Continuous time models

4.1 Learning outcomes

Mathematics and Physics

- Ordinary differential equations.
- Systems of ordinary differential equations.
- Euler and 4th order Runge-Kutta method.
- Dimensionless parameters.

Computing and Python

- Using numpy to solve systems of equations.

4.2 Introduction

So far in our population dynamic problems, we have assumed that the population was evolving in steps, monitoring the population every year. For most problems this does not make sense as populations evolve constantly, not in big steps. One thus needs to develop models which describe the continuous evolution of populations. This is quite easy to achieve and the simplest model is an adaptation of our rabbit model,

$$\frac{\mathrm{d}N(t)}{\mathrm{d}t} = \tilde{R}N(t)\,. \tag{4.1}$$

This differential equation is very similar to (2.1). The population N_t of that model, which could be evaluated only at $t = 0, 1, 2, \ldots$, has now been replaced by a population $N(t)$ which is a function of t which is a parameter which can take any real value. The equation states that the rate of change of the population per unit time is $\tilde{R}$. So if t is measured in years, $\tilde{R}$ is the relative increase of rabbit numbers per year. We actually have $\tilde{R} = R - 1 = r^+ - \lambda^{-1}$. The difference with this model is that the population will be evolving smoothly, not in steps. The solution is simply

$$N = N_0 \exp(R\,t)\,, \tag{4.2}$$

where N_0 is the rabbit population at $t = 0$.

4.3 The Schaefer model

The model (4.1) is unrealistic and of very limited interest but there is another simple model (which is used extensively to model fish stock): the Schaefer model [3]. It is defined by the equation

$$\frac{\mathrm{d}N}{\mathrm{d}t} = RN\left(1 - \frac{N}{K}\right) - Y(t)\,. \tag{4.3}$$

This differential equation is very similar to (2.2) except that we have a derivative on the left hand side and we have added a function $Y(t)$. The right hand side describes the variation of the population per unit of time, say per day. R is thus expressed in units per day and K is a saturation population. The function $Y(t)$ describes the amount of fish being caught, per day, and it is assumed to be independent of the population (the fishermen stop catching fish when their boats are full).

Our model contains two parameters, R and K, but these are not all mathematically relevant. By performing the following change of variables,

$$\hat{N} = \frac{N}{K}\,, \qquad \hat{t} = R\,t\,, \qquad \hat{Y}(\hat{t}\,) = \frac{Y(t)}{KR}\,, \tag{4.4}$$

(4.3) reduces to

$$\frac{\mathrm{d}\hat{N}(\hat{t}\,)}{\mathrm{d}\hat{t}} = \hat{N}(\hat{t}\,)\left(1 - \hat{N}(\hat{t}\,)\right) - \hat{Y}(\hat{t}\,). \tag{4.5}$$

This can be shown by direct substitution. Mathematically, all solutions of (4.5) can easily be converted into solutions of (4.3) by performing the inverse change of variables (4.4), so mathematically we only need to analyse equation (4.5). For simplicity, in Python programs we will use variable names without the hats.

When one must solve an equation like (4.5) the first thing to seek is a simple solution. In this case we can try to find constant solutions, *i.e.* solutions for which $\mathrm{d}\hat{N}/\mathrm{d}\hat{t} = 0$, assuming $\hat{Y}(\hat{t}\,)$ to be constant. In this case we have

$$\hat{N} = \frac{1 \pm \sqrt{1 - 4\hat{Y}}}{2}\,. \tag{4.6}$$

We then see that to have constant solutions, we must have $\hat{Y} < 1/4$ (too much fishing will inevitably make the population decline). We have two possible static solutions for each value of $\hat{Y}$ and when $\hat{Y} = 0$ these solutions are $\hat{N} = 0$ and $\hat{N} = 1$.

Problem 4.1:

We consider the following alternative fish population model,

$$\frac{\mathrm{d}N}{\mathrm{d}t} = R_0 N \exp(-R_1 N) - \frac{Y}{2}\left(1 + \tanh\left(\frac{N - N_h}{K}\right)\right)\,, \tag{4.7}$$

where N is the population variable and all the parameters are positive.

