BEBI5009 Homework 1

Student: Shao-Ting Chiu

Student ID: B03901045

Source Code on Github: https://github.com/stevengogogo/BEBI-5009-Mathematical-Biology (https://github.com/stevengogogo/BEBI-5009-Mathematical-Biology)

Reference: B. P.Ingalls, Mathematical Modelling in Systems Biology : An Introduction, vol. 53, no. 9. 2014.

Problem 1. Network Modelling

1. **2.4.7 Network Modelling.**

Consider the closed reaction network in Figure 2.16 with reaction rates v_i as indicated. Suppose that the reaction rates are given by mass action as $v_1 = k_1[A][B]$, $v_2 = k_2[D]$ and $v_3 = k_3[C]$.

(1) Construct a differential equation model for the network. Use moiety conservations to reduce your model to three differential equations and three algebraic equations.

(2) Solve for the steady-state concentrations as functions of the rate constants and the initial concentrations. (Note, because the system is closed, some of the steady-state concentrations are zero.)

(3) Verify your result in part (2) by running a simulation of the system from initial conditions (in mM) of ([A], [B], [C], [D], [E], [F]) = (1, 1, 0.5, 0, 0, 0).

Take rate constants $k_1 = 3$/mM/sec, $k_2 = 1$/sec, $k_3 = 4$/sec.

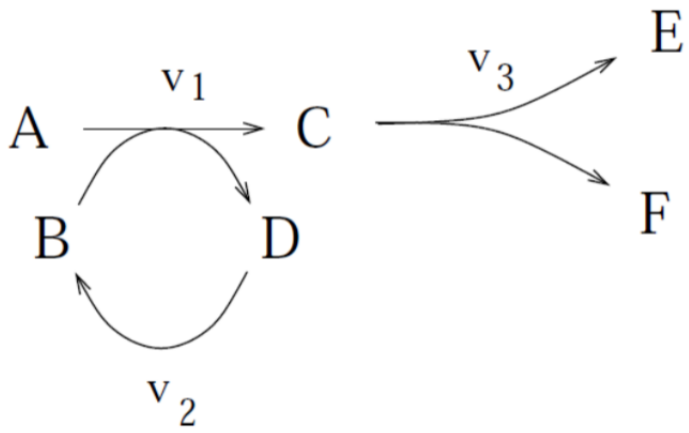


Figure 2.16: Closed reaction network for Problem 2.4.7(a).

a.i) Construct a differential equation model for the network.

Sol:

According the assumption in a)

$$\frac{d[A]}{dt} = -v_1 = -k_1[A][B]$$
$$\frac{d[B]}{dt} = -v_1 + v_2 = -k_1[A][B] + k_2[D]$$
$$\frac{d[C]}{dt} = v_1 - v_3 = k_1[A][B] - k_3[C]$$
$$\frac{d[D]}{dt} = v_1 - v_2 = k_1[A][B] - k_2[D]$$
$$\frac{d[E]}{dt} = \frac{d[F]}{dt} = v_3 = k_3[C]$$

Therefore, $-\frac{d[B]}{dt} = \frac{d[D]}{dt}$; $-\frac{d[C]}{dt} - \frac{d[A]}{dt} = \frac{d[E]}{dt} = \frac{d[F]}{dt}$

We can observe that $\frac{d[D]}{dt}$, $\frac{d[E]}{dt}$, $\frac{d[F]}{dt}$ are the linear combination of set: $[\frac{d[A]}{dt}, \frac{d[B]}{dt}, \frac{d[C]}{dt}]$. Because $-\frac{d[B]}{dt} = \frac{d[D]}{dt}$, $[D] = -[B] + c_0$, where c_0 is a constant. This network includes three equations and algebraic equations, and they are

$$\frac{d[A]}{dt} = -v_1 = -k_1[A][B]$$

$$\frac{d[B]}{dt} = v_1 - v_2 = k_1[A][B] - k_2[D] = k_1[A][B] - k_2(-[B] + c_0)$$

$$\frac{d[C]}{dt} = v_1 - v_3 = k_1[A][B] - k_3[C]$$

a.ii) Solve the steady state concentration by rate variables and initial concentrations.

