have a `pdf` file called `pop2.pdf` containing your figure. This can easily be included in LaTeX documents or returned as a homework.

savefig

As it is the program stops after 50 years of iteration. Modify it so that it stops either after 50 years or as soon as `N` is larger or equal to 1000 (which ever occurs first).

Check Point 2.1

2.4 Units and dimensions

Notice that in equation (2.1) the quantity N can be expressed in any units. We have implicitly assumed that it counts individual rabbits, but it could just as well count thousands of rabbits or even millions of them. The equation would not change and, in this case, R does not need to change either. The equation is said to be scale invariant. This is an important property for such systems and it can be used to simplify some problems, as we will see later.

Equation (2.1) can be used to describe any population which multiplies at a rate proportional to the population at the time of breeding. This property is satisfied by most living organisms, including unicellular organisms like bacteria.

2.5 A more realistic population model

In our first very simple model, we have assumed that rabbits can multiply indefinitely, but this is obviously not realistic. The fact is that there are environmental factors that limit the growth of populations. The most important one is usually food supplies. Modelling this can be quite complex, but there is a simple model which captures this limit very well: the *Logistic model*, given by

$$N_{t+1} = RN_t\left(1 - \frac{N_t}{K}\right) . \tag{2.2}$$

This is very similar to our first model except that we have added the factor $1 - N_t/K$ to the equation. The first point to notice is that, as N_t must be positive, the factor $1-N_t/K$ must be positive too and as a consequence $N_t < K$. The parameter K is thus a threshold population, called the *carrying capacity*. It describes the maximum population that the environment can potentially sustain.

We noticed before that our first model was scale invariant. If we change the units for N_t in (2.2), the equation changes unless we scale K by the same factor. To make our life simpler, we can count the population in units of K, meaning that we can substitute

$$\hat{N}_t = N_t/K \, , \tag{2.3}$$

which leads to

$$\hat{N}_{t+1} = R\hat{N}_t\left(1 - \hat{N}_t\right) . \tag{2.4}$$

which has the same form as (2.2) but with $K = 1$. This means that, without any loss of generality, we only have one real parameter in the model, R. The second parameter, K, is hidden in the units chosen for the population. Mathematically we do not need to consider it, and we simply solve (2.4) (and drop the hat on the N_t as well for brevity). A biologist would have to remember which units are used.

Problem 2.3:

Copy the program pop2.py into a file called log1a.py and modify it so that it solves equation (2.4). Before you run your code set the initial value of N to 0.01. Then take R=1.2 and compute the solutions for 100 iterations (instead of 50). Replace the plot format 'b-' by 'b.' (*bee* and *dot*) to plot dots instead of continuous lines.

Then:

a) Using a loop, solve the equation for 10 different initial values of N in the range $0 < N < 1$ and create a graph with all these solutions. Take $R = 1.2$. What difference does it make?

b) Do the same, in a file called log1b.py, but this time take 0.5 for the intial value of N and generate a separate graphs for the following values of R: 2.5; 2.8; 3.2; 3.4; 3.5; 3.9.

Check Point 2.2

2.6 The Population class

The two models we have used so far only differ by the right hand side of equation (2.1) and (2.4) which translates into a single line of code in our Python programs. If we want to study a different model it is tempting to make copies of previous programs and modify the few lines that need changing. While this is simple to do in the case we have looked at so far, when problems and programs are more complex, this can be difficult and often leads to errors which are hard to spot.

One way to circumvent this (though certainly not the only way) is to use a programming style called *object oriented programming*, which makes use of *classes* and *objects*. If you are not familiar with this from a previous programming module, please refer to the section *Object Oriented Programming* of chapter 10: Writing code. In the present section we will rewrite the population dynamics code which we have written in the previous sections using an object oriented approach. We will call the class Population and the module containing it population.py.

We already know that different models will differ by the right hand side of equation (2.1) or (2.2) which in general can be written as

$$N_{t+1} = F(N_t)\,. \tag{2.5}$$

Our class must thus contain a function which will compute the right-hand side of this equation. We will call it F(). The idea will now be to write a *base class* Population in which the F() function implements the simple model (2.1), and then construct a subclass of this class which only differs by the function F().

The base class must also keep track of the value of the population N (as a class variable), as well as the growth rate parameter R. It also contains a variable N_list which will contain a list of all the computed values of N which we will use to generate graphs.

It can then contain a generic function iterate which we use to compute the population evolution (for any of the population models). It takes two arguments: the initial population, N0 and the number of iterations to perform.

file: population.py

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

class Population:
  """ A class to compute the time evolution of a population.
  """

  def __init__(self, R=0):
    """ Initialise the class instance
        : param R : population parameter
        : return : (implicitely) an instance of Population
    """
    self.R = R       # Growth rate
    self.N = 0       # Current population
    self.N_list = [] # List of all computed values

  def F(self):
    """ Return the population at the next time.
    """
    return self.R*self.N

  def iterate(self, N0, n=0):
    """ Integrate the logistic equation n times, starting with population
        N0 from

      : param N0 :    initial population.
      : param n :     number of iteration

    self.N : contains the list of N values
    """

    self.N_list = []                # List of populations
    self.N = N0
    for i in range(n):
      self.N_list.append(self.N)
      self.N = self.F()