Use the following change of variable $t = \alpha\hat{t}$, $N = \beta\hat{N}$, $Y = \gamma\hat{Y}$, $K = \delta\hat{K}$, $N_h = \sigma\hat{N}_h$, $R_1 = \kappa\hat{R}_1$ and find the suitable expressions of these parameters as function of R_0, R_1, K and N_h so that (4.7) reduces to

$$\frac{\mathrm{d}\hat{N}}{\mathrm{d}\hat{t}} = \hat{N}\exp(-\hat{R}_1\hat{N}) - \frac{\hat{Y}}{2}\left(1 + \tanh(\frac{\hat{N} - 1}{\hat{K}})\right)\,. \tag{4.8}$$

In what follows we will typically drop the hats on the various symbols, with the implicit understanding that the we may still need a change of variables to get back to ‘biological quantities.

4.4 Numerical Solution of the Schaefer Model

Solving (4.5) numerically is quite simple and very similar to what we did for the models in chapter 2. The solution is to use a trick attributed to Euler and expand $N(t\,)$ in a Taylor series,

$$N(t_0 + \mathrm{d}t\,) \approx N(t_0) + \mathrm{d}t\,\frac{\mathrm{d}N}{\mathrm{d}t}(t_0) + \mathcal{O}(\mathrm{d}t^2)\,, \tag{4.9}$$

(remember, we drop the hats from now on), where $\mathrm{d}t$ is a small time interval which we have to chose carefully (often by trial-and-error). We then substitute (4.5) into (4.9) to find

$$N(t_0 + \mathrm{d}t) \approx N(t_0) + \mathrm{d}t\Big[N(t_0)\Big(1 - N(t_0)\Big) - Y(t_0)\Big] + \mathcal{O}(\mathrm{d}t^2)\,. \tag{4.10}$$

Defining $N_i = N(t_0 + i\mathrm{d}t)$ and knowing $N_0 = N(t_0)$, we can solve (4.5) by applying (4.9) recursively as follows:

$$N_{i+1} \approx N_i + \mathrm{d}t\Big[N_i\,(1 - N_i) - Y(t_0 + i\mathrm{d}t)\Big]\,. \tag{4.11}$$

Implementing this in a program is very similar to what we did with the class `Population`. We will create a class `ODE_Euler` which contains the functionality to solve a generic ordinary differential equation using the Euler method and define it in the module file `ode_euler.py`. In general, the differential equation we wish to solve can be written in the form

$$\frac{\mathrm{d}N(t)}{\mathrm{d}t} = F(t, N(t))\,, \tag{4.12}$$

where the right-hand side of the equation can depend explicitly on the integration variable t and the function $N(t)$. Sub-classes of `ODE_Euler` will be able to override this function F.

The class must contain the variable `N` to hold the current population value, and the list `N_list` to hold the history of the population (along with a list `t_list` which holds the corresponding values of t). (Remember these will be accessible in the class functions as `self.N` and `self.N_list`.) We also keep track of `dt`, the integration time step. The class functions are similar too:

- `reset(self, N, dt, t0)` : Reset the integration variables: `N` the initial population value, the integration time step `dt` and `t0` the initial time (with a default value set to `0`). This is actually used by the `__init__` function.
- `F(self, t, N)` : The right hand side of equation (4.12), i.e. $F(t, N)$. We pass t and N as parameters instead of using `self.t` and `self.N`, in anticipation of more accurate integration methods which require the value of `F` at `t`≠`self.t` and `N`≠`self.N` (see section 4.5.1).
- `one_step(self)` : Performs a single integration step. When we later consider integration methods which are more reliable than the Euler method, this is the function we will have to modify.
- `iterate(self, tmax)` : Repeatedly call `one_step()` to integrate the equation up to time `t=tmax`. It also saves the values of `N` and `t` in the lists `N_list` and `t_list` for later (to generate figures).
- `plot(self, style='b-')` : use the 2 lists `N_list` and `t_list` to generate a graph of $N(t)$.

file: `ode_euler.py` (part)

