Introduction to Chemical Engineering Computing:

Extension to Python

Bruce A. Finlayson

This pdf illustrates how to use the programming language Python to solve the problems posed in the book *Introduction to Chemical Engineering Computing*, Bruce A. Finlayson, Wiley (2006-2014). The material mirrors the use of MATLAB in the book, and solves the examples in Chapters 2, 3, 4, and 8. **In addition, a new Appendix F gives summaries of many Python commands and examples that can be used independently of any chemical engineering applications.** Together with the book, chemical engineering applications are illustrated using Microsoft Excel®, MATLAB®, Aspen Plus®, Comsol Multiphysics®, and Python. This pdf is intended to be used in conjunction with the book, and the treatment using Python mirrors that using MATLAB. Appendix F includes information for setting up Python on your computer. **This pdf is free for personal use and is available at www.chemecomp.com, Supplement Using Python.**

CHAPTER 2. EQUATIONS OF STATE USING PYTHON

SOLVING EQUATIONS OF STATE (SINGLE EQUATION IN ONE UNKNOWN)

Nonlinear algebraic equations can be solved using Python, too. First, you have to define the problem to solve by defining a function; then, you check it; finally, you issue a command to solve it. See Appendix F for additional details.

Step 1 Import the fsolve program from SciPy and define the function (note the indentation after the declaration def).

```
from scipy . optimize import fsolve
def f(x):
  return x ** 2 - 2 * x - 8
```

Step 2 Check the function. Issue the command: print(f(1.1)) to get the result: –8.99. You can easily calculate Eq. (2.8) to see that for $x = 1.1$, the function value is –8.99. Now you know the function f is correct. Note that we used a value for x that meant that every term in the function was important. If we had used x = 1e-5, then the $x*x$ term would be negligible unless we computed it to ten significant figures; hence we wouldn't have checked the entire function. The value x =1.0 is not a good choice either, since an incorrect function x-2*x-8 would return the same value as x*x-2*x-8, hence the error would not be discovered. This is a trivial example, and it is more important for more complicated problems.

Step 3 To find the value of x that makes $f(x) = 0$ in Python, use the 'fsolve' function. In the command window, issue the following command.

```
x = fsolve (f, 0) # one root is at x = -2.0
print (' The root is %5.3f.' % x)
```

This command solves the following problem for x: $f(x) = 0$ starting from an initial guess of 0. The answer is -2.0. You can test the result by saying:

```
print(f(x)))
```

which gives [-2.55795385e-13]. Sometimes the function will have more than one solution, and that can be determined only by using the command with a different initial guess.

To summarize the steps, step 1 defined the problem you wished to solve and evaluated it for some x, step 2 checked your programming, and step 3 instructed Python to solve the problem. It is tempting to skip the second step – checking your programming – but remember: if the programming is wrong, you will solve the wrong problem. The last command gives a further check that the zero of the function has been found.

When examining the command x = fsolve (f, x0), the f defines which problem to solve, the x0 is your best guess of the solution and fsolve tells Python to vary x, starting from x0 until the f is zero.

In all the commands, the f can be replaced by other things, say prob1. The answer can also be put into another variable name: z = x.

```
z = x
print(z)
```

In the last example the result is put into the variable *z*. The options vector allows you to set certain quantities, like the tolerance. Add to the fsolve command: xtol=1.5e-8. For the example used above, you can find the other root by running the program with x0 = 3. Multiple roots can be found only if you search for them starting with different guesses.

Example of a Chemical Engineering Problem Solved Using Python

Find the specific volume of n-butane at 500 °K and 18 atm using the Redlich-Kwong equation of state.

Step 1 First, you need to define the function that will calculate the f(x), here specvol(v), given the temperature, pressure, and thermodynamic properties. The file is shown below.

```
def specvol(v):
    # in K, atm, l/gmol
    # for n-butane
    Tc = 425.2
    pc = 37.5
    T = 393.3
    p = 16.6
    R = 0.08206
    aRK = 0.42748 * (R * Tc) ** 2 / pc
    aRK = aRK * (Tc / T) ** 0.5
    bRK = 0.08664 * (R * Tc / pc)
    return p * v ** 3 - R * T * v ** 2 + (aRK - p * bRK ** 2 - R * T * bRK) * v - aRK * bRK
```

This function, called specvol, defines the problem you wish to solve.

Step 2 To test the function specvol you issue the command:

```
print(specvol(2))
```

and get 25.98, which is correct. The specvol function causes Python to compute the value of the function specvol when v = 2. You should check these results line by line, especially the calculation of aRK, bRK, and y (just copy the code except for the return statement and calculate aRK and bRK with a calculator.. Alternatively, you can use the spreadsheet you developed, put in *v* = 1.506 and see what *f(v)* is; it should be the same as in MATLAB since the cubic function and parameters are the same.

Step 3 Next you issue the command:

```
v = fsolve(specvol,2)
```

```
print(v)
```

and get 1.5064. In specvol the 2 is an initial guess of the answer. To check, you might evaluate the function to find how close to zero f(v) is.

```
print (specvol(v))
```

and get 1.8e-15. Of course you expect this to be zero (or very close to zero) because you expect Python to work properly. If Python can't find a solution, it will tell you. If you use an initial guess of 0.2, you might get the specific volume of the liquid rather than the gas. Python gives 0.18147.

Another Example of a Chemical Engineering Problem Solved Using Python

Next rearrange the Python code to compute the compressibility factor for a number of pressure values. The compressibility factor is defined in Eq. (2.10).

$$Z = \frac{pv}{RT} \tag{2.10}$$

For low pressures, where the gas is ideal, the compressibility factor will be close to 1.0. As the pressure increases, it will change. Thus, the results will indicate the pressure region where the ideal gas is no longer a good assumption. The following code solves for the Redlich-Kwong, Redlich-Kwong-Soave, and Peng-Robinson equations of state and plots the compressibility factor versus pressure as in Figure 2.3.

```
from scipy . optimize import fsolve
import numpy as np
import pylab as plt

# n-butane Redlich-Kwong, Eq. (2.5)
def specvolRK(v, p):
    # in K, atm, l/gmol
    # for n-butane
    Tc = 425.2
    pc = 37.5
    T = 500
    R = 0.08206
    aRK = 0.42748 * (R * Tc) ** 2 / pc
    aRK = aRK * (Tc / T) ** 0.5
    bRK = 0.08664 * (R * Tc / pc)
    return p * v ** 3 - R * T * v ** 2 + (aRK - p * bRK ** 2 - R * T * bRK) * v - aRK * bRK
```

```
# n-butane Redlich-Kwong-Soave, Eq. (2.5)
def specvolRKS(v, p):
    # in K, atm, l/gmol
    # for n-butane
    Tc = 425.2
    pc = 37.5
    T = 500
    R = 0.08206
    acentric = 0.193
    mRKS = 0.480 + (1.574 - 0.176*acentric)*acentric
    alphaRKS = (1 + mRKS *(1-(T/Tc)**0.5)) ** 2
    aRKS = 0.42748 * alphaRKS * (R * Tc) ** 2 / pc
    bRKS = 0.08664 * (R * Tc / pc)
    return p * v ** 3 - R * T * v ** 2 + (aRKS - p * bRKS ** 2 - R * T * bRKS) * v - aRKS * bRKS

# n-butane Peng-Robinson, Eq. (2.6)
def specvolPR(v, p):
    # in K, atm, l/gmol
    # for n-butane
    Tc = 425.2
    pc = 37.5
    T = 500
    R = 0.08206
    acentric = 0.193
    mPR = 0.37363 + (1.54226 - 0.26992*acentric)*acentric
    alphaPR = (1 + mPR *(1-(T/Tc)**0.5)) ** 2
    aPR = 0.45724 * alphaPR * (R * Tc) ** 2 / pc
    bPR = 0.07780 * (R * Tc / pc)
    return p*v**3+(bPR*p - R*T)*v**2+(aPR- *p*bPR**2- *R*T*bPR)*v +
           (p*bPR**3 + R*T*bPR**2-aPR*bPR)

T = 500
R = 0.08206
pressure = np.arange(1, 27, 5)
print(pressure)
print(pressure[0])
print(pressure[5])
zcompRK = np.zeros(6,dtype=float)
zcompRKS = np.zeros(6,dtype=float)
zcompPR = np.zeros(6,dtype=float)
print(zcompRK)

for i in range(0, 6, 1):
    p = pressure[i]
    guess = R*T/p
    v = fsolve(specvolRK, guess, p)
    z = p * v / (R * T)
    zcompRK[i] = z
    v = fsolve(specvolRKS,v,p)
    z = p * v / (R * T)
    zcompRKS[i] = z
    v = fsolve(specvolPR,v,p)
    z = p * v / (R * T)
    zcompPR[i] = z

print(zcompRK)
print(zcompRKS)
print(zcompPR)
plt . plot (pressure,zcompRK,'o-g',label='Redlich-Kwong')
plt . plot (pressure,zcompRKS,'x-b',label='Redlich-Kwong-Soave')
plt . plot (pressure,zcompPR,'s-r',label='Peng-Robinson')
plt . legend(loc='best)')
plt . xlabel('Pressure (atm)')
plt . ylabel('Z')
plt . title ('n-Butane')
plt . show()
```