# This is run only when not importing the module.
if __name__ == "__main__":
  pop = Population(R=1.5)
  n_iter = 100                      # The number of iteration
  pop.iterate(0.1, n_iter)          # Iterate the equation F()
  t_val = range(n_iter)             # Create a list of t values
  plt.plot(t_val, pop.N_list, "r-") # Show N(t) on a graph
  plt.show()
```

Note how details for *users* of this class are documented with docstrings (the comments between 2 sets triple quotes), while details only relevant for people interested in the internals of the class are documented with comments. For the amount of code in this program the documentation may somewhat look like an overkill, but it is good to get used to a proper style from the start. Try to follow it.

Problem 2.4:

Run the program in `population.py`. It generates a figure showing how N evolves with time, showing how it explodes exponentially.

We will now write a program to solve the equation of the logistic model, reusing what we have already done for the simplest model. For this, all we have to do is to generate a new class called `PopulationLogistic` which is a subclass of the class `Population`. We do this in a new file called `population_logistic.py`:

file: `population_logistic.py`

```python
from population import Population
import matplotlib.pyplot as plt

class PopulationLogistic(Population):
  """ A class to compute the time evoluation of a logistic population. """

  def __init__(self, R=0):
    """ Initialise the class instance
        : param R : population parameter
        : return : (implicitely) an instance of Population
    """
    super().__init__(R)       # execute the Population class __init__ function

  # The only function we need to describe a population.
  def F(self):
      """ The scaled logistic population equation
      """
```

```
        return(self.R*self.N*(1-self.N))

# Only run this when not importing the module
if __name__ == "__main__":
  R = 2.5                       # The model parameter
  n_iter = 25                   # The number of iteration
  pop = PopulationLogistic(R)
  pop.iterate(0.1, n_iter)
  t_val = range(n_iter)
  plt.plot(t_val, pop.N_list, "r-", label="R="+str(R))

  plt.title("Logistic model") # Figure title
  plt.xlabel("t")             # Horizontal axis label
  plt.ylabel("N")             # Vertical axis label
  plt.legend()      # Displays the legends/label from the plot function "R
      ='.."
  plt.show()
```

Notice that all we have to change is the function F. At the bottom of the program, We have also improved the figure by adding labels on the 2 axis as well as a figure title.

Problem 2.5:

Run the program population_logistic.py and check that it works.

Problem 2.6:

Make a copy of the program population_logistic.py and call it population_logistic_loop.py. Modify it so that it generates, on the same figure, several curves for different values of R:

- Use a for loop so that R takes the values [2.5, 2.8, 3, 3.2, 3.5, 3.8]. Place it just before the creation of the instance of PopulationLogistic. The loop must include all the lines up to the plot function.
- Remove the format "r-" from the plot function. plot will select a different colour for each curve.

Run the program population_logistic_loop.py and check that it works. Is the result expected?

In the next problem sheet, we will compare the model to data. To use the model we will have to use an unscaled version of the logistic model.

Check Point 2.3

Problem 2.7:

Make a copy of the program population_logistic.py and call it population_real_logistic.py. We want to modify it so that the equation for the model is (2.2) with the saturation parameter K.

For this you need to:

- Change the class name PopulationLogistic to PopulationRealLogistic everywhere it appears.
- Add the parameters K to the __init__ function and give it the default value 1. Add the line self.K = K below the call to the parent initialiser.
- In the class function F, modify the equation so that it correspond to the real logistic equation (2.2).
- Where the instance of the PopulationRealLogistic is created add the value 1000 for K just after the value for R.

Run the program population_real_logistic.py and check that it works. We will use that program again later.

Check Point 2.4

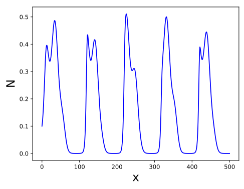
2.7 Extra questions

Problem 2.8:

An important part of modelling consists in writing computer programs. These programs must run correctly. Fixing programming issues is referred to as *debugging* (a term dating back to the 1940s, when actual moths would disturb the working of relays inside computers). The aim of this problem is to train you to find common mistakes in you code. You should first read the first section of chapter 11: Debugging and improving code.

The program `pop_modulate.py` solves a generalisation of the logistic model (2.2) where the parameters are time dependent:

$$N_{t+1} = N_t\Big(R + a\sin(2\pi\nu_1 t)\Big)\left(1 - \frac{N_t}{K + b\sin(2\pi\nu_2 t)}\right) . \tag{2.6}$$



Population profile

It should generate the figure shown in the margin and display the extremum values of N on the console, but unfortunately it contains 8 bugs.

The problem thus consists of correcting these 8 mistakes and annotating them with comments describing what the problem was. We also ask you to number the bugs that you find from 1 to 8, starting from the top. We also ask that the comment starts with the word `BUG` followed by the bug number and then a comment describing what the problem was. For example for the following program

```
for i in [1, 3, 5, 7 , 11]
    b = exp(i)
    v = i * B
```

you should produce something like

```
for i in [1, 3, 5, 7 , 11] : # BUG 1: missing colon ':'
    b = exp(i)
    v = i * b # BUG 2: wrong variable name B should be b
```

Problem 2.9:

Check that for the model (2.1), if one starts with a population of N_0 rabbits, after t years the population is

$$N_t = N_0 R^t . \tag{2.7}$$

Problem 2.10:

In the simplest model above, we have assumed that rabbits live forever. Which value must we take for R if each rabbit lives on average 5 years and if each rabbit couple has on average 4 youngs every year? You can assume a balanced number of male and female rabbits.