1

Introduction

1.1 Scope of the Book

This book is for advanced undergraduate students, post-graduate students, or engineers to acquire programming skills for dynamic system modelling and analysis using control theory. The readers are assumed to have a basic understanding of computer programming, ordinary differential equations (ODE), vector calculus, and probability.

Most engineering curricula at the undergraduate level include only an elementary-level programming course in the early of the undergraduate years. Only a handful of self-motivated engineering students acquire advanced level programming skills mainly from self-study through tedious time-consuming practices and trivial mistakes. As modern engineering systems such as aircraft, satellite, automobile, or autonomous robots are implemented through inseparable tight integration of hardware systems and software algorithms, the demand for engineers having fluent skills in dynamic system modelling and algorithm design is increasing. In addition, the emergence of interdisciplinary areas merging the experimental domain with mathematical and computational approaches such as systems biology, synthetic biology, or computational neuroscience further increases the necessity of the engineers who understand dynamics and are capable of computational implementations of dynamic models.

This book aims to fill the gap in learning practical dynamic modelling, simulation, and analysis skills in aerospace engineering, robotics, and biology. Learning programming in the engineering or biology domain requires not only domain knowledge but also a robust conceptual understanding of algorithm design and implementation. It is not, of course, the skills to learn in 14 days or less as many online courses claim. To be confident in dynamic system modelling and analysis takes more than several years of practice and dedication. This book provides the starting point of the long journey for the readers to equip and prepare better for real engineering and scientific problems.

Dynamic System Modelling and Analysis with MATLAB and Python: For Control Engineers,
First Edition. Jongrae Kim.

Companion Website: www.wiley.com/go/kim/dynamicmodeling

1.2 Motivation Examples

1.2.1 Free-Falling Object

Newton's second law of motion is given by

$$\sum_i F_i = \frac{d}{dt}(mv) \tag{1.1}$$

where F_i is the i-th external force in Newtons (N) acting on the object characterized by the mass, m, in kg, d/dt is the time derivative, t is the time in seconds, v is the velocity in m/s, and mv is the momentum of the object. Newton's second law states that *the sum of all external forces is equal to the momentum change per unit of time*.

Consider a free-falling object shown in Figure 1.1. There exists only one external force, i.e. the gravitational force acting downwards in the figure. Hence, the left-hand side of (1.1) is simply given by $\sum_i F_i = F_g$, where F_g is the gravitational force. Introduce the additional assumption that the object is within the reasonable range from the sea level. With the assumption, the gravitational force, F_g, is known to be proportional to the mass, and the proportional constant is the gravitational acceleration constant, g, which is equal to 9.81 m/s^2 in the sea level. Therefore, $F_g = mg$. Replace the left-hand side of (1.1), i.e. $\sum_i F_i$, by $F_g = mg$ provides

$$mg = F_g = \sum_i F_i = \frac{d}{dt}(mv) \tag{1.2}$$

where the downward direction is set to the positive direction, which is the opposite of the usual convention. *It highlights that establishing a consistent coordinate system at the beginning of modelling is vital in dynamic system simulation.*

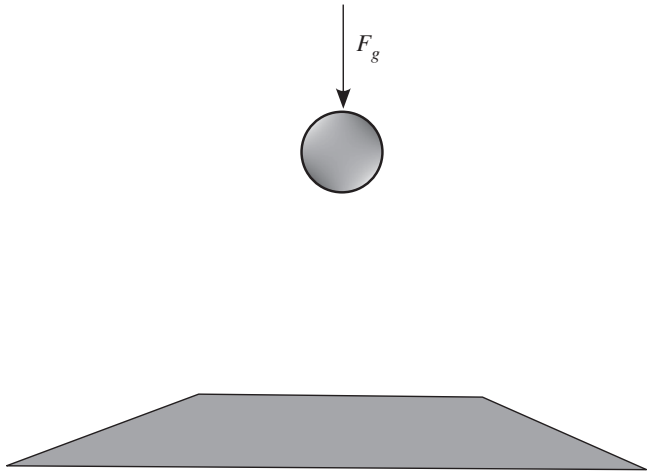


Figure 1.1 Free-falling object.

From the kinematic relationship between the velocity, v, and the displacement, x, we have

$$\frac{dx}{dt} = v$$

where the origin of x is at the initial position of the object, m, and the positive direction of x is downwards in the figure. The right-hand side of (1.2) becomes

$$mg = F_g = \sum_i F_i = \frac{d}{dt}(mv) = \frac{d}{dt}\left(m\frac{dx}{dt}\right)$$

Finally, the leftmost and the rightmost terms are equal to each other as follows:

$$mg = \frac{d}{dt}\left(m\frac{dx}{dt}\right)$$

and it is expanded as follows:

$$mg = \frac{dm}{dt}\frac{dx}{dt} + m\frac{d^2x}{dt^2}$$

Using the short notations, $\dot{m} = dm/dt$, $\dot{x} = dx/dt$, and $\ddot{x} = d^2x/dt^2$, and after rearrangements, the governing equation is given by

$$\ddot{x} = g - \frac{\dot{m}}{m}\dot{x} \tag{1.3}$$

For purely educational purposes, assume that the mass change rate is given by

$$\dot{m} = -m + 2 \tag{1.4}$$

We can identify now that there are three independent time-varying states, which are the position, x, the velocity, $\dot{x}$, and the mass, m. All the other time-varying states, for example, $\ddot{x}$ and $\dot{m}$, can be expressed using the independent state variables. Define the state variables as follows:

$$x_1 = x$$
$$x_2 = \dot{x}$$
$$x_3 = m$$

Obtain the time derivative of each state expressed in the state variable as follows:

$$\dot{x}_1 = \dot{x} = x_2 \tag{1.5a}$$

$$\dot{x}_2 = \ddot{x} = g - \frac{-m+2}{m}\dot{x} = g - \frac{-x_3+2}{x_3}x_2 \tag{1.5b}$$

$$\dot{x}_3 = \dot{m} = -m + 2 = -x_3 + 2 \tag{1.5c}$$

and this is called *the state-space form*.

