Part I

Basic Concepts of Modelling

Chapter 2

Systems of ordinary differential equations

We have already seen how to use a single differential equation to model a function that represents some quantity of interest (e.g., population of a species, height of a growing plant, amount of pollutant in a lake). The power of differential equations is considerably stronger when they are applied to modeling two or more functions simultaneously. The resulting model consists of a system of differential equations in one independent variable, and several dependent variables. The study of such systems will be the focus of this chapter.

2.1 Predator and prey model

In ecological modelling, the effect of predator-prey interaction on the population dynamics of the respective species is of interest. One of the best-known methods for describing such interactions is the Lotka-Volterra model, which consists of the following system of differential equations

$$\frac{dx}{dt} = k_1 x - k_2 xy \,, \tag{2.1}$$

$$\frac{dy}{dt} = k_3 xy - k_4 y \,, \tag{2.2}$$

Here x and y are functions of time t, and represent the prey and the predator population, respectively. Parameters k_1, k_2, k_3 and k_4 are positive constants, representing the prey growth rate, prey death rate, predator growth rate and predator death rate, respectively. Typically, the model description also includes the initial prey and predator populations, specified at some common time t_0.

The Lotka-Volterra model is not amenable to analytical solution methods, and is usually solved numerically. We illustrate the process using the forward Euler method, which essentially consists of replacing the time derivatives by forward difference approximations. If we take $t_0 = 0$, and denote the initial conditions as $x(0) = x_0$ and $y(0) = y_0$, the difference equations are

$$\begin{aligned} x_{n+1} &= x_n + (k_1 x_n - k_2 x_n y_n)\,\Delta t\,, \\ y_{n+1} &= y_n + (k_3 x_n y_n - k_4 y_n)\,\Delta t\,, \end{aligned}$$

Here $n = 0, 1, 2, ...,$ x_n and y_n are the approximate solutions at time t_n, and the time step is Δt (assumed constant here).

For the purpose of illustration, we use the following parameter values: $k_1 = 0.5$, $k_2 = 0.02$, $k_3 = 0.004$ and $k_4 = 0.4$. We also assume at time $t = 0$, $x_0 = 20$, $y_0 = 40$. The numerical simulation is carried out to a final time of $t_n = 50$, with time step 0.01. Plots of the resulting predator and prey populations are shown in Figure 1.1. As seen in the plots, both populations exhibit a periodic pattern in time, although the periods are out of phase. In other words, there is a time lag between the peaks and troughs of the two solutions. Oscillating trends in the population of predator and prey species have been observed by ecologists in real-world situations (**cite an appropriate reference**).

Code listing for solving the predator and prey model by the Euler method is shown in listing 1.1.

```python
import matplotlib.pyplot as plt

def Euler(fs, x0, y0s, dt):
    n = len(y0s)
    for i in range(n):
        y0s[i] = y0s[i] + fs[i](x0, y0s)*dt
    return y0s

def sys_ode(fs, t0, y0s, dt, tn):
    X = []
    Y = []
```

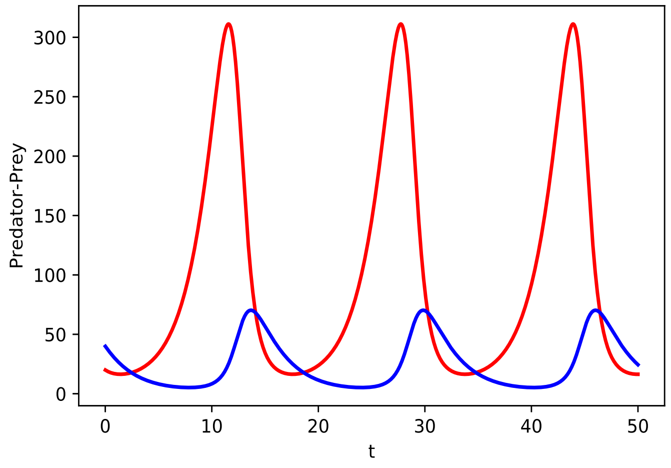

Figure 2.1: Predator (blue) and prey (red) dynamics

```
    T = []
    nt = int((tn-t0)/dt)
    for i in range(nt):
        y0s = Euler(fs, t0, y0s, dt)
        t0 = t0 + dt
        X.append(y0s[0])
        Y.append(y0s[1])
        T.append(t0)
    return T, X, Y

if __name__ == '__main__':

    k1 = 0.5
    k2 = 0.02
    k3 = 0.004
    k4 = 0.4
    dt = 0.01
    tn = 50
    t0 = 0
```

```
x0 = 20
y0 = 40
fs = [ lambda x, ys : k1*ys[0] - k2*ys[0]*ys[1],
       lambda x, ys : k3*ys[1]*ys[0] - k4*ys[1]]
T, X, Y = sys_ode(fs, t0, [x0,y0], dt, tn)
plt.plot(T,X,'r-',linewidth=2)
plt.plot(T,Y,'b-',linewidth=2)
plt.xlabel('t')
plt.ylabel('Predator-Prey')
plt.show()
```

Listing 2.1: Euler method for solving predator-prey model

We used the Euler method in this illustration because it makes the numerical solution process clear and easy to understand. However, a key drawback of this method is its low order of accuracy, which translates to smaller time-steps for attaining any specified level of accuracy. In practice, ordinary differential equation models of this type are typically solved using higher accuracy methods, such as the family of Runge-Kutta methods.

To implement more general numerical methods, it is convenient to rewrite our system of differential equations in vector form. Accordingly, equations (1.1-1.2) can be written as

$$\frac{d\mathbf{u}}{dt} = \mathbf{f}(\mathbf{u}, t) \tag{2.3}$$

where $\mathbf{u} = [x, y]^T$ is a vector function of the unknowns, and $\mathbf{f}$ denotes the vector of right hand sides, which will be a function $\mathbf{u}$ and t in general. In our predator-prey model, we have

$$\mathbf{f} = \begin{bmatrix} k_1 x - k_2 xy \\ k_3 xy - k_4 y \end{bmatrix} \tag{2.4}$$

Here the explicit dependence of $\mathbf{f}$ is only on $\mathbf{u}$. Such systems, where the dependence of the right hand side on t is implicit, are known as autonomous differential equations. To complete the model, the initial conditions are also written in vector form: $\mathbf{u}_0 = [x_0, y_0]^T$.