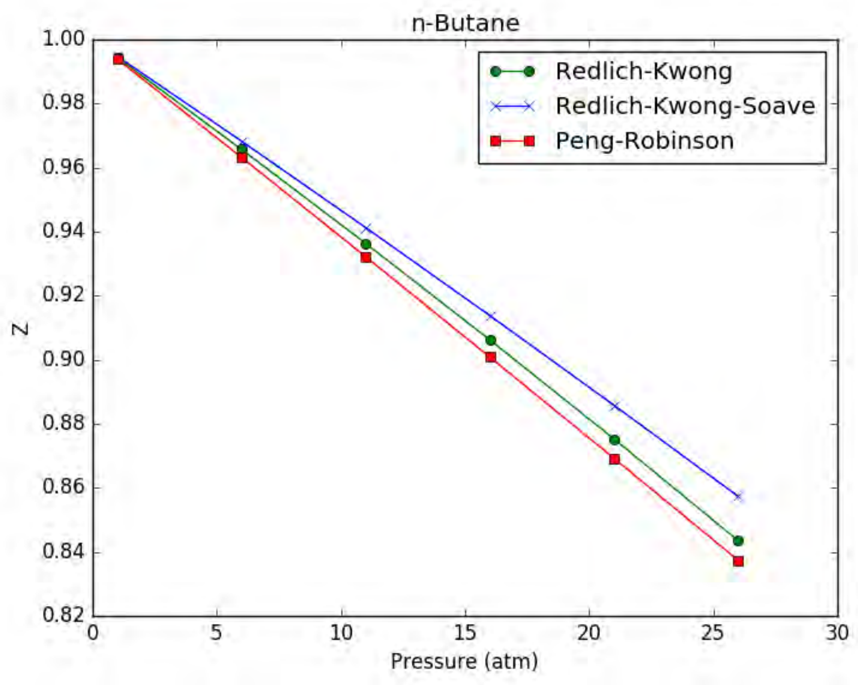


Figure 2.3. Compressibility factor for n-butane, using Python

The first three commands bring in the needed routines – fsolve, numpy, and pylab (for plotting. Then there are three definitions of functions that define the equation governing the specific volume, Eq. (2.5) and (2.6). The main program sets the temperature, provides a vector of 6 pressures, equidistant from 1 to 27; pressure = [1, 6, 11, 16, 21, 26]. The index goes from 0 to 5. The vectors of compressibilites are also set with 6 values for each equation of state, starting with 0, to be filled as the calculations proceed. Then a loop calculation is made for *i* from 0 to 5.

For each pressure in turn, the guess for the Redlich-Kwong equation of state is the result from the ideal gas law. The result from the Redlich-Kwong equation of state is used for the guess when solving the Redlich-Kwong-Soave equation of state, and that solution is used as the gues for the Peng-Robinson equation of state. For each *i*, after the compressibility is found it is put in a vector for that equation of state. Finally the results are plotted, with three curves on one plot.

CHAPTER 3. VAPOR-LIQUID EQUILIBRIA USING PYTHON

Example using Python

Before working this example, explore Appendix F so that you have all the program parts needed here.

We want to solve Eq. (3.9) using Python, given the K_i and z_i.

$$\sum_{i=1}^{NCOMP} \frac{(K_i - 1)z_i}{1 + (K_i - 1)v'} = 0 \tag{3.9}$$

This is a nonlinear equation to solve for v'. Thus, you can apply here the same methods used with Python in Chapter 2. Once the value of v' is known, you can calculate the value of the liquid compositions, $\{x_i\}$, and vapor compositions, $\{y_i\}$, using Eq. (3.8) and (3.1). The mole balance is then complete.

$$x_i = \frac{z_i}{1 + (K_i - 1)v'} \tag{3.8}$$

$$y_i = K_i x_i \tag{3.1}$$

The program to do this is shown here.

```
# vapor-liquid equilibrium
# this is necessary to get fsolve, which solves the non-linear equations
from scipy . optimize import fsolve

# function definition, Eq. (3.9)
def vpequil(v):
   z = [0.0, 0.1, 0.3, 0.4, 0.2]   #note the first value in the matrix is not used
   K = [0.0, 6.8, 2.2, 0.8, 0.052]
   # print(z): z and K can be printed when checking the program
   # print(K)
   sum1 = 0.
   for i in range(1,5):            # note: in python the range stops one before the last value
                                    # see the discussionin Appendix F on vectors
      num = (K[i] - 1.0) * z[i]
      denom = 1.0 + (K[i] - 1.0) * v
      sum1 = sum1 + num / denom
      #print(num)  #these can be printed when checking the function
      #print(denom)
      #print(sum1)
   return sum1
```

```
# check the function
print ('For testing %10.6f ' % vpequil(0.2))


# find the solution
v = fsolve (vpequil, 0.2)
print (v)

# find the composition
z = [0.0, 0.1, 0.3, 0.4, 0.2]
K = [0.0, 6.8, 2.2, 0.8, 0.052]
x = np.zeros(5,dtype=float)      # these are necessary to introduce vectors
y = np.zeros(5,dtype=float)
for i in range(1,5):         # remember that the indices go from 0 to 4
                     # we skip the first one, and the loop goes to one below 5, or 4
   x[i] = z[i]/(1.0 + (K[i]-1.0)*v)
   y[i] = K[i]*x[i]

print ('The liquid mole fractions are')
print (x)
print ('\nThe vapor mole fractions are')
print (y)
```

The output is

```
For testing   0.241549
The vapor fraction is   0.425838
The liquid mole fractions are
[ 0.         0.0288196   0.19854325  0.43723857  0.33539858]

The vapor mole fractions are
[ 0.         0.19597326  0.43679516  0.34979086  0.01744073]
```

These agree with the results from Excel and MATLAB.

CHAPTER 4. CHEMICAL REACTION EQUILIBRIA USING PYTHON

Solution of Tables 4.2 and 4.3 Using Python

To solve Eq. (4.15) using Python, you define a function that will calculate the right-hand side and use fsolve to find the value of x that makes it zero.

$$f(x) = 148.4 - \frac{x^2}{(1-x)^2} \tag{4.15}$$

The program that does this is shown here and explained step by step below.

```
from scipy . optimize import fsolve
def equil_eq(x):
    return 148.4 - x*x/(1.0-x)**2
x = fsolve (equil_eq, 0.5)
print ('The root x is %10.5f' % x)
```

Step 1 Import fsolve from scipy.optimize.

```
from scipy . optimize import fsolve
```

Step 2 Construct a function evaluates the function, given *x*. The name is equil_eq and it is listed below.

```
def equil_eq(x):
    return 148.4 - x*x/(1.0-x)**2
```

Step 3 Call fsolve to find the value of *x* that makes the function equil_eq zero, using 0.5 as the initial guess.

```
x = fsolve (equil_eq, 0.5)
```

Step 4 Print the result.

```
print ('The root x is %10.5f' % x)
```

The result is:

```
The root x is    0.92414
```

The program is easily changed to allow different inlet mole fractions, rather than pure carbon monoxide and water, *i.e.* Table 4.3. The data is entered in the function as a vector param that is set outside the function.

```
def equil_eq(x,param):
    COin = param[1]
    H2Oin = param[2]
    CO2in = param[3]
    H2in = param[4]
    Kequil = 148.4
    CO = COin - x
    H2O = H2Oin - x
    CO2 = CO2in + x
    H2 = H2in + x
    return Kequil-CO2*H2/(CO*H2O)

param = np.zeros(5,dtype=float)
param[1] = 1.
param[2] = 1.8
param[3] = 0.3
param[4] = 0.1
x = fsolve (equil_eq, 0.9, param)

print ('The root x is %10.5f' % x)
```

The solution is the same as that found in Table 4.3.

```
The conversion x is    0.98836
```

Some times the solution cannot be found and you must try again with a different initial guess. That was case here, and fsolve did not converge for an initial guess of 0.5.

Multiple Equations, Few Unknowns Using Python

Suppose you want to solve the following two equations:

$$\begin{aligned} 10x + 3y^2 &= 3 \\ x^2 - \exp(y) &= 2 \end{aligned} \tag{4.16}$$

These can be solved using the fsolve program. Since the exponential is used it must be imported from scipy, too.

```
from scipy . optimize import fsolve
from scipy import exp
import numpy as np

def prob2(p):
    # vector components are transferred to the function
    x = p[0]
    y = p[1]
    return 10*x + 3*y*y -3, x*x - exp(y) - 2
```

```
p = np.zeros(2,dtype=float)
p[0] = 1.5
p[1] = 2.5
p = fsolve(prob2,p)

print ('x =  %8.5f' % p[0])
print ('y =  %8.5f' % p[1])
```

The result is

```
x = -1.44555
y = -2.41216
```

The vector p could also be defined by the following command.

```
p = np.append(1.5, 2.5)
```

The function can also be calculated with the final solution to verify that it is correct.

```
x = p[0]
y = p[1]
z = 10*x + 3*y*y -3
print('The function is %10.5e ' % z)
z = x*x - exp(y) - 2
print('                %10.5e ' % z)
```

The result is

```
The function is -3.37508e-14
                 4.31655e-13
```

which is close to zero and indicates the solution is good.