Let the initial conditions be equal to $x_1(0) = x(0) = 0.0\,\text{m}$, $x_2(0) = \dot{x}(0) = 0.5\,\text{m/s}$, and $x_3(0) = m(0) = 5\,\text{kg}$. Equation (1.5) can be written in a compact form using the

matrix–vector notations. Define the state vector, $\mathbf{x}$, as follows:

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}$$

and the corresponding state-space form is written as

$$\dot{\mathbf{x}} = \mathbf{f}(\mathbf{x}) = \begin{bmatrix} x_2 \\ g + (x_3 - 2)(x_2/x_3) \\ -x_3 + 2 \end{bmatrix} \tag{1.6}$$

The second-order differential equation, (1.3), and the first-order differential equation, (1.4), are combined into the first-order three-dimensional vector differential equation, (1.6). Any higher order differential equations can be transformed into the first-order multi-dimensional vector differential equation, $\dot{\mathbf{x}} = \mathbf{f}(\mathbf{x})$. Numerical integration methods such as Runge–Kutta integration (Press et al., 2007) solves the first-order ODE. They can solve any high-order differential equations by transforming them into the corresponding first-order multi-dimensional differential equation.

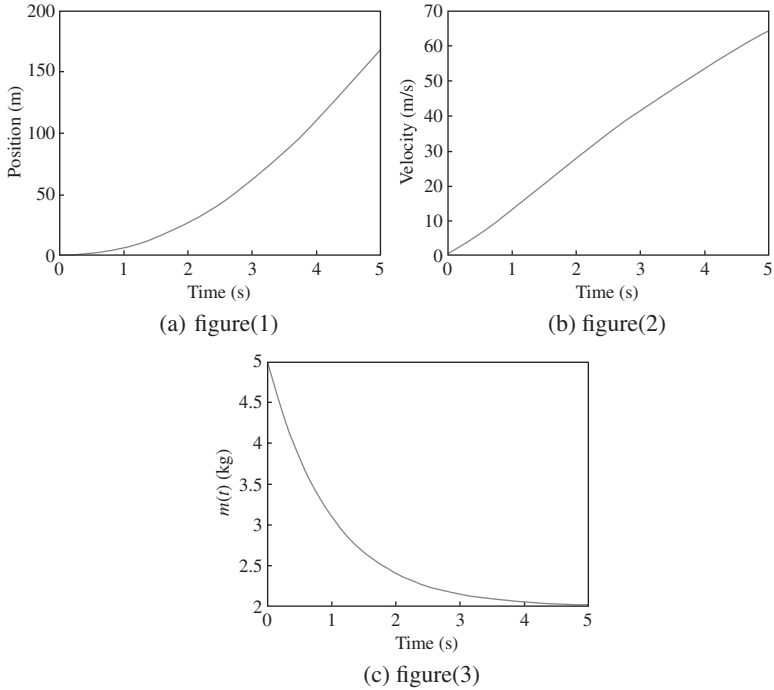
1.2.1.1 First Program in Matlab

We are ready to solve (1.6) with the initial condition equal to $\mathbf{x}(0) = [0.0\ 0.5\ 5.0]^T$, where the superscript T is the transpose of the vector. We solve the differential equation from $t = 0$ to $t = 5$ seconds using Matlab. Matlab includes many numerical functions and libraries to be used for dynamic simulation and analysis. A numerical integrator is one of the functions already implemented in Matlab. Hence, the only task we have to do for solving the differential equation is to learn how to use the existing functions and libraries in Matlab. The complete programme to solve the free-falling object problem is given in Program 1.1. Producing Figure 1.2 is left as an exercise in Exercise 1.1.

```
clear;

grv_const = 9.81; % [m/s^2]
init_pos = 0.0; %[m]
init_vel = 0.5; % [m/s]
init_mass = 5.0; %[kg]

init_time = 0; % [s]
final_time = 5.0; % [s]
time_interval = [init_time final_time];

x0 = [init_pos init_vel init_mass];
[tout,xout] = ode45(@(time,state) free_falling_obj(time,state,
    grv_const), time_interval, x0);

figure(1);
plot(tout,xout(:,1))
ylabel('position [m]');
```

```
18 xlabel('time [s]');
19
20 figure(2);
21 plot(tout,xout(:,2))
22 ylabel('velocity [m/s]');
23 xlabel('time [s]');
24
25 figure(3);
26 plot(tout,xout(:,3))
27 ylabel('m(t) [kg]');
28 xlabel('time [s]');
29
30 function dxdt = free_falling_obj(time,state,grv_const)
31     x1 = state(1);
32     x2 = state(2);
33     x3 = state(3);
34
35     dxdt = zeros(3,1);
36     dxdt(1) = x2;
37     dxdt(2) = grv_const + (x3-2)*(x2/x3);
38     dxdt(3) = -x3 + 2;
39 end
```

Program 1.1 (Matlab) Free-falling object



(a) figure(1) (b) figure(2)

(c) figure(3)

Figure 1.2 Free-falling object position, velocity, and mass time histories.

Now, we study the first program line by line. The m-script starts with the command 'clear'. The clear command removes all variables in the workspace. In the workspace, there would be some variables defined and used in previous activities. They may have the same names but different meanings and values in the current calculation. For example, the gravitational acceleration 'grv_const' in the third line is undefined in the current program and uses a variable of the same name used to analyse objects falling on the moon. A falling object program in the Moon was executed earlier, and 'grv_const' is still in the workspace. Without the clear command, the incorrect constant is used in the program producing wrong results. Hence, it is recommended to clear the workspace before starting new calculations. We must be careful, however, that the clear command erases all variables in the workspace. Before the clear command, we check if all values, which might be generated from a long computer simulation, were saved.