In this vector framework, a general step of the Euler method for solving the system is given by

$$\mathbf{u}_{n+1} = \mathbf{u}_n + \mathbf{f}(\mathbf{u_n})\,\Delta t \tag{2.5}$$

One of the most widely used higher-order methods for solving such systems is the classic four stage, fourth-order Runge-Kutta method, described by the following general step

$$\mathbf{u}_{n+1} = \mathbf{u}_n + (\mathbf{r}_1 + 2\mathbf{r}_2 + 2\mathbf{r}_3 + \mathbf{r}_4)\,\frac{\Delta t}{6} \tag{2.6}$$

where

$$\begin{aligned} \mathbf{r}_1 &= \mathbf{f}(\mathbf{u}_n, t_n) \\ \mathbf{r}_2 &= \mathbf{f}(\mathbf{u}_n + 0.5\Delta t\,\mathbf{r}_1,\ t_n + 0.5\Delta t) \\ \mathbf{r}_3 &= \mathbf{f}(\mathbf{u}_n + 0.5\Delta t\,\mathbf{r}_2,\ t_n + 0.5\Delta t) \\ \mathbf{r}_4 &= \mathbf{f}(\mathbf{u}_n + \Delta t\,\mathbf{r}_3,\ t_n + \Delta t) \end{aligned} \tag{2.7}$$

Written in this from, the method is relatively straightforward to implement in Python. The code for doing this is shown in listing 1.2.

Let us now take a closer look at the results we get from the predator-prey model. Since x and y are both functions of t, it is possible to sketch a graph of y versus x where the dependence on t is implicit, as shown in Figure 1.2. This is known as a phase plot, and the x-y plane a phase plane (or, phase space). An insight we get from the phase plot is that the predator and prey populations exhibit a kind of periodic stability known as *limit cycle*. The solution trajectory traces out a unique closed loop that it follows for all time. The trajectory shape is circular, and appears to be centered around a point in the interior. This point is actually an equilibrium point. In other words, the (x, y) values at that point produce a zero vector on the right hand side of equation (1.3). Thus, the time derivatives of x and y are both zero at the equilibrium point.

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

def Runge_Kutta(f, t0, y0s, dt):
  n = len(y0s)
  k1s = [fs[i](t0, y0s)            for i in range(n)]
  yts = [y0s[i] + 0.5*dt*k1s[i]    for i in range(n)]
  k2s = [fs[i](t0 + 0.5*dt, yts) for i in range(n)]
  yts = [y0s[i] + 0.5*dt*k2s[i]    for i in range(n)]
  k3s = [fs[i](t0 + 0.5*dt, yts) for i in range(n)]
  yts = [y0s[i] + dt*k3s[i]        for i in range(n)]
  k4s = [fs[i](t0 + dt, yts)       for i in range(n)]
  return [y0s[i] + (k1s[i] + 2*k2s[i] + 2*k3s[i] + k4s[i])*dt/6
    for i in range(n)]

def sys_ode(fs, t0, y0s, dt, tn):
  t=[]
  y1s=[]
  y2s=[]
  for i in range(int((tn-t0)/dt)):
    y0s = Runge_Kutta(fs, t0, y0s, dt)
    y1s.append(y0s[0])
    y2s.append(y0s[1])
    t.append(t0)
    t0 = t0 + dt
  return y1s, y2s

if __name__ == '__main__':
  fs = [
    lambda t, ys : 0.5*ys[0] - 0.02*ys[0]*ys[1],
    lambda t, ys : 0.004*ys[0]*ys[1] - 0.4*ys[1]
  ]

  y1, y2 = sys_ode(fs, 0, [20, 40], 0.01, 50)
  plt.plot(y1,y2)
  plt.xlabel("prey")
  plt.ylabel("predator")
  plt.show()
```

Listing 2.2: Runge-Kutta method for solving predator-prey model

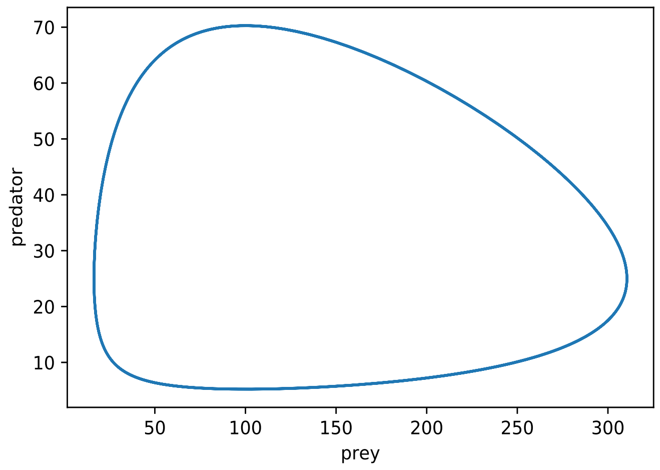

Figure 2.2: Phase plane of predator and prey model

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

def f(Y, t, k1, k2, k3, k4):
    x , y = Y
    x_ = k1*x-k2*x*y
    y_ = k3*x*y-k4*y
    return x_, y_

if __name__ == '__main__':

    k1 = 0.5
    k2 = 0.02
    k3 = 0.004
    k4 = 0.4

    y1 = np.linspace(0,350, 30)
    y2 = np.linspace(0, 70, 30)
    Y1, Y2 = np.meshgrid(y1, y2)

    t = 0
    u, v = np.zeros(Y1.shape), np.zeros(Y2.shape)
    NI, NJ = Y1.shape

    for i in range(NI):
        for j in range(NJ):
            x = Y1[i, j]
            y = Y2[i, j]
            yprime = f([x, y], t, k1, k2, k3, k4)
            u[i,j] = yprime[0]
            v[i,j] = yprime[1]
    plt.quiver(Y1, Y2, u, v, color='b')
    plt.xlabel('Prey')
    plt.ylabel('Predator')
    plt.show()
```