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

class ODE_Euler:
  """A class to compute the time evolution of a population using the
     Euler method. Contains variables which store the current
     population as well as a history.  You need to implement the
     function F(self, t, N) in a subclass in order to use this class.
  """

  def __init__(self, N=0, dt=0.001, t0=0):
      """ Initialise and set the parameters for an ODE integration.
        : param N  : initial population.
        : param dt : integration time step.
        : param t0 : initial time.
      """
      self.reset(N, dt=dt, t0=t0)

  def reset(self, N, dt, t0=0):
      """ Reset the integration parameters; see __init__ for more info.
        : param N  : initial population.
        : param dt : integration time step.
        : param t0 : initial time.
      """
      self.N  = N
      self.dt = dt
      self.t  = t0
      self.t_list = []        # to store t values for plots
      self.N_list = []        # to store N values for plots

  def F(self, t, N):
      """ Return the value of dN/dt = F(t,N). You need to implement this
          in a subclass.
        : param t : current value of integration variable t
        : param N : current value of function N
        : return : the value of the right hand side of the equation at time t
      """
      pass

  def one_step(self):
      """ Perform a single integration step using the Euler method.
      """

      self.N += self.dt*self.F(self.t, self.N)
      self.t += self.dt

  def iterate(self, tmax):
      """ Solve the equation dN(t)/dt = F(N(t))  until time tmax.
          Update N, t and append all values to N_list and t_list.
        : param tmax : upper bound of integration.
      """

      while(self.t < tmax):
        self.one_step()
        self.N_list.append(self.N)
        self.t_list.append(self.t)
```

The code below illustrates how to use the class `ODE_Euler`.

file: `euler_test.py`

```python
from ode_euler import ODE_Euler

class EulerTest(ODE_Euler):

    def __init__(self, N, dt, t0=0, R=-1):
        """ Initialiser: set all the parameter for the integration
        : param N  : initial value of N
        : param dt : integration time step
        : param t0 : initial time (usualy 0)
        : param R  : equation parameter value (default is -1)
        """
        super().__init__(N, dt=dt, t0=t0)
        self.R = R

    def F(self, t, N):
        """ The right hand side of the equation dN/dt = R*N
        : param t : current value of integration variable t
        : param N : current value of function N
        : return : the value of the right hand side of the equation at time t
        """
        return self.R*N

if __name__ == "__main__":
    pop = EulerTest(0.01, dt=0.01) # R takes the default value: -1
    pop.R = -0.5     # We change the value of R
    pop.iterate(10)  # We perform 10 steps of integration
    pop.plot()       # and display the result