CHAPTER 8. CHEMICAL REACTORS USING PYTHON

USING PYTHON TO SOLVE ORDINARY DIFFERENTIAL EQUATIONS

Simple Example

In Python, you define the problem by means of a function following the def command. You then tell Python to solve the differential equation. This process is illustrated using a single, simple differential equation:

$$\frac{dy}{dt} = -10y,\ \ y(0) = 1 \tag{8.16}$$

Integrate this equation from t = 0 to t = 1. The exact solution can be found by quadrature and is

$$y(t) = e^{-10t} \tag{8.17}$$

The program to do this is:

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

def f(y, t):
# this is the rhs of the ODE to integrate , i.e. dy/dt=f(y,t)
   return -10 * y

y0 = 1 # initial value
a = 0 # integration limits for t
b = 1
tspan = np. arange (a, b, 0.05) # values of t for which we require the solution y(t)

y = odeint (f, y0 , tspan) # actual computation of y(t)

# plot the solution
plt . plot (tspan, y)
plt . xlabel ('t'); plt . ylabel ('y(t)')
plt . show ()

# print the solution
print(tspan, y)
```

The import commands are necessary to get the programs that will be used. Then the function is defined, f(y,t), which is the right-hand side of the differential equation. The time span is set and values of the solution are to be obtained at intervals of 0.05. The odeint is called to solve the differential equation, and it is then plotted, giving the same graph Figure 8.1.

Passing Parameters

Still another way to introduce k into the function is to use it as a parameter in the calling argument. Modify the function and the calling arguments as follows. The red parts are the lines that are changed.

```
def rhs(y,t,krate):
# this is the rhs of the ODE to integrate , i.e. dy/dt=f(y,t)
   return - krate * y

y0 = 1 # initial value
a = 0 # integration limits for t
b = 1
tspan = np. arange (a, b, 0.05) # values of t for which we require the solution y(t)
krate = 10.0 # set the rate constant
y = odeint (rhs, y0 , tspan, args=(krate,)) # actual computation of y(t)
# note how the krate has to be put into an args tuple
```

The solution is the same as before.

Example: Isothermal Plug Flow Reactor

The equations for all three species in the plug flow reactor are (p. 144)

$$u\frac{dC_A}{dz} = -2kC_A^2, \quad u\frac{dC_B}{dz} = +kC_A^2, \quad u\frac{dC_C}{dz} = 0 \tag{8.21}$$

At the inlet

$$C_A(0) = 2 \text{ kmol/m}^3, C_B(0) = 0, C_C(0) = 2 \text{ kmol/m}^3 \tag{8.22}$$

and we take $u = 0.5$ m/s, $k = 0.3$ m^3/kmol s, and the total reactor length as $z = 2.4$ m.

Step 1 The Python program requires a function that defines the right-hand side. The input parameters to the function are the concentrations of all species. The function also needs the velocity, u, and the rate constant, k. The distance from the inlet, z, takes the place of time and is the independent variable. The code for the function follows. Note the fact that three values are returned, the right-hand sides of the three derivatives.

```
def ydot(y, t):
   # y(0) is CA, y(1) is CB, y(2) is CC
   # k = 0.3 and u = 0.5
   CA= y[0]
   rate = 0.3*CA*CA
   return (-2.*rate/0.5, +rate/0.5, 0.)
```

Step 2 You test this function by calling it with specific values for y to ensure that it is correct.

```
y0 =  np.zeros(3,dtype=float)
y0 [0] = 0.2
y0 [1] = 0.3
y0 [2] = 0.4
a = 0.
VR = 2.6
tspan = np. arange (a, VR, 0.2)
print(ydot(y0,tspan))
```

gives the same answer as with MATLAB (page 145): -0.048, 0.024, 0. This is a very important step, because this is where you add value. Python will integrate whatever equations you give it, right or wrong, and only *you* can ensure that the program has solved the right equations.

Step 3 Next, write the code that serves as the driver. This code must **(a)** set any constants (here they are just put into the function rate1 for simplicity), **(b)** set the initial conditions and total reactor length, and **(c)** call the odeint solver.

```
y0 = np.zeros(3,dtype=float)
y0[0] = 2.0
y0[1] = 0.0
y0[2] = 2.0
a = 0.
VR = 2.6
tspan = np. arange (a, VR, 0.2)
y = odeint(ydot,y0,tspan)
```

Step 4 The solution is then printed and plotted. The plot is the same as Figure 8.3.

```
# print the solution
print(tspan, y)
print(y[:,0])

# plot the solution
plt . plot(tspan, y[:,0],'*-')
plt . plot(tspan, y[:,1],'+-')
plt . plot(tspan, y[:,2],'x-')
plt . xlabel ('length (m)')
plt . ylabel ('concentrations (kgmol/m^3)')
plt .show()
```

Example: Nonisothermal Plug Flow Reactor

Consider the model of a simple reactor oxidizing SO_2 to form SO_3 treated on pp. 146-9. The equations are

$$\frac{dX}{dz} = -50R', \ \frac{dT}{dz} = -4.1(T - T_{surr}) + 1.02 \ 10^4 R' \tag{8.24}$$

where the reaction rate is

$$R' = \frac{X[1 - 0.167(1 - X)]^{1/2} - 2.2(1 - X)/K_{eq}}{[k_1 + k_2(1 - X)]^2} \tag{8.25}$$

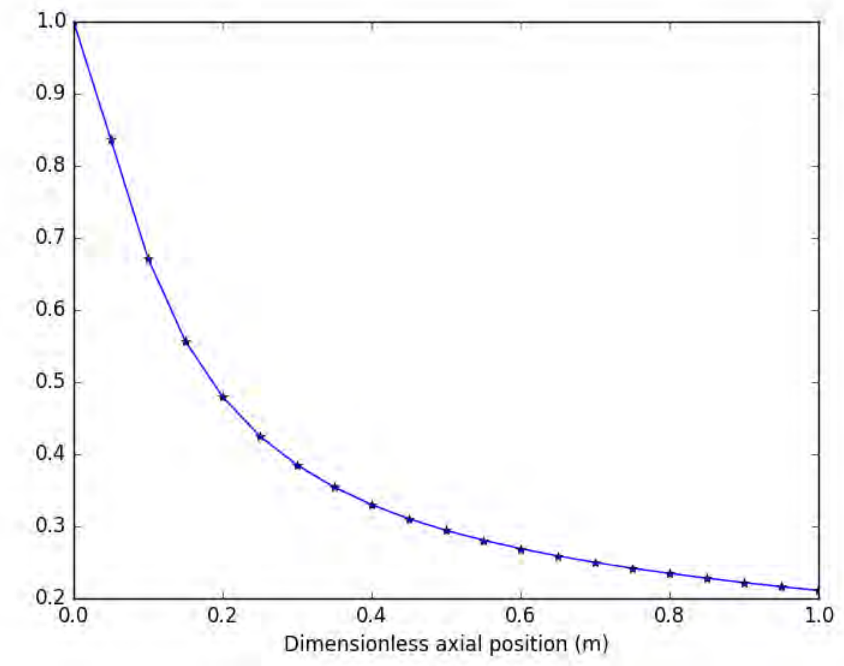
$$\ln k_1 = -14.96 + 11070/T, \ln k_2 = -1.331 + 2331/T, \ \ln K_{eq} = -11.02 + 11570/T \tag{8.26}$$

with the parameters: $T_{surr} = 673.2, T(0) = 673.2, X(0) = 1$. The variable X is the concentration of SO_2 divided by the inlet concentration, $1–X$ is the fractional conversion, and T is the temperature in K. The first equation is the mole balance on SO_2, and the second is the energy balance. The first term on the right-hand side of Eq. (8.24) represents cooling at the wall; the second term there is the heat of reaction. The Python program to solve these equations follows. Note that both the exp and sqrt functions need to be imported.

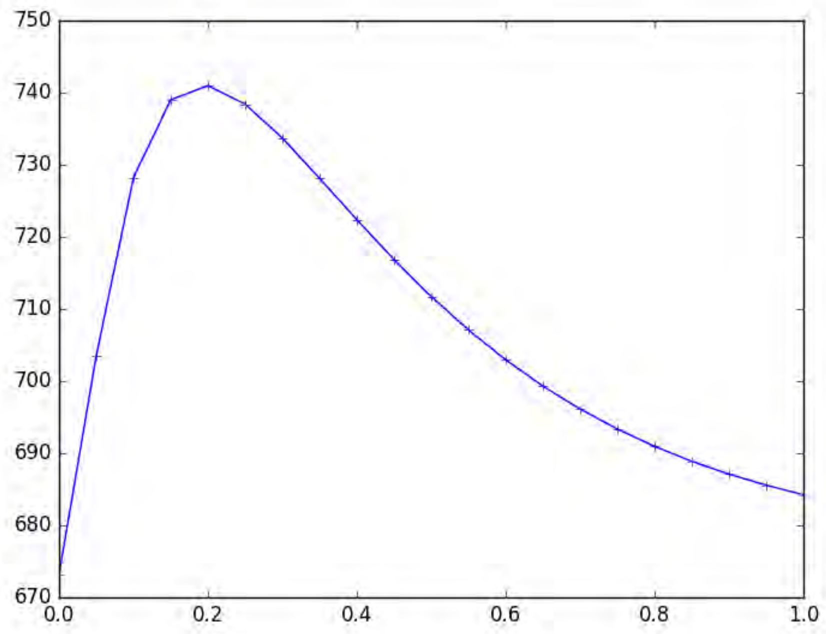
```
from scipy.integrate import odeint
from scipy import exp, sqrt
import matplotlib.pyplot as plt
import numpy as np