Listing 2.3: Vector field plot of predator and prey population

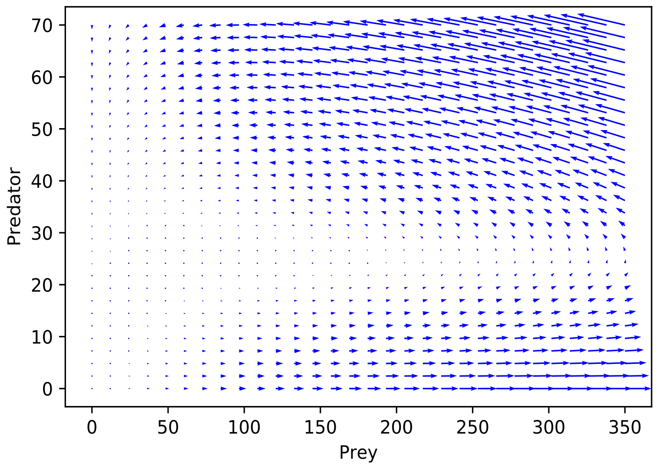

Figure 2.3: Vector field plot of predator and prey population

To find the equilibrium points of the system (1.3), we set the time derivatives of x and y to zero and solve for all the (x, y) pairs that satisfy the resulting equations

$$k_1 x - k_2 xy = 0 \tag{2.8}$$
$$k_3 xy - k_4 y = 0 \tag{2.9}$$

Thus, the system of ordinary differential equations reduces to a system of nonlinear algebraic equations that can be solved numerically by various methods including, for instance, the Newton method. It is easy to see in the above system $(x, y) = (0, 0)$ is one solution. This is the trivial solution and has no practically useful meaning in our current application context. Other solutions of (1.8-1.9) are straightforward to find algebraically, since the equations are relatively simple. But for the sake of treating more general cases, we will solve the system numerically by calling sympy package in Python, where the code is shown in listing 1.4. Using this code, we find there is another equilibrium point at $(x, y) = (100, 25)$. This implies the limit cycle has a center at $(x, y) = (100, 25)$ when $k_1, ...k_4$ are fixed at the values given earlier. When the initial conditions change, the size and shape of the limit cycle change, but the trajectory remains centered around the same equilibrium point.

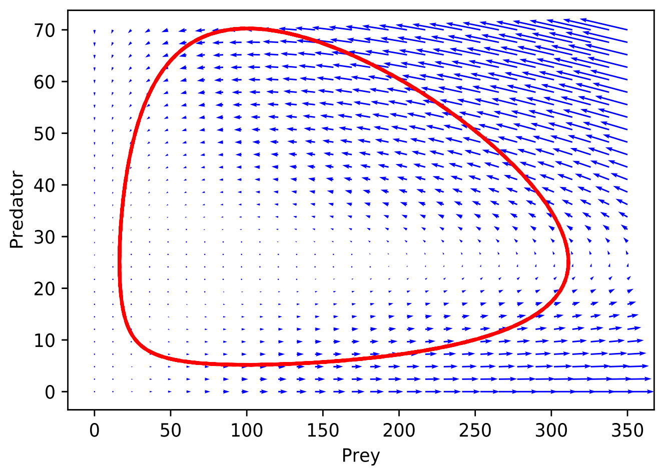

Figure 2.4: Vector field plot and limit cycle of predator-prey model

```
import sympy as sym

x, y= sym.symbols('x, y')

k1 = 0.5
k2 = 0.02
k3 = 0.004
k4 = 0.4

eq1 = k1*x-k2*x*y
eq2 = k3*x*y-k4*y
eqs = sym.solve([eq1,eq2],[x,y])[1]

print('x = %d, y = %d ' %(eqs[0], eqs[1]))
```

Listing 2.4: Solve the system of algebraic equations by Sympy package

Exercise

– Set $k_1 = 0.5, k2 = 0.02, k3 = 0.004$, and $k4 = 0.4$, change the initial conditions and then investigate the size of limit cycle.

– Change each value of parameters $k_1, ..., k_4$. Does the system has periodically stable ?

2.2 Lorenz system

Nonlinear differential equation systems can exhibit a fascinating and rich range of solution properties, even in low dimensional problems with relatively simple nonlinearities. An example of this is the Lorenz model, which exhibits chaotic behavior for certain choices of model parameters. Lorenz originally developed the model to study convection processes in the atmosphere, with the goal of studying weather-related phenomena. The model consists of the following system of three ordinary differential equations

$$\frac{dx}{dt} = \sigma(y - x)\,, \tag{2.10}$$

$$\frac{dy}{dt} = x(\rho - z) - y\,, \tag{2.11}$$

$$\frac{dz}{dt} = xy - \beta z\,. \tag{2.12}$$

(2.13)

This system represents a two-dimensional fluid layer with a temperature difference between the top and the bottom. The equations describe the rate of change of three quantities with respect to time, where $x(t)$ is the speed of rotation of the convection cell, $y(t)$ is the horizontal temperature, and $z(t)$ is the vertical temperature in two dimensional space. The system has been non-dimensionalized in such a way that the coefficients represent certain well-known constants in fluid mechanics: σ, ρ , and β are the Prandtl number, Rayleigh number, and the physical dimensions of the fluid layer.

Lorenz noticed some extremely unusual solution behaviors in this model. For instance, a solution trajectory that was initially calculated with six decimal places of accuracy was recomputed using three decimal places, as a check. But the two computations produced completely different trajectories! This led to the discovery of a major attribute of chaotic dynamical systems: sensitive dependence on initial conditions: very small changes in initial conditions can make very large differences in long-term behavior. This phenomenon has often been dubbed the "butterfly effect," suggesting that if a butterfly flaps its wings in some far-away land, it could cause a tornado locally, due to long-term effects of the small local change in atmospheric conditions caused by the butterfly!

One set of parameter values that produces this effect, and that has been widely studied, is $\sigma = 10, \rho = 28$ and $\beta = 8/3$. If the system is solved with these parameter values, and we plot x, y and z in three dimensional phase space, the solution trajectory shows a chaotic pattern, which also happens to resemble a butterfly, as seen in Figure 1.5. Here we have set the initial condition to be $x = 0$, $y = 1$ and $z = 2$. The Python code for solving the system using the Euler method is shown in listing 1.5 (from `matplotlib.org`).

