Population Dynamics

2.1 Learning outcomes

Mathematics and Physics

- Iterative systems.
- Units and dimensions.
- Phase plots and bifurcations.

Computing and Python

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

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 once species which evolve continuously in time (chapter 3), leading to (systems of) ordinary differential equations, or species which migrate (chapter 4), 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 bunny 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:

Question 2.1:

Check that if one starts with a population of N_0 rabbits, after t years the population is

$$N_t = N_0 R^t \tag{2.2}$$

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

```
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.

Action 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

```
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-") # plot the population
plt.show()                     # diplay the figure
```

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:

Coding task 2.1:

> 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?
>
> 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.
>
> 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.

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.

Homework 2.1:

> 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.

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

$$N_{t+1} = RN_t\left(1 - \frac{N_t}{K}\right) . \tag{2.3}$$

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 (2.3), the equation changes unless we scale K by the same factor. To make our life simpler, we can count the population is units of K, meaning that we can substitute

$$\hat{N}_t = N_t/K\,, \tag{2.4}$$

which leads to

$$\hat{N}_{t+1} = R\hat{N}_t \left(1 - \hat{N}_t\right). \tag{2.5}$$

which has the same form as (2.3) 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 (2.3) (and drop the hat on the N_t as well for brevity). A biologist would have to remember which units are used.

Coding task 2.2:

Copy the program `pop2.py` into a file called `log1.py` and modify it so that it solves equation (2.5).

Take `R=1.2`, set the initial value of N to 0.01 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) Take 10 different initial values for N in the range $0 < N < 1$. What difference does it make?

b) Set the initial value $N = 0.5$ and take successively the following values of R: `2.5; 2.8; 3.2; 3.4; 3.5; 3.9`.

2.6 Bifurcation plots

In the previous section we have seen that for large values of R, the population eventually (for late time) oscillates between 2 or more values. We would like to visualise how the number of values varies with R. To do this we can generate a figure in which, for a range of values of R, we plot a dot for each values taken by N. In order to capture what happens at late time, we should first solve (2.5) a number of times, say n_1, to let it settle. After that, we solve it for a further n_2 steps and plot all the values taken by N during these n_2 steps.

To generate the figures, we will evaluate (2.5) n_1 times followed by another n_2 times for which we will append the values of N into a list, say `N_list`. We can then generate an array or a list, say `R_list`, containing the same number of element as `N_list` but with each value set to the value of R used. Then the command `plt.plot(R_list, N_list, "b.")` will plot a series of dots over the value of R and by repeating this procedure for different values of R we will generate the diagram we are after.

Coding task 2.3:

Write a program to generate the plot described above, for the logistic equation, by proceeding as follows:

- Create a new file called `bifurcation_plot.py` in which you import the modules `matplotlib.pyplot` as `plt` and `numpy` as `np`.
- Create a function `iterate(N0, R, n1, n2)` where `N0`, `R`, `n1` and `n2` are the parameters. The function must
  - Create an empty list called `N_list`.
  - Initialise the variable `N` to the parameter value `N0`.
  - Use a for loop to iterate equation (2.5) `n1+n2` times and update `N` each time. After `n1` steps, append the value of `N` to the list `N_list`.
  - Return the list `N_list`.
- Terminate your program with the following lines


```
for R in np.linspace(0.2, 4, 500):
    N_list = iterate(0.5, R, 500, 1500)
    R_list = np.ones(len(N_list))*R
    plt.plot(R_list, N_list, "b.", markersize=1)