# define the functions or right-hand sides
def ydot(y, t):
    # y(0) is X, y(1) is T
    X = y[0]
    T = y[1]
    k1 = exp(-14.96 + 11070 / T)
    k2 = exp(-1.331 + 2331 / T)
    Keq = exp(-11.02 + 11570 / T)
    term1 = X * sqrt(1 - 0.167 * (1 - X))
    term2 = 2.2 * (1 - X) / Keq
    denom = (k1 + k2 * (1 - X)) ** 2
    rate = (term1 - term2) / denom
    return (-50 * rate , -4.1 * (T - 673.2) + 1.02e4 * rate)

# set the initial conditions
y0 = np.zeros(2,dtype=float)
y0[0] = 1.0
y0[1] = 673.2
```

```
# set the integration limits and call odeint
a = 0.
L = 1.05
tspan = np. arange (a, L, 0.05)
y = odeint(ydot,y0,tspan)

# print the solution
print(tspan, y)
print(y[:,0])

# plot the solution in two plots
plt . plot(tspan, y[:,0],'*-')
plt . xlabel ('Dimensionless axial position (m)')
plt . show()
plt . plot(tspan, y[:,1],'+-')
plt . show()
```

The solution is the same as shown on page 149, Figure 8.3.



(a)



(b)

Figure 8.3. (a) Fraction converted; (b) temperature, using Python

CHEMICAL REACTORS WITH MASS TRANSFER LIMITATIONS

When there are mass transfer limitations the reaction rate is calculated based on the concentration on the solid catalyst, not the concentration in the fluid. As shown on p. 155, *et. seq*. it is necessary to solver for the mass transfer as well as integrate the equation. The equations are:

$$u\frac{dC_A}{dz} = -2k_s C_{A,s}^2 \ , \quad u\frac{dC_B}{dz} = +k_s C_{A,s}^2 \ , \quad u\frac{dC_C}{dz} = 0 \qquad (8.39)$$

The mass transfer equation is:

$$k_m a(C_A - C_{A,s}) \ = \ k_s C_{A,s}^2 \qquad (8.40)$$

Looking closely at Eq. (8.39)-(8.40) you can see that in order to solve the differential equations in Eq. (8.39) you must solve Eq. (8.40) *at every position z*. Thus, this is a problem that combines ordinary differential equations with nonlinear algebraic equations.

Python easily handles these kind of problems. We have to define a function to solve Eq. (8.40) and call that function inside the function that defines the ordinary differential equations (8.39). The function for Eq. (8.40) must appear in the code before the one for Eq. (8.39). Basically you call a routine to integrate the ordinary differential equations (e.g., odeint). You construct a right-hand side function (here called ydot) to evaluate the right-hand side. The input variables are z and the three concentrations, and the output variables are the three derivatives. Take C_A and solve Eq. (8.40) using the function mass_rxn to find $C_{A,s}$ at this location, z. Then evaluate the rates of reaction in Eq. (8.39). The program follows.

```
def mass_rxn(CAs,CA): # define the mass balance equation
   k = 0.3
   km = 0.2
   return km * (CA - CAs) - k * CAs * CAs

def ydot(y, t): # define the ordinary differential equations
   CA= y[0]
   CAguess = CA
   CAs = fsolve(mass_rxn,CAguess,CA)
   rate = 0.3*CAs*CAs
   return (-2.*rate/0.5, +rate/0.5, 0.)

# set the initial conditions
y0 = np.zeros(3,dtype=float)
y0[0] = 2.0
y0[1] = 0.0
y0[2] = 2.0

# set the integration parameters
a = 0.
VR = 2.6
tspan = np. arange (a, VR, 0.2)

# integrate the ordinary differential equations
y = odeint(ydot,y0,tspan)

print(tspan, y) # print the solution
print(y[:,0])

# plot the solution
plt . plot(tspan, y[:,0],'*-',label='$A$')
plt . plot(tspan, y[:,1],'+-',label='$B$')
plt . plot(tspan, y[:,2],'x-',label='$C$')
plt . legend()
plt . xlabel ('Length (m)')
plt . ylabel ('Concentrations (kgmol/m^3)')
plt .show()
```

The solution is shown in Figure 8.10, which is the same as found with MATLAB. With mass transfer resistance included, the outlet concentration of B is 0.61. When there was no mass transfer limitation, the outlet concentration of B was 0.85. Thus, the reactor is not able to produce as much product, and a bigger reactor is required.

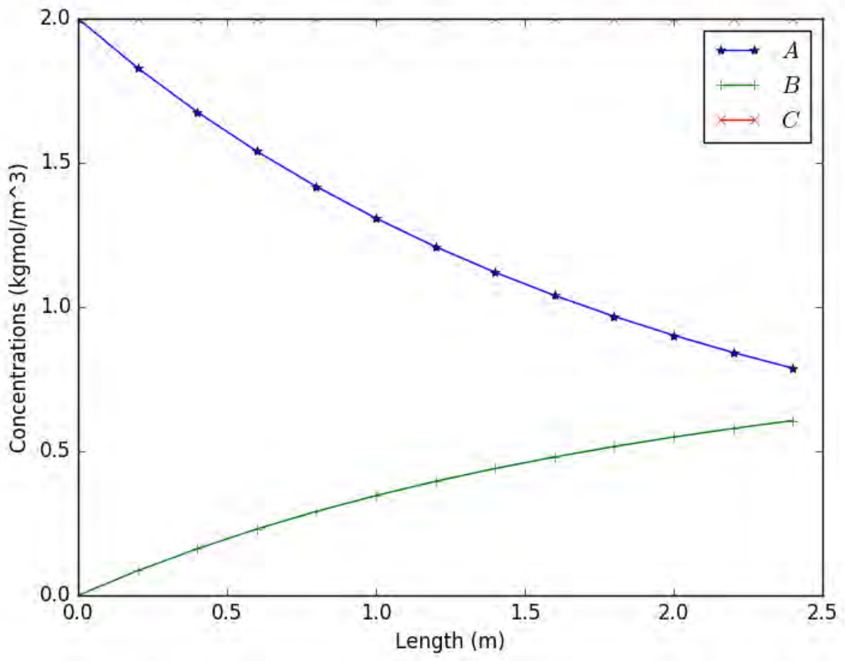


Figure 8.10. Solution to Eq. (8.39)-(8.40) when mass transfer is important, using Python

CONTINUOUS STIRRED TANK REACTORS

Eq. (8.15) gave the mass balance for a continuous stirred tank reactor (CSTR). A similar equation can be written as an energy balance. This example considers a CSTR in which a first-order reaction occurs, but the temperature also changes due to the heat of reaction. The equations to be solved are:

$$\frac{Q}{V_R}(1-c) = c \ \exp[\gamma(1-1/T)]$$
$$\frac{Q}{V_R}(1-T) = -\beta \ c \ \exp[\gamma(1-1/T)] \tag{8.53}$$

The left-hand sides are the flow rate times a concentration or temperature difference between the input and output, divided by the volume. The equations have been normalized by the inlet concentration and temperature. The right-hand sides are the rate of reaction and the rate of energy generation due to reaction, respectively.

The case described by Eq. (8.53) is for an adiabatic reactor. When the reactor is adiabatic, the equations can be combined by multiplying the first equation by β and adding it to the second equation; then the right-hand side vanishes.

$$\beta(1-c)+(1-T)=0 \tag{8.54}$$

This equation can be solved for T.

$$T = 1 + \beta(1 - c) \tag{8.55}$$

Now the mass balance can be considered a single equation in one unknown (c).

$$\frac{Q}{V_R}(1 - c) = c \ \ \exp[\gamma(1 - 1/\{1 + \beta - \beta c\})] \tag{8.56}$$

Solution using Python

The program to solve this equation just requires using fsolve to find the roots to a single equation.