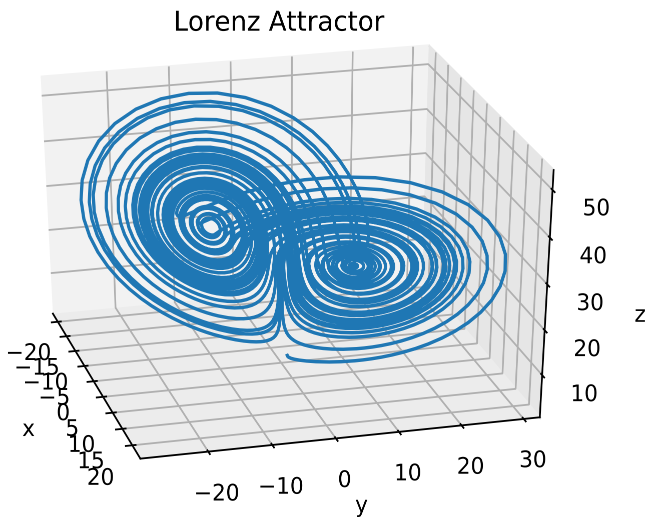

Figure 2.5: Butterfly effect from Lorenz system

```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

def lorenz(x, y, z, s=10, r=28, b=2.667):
    x_dot = s*(y - x)
    y_dot = r*x - y - x*z
    z_dot = x*y - b*z
    return x_dot, y_dot, z_dot

dt = 0.01
num_steps = 5000

xs = np.empty(num_steps + 1)
ys = np.empty(num_steps + 1)
zs = np.empty(num_steps + 1)

xs[0], ys[0], zs[0] = (0, 1, 2)

for i in range(num_steps):
    x_dot, y_dot, z_dot = lorenz(xs[i], ys[i], zs[i])
    xs[i + 1] = xs[i] + (x_dot * dt)
    ys[i + 1] = ys[i] + (y_dot * dt)
    zs[i + 1] = zs[i] + (z_dot * dt)

fig = plt.figure()
ax  = plt.axes(projection='3d')

ax.plot(xs, ys, zs)
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.set_zlabel('z')
ax.view_init(30, -15)
ax.set_title("Lorenz Attractor")
plt.savefig('Lorenz.eps', bbox_inches='tight')
plt.show()
```

Listing 2.5: Lorenz system solved by the Euler method

The Lorenz system is deterministic, which means that if you know the exact initial conditions, the rest of the solution is completely and uniquely determined by the differential equation system. As we pointed out earlier, Lorenz demonstrated that even the slightest change in the initial conditions causes dramatic changes in the long-term solution dynamics. Such solution behavior is called "chaotic," and is often seen in weather prediction models.

Returning to Figure 1.5, observe that the two "wings" of the butterfly correspond to two different sets of physical behavior of the system. Solution trajectories typically loop arond one wing a few times before switching to the other wing, and then continue this back and forth pattern in a seemingly random fashion.

Exercise

- Try to change the initial conditions and investigate the chaotic behavior.
- Discuss why we call the Lorenz system as chaos.
- Try to explain the physical meaning of x, y and z over time.
- What is the attractor ?
- Why do we have to set the parameters as $\sigma = 10, \rho = 28$ and $\beta = 8/3$?

2.3 SIR epidemic model

Differential equation systems have been widely used for modeling the spread of infectious diseases, and to study the effect of various treatment and control mechanisms. One of the most well-known of these is the SIR model [Kermack and McKendrick, 1927], also known as the epidemic model. It consists of a system of three differential equations that describe the spread of a disease in a fixed population of interest. The modeling strategy consists of splitting the given population of size N, say, into three separate, non-overlapping "compartments," whose sizes vary with time as the disease spreads through the population. Each compartment consists of a certain number of individuals who share some common attribute, as follows: $S(t)$ is the number of individuals who are susceptible, but not yet infected; $I(t)$ is the number of infectious individuals, and $R(t)$ is number of recovered individuals, who are assumed to be immune from the disease. It is also assumed that N is a constant, and that $S(t) + I(t) + R(t) = N$.

The system of differential equations describing the evolution of (S, I, R) over time is expressed by

$$\begin{aligned} \frac{dS}{dt} &= -\beta\frac{SI}{N} \\ \frac{dI}{dt} &= \beta\frac{SI}{N} - \gamma I \\ \frac{dR}{dt} &= \gamma I \end{aligned} \tag{2.14}$$

The SIR model describes the change in each compartment over time, with β being the contact rate of the disease and γ the mean recovery rate. The model is complete once we specify initial conditions for the variables S, I, R.

To illustrate the behavior of this model, we choose the initial conditions $S(0) = 999$, $I(0) = 1$ and $R(0) = 0$, with parameter values $\gamma = 0.1$ (??**please include units**) and $\beta = 0.2$ (??**please include units**). Thus, the total popultion is $N = S + I + R = 1000$. Figure 1.6 shows numerical simulation results for a time period of 160 days.

It is useful to observe the following features in the solution, most of which also seem intuitively reasonable:

- The number of susceptibles decreases rapidly in the early stages as they become infected, and $S(t)$ is a decreasing function.
- The infected population increases to some maximum value, and then decreases, until it asymptotes zero at some later stage.
- The recovered population increases until there is no infected person in the system.

The Python code for solving the SIR model is shown in listing 1.6. Here we integrate the system of ODEs by calling the built-in function 'odeint' in scipy package. This code was downloaded from `scipython.com/book/chapter-8-scipy/additional-examples/the-sir-epidemic-model/`.