plt.xlabel("R",fontsize=24)
plt.ylabel("N",fontsize=24)
plt.show()
```

The function call `np.linspace(0.2, 4, 500)` generates an array of `500` real numbers between `0.2` and `4` including both bounds. This is similar to the range function except that it generates real numbers and that the upper bound is included as well. `np.ones(len(N_list))` creates an array with the same number of elements as the list `N_list` and sets all its value to `1`. It is then multiplied by `R` so that the array contains the value R repeated `len(N_list)` times. The argument `markersize=1` is there to make the dot on the figure very small.

Homework 2.2:

Run your program `bifurcation_plot.py`. Compare the figure (a so-called *bifurcation plot*) with the numerical results you obtained in coding task 2.2 for the asymptotic behaviour, and explain how they match.

2.7 The Population class

The two models we have used so far only differ by the right hand side of equation (2.1) and (2.5) which translate 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 in `bifurcation_plot.py` which we have written for coding task 2.3 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 (2.3) which in general can be written as

$$N_{t+1} = F(N_t)\,. \tag{2.6}$$

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 can then contain a generic function `iterate` which we use to compute bifurcation plot data (for any of the population models). We will also create a function called `bifurcation_plot` which similarly can generate bifurcation plots for both of the models. Finally we will add a function called `transfer_plot` which will plot the right hand side of the population equation as a function of `N`.

file: `population.py`

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

class Population:
```

```
5   """ A class to compute the time evolution of a population. """
6
7   def __init__(self, R=0):
8     self.R = R  # growth rate
9     self.N = 0  # current population
10
11  def F(self):
12    """ Return the population at the next time. """
13    return self.R*self.N
14
15  def iterate(self, N0, n1, n2):
16    """ Iterate the logistic equation starting at N0 for n1+n2 steps.
17    Return the last n2 values of N in a list.
18
19    :param N0:    initial population.
20    :param n1:    number of iterations to ignore.
21    :param n2:    number of iterations to store.
22    :return:      the n2 values of N as a list.
23    """
24
25    N_list = []  # list of populations
26    self.N = N0
27    for i in range(n1+n2):
28      if i>=n1:
29        N_list.append(self.N)
30      self.N = self.F()
31
32    return N_list
33
34  def bifurcation_plot(self, Rmin, Rmax, NR, N0):
35    """ Display the bifurcation plot.
36
37    :param Rmin: starting value of growth rate.
38    :param Rmax: ending value of growth rate.
39    :param NR:   number of steps along R axis.
40    :param N0:   initial population.
41    """
42
43    for R in np.linspace(Rmin, Rmax, NR):
44      self.R = R
45      N_list = self.iterate(N0, 500, 1500)
46      R_list = np.ones(len(N_list))*R
47      plt.plot(R_list, N_list, "b.", markersize=1)
48    plt.xlabel('$R$')
49    plt.ylabel('asymptotic $N$')
50    plt.show()
51
52  def transfer_plot(self):
53    """ Display the transfer plot of the population. """
54
55    for self.N in np.linspace(0, 1, 200):
56      plt.plot(self.N, self.F(), "b.")
57    plt.xlabel('$N_t$')
58    plt.ylabel('$N_{t+1}$')
59    plt.show()
60
61
62 # Only run this when not importing the module.
63
64 if __name__ == "__main__":
65   pop = Population(R=1.5)
66   pop.transfer_plot()
67   pop.bifurcation_plot(0.2, 1.01, 10, 0.1)
```

Note how details for *users* of this class are documented with docstrings, 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 look a bit like overkill, but it is good to get used to a proper style from the start. Try to follow it.

Action 2.2:

Run the program in `population.py` (for the unconstrained model we have chosen, the figures are not particularly useful).

We must thus use our `population` class for the logistic model by generating a class called `PopulationLogistic` which is a subclass of the class `Population` in a module file called `population_logistic.py`:

file: `population_logistic.py`

```
from population import Population

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

  # The only function we need to describe a population.
  def F(self):
      return(self.R*self.N*(1-self.N))


# Only run this when not importing the module
if __name__ == "__main__":
  pop = PopulationLogistic(2.5)
  print(pop.R)
  pop.transfer_plot()
  pop.bifurcation_plot(0.2, 4, 500, 0.1)
```

Notice that all we have to change is the function `F`. Note that we have left out the docstring for this function; if you ask for it using e.g. `help(PopulationLogistic)` you will be presented with the docstring of `F` in the `Population` class.

Action 2.3:

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

2.8 The Ricker model

The power of classes will become more explicit when we consider a new model. The Ricker model is defined by the equation

$$\hat{N}_{t+1} = \hat{N}_t \exp\left[R\,(1-\hat{N}_t)\right] + Q\,, \tag{2.7}$$

where Q is a parameter describing the spontaneous generation of the population which is independent of the existing population (like an immigration). R is a saturation parameter. In this model the population also saturates, but the saturation value is not easily read off from the above expression. When $\hat{N}$ is very small, the growing rate is e^R, but when $\hat{N}$ is large (comparable to one), the growing rate is very small.

Coding task 2.4:

- Copy the module `population_logistic.py` into a file called `population_ricker.py`.
- Modify the name of the class to `PopulationRicker`. It must remain a subclass of `Population`.
- Give it the following `__init__` function,

  ```
  def __init__(self,Q=0,R=1):
      super().__init__()  # initialise the parent object
      self.Q = Q
      self.R = R
  ```

  at the top of the class definition (before the function `F`).
- Modify the class function `F` so that it describes the Ricker model (2.7).
- Modify the bottom of the program so that it creates a `PopulationRicker` object and generates the transfer plot and bifurcation diagram for $Q = 0.06$.

Homework 2.3:

In the Ricker model, what would be the interpretation of $R < 0$? What property would the species have? What condition can you then derive for $Q/\hat{N}_{\text{stat}}$ where $\hat{N}_{\text{stat}}$ is a constant solution (i.e. $\hat{N}_t = \hat{N}_{t+1} = \hat{N}_{\text{stat}}$ for all t)? What condition can you deduce on Q for steady states to exist?