```
from scipy optimize import fsolve
import numpy as np

# define the function
def rate_T(c,param):
   beta = param[1]
   gamma = param[2]
   flowvol = param[3]
   T = 1 + beta*(1.0-c)
   rate = c * exp(gamma * (1.0 - 1.0/T))
   return flowvol * (1.0 - c) - rate

# set the parameters
param = np.zeros(4,dtype=float)
param[1] = 0.15
param[2] = 30.0
param[3] = 8.7
print(param)
# print(rate_T(0.5,param)) #used for testing

# solve the problem
c = fsolve(rate_T,0.5,param)
print ('The root c is %10.4f' % c)
```

The answer is the same as before, 0.7311.

CSTR with multiple solutions

For a different set of parameters, the CSTR can have more than one solution. For this problem, the solutions all lie between 0 and 1, because the concentration has been normalized by the inlet value, where the normalized concentration is 1.0, and the reaction uses up the material. Which solution you get depends upon the initial guess of c. Use Python to solve the problem when $Q/V_R = 25$, $\beta = 0.25$, keeping $\gamma = 30$. Successive trials led to the results shown in Table 8.1, which are the same solutions obtained using Excel.

Initial guess of c	Final result
0	0.0863
0.5	0.5577
1.0	0.9422

Table 8.1. Multiple solutions to Eq. (8.56) when $Q/V_R = 25$, $\beta = 0.25$, $\gamma = 30$, using Python

Solutions to multiple equations using MATLAB.

When two or more variables must be found, as in Eq. (8.53), a solution can be found to make both the equations zero without rearrangement like that to produce Eq. (8.56). The main requirement is that the return statement has two parts to it (separated by commas).

```
def rate_T(y,param):
   beta = param[1]
   gamma = param[2]
   flowvol = param[3]
   c = y[0]
   T = y[1]
   rate = c * exp(gamma * (1.0 - 1.0/T))
   return (flowvol * (1.0 - c) - rate, flowvol * (1.0 - T) + beta * rate)

param = np.zeros(4,dtype=float)
param[1] = 0.15
param[2] = 30.0
param[3] = 8.7
y = fsolve(rate_T,[0.5, 1.1],param)
print ('The concentration is %10.4f ' % y[0])
print ('The temperature is %10.4f ' % y[1])
```

The answer is the same as before, 0.7311 and 1.0403.

TRANSIENT CONTINUOUS STIRRED TANK REACTORS

Reactors don't always run at steady state. In fact, many pharmaceuticals are made in a batch mode. Such problems are easily solved using the same techniques presented above because the plug flow reactor equations are identical to the batch reactor equations. Even CSTRs can be run in a transient mode, and it may be necessary to model a time-dependent CSTR to study the stability of steady solutions. When there is more than one solution, one or more of them will be unstable. Thus, this section considers a time-dependent CSTR as described by Eq. (8.57).

$$V\frac{dc'}{dt'} = Q(c'-c'_{in}) - Vk_0c'\exp(-E/RT')$$
$$[\phi(\rho C_p)_f + (1-\phi)(\rho C_p)_s]V\frac{dT'}{dt'} = -(\rho C_p)_f Q(T'-T'_{in}) + (-\Delta H_{rxn})Vk_0c'\exp(-E/RT') \tag{8.57}$$

The variables are

V = reactor volume, Q = volumetric flow rate
c'= concentration, T'= temperature, t'= time
k_0 = reaction rate constant, E = activiation energy
ϕ = void fraction, ρ = density, C_p = heat capacity per mass (8.58)
subscript f for fluid, s for solid
$-\Delta H_{rxn}$ = heat of reaction, energy per mole

The non-dimensional form of these equations is

$$\frac{dc}{dt} = (1-c) - c \bullet Da \bullet \exp[\gamma(1-1/T)]$$
$$Le\frac{dT}{dt} = (1-T) + \beta \bullet c \bullet Da \bullet \exp[\gamma(1-1/T)] \tag{8.59}$$

The parameters are defined as

$$c = \frac{c'}{c'_{in}}, T = \frac{T'}{T'_{in}}, t = \frac{Qt'}{V}, Da = \frac{V}{Q}k_0\exp(-\frac{E}{RT'_{in}})$$
$$Le = \frac{\phi(\rho C_p)_f + (1-\phi)(\rho C_p)_s}{(\rho C_p)_f}, \beta = \frac{(-\Delta H_{rxn})c'_{in}}{(\rho C_p)_f T'_{in}} \tag{8.60}$$

The parameter *Le* is a Lewis number, and it includes the heat capacity of the system. The *Da* is a Damköhler number and includes the rate of reaction. The parameters are taken as

$$\beta = 0.15,\ \gamma = 30,\ Da = 0.115,\ Le = 1080,\ c(0) = 0.7,\ T(0) = 1 \tag{8.61}$$

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

def ydot(y, tspan, param):
    c = y[0]
    T = y[1]
    Damk = param[0]
    Tin = param[1]
    beta = param[2]
    Lewis = param[3]
    gamma = param[4]
    rate = c*Damk*exp(gamma * (1.0 - 1.0/T))
    return (1 - c - rate, (Tin - T - beta*rate)/ Lewis)

# set the initial conditions
y0 = np.zeros(2,dtype=float)
y0[0] = 0.7
y0[1] = 1.0

# set the integration conditions
tstart = 0.
tend = 2.1
tspan = np. arange (tstart, tend, 0.1)

# set the parameters
# Damk, Tin, beta, Lewis, gamma
param = (0.115, 1.0, 0.15, 1080.0, 30.0)
print(param)

# integrate the equations
y = odeint(ydot,y0,tspan,args=(param,))

# print the solution
print(tspan, y)

# plot the solution
plt . plot(tspan, y[:,0],'*-',label='$Concentration$')
plt . xlabel ('Time')
plt . ylim(0.65, 1.05)
plt . plot(tspan, y[:,1],'+-',label='$Temperature$')
plt . legend(loc=(0.1,0.5))
plt . xlabel ('Time')
plt . show()
```

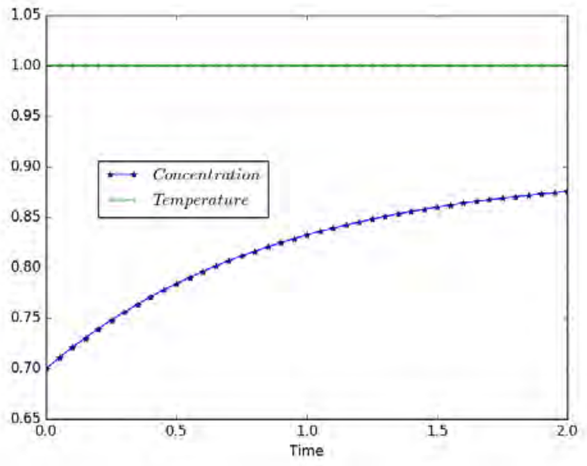


Figure 8.15. Transient CSTR, up to *t* = 2, using Python

The steps in the program are the same as in other applications: define the function, set the initial conditions, integration limits, parameters, integrate the differential equations, print the results, and plot the graph. The only difference is that parameters are passed to the function, and these need to be defined as a tuple, the tspan and param.

The result is shown in Figure 8.15. It looks like steady state is achieved by the time that t = 2. This is not true, however. Integrate to t = 1000 and look at the results in Figure 8.16. It still has not reached steady state. The reason is that the temperature responds much more slowly than does the concentration. Thus, the concentration comes to a steady state value appropriate to the current temperature, but then the temperature keeps changing and the concentration must change to keep up. Notice the very rapid change of c from the initial value of 0.7 to about 0.89 in Figure 8.16. This is because the value of c = 0.7 was not appropriate for a temperature of 1.0. In mathematical terms, the time response of the two variables is very different (the eigenvalues of the equation are widely separated), and the system is called stiff. See Appendix E for more discussion about stiff equations.

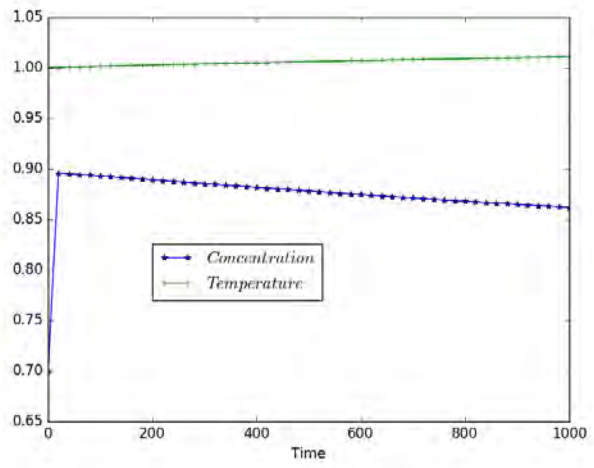


Figure 8.16. Transient CSTR, up to t = 1000, using Python

Next we integrate to t = 40,000, as shown in Figure 8.17 with the following changes in the program.

```
tstart = 0.
tend = 41250.0
tspan = np. arange (tstart, tend, 1250.0)
```

This looks funny, but it is because the first data point for concentration (after the initial condition) misses some of the detail shown in Figure 8.16. One way to improve the figure is to just leave out the 1250.0 in tspan. Then the odeint will plot whatever points it has solved for. Another solution is to integrate to 1000 and use that as the initial guess for another calculation to 40,000. The changes in the program are as follows.

```
# set the integration conditions
tstart = 0.
tend = 1020.0
tspan = np. arange (tstart, tend, 20.0)

# integrate to t = 1000.
y = odeint(ydot,y0,tspan,args=(param,))

# set the integration conditions again
tstart = 1000.0
tend = 40000.0
tspan = np. arange (tstart, tend, 1250.0)

# set the initial conditions to the ending conditions of the previous calculation
y0[0] = y[50,0]
y0[1] = y[50,1]

# continue the integration
y2 = odeint(ydot,y0,tspan,args=(param,))

# plot the extension
plt . plot(tspan, y2[:,0],'*-')
plt . plot(tspan, y2[:,1],'+-')
plt .show()
```