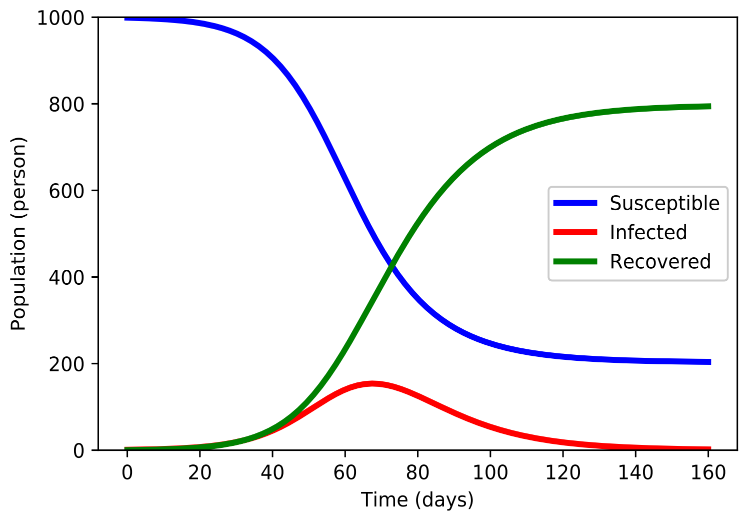

Figure 2.6: The results of SIR model over 160 days

```
import numpy as np
from scipy.integrate import odeint
import matplotlib.pyplot as plt

# Total population, N.
N = 1000

# Initial number of infected and recovered individuals.
I0, R0 = 1, 0

# Assume S + I + R = N
S0 = N - I0 - R0

# beta is infection rate, gamma is recovery rate, (unit 1/days).
beta = 0.2
gamma = 0.1

# Discretize time 160 days
```

```
t = np.linspace(0, 160, 160)

# Define SIR model.
def deriv(y, t, N, beta, gamma):
    S, I, R = y
    dSdt = -beta*S*I / N
    dIdt =  beta*S*I / N - gamma*I
    dRdt =  gamma*I
    return dSdt, dIdt, dRdt

# Set Initial values in term of vector
y0 = S0, I0, R0

# Integrate the ODE system using builtin odeint
result = odeint(deriv, y0, t, args=(N, beta, gamma))

# Get the numerical results over time
S, I, R = result.T

# Visualization
plt.plot(t, S, 'b', alpha=0.5, lw=3, label='Susceptible')
plt.plot(t, I, 'r', alpha=0.5, lw=3, label='Infected')
plt.plot(t, R, 'g', alpha=0.5, lw=3, label='Recovered')
plt.xlabel('Time (days)')
plt.ylabel('Population (person)')
plt.ylim(0,1000)
legend = plt.legend()
legend.get_frame().set_alpha(0.8)
plt.savefig('SIR.eps', bbox_inches='tight')
plt.show()
```

Listing 2.6: SIR model solved by scipy package

Exercise

– Change the parameters and investigate the S, I and R over time, interpret the results.

– Change the initial condition, interpret the results, and investigate equilibrium solutions.

– Identify the main assumptions of this model. Describe the situations that this model cannot be applied.

– Plot the trajectory of S, I and R in the three dimensional space and investigate the dynamics approaching the equilibrium point.

2.4 Time delay epidemic model for COVID-19

The coronavirus pandemic of 2020, caused by the COVID-19 respiratory virus, brought the entire world to a near standstill, and wreaked enormous damage worldwide. Although many studies have been carried out to understand the virus propagation mechanisms and post-infection prognosis (see [Anderson and May, 1979], [Diekmann and Heesterbeek, 2000] and [Xu and Ma, 2009]), many uncertainties and open questions remain.

Understanding the propagation mechanism is a very important issue in the control of the global spread of COVID-19 and other similar pathogens in the future. Mathematical modelling is an important tool for understanding the transmission mechanism. To quantify the spread of the virus, a dimensionless constant called the basic reproduction number R^* is usually estimated to predict the number of new infections caused by each currently infected individual. For example, $R^* \approx 3 - 4$ for influenza, but it is much larger for measles, with $R^* \approx 16 - 18$ (see [Keeling and Rohani, 2008] and [Anderson and May, 1998]). However, the value of R^* for COVID-19 is unclear. In this section, we will apply the standard SIR model [Kermack and McKendrick, 1927] to estimate R^* for COVID-19. We will also describe an extension of the SIR model with two-time delays for infected individuals, in order to obtain the best fit to the actual data, using Monte Carlo Simulation. The transmission and recovery rates will be approximated and a time delay effect will be developed. Finally, we will define the infected ratio, and use it as an aid to monitoring and controling of the spread of the virus.

2.4.1 Basic epidemic models

The SIR system of differential equations introduced earlier is among the most commonly used epidemic models. We will now use it to model the spread of COVID-19. The model consists of the system of equations (1.15), repeated below for convenience

$$\frac{dS}{dt} = -\beta S I \tag{2.15}$$

$$\frac{dI}{dt} = \beta S I - \gamma I \tag{2.16}$$

$$\frac{dR}{dt} = \gamma I \tag{2.17}$$

As before, $S(t)$ denotes the number of susceptible individuals, $I(t)$ is the number of infectious individuals, and$R(t)$ is number of recovered individuals. We retain the assumption of a constant population size N, with $S(t)+I(t)+R(t) = N$. The following additional assumptions are made to build the mathematical model

- The populations of susceptible individuals and contagious infectives are very large, so that random differences between individuals can be neglected.
- There are no natural births and deaths in this model. The disease is spread by contact from individual to individual.
- Individuals who recover from the disease have immunity.
- The population system is closed with no migration.

We note that the model equations (1.15)-(1.17) have been written in dimensionless form, where each population variable is divided by N.

The epidemic model described by the differential equations (1.15)-(1.17) has been extensively studied from various viewpoints (see [Kermack and McKendrick, 1927]). A key feature – and limitation – of this model is that it uses positive constants (β and γ) to model the mean infection and recovery rates. In a real-world situation the infection and recovery rates may vary considerably with time, and it may not work well to use a constant mean value for them. One way to incorporate the time-dependent behavior while preserving most other features of the model is by augmenting the mean infection and recovery rate constants with a time- delay effect.

For the case of COVID-19, we will now develop a more general epidemic model which includes the time delay effect for both infected and recovered individuals. One way to interpret this effect is that susceptible individuals are not immediately infected when they come in contact with infected individuals. Instead there is a time-delay period of disease incubation. Similarly, infected individuals do not immediately start recovering, and transitioning into the population of the recovered individuals. The resulting SIR model, generalized with time delay effects, is then given by the following system

$$\frac{dS}{dt} = -\beta S I(t-d_1) \tag{2.18}$$

$$\frac{dI}{dt} = \beta S I(t-d_1) - \gamma I(t-d_2) \tag{2.19}$$

$$\frac{dR}{dt} = \gamma I(t-d_2) \tag{2.20}$$

where d_1 and d_2 are constants that represent the effects of time delay for the infective and recovered individuals, respectively.

2.4.2 Simulation results

We show the numerical results and compare them with real data collected by the Johns Hopkins Github repository and summarized in reference [Nadu, 2020]. The case study considered is from China, since a complete data set is available for describing and fitting parameters in the present model. These data are from January 22, 2020, to March 15, 2020. Looking at the broad trends in the data, we can identify three distinct time stages for infective individuals: the early stage where the number of cases is slowly increasing, the middle stage of a fast increase, and the final stage where it slows again to a steady-state.

We assume the total population size under consideration is $N = 83,000$. Based on the daily report on January 22, 2020, which is taken as the beginning of the study period, the initial numbers of susceptible, infective and recovered individuals is $S_0 = 82408/N$, $I_0 = 547/N$, and $R_0 = 45/N$, respectively. Figures 1.7 and 1.8 display show plots of the actual data for susceptible, infective and recovered individuals.