From line 3 to line 12, several constants are defined. Based on the equations we have seen earlier, it is tempting to write a code as follows:

```
g = 9.81
x = 0.0
v = 0.5
t = [0 5]
x0 = [x v m]
```

Program 1.2 (Matlab) Poor style constant definitions

These seem to look compact and closer to the equations we derived. It is a bad habit to write a program in this way. The list of problems in the above programming style is as follows:

- It defines a variable with a single character, 'g', 'x', 'v', etc. Using a single character variable might cause confusion on the meaning of the variable and lead to using them in wrong places with incorrect interpretations.
- Numerical numbers are written without units. There is no indication of units of the numerical values, e.g. 9.81, is it m/s^2 or ft/s^2?
- It uses magic numbers. What do the numbers, 0 and 5, mean in defining 't'?

Program 1.1 uses a better style. The initial position is defined using the variable name, 'init_pos', whose value is 0.0 and the unit is in metres. Appropriately named variables reduce mistakes and confusion in the program. Program 1.1 indicates the corresponding unit for each numerical value, e.g. the 'init mass' value 5.0 is in kg. We understand the meaning of each variable by its name. The texts after '%' are the comments, where we could add various information such as the unit of each numerical value.

In line 13, the built-in Runge–Kutta integrator, *ode45()*, is used to integrate the differential equation provided by the function, 'free_falling_obj', at the end of the m-script. Frequently, each function is saved as a separate m-script. It could also be included in the m-script for the cases that the functions might be used in the specific m-script only. To include functions in the m-script, they must be placed at the end of the m-script as in this example.

Functions in Matlab begin with the keyword *function* and close with the keyword *end*. In line 30, 'dxdt' is the return variable of the function and 'free_falling_obj' is the function name. The function has three input arguments. A function can have any input argument used by the function. This particular function, 'free_falling_obj', is not an ordinary function, however. This is the function to describe the ODE. The function is to be passed into the built-in integrator, *ode45*. The first two arguments of the function for *ode45* must be time and states, i.e. t and $\mathbf{x}$ in (1.6).

In lines 31–33, the variable 'state' is assumed to be a three-dimensional vector, and each element of the vector corresponds to the states, x_1, x_2, and x_3. In line 35, the return variable 'dxdt' is initialized as [0 0 0] by the built-in function *zeros(3,1)*. *zeros(m,n)* creates the m × n matrix filled in zeros. Lines 36, 37, and 38 define the state-space form ODE, (1.6).

The function works perfectly well without the initialization line for 'dxdt', line 35. However, it is not good programming if line 35 is removed. Without the initialization, 'dxdt' in line 36 is a one-dimensional scalar value. In the next lines, it becomes a two-dimensional value and a three-dimensional value. Each line, the size of 'dxdt' changes, and this requires the computer to find additional memory to store the additional value. This could increase the total computation time longer and could be noticeably longer if this function is called a million times or more. Hence, it is better to acquire all the required memory ahead as in line 35.

Efficiency vs. development cycle: We strive to create efficient programs, but the prototyping phase requires a fast development cycle.

It is vital to have the habit of being conscious of the efficiency of algorithm implementation. On the other hand, try not to overthink the efficiency of the program. Script languages such as Matlab and Python are for rapid implementation and testing. Hence, it needs a proper balance between optimizing codes and saving the development time.

Now, we are ready to solve the differential equation using the built-in numerical integrator, *ode45*. *ode45* stands for ODE with Runge–Kutta fourth- and fifth-order

methods. Details of the Runge–Kutta integration methods can be found in Press et al. (2007).

Recall, the following line from Program 1.1:

```
13 [tout,xout] = ode45(@(time,state) free_falling_obj(time,state,
       grv_const), time_interval, x0);
```

When we use ode45, the input argument starts with @ symbol, which is the function handle. The function handle, @, is used when we pass function A, e.g. 'free_falling_obj', to function B, e.g. *ode45*, where function B would call function A multiple times. With the function handle, we can control or construct the function to be passed with some flexibility. '@(time,state)' explicitly indicates that the function to be passed has two arguments, 'time' and 'state', and they will be passed between *ode45* and 'free_falling_obj' function in the specific order, i.e. 'time' be the first and 'state' be the second argument. This order is required by the integrator, ode45.

With the function handle, we can take some freedom to order the function arguments differently in the function definition of 'free_falling_obj'. For example, we could write the function as follows:

```
function dxdt = free_falling_obj(time,grv_const,state)
    x1 = state(1);
    x2 = state(2);
    x3 = state(3);

    dxdt = zeros(3,1);
    dxdt(1) = x2;
    dxdt(2) = grv_const + (x3-2)*(x2/x3);
    dxdt(3) = -x3 + 2;
end
```

and the integration part is updated to follow the updated function definition as follows:

```
[tout,xout] = ode45(@(time,state) free_falling_obj(time,grv_const,
    state), time_interval, x0);
```

The program works the same as the ones before the modifications. Also, we notice that we have an additional input argument, 'grv_const'. Similarly, we could add more input parameters if they are necessary. As long as the first argument, 'time', and the second argument, 'state', are indicated in the function handle, the function can have any number of input arguments in any order to pass to the integrator, *ode45*.

Once the integration is completed, the results return to two output variables, 'tout' and 'xout'. Execute the command, *whos*, in the Matlab command prompt, the following information is displayed:

```
>> whos
  Name                Size                Bytes  Class     Attributes

  final_time          1x1                     8  double
  grv_const           1x1                     8  double
  init_mass           1x1                     8  double
  init_pos            1x1                     8  double
  init_time           1x1                     8  double
  init_vel            1x1                     8  double
  time_interval       1x2                    16  double
  tout               61x1                   488  double
  x0                  1x3                    24  double
  xout               61x3                  1464  double
```

The first column shows all variables created including the two output results from the integrator. The second column shows the size of each variable: 'tout' is 61 rows and 1 column and 'xout' is 61 rows and 3 columns. Hence, each row of 'xout' corresponds to the time instance of the corresponding row values of 'tout'. Why is the number of row 61? This is determined by the integrator automatically to adjust the integration accuracy and computation time. We can assign the number of rows or the number of time steps explicitly, and this is covered in the later chapters. The three columns of 'xout' correspond to the state, x, $\dot{x}$, and m. The first column of 'xout' is for x, the second column of 'xout' is for $\dot{x}$, and the last column of 'xout' is for m.