Sol:

Under the steady state, all the chemicals will reach the dynamic equibrilium. In other words, the derivative of chemicals in respect of time is zero. Besides, I assume all the rate variables aren't equal to zero.

$$-k_1[A]^{ss}[B]^{ss} = 0$$
$$k_1[A]^{ss}[B]^{ss} - k_2(-[B]^{ss} + c_0) = 0$$
$$k_1[A]^{ss}[B]^{ss} - k_3[C]^{ss} = 0$$

According to the equations above,

$$k_1[A]^{ss}[B]^{ss} = 0$$

$$k_3[C]^{ss} = k_1[A]^{ss}[B]^{ss} = 0$$

$$k_2[B]^{ss} - k_2 c_0 = 0$$

where $c_0 = [B]_0 + [D]_0$

The steady state concentraions are:

$$
\begin{aligned}
[A]^{ss} &= 0 \\
[B]^{ss} &= [B]_{t=0} + [D]_{t=0} \\
[C]^{ss} &= 0 \\
[D]^{ss} &= \frac{k_1}{k_2}[A]^{ss}[B]^{ss} = 0 \\
[E]^{ss} &= \int_0^{\infty} [C]_t dt + [E]_{t=0} \\
[F]^{ss} &= \int_0^{\infty} [C]_t dt + [F]_{t=0}
\end{aligned}
$$

a.iii) Verify the Result by Simulation

Here, I used Scipy ODE solver, odeint which I learned from Scipy Documentation (https://docs.scipy.org/doc/scipy/reference/generated/scipy.integrate.odeint.html), and Scipy Cookbook (http://scipy-cookbook.readthedocs.io/items/CoupledSpringMassSystem.html)

The following code defines the system of equations (also known as vector field). Note that the time t must be the second argument of the function.

In [63]:

```
def reaction(agent, t, k):
    """
    Define the differential equations for the network system.

    Arguments:
            agent: vector of the chemical concentrations:
                    agent = [a,b,c,d,e,f]
                k: vector of rate constants:
                    k = [k1, k2, k3]
                t: time
    """
    a, b, c, d, e, f = agent
    k1, k2, k3 = k

    # Create dfdt = [a', b', c', d', e', f']:
    dfdt = [
            -1*k1*a*b,
            -1*k1*a*b+k2*d,
            k1*a*b-k3*c,
            k1*a*b-k2*d,
            k3*c,
            k3*c
           ]

    return dfdt
```

Next, here is a script that uses odeint to solve the equations for a given set of parameter values, initial conditions, and time interval.

In [92]:

```
# Use ODEINT to solve the differential equations defines by the vector field
import numpy as np
from scipy.integrate import odeint

# Initial condition and rate constants
a0, b0, c0, d0, e0, f0 = 1.0, 1.0, 0.5, 0.0, 0.0, 0.0 # Initial condition
k1, k2, k3 = 3.0, 1.0, 4.0 # Rate constants

# Pack up the parameters and initial conditions:
agent0 = [a0, b0, c0, d0, e0, f0]
k = [k1, k2, k3]

# ODE solver parameters
abserr = 1.0e-8
relerr = 1.0e-6
stoptime = 10.0
numpoints = 500

# Create the time samples for the output of the ODE solver.
t = np.linspace(0, stoptime, numpoints)

# Call the ODE solver
wsol = odeint(reaction, agent0, t, args=(k,), atol=abserr, rtol=relerr)
```

In [96]:

```
# Check the shape of wsol (numpoints, chemicals)
wsol.shape
```

Out[96]:

```
(500, 6)
```

Print out the steady-state concentrations of network systmem

In [94]:

```
label = [r'[A]',r'[B]',r'[C]',r'[D]',r'[E]',r'[F]']

# Print out Steady-state concentrations
print('Steady-state concentration')
for i in range(wsol.shape[1]):
    print(label[i],': ',round(wsol[-1,i], 2),'\tmM')
```

```
Steady-state concentration
[A] :  0.0      mM
[B] :  1.0      mM
[C] :  0.0      mM
[D] :  0.0      mM
[E] :  1.5      mM
[F] :  1.5      mM
```