    # Another way to do the same
    pop2 = EulerTest(0.01, dt=0.01, t0=0, R=-0.5) # Set R value
    pop2.iterate(20) # We perform 20 steps of integration
    pop2.plot("r-")  # and display the result in red this time
```

We start by importing the module `ode_Euler`. We then create a the class `EulerTest` as a sub class of `ODE_Euler`. The initialiser function, `__init__`, calls the initialiser function of the parent class. This is what `super().__init__(...)` does. We also add 1 parameter to the class: `R`. The function `F` compute the right hand side of the equation of interest.

We can then create one or more instance of the class `EulerTest` and integrate the equation for different parameter values of different initial conditions.

Problem 4.2:

Write a program called `schaefer.py` which creates a subclass of the class `ODE_Euler` named `Schaefer` and which uses it to solve the Schaefer equation (4.5). (The program `euler_test.py` illustrates how to solve the equation $dN/dt = RN$, so you might want to make a copy of that program and modify it). We will assume $Y(t)$ to be a constant: Y. Proceed as follows:

- Import the modules `ode_euler.py` using `from ode_euler import ODE_Euler`.
- Define `Schaefer` as a subclass of `ODE_Euler` using `class Schaefer(ODE_Euler)`.
- Add to the class an `__init__(self, N, dt, t0, Y)` function which must initialise the `ODE_Euler` class variables as well as `self.Y`.
- Implement the function `F(self, t, N)` so that it corresponds to equation (4.5) but for a constant Y.
- Copy the code at the bottom of `euler_test.py`, starting with the `if` test, and modify it so that
  - It creates an instance of the `Schaefer` class instead of `EulerTest`. Use the following values `N=0.1`, `t0=0`, `dt=0.01`.
  - It sets the equation parameter `Y` to `0`.
  - It sets the parameter `t_max` of the `iterate` function to `20`.

  As we assume that $Y(t)$ is constant, you need to initialise its value. The program `euler_test.py` illustrates 2 equivalent ways to create a class object and set its parameter. You can use either method but there is no need to use both.

Check Point 4.1

Problem 4.3:

Consider the Schaefer model (4.6).

a) If the initial population is $N = 0.25$, for which value of Y does the population remain at that value? (solve this analytically to justify your answer). Hint: use equation (4.6) or (4.5)

b) What happens when the value of Y is increased to reach the value found in a)? (run your program with $N = 0.25$ for 5 different values of Y, starting with $Y = 0$ en ending with the value found in (a).)

c) What happens when Y is larger than the value found in a)? Why is your program giving you these unexpected results?

Check Point 4.2

4.5 Multiple Population Model

So far our model has only involved one population, but most ecosystems involve a variety of species all depending on each other. The simplest model involves two populations and is a predator prey model called the Lotka-Volterra model [1]. Using $N(t)$ to represent the prey population and $P(t)$ for the predator population, the model is given by the pair of equations,

$$\begin{aligned} \frac{\mathrm{d}N}{\mathrm{d}t} &= N(t)\Big(a - bP(t)\Big), \\ \frac{\mathrm{d}P}{\mathrm{d}t} &= P(t)\Big(cN(t) - d\Big), \end{aligned} \tag{4.13}$$

The 4 parameters play the following role:

a: The birth rate of the prey. Without predator, the population would grow exponentially with this rate. It is assumed there is an unlimited amount of space and food.

b: The killing rate of the prey by the predator. The predators are like gluttons and eat all prey that come their way. The more predators there are, the more prey will be eaten, hence the term $-bNP$ in the equation.

d: The death rate of the predator.

c: The birth rate of the predator. The actual birth rate is proportional to the amount of food, the prey, and the number of predators, hence the term cPN in the equation.

As shown in the appendix, one can redefine the parameters so that (4.13) reduces to the following equations.

$$\begin{aligned} \frac{d\hat{N}}{d\hat{t}} &= \hat{N}(1-\hat{P}) \\ \frac{d\hat{P}}{d\hat{t}} &= K\hat{P}(\hat{N}-1)\,. \end{aligned} \tag{4.14}$$

We are thus left with only one mathematical parameter, K, instead of the initial four.

4.5.1 Numerical solutions of a system of ODEs

Solving (4.14) numerically is very similar to what we have done so far. The main change is to make the variable `N` to be an array so that it can contain the values of the different populations ($\hat{N}$ and $\hat{P}$ in our case). We will also make a few other changes and improvements:

- The variable used to keep the function values in the class will be called `V` rather than `N` (to remember it is a vector).
- The class function `one_step` does not need to be modified thanks to the use of numpy arrays for `self.V` and `self.F()`. If we had decided to use lists this would not have been as simple as this.