By executing the following line in the Matlab command prompt, we can print out all values of $x(t)$ in the command window:

```
>> xout(:,1)
```

where ':' indicates all rows. If we want to see the values of x from the 11th row to the 15th row, then

```
>> xout(11:15,1)
```

Similarly, the time history of $\dot{x}$ is xout(:,2) and the time history of m is xout(:,3).

The plot command in Matlab plots the results as follows:

```
plot(tout, xout(:,1))
```

Before plotting each figure, open a new figure window using *figure(1)*, *figure(2)*, and *figure(3)*, respectively. The label for each axis is created using the commands *xlabel* and *ylabel* for the horizontal and the vertical axes, respectively, where each axis must indicate what quantity and what units are used.

1.2.1.2 First Program in Python

Program 1.3 solves the free-falling object differential equation. The program is remarkably similar to the Matlab script in Program 1.1. There are, however, many differences between the two languages.

```
from numpy import linspace
from scipy.integrate import solve_ivp

grv_const = 9.81 # [m/s^2]
init_pos = 0.0 # [m]
init_vel = 0.5 # [m/s]
init_mass = 5.0 #[kg]

init_cond = [init_pos, init_vel, init_mass]

init_time = 0 # [s]
final_time = 5.0 # [s]
num_data = 100
tout = linspace(init_time, final_time, num_data)


def free_falling_obj(time, state, grv_const):
    x1, x2, x3 = state
    dxdt = [x2,
            grv_const + (x3-2)*(x2/x3),
            -x3 + 2]
    return dxd


sol = solve_ivp(free_falling_obj, (init_time, final_time),
    init_cond, t_eval=tout, args=(grv_const,))
xout = sol.y

import matplotlib.pyplot as plt
plt.figure(1)
plt.plot(tout,xout[0,:])
plt.ylabel('position [m]');
plt.xlabel('time [s]');

plt.figure(2);
plt.plot(tout,xout[1,:])
plt.ylabel('velocity [m/s]');
plt.xlabel('time [s]');

plt.figure(3);
plt.plot(tout,xout[2,:])
plt.ylabel('m(t) [kg]');
plt.xlabel('time [s]');
```

Program 1.3 (Python) Free-falling object

On lines 4 through 14, the constants are defined with the proper naming and the units indicated in the comments. In Python, comments are placed after #.

The first two lines shown are not trivial to understand for the beginners of the Python language. Python has many packages, and each package is a collection of functions. There are several different ways to load these functions and the first line in the program,

```
1 from numpy import linspace
```

shows one of the methods. *from* and *import* are the keywords in Python. It loads the function *linspace* from the library called *numpy*. *numpy* is one of the scientific and engineering libraries and includes many useful functions such as matrix manipulations, and maths functions.

Numpy vs. scipy: The two packages are very similar and have many common functions. The execution speed of numpy is faster than scipy; in general, as numpy is written in C-language while scipy is written in Python. Scipy, however, has more specialized functions, which are not implemented in numpy.

We might wonder why each function is manually loaded before it is used, unlike in Matlab. This is one of the design principles of the Python language. If all functions are pre-loaded or they are automatically searched and loaded when they are used, then the search time or the size of the memory storing the function lists is long or larger. Hence, it is more efficient to load the functions manually when they are used.

The function linspace has three input arguments, for example, line 14 generates an array of numerical values starting from the initial time, 0.0, to the final time, 5.0, whose number of elements is equal to 'num data', 100. Unlike the integrator in Matlab, the Python integrator, discussed shortly later, needs the explicit time lists as one of the input arguments.

In the second line, the numerical integrator, *solve_ivp*, is loaded

```
2 from scipy.integrate import solve_ivp
```

This is slightly different from the way to load a function shown in the first line. *scipy* is another science and engineering function library. Some library divides the functions in the library into several categories. *integrate* is one of the categories in the scipy library. To access the functions under the category, *integrate*, the period is used after the library name, i.e. *scipy.integrate*. The numerical integrator, *solve_ivp*,

is defined in the integrate category of the scipy library. If we try to load the function using *from scipy import solve_ivp*, it cannot find the integrator and generates an import error.

The ODE are defined between lines 17 and 22. The first line of the function definition begins with the keyword, '**def**', the function name, 'free_falling_obj', the three input arguments, and the colon, ':' as follows:

```
def free_falling_obj(time, state, grv_const):
```

In general, the function to be defined could have any input arguments. The function to be passed to *solve_ivp*, however, must have the first two input arguments, time and state, in this order. *solve_ivp* assumes that the first arguments and the second argument of the function passed are t and $\mathbf{x}$ in $\dot{\mathbf{x}} = d\mathbf{x}/dt$ in (1.6). The main body of the function is between the line below the function heading and the *return* line. Those lines that belong to the main part of the function are indented. The indentation in Python is not a decoration to simply improve the readability as in many other programming languages. The indentation in Python is the way to indicate which lines belong to the function body. The following is the first line of the function body:

```
    x1, x2, x3 = state
```

where 'state' is presumed to have three elements, and they are assigned to the three new variables on the left-hand side of the equal sign, 'x1', 'x2', and 'x3'. Instead of unpacking the three elements one by one, it unpacks all the three elements in one line.

'dxdt' is the list element in Python. In the list, each element is separated by the comma, ','. Finally, 'dxdt' becomes the return value of the function by the keyword, *return*, and the function is passed to the integrator, *solve_ivp*.