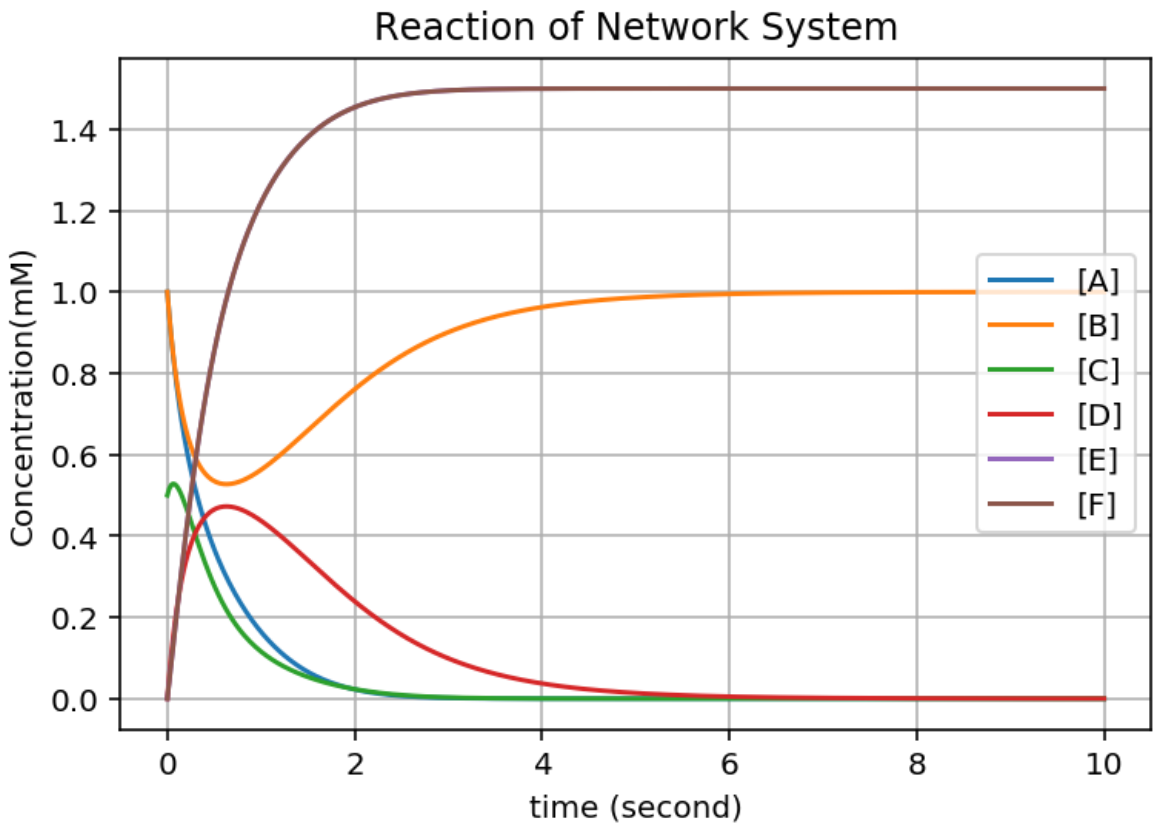
The following script uses Matplotlib to plot the solution generated by the above script.

In [95]:

```
import matplotlib.pyplot as plt

fig_reaction = plt.figure()

for i in range(wsol.shape[1]):
    plt.plot(t,wsol[:,i], label=label[i])

plt.legend()
plt.grid()
plt.title('Reaction of Network System')
plt.xlabel('time (second)')
plt.ylabel('Concentration(mM)')
```

Out[95]:

```
Text(0,0.5,'Concentration(mM)')
```



According to the results, $[A]^{ss} = [C]^{ss} = [D]^{ss} = 0$, $[B]^{ss} = [B]_0 + [D]_0 = 1 + 0$, $[E]^{ss} = [F]^{ss} = 1.5mM$ which mutually support the calculation in a.ii). Besides, the solver even told us the value of $[E]^{ss}$ and $[F]^{ss}$. In conclusion, the simulation support the paper-and-pancil calculation of a.i), and a.ii).

Problem 2. Numerical methid: Fourth-order Runge Kutta Method (RK4)

SAMPLE CODE: https://github.com/SosirisTseng/BEBI-5009/tree/master/ch2 (https://github.com/SosirisTseng/BEBI-5009/tree/master/ch2)

2. Numerical method: **Fourth-Order Runge-Kutta Method (RK4)**

Ref: Chapter 7, Numerical Methods in Engineering with Python 3 by Jaan Kiusalaas (2011)

Euler's method is based on the truncated Taylor series of **y** about *x*:

$$\mathbf{y}(x+h) \approx \mathbf{y}(x) + \mathbf{y}'(x)h \qquad \text{Eq.(1)}$$

Because Eq.(1) predicts **y** at $x + h$ from the information available at *x*, it can be used to move the solution forward in steps of *h*, starting with the given initial values of *x* and **y**. The error in Eq. (1) caused by truncation of the Taylor series is given by

$$\mathrm{E} = \frac{1}{2}\mathrm{y}''(\xi)h^2,$$

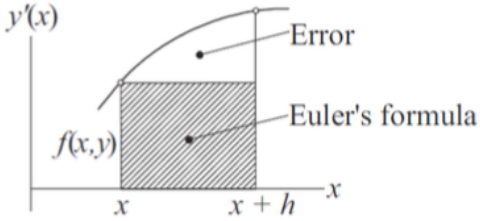


Figure 7.1. Graphical representation of Euler's formula.