To numerically approximate the parameters β and γ in the standard SIR model in (1.15)-(1.17), we use Monte Carlo simulation with 1,000 random iterative steps. The system of ODEs is integrated using the *ddeint* package [Zulko, 2019]. Two sets of uniformly distributed random numbers are generated to assign initial guesses for the values of β and γ, and we also specify that $0.1 < \beta < 0.6$, and $0.02 < \gamma < 0.08$. The optimal values are obtained by minimizing the percentage of the root mean square error (RMSE) between the actual data and the simulation result for infected individuals over 61 days. We find the minimum value of the RMSE is 6.438%, and the corresponding optimal values are $\beta = 0.345$ and $\gamma = 0.045$. Plots comparing the actual data with simulation results from the standard SIR model are shown in Figure 1.9. If the basic reproduction number R^* is defined by $\beta S_0/\gamma$, we obtain $R^* = 7.61$, which is considerably larger than the value for influenza, where $R^* = 3\text{-}4$ (see [Keeling and Rohani, 2008] and [Anderson and May, 1998]).

Our value of $R^* = 7.61$ for COVID-19 in China turns out to be an over-estimation. One reason for this may be that we use S_0 to approximate R^*, and this might be accurate only in the early stages. Instead of using S_0, we could define $\bar{S}$ as the average number of susceptible individuals over the simulated period. With this approach, we get $\bar{S} = 0.1587$, and thus $R^* \approx \beta\bar{S}/\gamma = 2.151$. This value is likely an under-estimate, since the simulation result in Figure 1.9 does not fit well with the actual data.

To obtain a better fit than the standard SIR model, we will next try using the time delay SIR variant given in equations (1.18)-(1.20). Recall, d_1 is the time delay effect for transitioning from susceptible to infected, and d_2 for transitioning from infected to recovered. As before, we use Monte Carlo simulation to obtain the best fit, but now d_1 and d_2 are two additional parameters which must be considered in the fit as well. So, four sets of uniformly distributed random numbers are initially generated, to iteratively compute the optimal values of β, γ, d_1, and d_2. We specify the parameter ranges as: $1.5 \leq \beta \leq 2.3$, $0.03 \leq \gamma \leq 0.08$, $7 < d_1 \leq 14$, and $0 < d_1 \leq 7$. After performing the simulation using 1,000 iteration steps, we obtain the optimal values $\beta = 1.747$, $\gamma = 0.05$, $d1 \approx 12$, $d_2 \approx 5$ with a percentage RMSE of 4.466%. Plots comparing the actual data with results from the time delay SIR model are shown in Figure 1.10. It can be seen that the simulation results for both infected and recovered individuals agree well with the actual data. However, it is interesting to compute and find the ratio $\beta S_0/\gamma \approx 34.69$, which is very large and cannot possibly represent the usual basic reproduction number. It can be shown that due to the time delay effect, we must multiply

$\beta S_0/\gamma$ by a correction factor of the form $I(t-d_1)/I(t-d_2)$. Even so, it is not entirely clear how to find the exact numerical values needed here. It may require stability analysis at equilibrium points, or the use of next-generation matrix methods (see [Xu and Ma, 2009]). Linear or nonlinear analysis is beyond the scope of our present work, as our primary aim here is exposition of modeling concepts. Since the present model can reasonably predict the time-evolution of susceptible, infected and recovered individuals, this will suffice until further analysis and newer developments become available in the future.

It is interesting to take a closer look at the ratio $I(t-d_1)/I(t-d_2)$ that arises naturally in the time delay variant of the SIR model. We assume that d_1 and d_2 are two integers representing a certain number of days before the present day, t. From equation (1.19), we see that $dI/dt > 0$ when

$$\beta S I(t-d_1) - \gamma I(t-d_2) > 0$$

which holds initially, at the early stage of disease spread. Furthermore, since $0 < S = S_0 < 1$, we have

$$\frac{\beta S_0}{\gamma}\frac{I(t-d_1)}{I(t-d_2)} > 1$$

This suggests the ratio $R_I = I(t-d_1)/I(t-d_2) > 0$ is a non-dimensional factor multiplying the basic reproduction number $R^* = \beta S_0/\gamma$ in the standard SIR model. When the disease is spreading, $I(t)$ is an increasing function, so $dI/dt > 0$ and $I(t-d_1) < I(t-d_2)$. When $R_I \ll 1$ and $0 < d_2 < d_1$, the disease spreads rapidly. Thus, it provides an alternative indicator that shows how fast the disease is spreading. In other words, we can use this indicator to monitor the progress of the disease. A plot of the ratio $I(t-d_1)/I(t-d_2)$ for China during January 22, 2020 to March 15, 2020 is shown in Figure 1.11 and reveals that China successfully controlled the spread of COVID-19 during this period, since R_I (which is actually a function of time) increases monotonically and reaches a steady value at nearly the endemic state. Observe that R_I is approaching an asymptotic constant as $t \to \infty$.

Next, we define a new reproduction number $\bar{R} = \bar{R}^* \bar{R}_I$ where $\bar{R}^* = \beta \bar{S}/\gamma$, and $\bar{R}_I$ is the average of the ratio R_I. From a simulation over 49 days (61-12 days), we obtain $\bar{S} = 0.1587$ and $\bar{R}_I = 0.7747$, which gives $\bar{R} = 4.2953$. So, we conclude that the basic reproduction number for the spread of COVID-19 in China is approximately 4.2953. A more precise closed form, together with a better estimate of the infected ratio, is needed to derive a more accurate reproduction number. It should be emphasized that all parameters shown here are obtained from fitting the data from China. For other countries, all parameters in the model need to be calibrated again, including the delay times d_1 and d_2.

The Python code for solving the SIR with time delay is shown in listing 1.7-1.10.

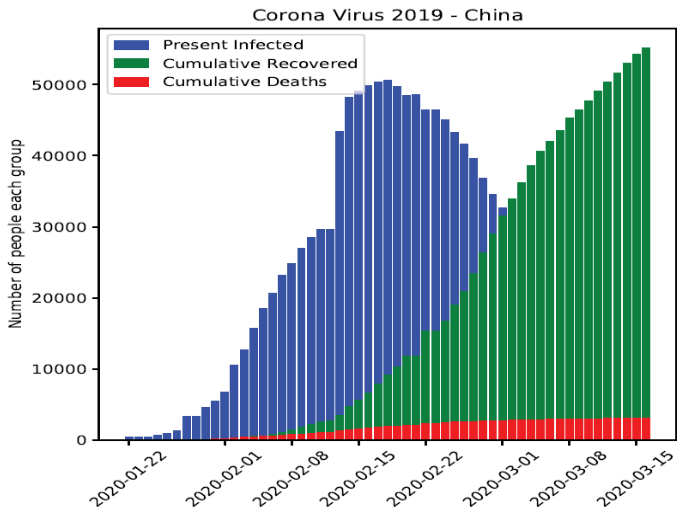

Figure 2.7: Bar plot of actual data for infected, recovery and death people due to COVID-19.