The first input argument of the integrator is the function name describing the ODE. The second one is the integration time interval. The third one is the initial condition. 't eval' is the list of time points, where the solution, $x(t)$, is stored to the output of the integrator. The last one is the arguments, whose name is reserved by *args*. As the function 'free_falling_obj' has the additional input variable apart from the time and the state, i.e. 'grv_const', this value must be sent to 'solve_ivp'. *args* is the input variable of 'solve_ivp' to pass additional input variables. 'grv const' is passed to the integrator by 'arg=(grv const,)'. The data type of *args* is a tuple. (1.3, 4.2, 4.3) or (1.3, 2.3) is a tuple. When there is only one element in a tuple, for example, (1.2,), the comma at the end must not be omitted. (1.2) is interpreted as floating-point 1.2, not a tuple. To make it a tuple, it must be (1.2,). Hence, there is the comma after 'grv const' in 'args=(grv const,)'.

Similar to Matlab, typing 'whos' at the command prompt in Python prints out the following list to the screen:

```
Variable          Type          Data/Info
------------------------------------------
final_time        float         5.0
free_falling_obj  function      <function free_falling_obj
grv_const         float         9.81
init_cond         list          n=3
init_mass         float         5.0
init_pos          float         0.0
init_time         int           0
init_vel          float         0.5
linspace          function      <function linspace at 0x7f
num_data          int           100
plt               module        <module 'matplotlib.pyplo<...
sol               OdeResult     message: 'The solver su<.
solve_ivp         function      <function solve_ivp at 0x7f
tout              ndarray       100: 100 elems, type 'float64',
xout              ndarray       3x100: 300 elems, type 'float64
```

The solution of the ODE is stored in 'sol', whose type is OdeResult, and it includes various information about the integration results. Typing 'sol' in the command prompt and hitting enter shows what variables are in 'sol'. We can access $\mathbf{x}(t)$ through 'sol.y'. To avoid keep adding the dot to access $\mathbf{x}(t)$ inside 'sol', create a new variable, 'xout', and store 'sol.y' into 'xout'. We can also see from the variable list that the size of 'xout' is 3×100. Each of the rows corresponds to $x(t)$, $\dot{x}(t)$, and $m(t)$, respectively.

To plot the results, a plotting library must be loaded. *matplotlib* is the most widely used plotting library in Python. More specifically, plot functions under *matplotlib.pyplot* category are the most frequently used. Load the functions as follows:

```
import matplotlib.pyplot
```

The way to access the functions under a specific category is using the dot next to the package name. *matplotlib.pyplot* means that we want to access the functions under the sub-category called *pyplot* in *matplotlib* instead of loading all functions in *matplotlib*. Now, we can use the *plot* command in *pyplot* as follows:

```
matplotlib.pyplot.plot(tout, xout[0,:])
```

This is inconvenient as the name becomes very long. To reduce the length of the name, *pyplot* is loaded as follows:

```
import matplotlib.pyplot as plt
```

After the keyword *as*, any convenient name we would call it could be used. By convention or almost standard, *matplotlib.pyplot* is called 'plt'. Hence, the long name to call 'plot' is shortened to

```
plt.plot(tout, xout[0,:])
```

This plots $x(t)$ vs. time t. Unlike Matlab, array indices in Python start at 0, not 1. The first row of 'xout' is 'xout[0,:]', the second row of 'xout' is 'xout[1,:]', and so forth. *xlabel* and *ylabel* commend work the same way as the ones in Matlab.

1.2.2 Ligand–Receptor Interactions

Ligand–receptor interactions are one of the most common interactions in biomolecular systems. As shown in Figure 1.3, the ligands, L, bind to the receptors, R, which spread on the cell boundary, form the ligand–receptor complex, C, and the complex evokes further reactions through various cascade signalling pathways inside the cell. L is produced with the rate given by a function of time, $f(t)$. From the control point of view, $f(t)$ is considered as the input, R is the internal state, and the concentration of C is the output of the ligand–receptor interactions.

The following molecular interactions describe the interactions between L, R, C, and $f(t)$:

$$\mathrm{R} + \mathrm{L} \xrightarrow{k_{\mathrm{on}}} \mathrm{C} \tag{1.7a}$$

$$\mathrm{C} \xrightarrow{k_{\mathrm{off}}} \mathrm{R} + \mathrm{L} \tag{1.7b}$$

$$\mathrm{R} \xrightarrow{k_{\mathrm{t}}} \emptyset \tag{1.7c}$$

$$\mathrm{C} \xrightarrow{k_{\mathrm{e}}} \emptyset \tag{1.7d}$$

$$\mathrm{f(t)} \xrightarrow{1} \mathrm{L} \tag{1.7e}$$

$$Q_{\mathrm{R}} \xrightarrow{1} \mathrm{R} \tag{1.7f}$$

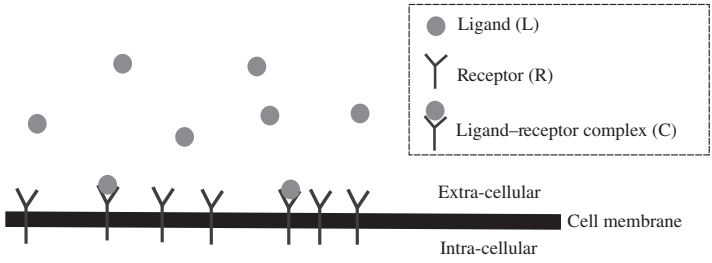


Figure 1.3 Ligand–receptor interactions form ligand–receptor complex.

where k_{on} and k_{off} are the reaction rates of binding or unbinding the receptor and the ligand, R and L, respectively, to form or destroy the complex, C, the receptor is destroyed with the rate of k_t, the complex is also destroyed with the rate of k_e, $f(t)$ is the stimulus that produces the ligand at the unit rate, and Q_R is the internal receptor generation at the unit rate.

We derive a set of ODE using the molecular interactions. To this end, we introduce the following two assumptions:

- All the molecules and the sources are uniformly distributed in the reaction space
- There are a sufficient number of molecules for every molecular species to consider concentration alone.