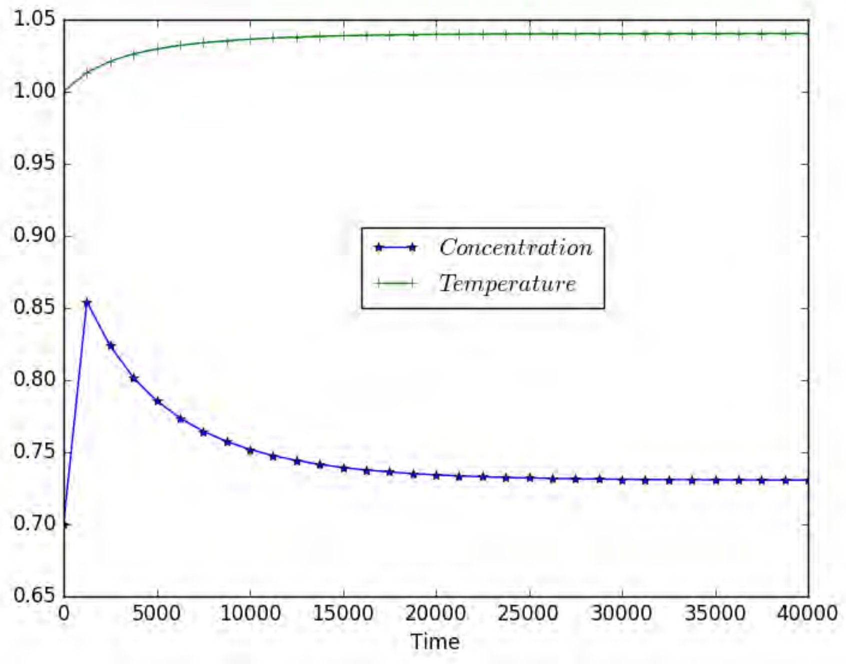


Figure 8.17. Transient CSTR, up to *t* = 40.000, using Python

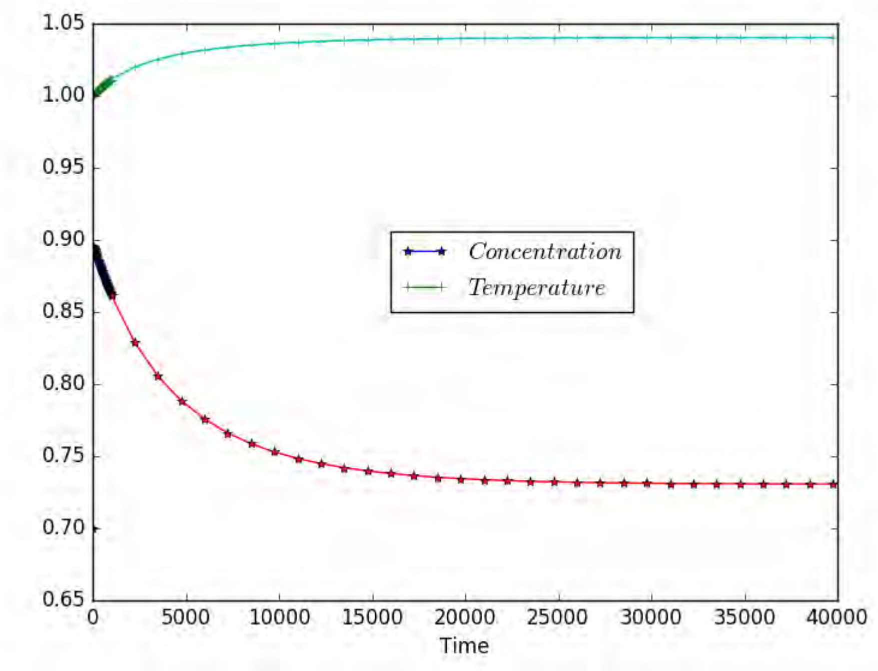


Figure 8.17_revised. Transient CSTR, up to *t* = 40,000, using Python

Next change the parameter *Le* from 1080 to 0.1 and integrate to *t* = 100. The changes to the code are:

```
param = (0.115, 1.0, 0.15, 0.1, 30.0)
```

and

```
tend= 100.025
tspan = np. 27rrange (tsta, tend, 0.025)
```

Figure 8.18 shows the limit cycle as the concentration and temperature never reach steady state.

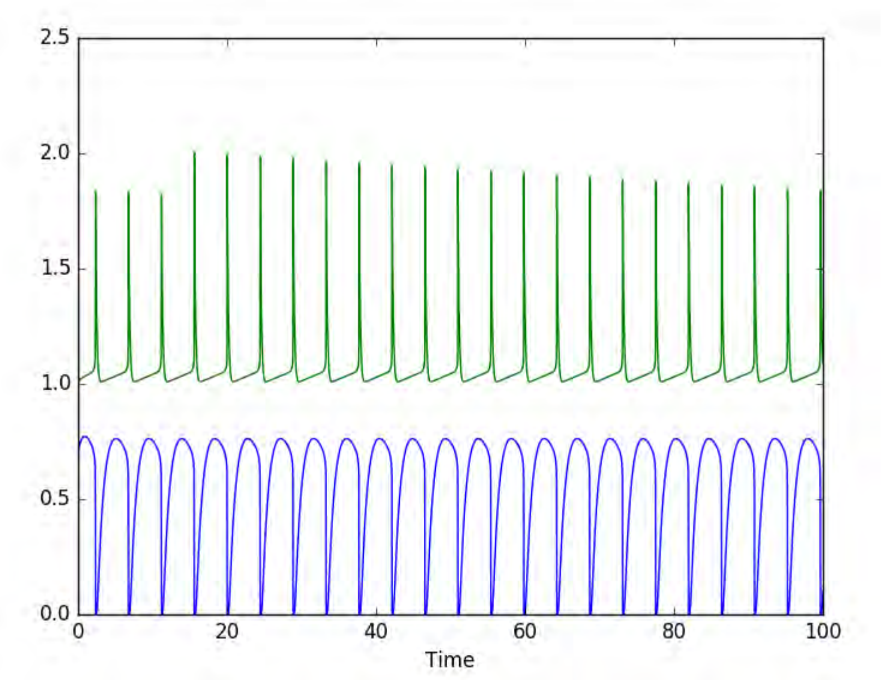


Figure 8.18. Transient CSTR, *Le*= 0.1, up to *t* = 100, using Python

Figure 8.19 shows the temperature plotted versus the concentration, and the limit cycle is clear.

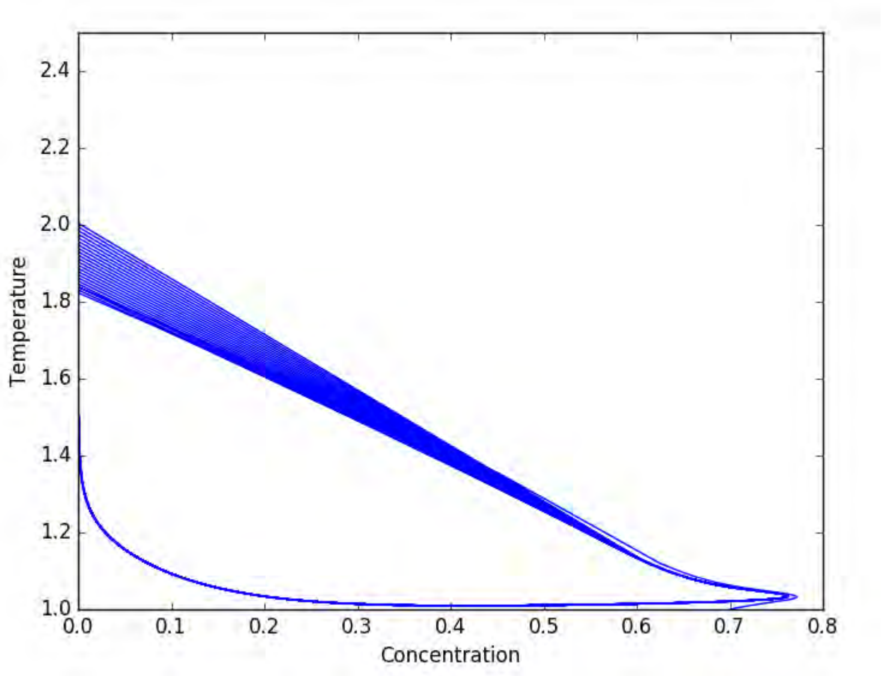


Figure 8.19. Limit cycle display of Figure 8.18, using Python

APPENDIX F. HINTS WHEN USING PYTHON

This appendix provides hints and tips when using Python[1]. Python is a programming language first developed by Guido van Rossum in 1991, but it has been a good programming language for scientific work only since about 2008. Since then mutually incompatible add-on packages have been consolidated into a few compatible packages (see below).[2] Python assumes that you are a beginner in using Python, but not an absolute beginner in computer programming. Most likely, you remember concepts from a computer programming class taken earlier. Included in Appendix F are general features that are useful in all the applications solved with Python. Other features are illustrated in the context of specific examples; a list of examples is provided at the end of the appendix for handy reference. You'll probably want to skim this appendix first, then start working some of the problems that use Python to gather experience, and finally come back and review this appendix in more detail. That way you won't be burdened with details that don't make sense to you before you see where and how you need them.