Euler's method is seldom used in practice because of its computational inefficiency. Suppressing the truncation error to an acceptable level requires a very small h, resulting in many integration steps accompanied by an increase in the roundoff error. The value of the method lies mainly in its simplicity, which facilitates the discussion of certain important topics, such as stability.

The accuracy of numerical integration can be greatly improved by keeping more terms of the Taylor's series. *Runge-Kutta methods* that are based on truncated Taylor series, but do not require computation of higher derivatives of $\mathbf{y}(x)$. The most popular version is the forth order Runge-Kutta Method which entails the following sequence of operations:

$$K_0 = hF(x, y)$$

$$K_1 = hF\left(x + \frac{h}{2}, y + \frac{K_0}{2}\right)$$

$$K_2 = hF\left(x + \frac{h}{2}, y + \frac{K_1}{2}\right)$$

$$K_3 = hF(x + h, y + K_2)$$

$$y(x + h) = y(x) + \frac{1}{6}(K_0 + 2K_1 + 2K_2 + K_3) \qquad (Eq.2)$$

(1) Use the fourth-order Runge-Kutta method (Eq.2) to integrate the initial value problem as in Figure 2.7 the textbook:

$$\frac{d}{dy}a(t) = -a(t)$$

from $x = 0$ to 4 in increments of $h = 1/3$ and $2/3$. The initial condition is a(0)=1. Plot the computed y together with the analytical solution.

(2) Compare the y computed y by Runge-Kutta method with the y computed by Euler method (Fig2-07 Euler method Matlab or Python codes could be downloaded in Gibhub:

https://github.com/SosirisTseng/BEBI-5009/tree/master/ch2).

You could also try different h values.

Sol:

2.1 Analytical Solution

$\frac{da(t)}{dt} = -a(t)$ is an autonomous equation. It can be solved by "integral-by-part" method:

$$\begin{aligned} \frac{da(t)}{dt} &= -a(a) \\ \frac{-1}{a(t)} da(t) &= dt \\ \int \frac{-1}{a(t)} da(t) &= \int dt \\ -ln(a(t)) &= t + c_0 \\ a(t) &= c_1 e^{-t} \end{aligned}$$

where c_0 and c_1 are constants.

With initial condition $a(0) = 1$:

$$a(t) = e^{-t}$$

2.2 Numerical Solution

In the following code, both RK and Euler method are implemented and wrapped inside an object-class. In the beginning, I define the automous ODE in a Python function.

In [2]:

```
def autoODE(a, t):
    """
    Define the differential equations.

    Arguments:
            a: variable
            t: time
    """
    # Create dfdt = [a']:
    dfdt = [
            -a
           ]

    return dfdt
```

Second, I created an Python class including RK and Euler ODE solver, which provides flexibility for future add-ons.

In [3]:

```
class ode_equation:
    """
    Class object provide RK estimation for ODE equation

    Functions:
        __init__
        iteration
        rk_step
        k
        euler_step
    """
    def __init__(self, equation):
        """
        f(a) = da/dt
        """
        self.equation = equation # Store the ODE function

    def iteration(self, init=1, start=0, end=4, h=1/3.0, method='rk'):
        """
        Using iteration process to simulate the ODE equation

        Arguments:
                init: initial value, a(0)
                start: start time
                end: end time
                h: step size
        """

        # Set time step and to store iteration process
        # h_r is the real step. Approximately equal to h
        self.t, h_r = np.linspace(start, end, int((end-start)/h), retstep=True)
        self.a = []

        # Set initial value
        self.a.append(init)

        # Iteration process
        for i, time in enumerate(self.t[1:]):

            if method == 'rk':
                a_next = self.a[i] + self.rk_step(self.a[i], time, h_r)

            elif method == 'euler':
                a_next = self.a[i] + self.euler_step(self.a[i], time, h_r)

            else:
                raise ValueError('This class only provides rk and euler Method')

            self.a.append(a_next) # Add current step

        return self.a, self.t, h_r



    def rk_step(self, a, t, h):
        """
        Return one step difference by Rung-Kutta Method.

        Arguments:
```

```
                a: a(t) value
                t: current time
                h: step size
        """
        k0, k1, k2, k3 = self.k(a, t, h)
        return (1/6.0)*( k0 + 2*k1 + 2*k2 + k3 )

    def k(self, a, t, h):
        """
        Calculate k of 3 order

        Arguments:
                a: value at a(t)
                t: time
                h: step size
        """

        k0 = h*self.equation(a, t)[0]
        k1 = h*self.equation(a + k0/2.0, t + h/2.0)[0]
        k2 = h*self.equation(a + k1/2.0, t + h/2.0)[0]
        k3 = h*self.equation(a + k2, t + h)[0]

        return [k0, k1, k2, k3]

    def euler_step(self, a, t, h):
        """
        Return one step difference by Euler Method.

        Arguments:
                a: a(t) value
                t: current time
                h: step size
        """
        return h*self.equation(a,t)[0]
```