The first assumption makes the modelling being ODE. Otherwise, partial differential equations with the spatial coordinates are solved. Solving partial differential equations is computationally a lot more challenging than solving ODE. The second assumption indicates that the population of each molecular species is far away from 0. The randomness of molecular interactions and the integer nature of the number of molecules are ignored in the modelling.

Molecular interactions are stochastic. The probability of the occurrence of each reaction is calculated in stochastic simulations. We will discuss the details of stochastic modelling and simulation in the later chapter. On the other hand, deterministic simulations are performed by assuming a large number of molecules. The average molecular numbers show deterministic trajectories, where the random fluctuations are negligible.

Consider the receptor, R, which is directly involved in the three reactions. L binds to R and becomes C in (1.7a). The concentration of R is decreased by this reaction. The change rate is proportional to the concentrations of R and L as follows:

$$\frac{d[\mathrm{R}]}{dt} \propto -[\mathrm{R}] \times [\mathrm{L}] \tag{1.8}$$

where $[\cdot]$ is the concentration of the molecules. The proportional constant is given by k_{on} in the reaction. The concentration unit is nanomolar (nM). Molar is equal to $N/(N_A V)$, where N is the number of molecules, N_A is Avogadro's number equal to 6.022×10^{23}, and V is the reaction space volume in litres.

In (1.7b), C is decomposed into R and L. The concentration of R is increased by this reaction. The decreasing rate is proportional to the concentration of C as follows:

$$\frac{d[\mathrm{R}]}{dt} \propto [\mathrm{C}] \tag{1.9}$$

where the proportional constant is k_{off}. The receptor is destroyed by itself at the rate of k_t as follows:

$$\frac{d[\mathrm{R}]}{dt} \propto -[\mathrm{R}] \tag{1.10}$$

Finally, in (1.7f), R is created at the rate of Q_R:

$$\frac{d[\mathrm{R}]}{dt} \propto [Q_R] \tag{1.11}$$

where the proportional constant is 1.

Combining (1.8)–(1.11) as follows: Shankaran et al. (2007)

$$\frac{d[\mathrm{R}]}{dt} = -k_{\mathrm{on}}[\mathrm{R}][\mathrm{L}] + k_{\mathrm{off}}[\mathrm{C}] - k_t[\mathrm{R}] + [Q_R] \tag{1.12}$$

Similarly, the following differential equations are established for L and C:

$$\frac{d[\mathrm{L}]}{dt} = -k_{\mathrm{on}}[\mathrm{R}][\mathrm{L}] + k_{\mathrm{off}}[\mathrm{C}] + [f(t)] \tag{1.13a}$$

$$\frac{d[\mathrm{C}]}{dt} = k_{\mathrm{on}}[\mathrm{R}][\mathrm{L}] - k_{\mathrm{off}}[\mathrm{C}] - k_e[\mathrm{C}] \tag{1.13b}$$

where $k_{\mathrm{off}} = 0.24\,[1/\mathrm{min}]$, $k_{\mathrm{on}} = 0.0972\,[1/(\mathrm{min\,nM})]$, $k_t = 0.02\,[1/\mathrm{min}]$, $k_e = 0.15$ $[1/\mathrm{min}]$, and $[f(t)] = 0.0\,[\mathrm{nM/min}]$, i.e. no external stimulation. The values are the ones for the epidermal growth factor receptor (EGFR), which plays an important role in understanding tumour formation and growth.

Because of Q_R in $d[\mathrm{R}]/dt$, R would increase to infinity, which does not coincide with the reality as there would be the possible maximum number of receptors to be present in the cell. It is known that the maximum number of receptors for the EGFR is around 100,000 (Wee and Wang, 2017, Carpenter and Cohen, 1979). As the volume of the reaction space is given by $4 \times 10^{-10}\ell$ in Shankaran et al. (2007), the maximum concentration of R is $10,000/(N_A V) \approx 0.415\,\mathrm{nM}$. We model Q_R as follows:

$$[Q_R] = \begin{cases} 0.0166\ [\mathrm{nM/min}], & \text{for } [\mathrm{R}] \le [\mathrm{R}]_{\max} \\ 0, & \text{otherwise} \end{cases} \tag{1.14}$$

where $[\mathrm{R}]_{\max}$ is equal to 0.415 nM.

The initial conditions for the following simulation are set as follows: $[\mathrm{R}(0)] = 0.1\,\mathrm{nM}$, $[\mathrm{L}(0)] = 0.0415\,\mathrm{nM}$, and $[\mathrm{C}(0)] = 0\,\mathrm{nM}$. In biomolecular network simulations, we must confirm that the molecular quantities such as the number of molecules or the concentrations must be non-negative. [C] at the beginning of the simulation could become negative if the time rate is negative. In the above initial conditions, [C] is strictly increasing because $d[\mathrm{C}(0)]/dt = k_{\mathrm{on}}[\mathrm{R}(0)][\mathrm{L}(0)]$ is positive at the beginning. As we can see from (1.13b), $d[\mathrm{C}]/dt$ is only negative when [C] is high enough, i.e. $[C] > k_{\mathrm{on}}[R][L]/(k_{\mathrm{off}} + k_e)$.