Outline of Appendix F:

1. General features: loading Python, screen format, and creating a program, classes of data
2. Programming options: input/output, functions, loops, conditional statements, timing, matrices, matrix multiplication
3. Finding and fixing errors
4. Solving linear equations and finding eigenvalues of a matrix
5. Evaluate of an integral
6. Interpolation: splines and polynomials: spline interpolation, polynomial interpolation, polynomials of degree n, fit a function to data and plot the result, fit a polynomial to data
7. Solve algebraic equations
8. Integrate ordinary differential equations that are initial value problems: differential-algebraic equations, checklist for using odeint
9. Plotting: simple plots, multiple plots, bold, italics, and subscripts, Greek letters, contour plots, 3D plots
10. Python help
11. Applications of Python

F.1. GENERAL FEATURES

Loading Python, Screen Format, and Creating a Program

The web site https://www.python.org/ has options for downloading Python. The author has chosen to use Pycharm, which is available at https://www.jetbrains.com/pycharm/; click download. There is a Community version that is free, and that is adequate here. Documentation

[1] Python is a programming language that is user-supported and available from the Python Foundation: https://www.phthon.org The examples here use PyCharm ver. 3.6.
[2] *Python for Scientists*, John M. Stewart, Cambridge University Press, Cambridge, England, 2014.

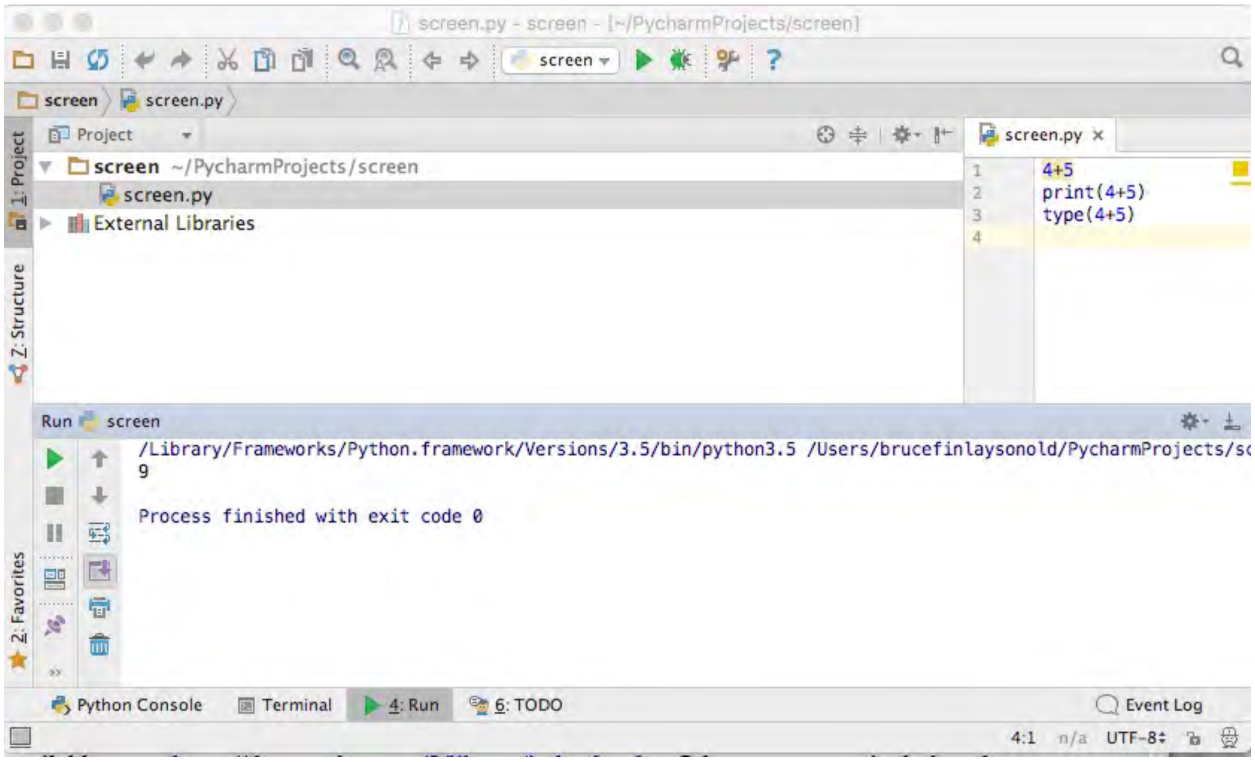
is available at https://docs.python.org/3/library/index.html. Other sources include the *Introduction to Python for Computational Science and Engineering*, by Hans Fangohr of the University of Southhampton, available as a pdf from https://www.southampton.ac.uk/~fangohr/training/python/pdfs/Python-for-Computational-Science-and-Engineering.pdf and the books *Chemical and Biomedical Engineering Calculations Using Python*, by Jeffrey J. Heys, Wiley, 2017 and *Python For Scientists*, by John M. Stewart, Cambridge University Press, Cambridge, England, 2012.

The following websites may be useful.
http://docs.python.org/2/tutorial
http://matplotlib.org/api/pyplot_summary.html
http://matplotlib.org/gallery.html
http://www.scipy.org/NumPy_for_Matlab_Users

The following pdf documents are also useful, obtained from https://docs.scipy.org/doc/.
scipy-ref-0.18.1.pdf
numpy-user-1.11.0.pdf

The Python program includes many features, but not everything needed for scientific calculation. Other modules can be added. The ones used here are math, numpy, scipy, and matplotlib. The last three must be added to the same area where the Python program is stored. On a Macintosh, open Terminal, back up one directory (cd ..) twice, then choose the directories in turn: cd usr, cd local, cd bin. Then say python3.5 -m pip install numpy, etc. This need be done only once.

Then when you open Python you will be asked if you want to open an existing project or create a new one. For a new one, provide a name. Then in the upper left choose the project name, then File/New… in the pull-down menu. Several options are there; choose File and give it a name ending with .py, such as screen.py. The upper right area is the editor where you type the program screem.py, and the lower left area is where the output appears (unless you are writing to a file). The screen will look like Figure F.1. Along the bottom there are several options. Python Consol is where you can type in a Python command and immediately have it executed. That is useful for testing commands and their syntax. Terminal is the area mentioned below for importing other programs and can be accessed from this Python window. Run is where the output appears when you run a program.



```
4+5
print(4+5)
type(4+5)
```

```
/Library/Frameworks/Python.framework/Versions/3.5/bin/python3.5 /Users/brucefinlaysonold/PycharmProjects/sc
9

Process finished with exit code 0
```

Figure F.1. Python Screen

However, when you type a command in the Python editor it will not execute until you say Run (in the Run pull-down menu or the green triangle). If you just want to try some commands and get the answer directly, there are several options. You can choose Python Console and enter the command there. Or on the Macintosh open terminal independently of Python and say python. The fact that you see >>> indicates that Python is waiting for input. The symbol >>> indicates a computer command is expected and the result is printed immediately after you press return.

```
>>> 4+5
9
>>> type(4+5)
<class 'int'>
```

In the terminal window you can say: import math, then dir(math) and see the functions that are included. They include trigonometric functions, inverse trigonometric functions, hyperbolic trigonometric functions, exponentials, error functions, as well as others. If you want to see what modules you can import, go to the terminal window and say help('modules'). A long list is provided which includes math, numpy, scipy[3], and matplotlib. To stop using the terminal as an interpreter or consol say exit().

[3] Some of the scipy programs require a FORTRAN compiler, but none of the examples here need that. Legacy FORTRAN programs can be used; see Chapter 8 in *Python For Scientists*.

Once you are ready to create a program, type it in the editor window. Comment lines in your code begin with a #. If you wish to have several lines as a paragraph of comments, then use """ paragraph goes here """. In the examples below, the symbol >>> indicates a computer command and the result is printed here immediately after. In actuality, you would put all the computer commands together in the file and get all the output in the run window at the lower left unless you were executing commands line by line in terminal.

The usual arithmetic symbols +, –, x, and / are used. Exponentiation is done using **, as in x**2.

In print commands with two or more variables (separated by commas) the comma itself provides a space between the variables. The \n indicates a carriage return.

There are also specialized programs submitted by others, but these require a compiler for another language and are not treated here. You can see them at http://pypi.python.org/pypi.