Then, create a ode solver which has defined above, and input parameters given in the question.

In [4]:

```
import numpy as np

# Set the ODE equation
ode = ode_equation(autoODE)

# Analytical Solution
t_ana = np.linspace(0, 4, int((4-0)/(1/3.0)))
a_ana = np.exp(-t_ana)

# Approximate by RK
a1, t1, h1 = ode.iteration(init=1, start=0, end=4, h=1/3.0, method='rk')
a2, t2, h2 = ode.iteration(init=1, start=0, end=4, h=2/3.0, method='rk')

# Approximate by Euler
a1e, t1e, h1e = ode.iteration(init=1, start=0, end=4, h=1/3.0, method='euler')
a2e, t2e, h2e = ode.iteration(init=1, start=0, end=4, h=2/3.0, method='euler')
```

The following script uses Matplotlib (https://matplotlib.org/tutorials/index.html) to plot the results generated by the above ode solver. Besides, I used TEX inside the plot. (tutorial (https://matplotlib.org/users/usetex.html))

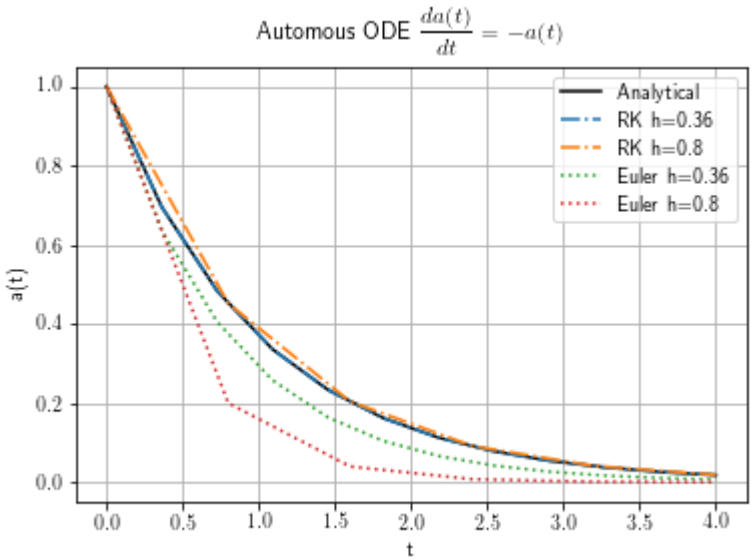
Besides, analytical solution was also plotted together as comparison.

In [16]:

```
import matplotlib.pyplot as plt
%matplotlib inline

# To use latex format
from matplotlib import rc
rc('text', usetex=True)

# Create a figure
plt.figure()
#fig_rk = plt.figure(num=None, figsize=(8, 6), dpi=80, facecolor='w', edgecolor='k',

# Plot Analytical solution
plt.plot(t_ana, a_ana, label='Analytical', linestyle='-', color='black')

# Plot RK approximation
plt.plot(t1, a1, label='RK h='+str(round(h1,2)), linestyle='-.')
plt.plot(t2, a2, label='RK h='+str(round(h2,2)), linestyle='-.')

# Plot Euler approximation
plt.plot(t1e, a1e, label='Euler h='+str(round(h1e,2)), linestyle=':')
plt.plot(t2e, a2e, label='Euler h='+str(round(h2e,2)), linestyle=':')

plt.legend()
plt.grid()
plt.title('Automous ODE 'r'$\displaystyle\frac{da(t)}{dt}=-a(t)$')
plt.xlabel('t')
plt.ylabel('a(t)')
```

Out[16]:

```
Text(0,0.5,'a(t)')
```



2.3 Conclusion

1. According to the results, the RK approximation overestimates the analytical solution
2. RK approximation provides better accuracy compared to the Euler method with same step size.
3. Unlike RK method, Euler approximation underestimates the analytical solution.