- The class function `iterate` has an extra parameter called `fig_dt` to specify how often we want to keep the computed data for figures. The reason for this is that we usually only need a few hundred points for a figure but if we integrate a system for a very long time or with a very small dt, then the 2 lists holding the data will become very large and might even force the program to stop. By default, (`fig_dt` < 0), it is set to `dt` and all the data points are saved. (Do not worry if you do not fully understand how this is done at this stage.)
- The `plot` function must be amended so that we can decide which of the functions we want to plot. It now takes 3 arguments: 2 indices and a plot style string. The first index, i, specifies what to use for the y-axis. If it is positive the i^{th} function will be plotted (corresponding to `V[i-1]` as array indices start with 0 in python). If $i = 0$, the time values are used. The second index, j, specifies what to use for the x axis. If $j = 0$, the default, the time values are used. If j is a positive integer then the j^{th} function will be plotted. The class function `plot` does not call the function `show()` which must be called separately. This is to allow the superposition of several figures. So for (4.14), `plot(1,0,''-b'')` will plot $N(t)$ as a blue line while `plot(2,1,''-r'')` will plot the trajectory $P(N)$ in red.
- The system of equations solved in the test part of the `ode_euler_N` module is

$$\begin{aligned} \frac{d\hat{N}}{d\tau} &= \hat{P}\,, \\ \frac{d\hat{P}}{d\tau} &= -\hat{N}. \end{aligned} \tag{4.15}$$

file: ode_euler_N.py

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

class ODE_Euler_N:
  """A class to compute the time evoluation of a population of multiple
     species using the Euler method. Contains variables which store
     the current population as well as a history.  You need to
     implement the function F(self, t, V) in a subclass in order to
     use this class.
  """

  def __init__(self, V0=np.array([], dtype='float64'), dt=0.001, t0=0):
      """ Set the variables used by the class:

      : param V0 : initial value (as an array or a list)
      : param dt : integration time step
      : param t0 : initial time
      """
      self.reset(V0, dt=dt, t0=t0);

  def reset(self, V0, dt=0.001, t0=0):
      """ Reset the integration parameters :

      : param V0 : initial value (as an array or a list)
      : param dt : time step
      : param t0 : initial time
      """
      self.V = np.array(V0, dtype='float64') # ensure we use floats!
      self.t = t0
      self.dt = dt
      self.t_list = []
      self.V_list = []

  def F(self, t, V):
      """ Return the right hand side of the equation dV/dt = F(t,V)

      : param t  : current time
      : param V0 : current function values
      : return : the right hand side of the equation at time t
      """
      pass

  def one_step(self):
      """ Performs a single integration step using the Euler method.
      """
      self.V += self.dt*self.F(self.t, self.V)
      self.t += self.dt

  def iterate(self, tmax, fig_dt=-1):
      """ Solve the system of equations DN/dt = F(N)  until tmax
          Save N and t in lists N_list and t_list every fig_dt

      : param tmax   : integration upper bound
      : param fig_dt : interval between data point for figures (use dt if <
          0)
      """

      if(fig_dt < 0) : fig_dt = self.dt*0.99 # Save all data

      next_fig_t = self.t*(1-1e-15) # Ensure we save the initial values

      tmax -= self.dt*0.1          # Stop as close to tmax as possible
      while(self.t < tmax):        # Integrate until tmax
        self.one_step()
        if(self.t >= next_fig_t): # Save fig when next_fig_t is reached
          self.V_list.append(np.array(self.V)) # force a copy of V!
          self.t_list.append(self.t)
          next_fig_t += fig_dt    # Set the next figure time

  def plot(self, i, j=0, style="k-"):
    """ plot V[i] versus t    (i > 1 and j = 0)  using style
        plot V[i] versus V[j] (i > 1 and j > 1)  using style
        plot t    versus V[j] (i = 0 and j > 1)  using style

      : param i    : index of function for y-axis
      : param j    : index of function for x-axis
      : param style: style string for the plot function
    """

    if(j==0):
      lx = self.t_list
    else: # Extra item i-1 from each element of f_list
      lx = list(map (lambda v : v[j-1] , self.V_list))
    if(i==0):
      ly = self.t_list
    else:
      ly = list(map (lambda v : v[i-1] , self.V_list))
    plt.plot(lx, ly, style);


# Only run this when not importing the module.

if __name__ == "__main__":
    class EulerNTest(ODE_Euler_N):
        def F(self, t, V):
          """Example below:
             du/dt = v;
             dv/dt = -u.
          """
          u = V[0]
          v = V[1]
          return(np.array([v, -u]))

    pop = EulerNTest(V0=[1.0,0.0], dt=0.001, t0=0)
    pop.iterate(tmax=10, fig_dt=0.1)
    pop.plot(1, 0, "r-")          # Plot u(t) in red
    pop.plot(2, 0, "b-")          # Plot v(t) in blue
    plt.show()                    # Show the 2 curves on the same figure
    pop.plot(1, 2, "g-")          # Plot u(v) in green
    plt.axis('equal')             # Ensures a square is shown as a square
    plt.margins(0.1, 0.1)         # add extra space on the edges
    plt.show()                    # Show the trajectory
```