Homework 2.4:

a) Use the program `population_ricker.py` to generate the bifurcation plots for the Ricker model for $R = 1$ and the following values of Q: 0, 0.04, 0.05, 0.06 and 0.1.

b) What is the smallest value of Q, up to the 3rd decimal, for which the bifurcation diagram has at most 2 values (for $R = 1$)? You can answer the question visually by looking at the figures generated by your program.

2.9 Comparing with data

No modelling project is complete without the final step, in which you compare its predictions with real-world data. Sometimes that is easy: we could easily discard our simplest model (2.1) without looking at population data, because everyone knows rabbit growth is not unbounded. But in most cases, of course, the situation will be less clear.

To see how well the Ricker model describes the real world, we will compare its predictions with some salmon population data provided by [2]. Salmon populations in a river have non-overlapping generations, in the sense that adults leave the river before their young are born. This means that N_{t+1} in our model represents just the total number of newborn (or *recruits*), not the sum of the number of newborn and the number of parents.

A salmon data file is given by `data/LakeTroutMI.csv`. The first few lines of this file read

```
year, recruits, wild, stocked, area
1971, 1.1,      0.2,  20,      MI3
1972, 2.1,      0.5,  21,      MI3
1973, 4.2,      12,   182,     MI3
...
```

The `area` column refers to different geographical areas; we will restrict to a single one in the code below. We can now make a plot of the number of recruits (or N_{t+1}) against the number of stocked (or N_t, also called spawners) by running the following code.

file: `salmon_basic.py`

```
1  import matplotlib.pyplot as plt
2  import numpy as np
3  import pandas as pd
4
5  data = pd.read_csv('data/LakeTroutMI.csv')
6
7  filtered = data[data['area'] == 'MI5'].sort_values(by='stocked')
8
9  recruits = filtered['recruits'].values
10 stocked  = filtered['stocked'].values
11
12 plt.plot(stocked, recruits, 'o')
13 plt.xlabel('stock')
14 plt.ylabel('recruits')
15 plt.show()
```

Here we use the Pandas library [1] to read in the data file, filter by 'area MI5' in line 7, and then plot the recruitment against the stock.

This way of representing the data is not particularly convenient if we want to compare it with the predictions of the Ricker model. For that purpose, it is convenient to rearrange (2.7) as (we will only consider $Q = 0$ here)

$$\log \frac{N_{t+1}}{N_t} = R\left(1 - \frac{N_t}{K}\right), \tag{2.8}$$

where we have re-instated the normalisation constant K. The above relation expresses the left-hand side as a linear function of N_t. This is nice because it enables us fit the Ricker model to the data by using a best-fit straight line.

Coding task 2.5:

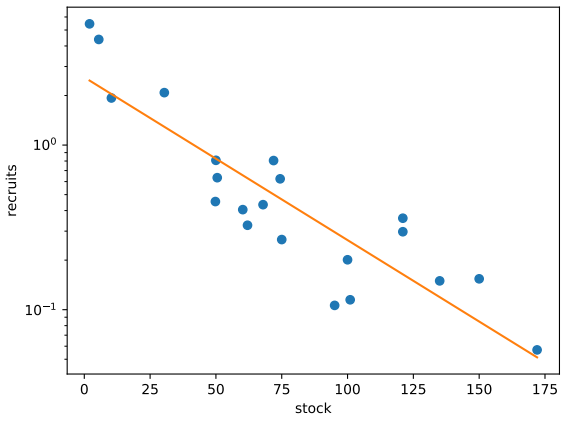
Copy the `salmon_basic.py` program to `salmon_ricker.py` and modify it so that it displays the logarithm of N_{t+1}/N_t versus N_t (use what you learned in coding task 2.1). If the Ricker model has any validity, you should find a roughly linear relation.

In order to fit the coefficients R and $-R/K$ of the Ricker model, use the `np.polyfit` function. It accepts two arrays `x` and `y`. Calling it produces the coefficients for the best linear fit, which you can then use to construct a function which represents this linear fit. Trivial example (with only two data points):

```
coefficients = np.polyfit([1,4], [2,14], 1)
bestfit      = np.poly1d(coefficients)
print(coefficients)
print(bestfit(3))
```

produces `[4, -2]` and `10`, representing a 'fit' $y = 4x - 2$ and $y(x = 3) = 10$ respectively.

Use the above information to extend your program so that it generates a combined plot which contains the data and the linear fit. You should find a figure like the one displayed below.



If you do not see a line, or you get strange results for the fit, think carefully about what `plt.semilogy` does with the data that you give it.

What are the values of R and K for the data set we used here? Add a line to your `salmon_ricker.py` program to display these values.

Run you code so that it uses the data set MI7. What value of R and K do you get and how do you interpret them?

2.10 References

[1] Wes McKinney and PyData Development Team. *Pandas: Python Data Analysis Library*. URL: http://pandas.pydata.org/.

[2] Derek H. Ogle. *R for Fisheries Analyses*. URL: http://derekogle.com/fishR/.