The Matlab script to simulate the EGFR concentration kinetics is given in Program 1.4.

```
clear;

init_receptor = 0.1; % [nM]
init_ligand = 0.0415; %[nM]
init_complex = 0.0; %[kg]

init_time = 0; % [min]
final_time = 180.0; % [min]
time_interval = [init_time final_time];

kon = 0.0972; % [1/(min nM)]
koff = 0.24; % [1/min]
kt = 0.02; %[1/min]
ke = 0.15; % [1/min]

ft = 0.0; % [nM/min]
QR = 0.0166; % [nM/min]
R_max = 0.415; %[nM]

sim_para = [kon koff kt ke ft QR R_max];

x0 = [init_receptor init_ligand init_complex];
[tout,xout] = ode45(@(time,state) RLC_kinetics(time,state,sim_para)
    , time_interval, x0);

figure(1); clf;
subplot(311);
plot(tout,xout(:,1))
ylabel('Receptor [nM]');
xlabel('time [min]');
axis([time_interval 0 0.5]);
subplot(312);
plot(tout,xout(:,2))
ylabel('Ligand [nM]');
xlabel('time [min]');
axis([time_interval 0 0.05]);
subplot(313);
plot(tout,xout(:,3))
ylabel('Complex [nM]');
xlabel('time [min]');
axis([time_interval 0 0.004]);

function dxdt = RLC_kinetics(time,state, sim_para)
    R = state(1);
    L = state(2);
    C = state(3);

    kon = sim_para(1);
    koff = sim_para(2);
    kt = sim_para(3);
    ke = sim_para(4);
    ft = sim_para(5);
    QR = sim_para(6);
```

```
53      R_max = sim_para(7);
54
55      if R > R_max
56          QR = 0;
57      end
58
59      dxdt = zeros(3,1);
60      dxdt(1) = -kon*R*L + koff*C - kt*R + QR;
61      dxdt(2) = -kon*R*L + koff*C + ft;
62      dxdt(3) = kon*R*L - koff*C - ke*C;
63  end
```

Program 1.4 (Matlab) EGFR receptor, ligand, and complex kinetics

Figure 1.4 shows the simulation results. The receptor concentration increases almost linearly at the beginning and fluctuates later around the maximum concentration limit. The ligand–receptor reaction steadily consumes the ligand when they bind together and become the ligand–receptor complex. The complex has a peak concentration that occurred around 20 minutes and then slowly decayed.

Figure 1.5 shows the simulation results of the Python program, Program 1.5. Unlike the figure commands in Matlab for Figure 1.4, plotting subfigures in *matplotlib* is not as simple as in Matlab. We need advanced features in *matplotlib*. The advanced features of *subplots* in *matplotlib* are introduced in detail later in Program 2.2. As we notice in the figure, the figure fonts are too small to read. How to adjust the figure font sizes is also discussed in Program 2.2.

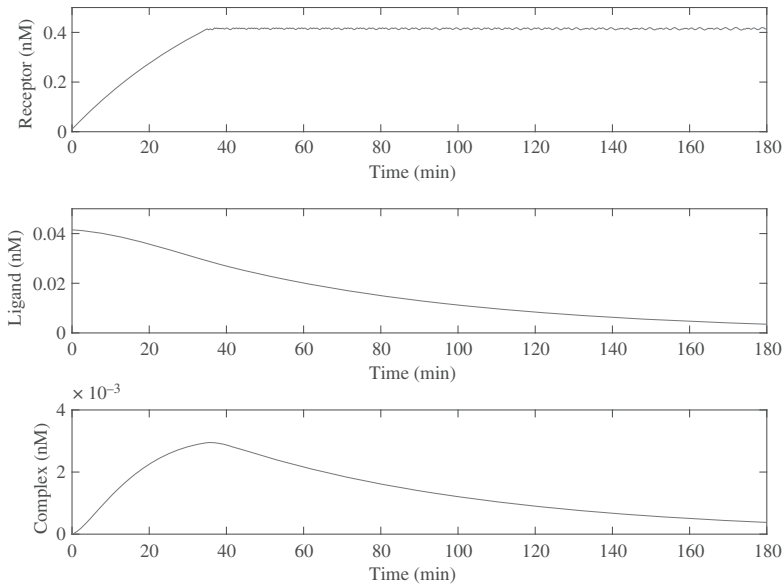


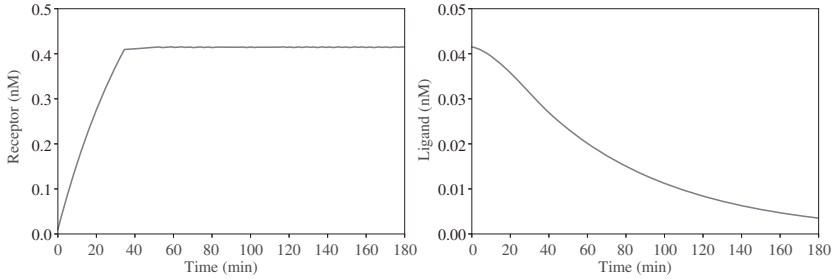
Figure 1.4 (Matlab) EGFR receptor, ligand, and complex time histories.

Program 1.5 uses two different integrators, i.e. *solve_ivp* and *odeint*. The ODE includes the discontinuous part, Q_R, given in (1.14). *odeint* cannot handle the differential equations with the discontinuity, and the solutions diverge. *solve_ivp* returns the correct numerical results. We recommend using *solve_ivp* instead of *odeint*.