The code at the bottom of the module shows how to use the object `ODE_euler_N`. As before, we create a subclass which implements the `F` function. The initial populations are passed as a list to the reset function (which converts it to an array, but passing an array would work too). The program then solves the system of equations and displays the two functions on the same figure and then creates a figure of $\hat{P}(\hat{N})$. From the figure, can you guess what the analytical solution of (4.15) is?

Notice also that the module `ode_euler_N.py` can be subclassed so as to solve any system of any number of equations (not just two, as in this example). All that needs to be changed is the definition of the class function `F` and the initialisation.

Problem 4.4:

Write a program called `lotka_volterra.py` by creating a subclass of `ODE_euler_N` called `LotkaVolterra` so that it solves the Lotka Volterra equation (4.14). Use the template `lotka_volterra_template.py` and complete the definition of the class function `F()`.

Problem 4.5:

Run the program `lotka_volterra.py` using a time-step of `dt=0.01`. The first figure shows $\hat{N}(t)$, in red, and $\hat{P}(t)$, in blue, and they look nearly periodic. The second figure corresponds to the trajectory $\hat{P}(\hat{N})$ which looks nearly periodic.

With an integration time step `dt=0.01` the trajectory is irregular. Is this a genuine effect or is the expansion of the trajectory an artefact of the numerical method that we have used? To get a clue, rerun the program but take an integration time step `dt=0.0001`. Is this more regular?

Check Point 4.3

The answer to the question above suggest that the Euler method is not very accurate and indeed it can be shown that it isn't, but this is beyond the scope of this course. There are better methods and the most widely used is the so called 4th order Runge Kutta method. It works by computing successive approximations to the derivative and then combining them all. It has been shown to be stable and accurate.

The module `ode_rk4.py` is a copy of the module `ode_euler_N.py` where we have only changed the name of the class to `ODE_RK4` as well as the definition of the function `one_step()` so that it implements the 4th order Runge-Kutta method. (You are welcome to look at the code, but do not try to guess how it works. The method is described in [2] and derivation of the simpler, but conceptually identical 2nd order method is given in many numerical analysis books and many resources on the web.)

Problem 4.6:

Copy your program `lotka_volterra.py` into a file called `lotka_volterra_rk4.py` and import that class `ODE_RK4` from `ode_rk4` (instead of importing `ODE_euler_N`). Then replace the name of the parent class `ODE_euler_N` by `ODE_RK4` so that your program uses the Runge-Kutta method.

Problem 4.7:

How do the Euler and Runge-Kutta methods compare in term of accuracy? To answer that question, we will solve the Lotka-Volterra equation with the initial value $\hat{N} = \hat{P} = 0.1$ using the Runge-Kutta method and a small integration time step $\mathrm{d}t$. We will then solve the same equation using the same initial value and a different, but larger, values of $\mathrm{d}t$ both for the Euler and the Runge-Kutta method. We will then compute the distance, *i.e.* the error, between the obtained solution and the reference one and see how it varies with $\mathrm{d}t$.

The program `lotka_volterra_compare.py` imports the module `lotka_volterra.py` and `lotka_volterra_rk4.py` to compute solutions of the Lotka Volterra equations using the 2 methods, for the following values of dt: 0.2, 0.1, 0.05, 0.02 and 0.01.

Run the program `lotka_volterra_compare.py` to fill in the following table

dt	RK4 Error	Euler Error
0.2		
0.1		
0.05		
0.02		
0.01		

It can be shown that the errors for such numerical methods are approximately proportional to $\mathrm{d}t^n$ where n is an integer (this is derived from the Taylor expansion). More explicitly, one can write $f_{\text{exact}}(t) = f_{\text{approx}}(t) + K dt^n$ where K is an unknown constant which depends on the equation but is independent of dt. What value of n can you infer from the values obtained in your table (one value for the Euler method and one for the Runge-Kutta methods)? Justify your answer.

Check Point 4.4

4.6 Extra Problems

Problem 4.8:

We consider the following alternative fish population model,

$$\frac{\mathrm{d}N}{\mathrm{d}t} = R_0 N \exp(-R_1 N) - \frac{Y}{2}\left(1 + \tanh\left(\frac{N - N_h}{K}\right)\right) , \tag{4.16}$$

where N is the population variable and all the parameters are positive.

a) If Y is a fishing parameter, what does the function $(1 + \tanh(\frac{N-N_h}{K}))/2$ attempt to model? Hint: plot it first.

b) What do the five parameters R_0, R_1, Y, N_h and K each control/describe?

c) Perform a change of variables to reduce the number of mathematical parameters down to three.

Problem 4.9:

Copy the program `schaefer.py` into `schaefer_periodic.py` and modify it so that

$$Y(t) = Y_0\left(1 - \cos(2\pi\nu t)\right) , \tag{4.17}$$

where Y_0 is constant. This corresponds to a catching which varies periodically in time with period ν and average amplitude Y_0.