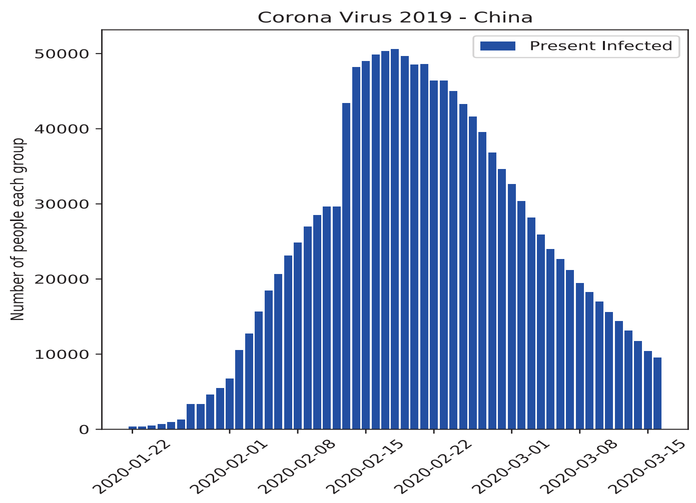

Figure 2.8: Bar plot of actual data for infected people due to COVID-19.

```
import numpy as np
import pandas as pd
import dateutil
import matplotlib.pyplot as plt
import seaborn as sns
import random

df = pd.read_csv('covid_19_data.csv')
df1 = df[df['Country/Region']=='Mainland China']
df1['ObservationDate'] = pd.to_datetime(df1['ObservationDate'])
df_China = df1[{"ObservationDate","Confirmed","Deaths","
    Recovered"}]
df_China.isnull().sum()

num_death = df_China['Deaths'].values
num_confirmed = df_China['Confirmed'].values
num_recovered = df_China['Recovered'].values
```

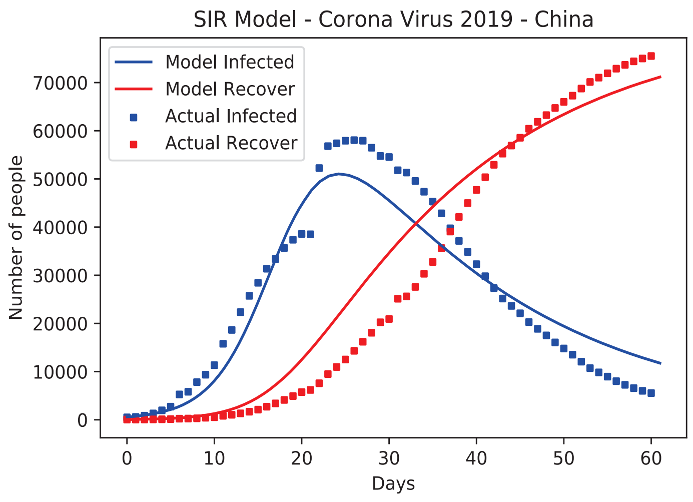

Figure 2.9: The results of SIR model

```
infect_daily = num_confirmed - num_recovered - num_death

df1['datetime'] = pd.to_datetime(df_China.ObservationDate)
df1.index = df1['datetime']

df2 = df1.resample('D').sum()
inf_con = df2['Confirmed']
inf_re = df2['Recovered']
inf_dead = df2['Deaths']
inf = inf_con.values - inf_re.values - inf_dead.values

actual_confirm = inf
actual_red = inf_re.values + inf_dead.values
```

Listing 2.7: SIR with time delay model (Dataframe)

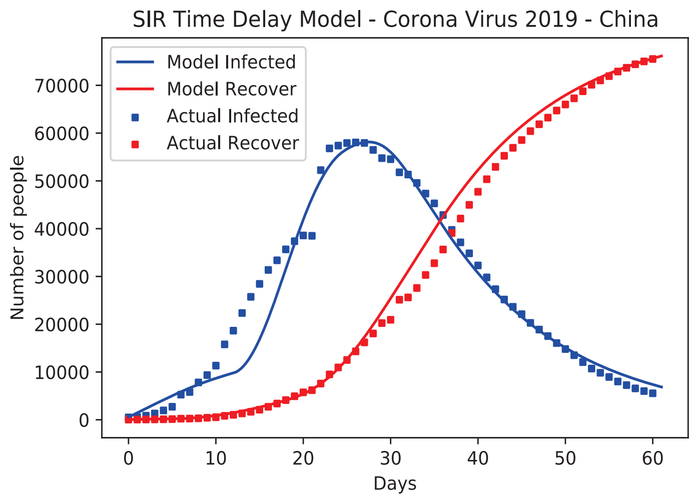

Figure 2.10: The results of SIR with time Delay model

```
import numpy as np
from scipy.integrate import odeint
import matplotlib.pyplot as plt

# Total population, N.
N = 83000

# Initial number of infected and recovered individuals.
I0, R0 = 500, 50

# Assume S + I + R = N
S0 = N - I0 - R0

# beta is infection rate, gamma is recovery rate, (unit 1/days).

beta = 0.3
gamma = 0.05
```