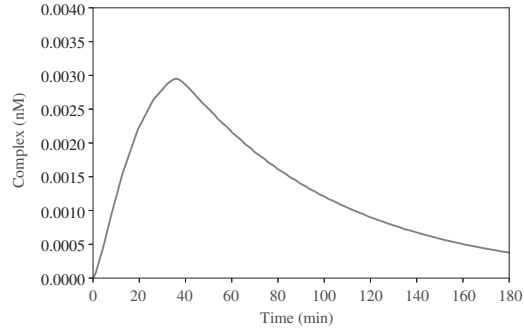
```
from numpy import linspace
from scipy.integrate import solve_ivp


init_receptor = 0.01 #[nM]
init_ligand = 0.0415 #[nM]
init_complex = 0.0 #[kg]

init_time = 0 #[min]
final_time = 180.0 #[min]
time_interval = [init_time, final_time]

kon = 0.0972 #[1/(min nM)]
koff = 0.24 #[1/min]
kt = 0.02 #[1/min]
ke = 0.15 #[1/min]

ft = 0.0 #[nM/min]
QR = 0.0166 #[nM/min]
R_max = 0.415 #[nM]

sim_para = [kon, koff, kt, ke, ft, QR, R_max]

init_cond = [init_receptor, init_ligand, init_complex]


num_data = int(final_time*10)
tout = linspace(init_time, final_time, num_data)


def RLC_kinetics(time,state,sim_para):
    R, L, C = state

    kon, koff, kt, ke, ft, QR, R_max = sim_para

    if R > R_max:
        QR = 0

    dxdt = [-kon*R*L + koff*C - kt*R + QR,
            -kon*R*L + koff*C + ft,
            kon*R*L - koff*C - ke*C]
    return dxdt

sol_out = solve_ivp(RLC_kinetics, (init_time, final_time),
    init_cond, args=(sim_para,))

tout = sol_out.t
xout = sol_out.y
```

```
48
49 from scipy.integrate import odeint
50 xout_odeint = odeint(RLC_kinetics, init_cond, linspace(init_time,
       final_time, num_data), args=(sim_para,),tfirst=True)
51
52 import matplotlib.pyplot as plt
53 plt.figure(1)
54 plt.plot(tout,xout[0,:])
55 plt.ylabel('Receptor [nM]')
56 plt.xlabel('time [min]')
57 plt.axis([0, final_time, 0, 0.5])
58
59 plt.figure(2)
60 plt.plot(tout,xout[1,:])
61 plt.ylabel('Ligand [nM]')
62 plt.xlabel('time [min]')
63 plt.axis([0, final_time, 0, 0.05])
64
65 plt.figure(3)
66 plt.plot(tout,xout[2,:])
67 plt.ylabel('Complex [nM]')
68 plt.xlabel('time [min]')
69 plt.axis([0, final_time, 0, 0.004])
```

Program 1.5 (Python) EGFR receptor, ligand, and complex kinetics



(a) plt.figure(1) (b) plt.figure(2)



(c) plt.figure(3)

Figure 1.5 (Python) EGFR receptor, ligand, and complex time histories.

1.3 Organization of the Book

Chapters 2 and 3 cover the dynamics, control, and estimation algorithms of autonomous vehicles. Chapters 4 and 5 cover modelling and analysis of biological systems. Each of the chapters provides examples and exercises. We discuss additional readings and topics in the last chapter, Chapter 6.

Exercises

Exercise 1.1 (Matlab) Run Matlab, open the editor, type Program 1.1, save it as an m-script, execute the m-script in the Matlab command prompt, and obtain Figure 1.2.

Exercise 1.2 (Matlab) Using the *ode45* results from Program 1.1, plot Figure 1.6 using the *subplot* command in Matlab. Hint: Check the help for *subplot* in Matlab.

Exercise 1.3 (Python) Plot Figure 1.6 using the functions under *matplotlib.pyplot* in Python.

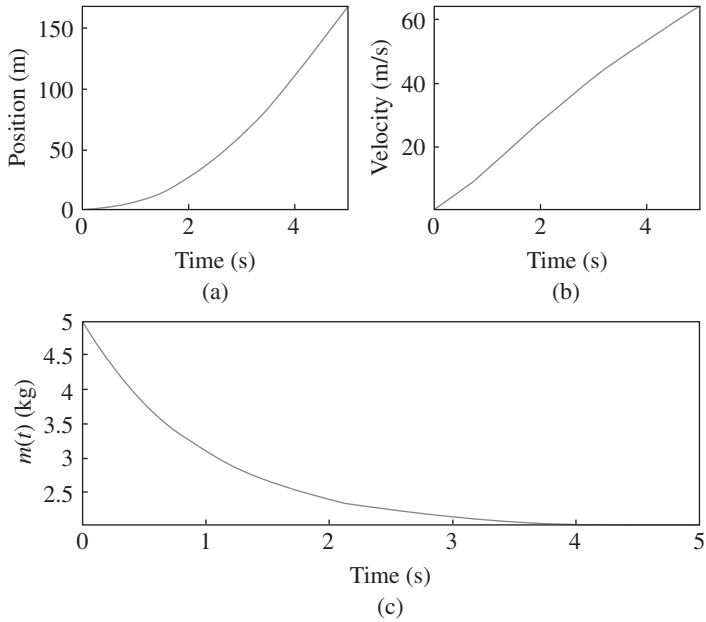


Figure 1.6 The time histories of (a) position (x), (b) velocity ($\dot{x}$), and (c) mass (m).

Exercise 1.4 Derive (1.13) from the molecular interactions in (1.7).

Exercise 1.5 (Python) What is the purpose of 'tfirst=True' in the arguments of *odeint* in Program 1.5?

Exercise 1.6 (Matlab/Python) Run the EGFR kinetic simulation 1000 times using the Matlab or the Python script, randomly selecting the initial concentration values in the following range: $[R(0)] \in [0, 0.2]$ nM, $[L(0)] \in [0, 0.05]$ nM, and $[C(0)] \in [0, 0.01]$ nM. Check if the concentrations are always positive.

Bibliography

G. Carpenter and S. Cohen. Epidermal growth factor. *Annual Review of Biochemistry*, 48(1):193–216, 1979. https://doi.org/10.1146/annurev.bi.48.070179.001205. PMID: 382984.

W.H. Press, S.A. Teukolsky, W.T. Vetterling, and B.P. Flannery. *Numerical Recipes 3rd Edition: The Art of Scientific Computing*. Cambridge University Press, 2007. ISBN 9780521880688.

Harish Shankaran, Haluk Resat, and H. Steven Wiley. Cell surface receptors for signal transduction and ligand transport: a design principles study. *PLOS Computational Biology*, 3(6):1–14, 2007. https://doi.org/10.1371/journal.pcbi.0030101.

Ping Wee and Zhixiang Wang. Epidermal growth factor receptor cell proliferation signaling pathways. *Cancers*, 9(5), 2017. ISSN 2072-6694. https://doi.org/10.3390/cancers9050052. https://www.mdpi.com/2072-6694/9/5/52.