In `schaefer_periodic.py` you will have to add the parameter ν to the initialiser. Remember that the class `ODE_euler` keeps the value of the current time in the class variable `self.t`.

Problem 4.10:

a) Set the initial population to `N=0.25`, take $\nu = 1$ and `tmax=50`. Then increase Y_0 from 0 to the value obtained for Y in homework question 4.3a. Can one take Y_0 larger than that critical value?

b) What can you conclude from the results above from a modelling point of view?

Problem 4.11:

A zoologist has estimated the following parameters for the predator prey model in the natural reserve he is studying:

$$\begin{aligned} a &= 0.5 \text{ year}^{-1} , & b &= 10^{-4} \text{ year}^{-1}\text{predator}^{-1} , \\ d &= 0.1 \text{ year}^{-1} , & c &= 10^{-7} \text{ year}^{-1}\text{prey}^{-1} . \end{aligned} \tag{4.18}$$

The initial populations in the natural reserve are 10^6 preys and 100 predators. He asks you to predict the population after 10 years and you find that $\hat{N} = 0.0474711$, $\hat{P} = 5.15248011$ (you do not need to solve (4.18) as we have done it for you).

- Compute the value of K, specifying the correct units.
- What value of $\hat{t}$ does 10 years correspond to?
- What are the real populations N and P after 10 years?

(You do not need to solve the equations to answer this problem.)

4.7 Appendix : reducing the number of parameters

In this section we will reduce the number of parameters of (4.13) using a systematic method. It consists in defining new variables and functions, $\hat{t}$, $\hat{N}$ and $\hat{P}$, which differ from the original ones, t, N and P, by parameters to be determined later (this is called a rescaling):

$$t = \alpha\hat{t}, \qquad N = \beta\hat{N}, \qquad P = \gamma\hat{P}. \tag{4.19}$$

Substituting (4.19) in (4.13) we obtain

$$\begin{aligned} \frac{\beta}{\alpha}\frac{\mathrm{d}\hat{N}}{\mathrm{d}\hat{t}} &= \beta\hat{N}(a - b\gamma\hat{P}) \\ \frac{\gamma}{\alpha}\frac{\mathrm{d}\hat{P}}{\mathrm{d}\hat{t}} &= \gamma\hat{P}(c\beta\hat{N} - d). \end{aligned} \tag{4.20}$$

We must know find expression for α, β and γ so that as few parameters as possible are left in the equations. We start by multiplying the top equation by α/β and the second by α/γ to remove all parameters from the left hand side of the equations:

$$\begin{aligned} \frac{d\hat{N}}{d\tau} &= \hat{N}(\alpha a - b\alpha\gamma\hat{P}) \\ \frac{d\hat{P}}{d\tau} &= \alpha d\hat{P}(\frac{c\beta}{d}\hat{N} - 1). \end{aligned} \tag{4.21}$$

We then try to set as many of the equation coefficients to 1. First we take $\alpha = 1/a$ to *remove* the first parameter of the first equation. Then taking $\gamma = 1/(b\alpha) = a/b$, the second parameter also becomes 1. Finally $\beta = d/c$ sets the second parameter of the second equation to 1 and we are left with defining $K = d/a$ which is the only remaining parameter:

$$\begin{aligned} \frac{\mathrm{d}\hat{N}}{\mathrm{d}\hat{t}} &= \hat{N}(1 - \hat{P}) \\ \frac{\mathrm{d}\hat{P}}{\mathrm{d}\hat{t}} &= K\hat{P}(\hat{N} - 1). \end{aligned} \tag{4.22}$$

The only remaining parameter is thus K.

4.8 References

[1] J. D. Murray. *Mathematical Biology*. Springer, 2002.

[2] W. H. Press et al. *Numerical Recipes in C*. Cambridge University Press, 1992.

[3] M. B. Schaefer. "Some aspects of the dynamics of populations important to the management of the commercial marine fisheries". In: *Journal of the Fisheries Research Board of Canada* 14 (1957). `http://aquaticcommons.org/3530/1/Vol._1_no._2.pdf`, pp. 669–681.