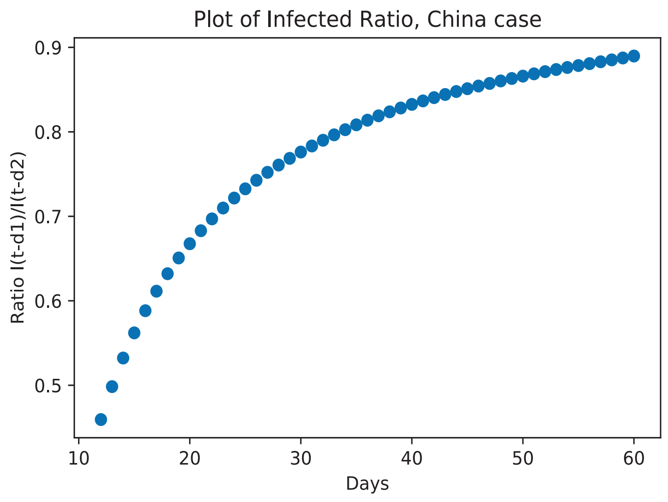

Figure 2.11: The ratio of infected people $I(t-d_1)/I(t-d_2)$ over 61 days where $d_1 = 12$ and $d_2 = 5$.

```
# Discretize time 160 days
t = np.linspace(0, 54, 55)

# Define SIR model.
def deriv(y, t, N, beta, gamma):
    S, I, R = y
    dSdt = -beta*S*I / N
    dIdt =  beta*S*I / N - gamma*I
    dRdt =  gamma*I
    return dSdt, dIdt, dRdt

# Set Initial values in term of vector
y0 = S0, I0, R0

# Integrate the ODE system using builtin odeint
result1 = odeint(deriv, y0, t, args=(N, beta, gamma))

# Get the numerical results over time
```

```
S1, I1, R1 = result1.T

# Visualization
plt.plot(t, S1, 'b', alpha=0.5, lw=3, label='Susceptible')
plt.plot(t, I1, 'r', alpha=0.5, lw=3, label='Infected')
plt.plot(t, R1, 'g', alpha=0.5, lw=3, label='Recovered')
plt.xlabel('Time (days)')
plt.ylabel('Population (person)')
plt.ylim(0,100000)
legend = plt.legend()
legend.get_frame().set_alpha(0.8)
plt.show()



x = range(55)
y = inf
z = actual_red

fig = plt.figure()
ax1 = fig.add_subplot(111)

ax1.scatter(x, y, s=10, c='b', marker="s", label='Actual Infect'
    )
ax1.plot(t,I1,'-b', label='Model Infect')

ax1.scatter(x, z, s=10, c='r', marker="s", label='Actual Recover
    ')
ax1.plot(t,R1,'-r', label='Model Recover')

ax1.set(xlabel="Days",
        ylabel="Number of people",
        title="Corona Virus 2019 - China")

plt.legend(loc='upper left');
plt.show()
```

```
print('R0 = ',beta/gamma)
```

Listing 2.8: SIR with time delay model

```
# https://pypi.org/project/ddeint/

from pylab import array, linspace, subplots
from ddeint import ddeint

from random import seed
from random import random

nstep = 10  # number steps of Monte Carlo simulation

def model(Y, t, d1, d2, beta, gamma):
    x, y, z = Y(t)
    xd, yd, zd = Y(t - d1)
    xdd, ydd, zdd = Y(t - d2)
    return array([-beta*x*yd, beta*x*yd - gamma*ydd, gamma*ydd])

N = 82000
S0 = 81408
I0 = 547
R0 = 45

g = lambda t: array([S0/N, I0/N, R0/N])
tt = linspace(0, 54, 541)


rmse_inf_percent = []
beta = []
gamma = []
delay = []
delay_re = []

for i in range(nstep):
    value1 = random()
    value2 = random()
    value3 = random()
    value4 = random()
    #scaled = minv + (value * (maxv - minv))

    betar = 1.7 + (value1 * (2.4 - 1.7))
    gammar = 0.03 + (value2 * (0.08 - 0.013))
    delayr = round(7 + (value3 * (14 - 7)))
    delayre = round(1 + (value4 * (7 - 1)))

    yy = ddeint(model, g, tt, fargs=(delayr,delayre,betar,gammar
    ))

    Ia = []
    for j in range(0,55):
        ii = j*10
        Ia.append(N*yy[ii,1])


    rmse_inf = np.sqrt(sum(np.power(np.abs(inf-Ia),2))/len(inf))
    rmse_inf_percent.append((rmse_inf/N)*100)

    beta.append(betar)
    gamma.append(gammar)
    delay.append(delayr)
    delay_re.append(delayre)

index_min = np.argmin(rmse_inf_percent)
print(index_min)
print('Mininum beta, gamma, delay1, delay2 = ',beta[index_min],
   gamma[index_min], delay[index_min], delay_re[index_min])


# Final results
yf = ddeint(model, g, tt, fargs=(delay[index_min],delay_re[
   index_min], beta[index_min], gamma[index_min]))

x = range(55)
y = inf
```

```
z = actual_red

fig = plt.figure()
ax1 = fig.add_subplot(111)

ax1.scatter(x, y, s=10, c='b', marker="s", label='Actual Infect'
    )
ax1.plot(tt,N*yf[:,1],'-b', label='Model Infect')

ax1.scatter(x, z, s=10, c='r', marker="s", label='Actual Recover
    ')
ax1.plot(tt,N*yf[:,2],'-r', label='Model Recover')

ax1.set(xlabel="Days",
       ylabel="Number of people",
       title="Corona Virus 2019 - China")

plt.legend(loc='upper left');
plt.show()
```

Listing 2.9: SIR with time delay model (Solve time delay)

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

colors = (0.6,0.6,0.9)

nd = len(Ia)

ts = []
Ratio = []

for i in range(d1,nd):
    Ratio.append(Ia[i-d1]/Ia[i-d2])
    ts.append(i)

plt.scatter(ts,Ratio, c=colors, alpha=0.8)
plt.title('Plot of Infection Ratio China case')
plt.xlabel('Days')
plt.ylabel('Ratio I(t-d1)/I(t-d2)')
plt.show()
```

Listing 2.10: SIR with time delay model (Plot infected ratio)

Bibliography

[Anderson and May, 1979] Anderson, R. M. and May, R. M. (1979). Population biology of infectious diseases: part i. *Nature*, 280:361–367.

[Anderson and May, 1998] Anderson, R. M. and May, R. M. (1998). Infectious diseases of humans: Dynamics and control. *Oxford University Press.*

[Diekmann and Heesterbeek, 2000] Diekmann, O. and Heesterbeek, J. A. P. (2000). Mathematical epidemiology of infectious disease. *John Wiley and Sons UK.*

[Giordano et al., 2003] Giordano, F. R., Weir, M. D., and Fox, W. P. (2003). *A first course in mathematical modeling, 3th edition.*

[Keeling and Rohani, 2008] Keeling, M. J. and Rohani, P. (2008). Modelling infectious diseases in humans and animals. *Princeton University Press.*

[Kermack and McKendrick, 1927] Kermack, W. and McKendrick, A. (1927). A contribution to the mathematical theory of epidemics. *Proceedin of the Royal Society. A*, 115:700–721.

[Nadu, 2020] Nadu, C. T. (2020). Novel corona virus 2019 dataset. `https://www.kaggle.com/sudalairajkumar/novel-corona-virus-2019-dataset`.

[Xu and Ma, 2009] Xu, R. and Ma, Z. (2009). Stability of a delayed sirs epidemic model with a nonlinear incidece rate. *Chaos, Soliton and Fractals*, 41:2319–2325.

[Zulko, 2019] Zulko (2019). Scipy-based delay differential equation (dde) solver. `https://pypi.org/project/ddeint`.