Classes of Data

Variables can be integers, floating point numbers, strings, and Boolean variables. Setting a = 4.0 provides a floating point number but setting b = 4 provides an integer.

```
>>>import numpy as np
>>>a = 4.0
>>>print(a)
4.0
>>>print(type(a))
<class 'float'>
b = 4
>>>print(b)
4
>>>print(type(b))
<class 'int'>
```

The floating point numbers involve 16 digits of precision, which is called double precision in C and MATLAB. You can see all the digits in this way. Messages can be combined with the numerical output for clarity.

```
b = 2.3456789
>>> print('long format %20.16e' % b)
long format 2.3456788999999998e+00

>>> print('long format %20.16f' % b)
long format   2.3456788999999998
```

Most of your work will involve numbers, but sometimes you will want to use text or words. Strings are text.

```
>>> print("Chemical Engineering rules.")
```

```
Chemical Engineering rules.
>>>print(type(c))
<class 'str'>
```

Information read in by a file is read as a string and must be converted to float or int to use in calculations.

A number can be converted to a character using the str command and back using the float command.

```
>>>d = str(b)
>>>print(type(d))
<class 'str'>
>>>e = float(d)
>>>print(type(e)
<class 'float'>
```

Boolean variables are discussed below in conjunction with conditional statements.

Python is case sensitive so that t(i) and T(i) are different.

F.2. PROGRAMMING OPTIONS: INPUT/OUTPUT, FUNCTIONS, LOOPS, CONDITIONAL STATEMENTS, TIMING, MATRICES

Input/Output

You can ask the user for input with the following command.

```
>>> viscosity = input('What is the viscosity (Pa s)?')
```

When you see 'What is the viscosity (Pa s)?' in the run-time window, type the value and press return. Keep in mind that what comes in is treated as a string. Thus, if you typed 3.456 and asked print(type(viscosity)) you would get <class 'str'>. You would have to use the float(viscosity) command to convert it to another variable name as a floating point number. You can test this with print(viscosity+viscosity), and you will get 3.4563.456. If you convert viscosity to a float: viscosityf = float(viscosity) and printed viscosityf+viscosityf you would get 6.912. You can display in a specified format:

```
>>> a =  1.25456
>>>b = 5.4521
>>>print ('\nThe values of a and b are %7.4f %5.3e' % (a, b))
```

gives

```
The values of a and b are  1.2546 5.452e+00
```

These are C-commands, and the notation means:

f – floating point
\n is a carriage return
e – exponential format
5.3 means five characters, three after the decimal point.

To write a text file, you must open it, write it, and close it. The following commands write a text file, closes the file, opens it and reads it.

```
# Open the file, write to it, and close it
>>>out_file = open ("test.txt", "w")
>>>out_file . write ("Writing text to file. This is the first line.\n"+\
"And the second line.")
>>>out_file . close ()

# Open the file, read it, and close it
>>>in_file = open ("test.txt", "r")
>>>text = in_file . read ()# read into a string variable 'text'
>>>in_file . close ()

# Display the text
>>>print (text)
Writing text to file. This is the first line.
And the second line.
```

Numbers written to a file are written as text and must be converted to float or int when read in. The first step is to put them into an array, then save the array as a text file. It is read as a text file and is converted to a numbers.

```
>>>import numpy as np

>>> a = [2.345, 4.567, 6.789]
>>>np.savetxt('x.txt', a)
>>>xc=np.loadtxt('x.txt')
>>>print(xc)
[ 2.345  4.567  6.789]

>>>print (type(xc))
<class 'numpy.ndarray'>

>>>print (xc[1]*xc[2])
31.005363
```

Functions

A function is defined with the def command. The commands within the function must be indented, and the last line returns the results. For example

```
>>>def f(x):
>>>    return x ** 3 - 2 * x ** 2
```

will take the value of x and compute the number, returning it. The program following this definition will not be indented – that is how the function definition is ended. The command

```
>>>print(f(3))
```

gives the value 9. There may be more than one argument, such as f(x,y,z), and numbers may be assigned to them.

```
>>>def f(x=3,y=4,z)
```

In this case x will take the value 3, unless it is specified otherwise, and z must be specified before the function is called. The function definition can be anywhere in the program as long as it is defined before it is called. Examples with many input variables are given on pages 11, 21, 22, 44.

Loops

It is sometimes useful to execute a command over and over, putting each result in one element of a vector. One option is to compute something when an index goes from a starting value to an ending value.

```
>>>print ('loop using range')
>>>i = 0
# range(start, end, increment)
# does not include the last number, end
>>>for i in range(0,10,1):
>>>    print(i)
>>>    i = i + 1
```

The output is:

```
loop using range
0
1
2
...
8
9
```

Note that the loop *does not* include the ending value. The increment is optional. Another option is to use the while command.

```
>>>print('\nloop using while')
```

```
>>>i=0
# while(condition)
>>>while (i < 5):
>>>    print(i)
>>>    i=i+1
```

The output is:

```
loop using while
0
1
2
3
4
```

If you want the loop not to do the remaining calculations if a condition is true, the program will continue the loop but not execute the rest of the commands, going back to the start of the loop.

```
>>>for i in range(...):
>>>     <block1> # commands to be executed every time
>>>         if <condition> #don't execute the commands <block2>, but continue the loop
>>>     continue
>>><block2>
```

If you want to stop before the end depending on a condition:

```
>>>for i in range(...):
>>>    <block1> # commands to be executed every time
>>># don't execute the commands <block2>, and get out of the loop
>>>    if <condition>
>>>         break
>>>    <block2>
```

Conditional Statements

A Boolean number is either True or False. The following command produced 3.

```
>>>c = True
>>>if (c):
>>>     print(3)
>>>else:
>>>    print(2)
3
```

The following commands produced 2.

```
>>>c = False
>>>if not c:
```

```
>>>    print(2)
>>>else:
>>>    print(3)
2
```

Note the indentation of four spaces after the if/else commands. This is done automatically by using a tab but is essential for telling where the section ends. Also note the : at the end of the conditional statement.

Sometimes the program needs to execute different instructions depending upon a calculated result. The following program prints negative if *i* is negative, 5 if it is between 0 and 5 (not including 5) and 6 otherwise.

```
>>>if (i<0):
>>>    print("negative")
>>>elif (i<5):
>>>    print ("5")
>>>else:
>>>    print ("6")
```

The other command are:

==	which means the left-hand side is equal to the right-hand side,
not (var) == (condition)	which means they are not equal,
(var) != (condition)	which means they are not equal,
>=	which means greater than or equal,
<=	which means less than or equal.

One can also test two conditions with the <condition1> and <condition2> command. They both have to be true for the expression following to be used. If one uses or then the expression following is used if either condition1 or condition2 is true. The command not switches from true to false.

Timing information

To measure the time a calculation takes, import time. Then say starttime = time.time() to begin the clock. At a later point in the program say endtime = time.time(). Then the following commands give you the time between those two calls.

```
>>>elapsedTime = endtime - starttime
>>>print('total time is %10.3f seconds' % elapsedTime
total time is      2.211 seconds
```

Matrices

Vectors are row (1x3)

```
>>>x = [2, 4, 6]
```

establishes a vector with three elements, numbered 0, 1, and 2; x(0) = 2,…x(2) = 6.

The matrix A

$$A = \begin{bmatrix} 2 & 6 & 6 \\ 8 & 10 & 12 \\ 14 & 10 & 18 \end{bmatrix}$$

is set by

```
>>>A = ([2, 6, 6],[8, 10, 12], [14, 10, 18])
```

In Python these are called arrays, and each element of an array must be the same type. Python has the capability of having lists, in which the elements need not be the same; some can be numbers, some text, even pictures.

Two arrays can be added, multiplied, or divided, element by element.

```
>>>import numpy as np

>>>a = np.array([0,1,5])
>>>print(a)
[0 1 5]
>>>c = np.array([1,3,5])
>>>print(c)
[1 3 5]
>>>print(a+c)
[ 1  4 10]
>>>print(a*c)
[ 0  3 25]
>>>print(a/c)
[ 0.          0.33333333  1.        ]
```

You can operate on one element of the array by using indices. In this example c[1] += 1 means add 1 to c[1]. It could also be written c[1] = c[1] + 1.

```
>>>c[1] += 1
>>>print(c)
[1 4 5]
```

Matrix Multiplication

Take the matrix $A = \begin{bmatrix} 2 & 6 & 6 \\ 8 & 10 & 12 \\ 14 & 10 & 18 \end{bmatrix}$ and the vector $x' = \begin{bmatrix} 2 \\ 3 \\ 4 \end{bmatrix}$.

and compute the dot product.

$$A = \begin{bmatrix} 2 & 6 & 6 \\ 8 & 10 & 12 \\ 14 & 10 & 18 \end{bmatrix} \begin{bmatrix} 2 \\ 3 \\ 4 \end{bmatrix} = \begin{bmatrix} 46 \\ 94 \\ 130 \end{bmatrix} \qquad \text{(F.1)}$$

```
import numpy as np

>>>AA=np.matrix([[2, 6, 6], [8, 10, 12], [14, 10, 18]])
>>>print(AA)
[[ 2  6  6]
 [ 8 10 12]
[14 10 18]]

>>>Bb=np.array([[2], [3], [4]])
>>>print(Bb)
[[2]
 [3]
 [4]]

>>>ans = np.dot(AA,Bb)
>>>print(ans)
[[ 46]
 [ 94]
[130]]
```

is [3x3][3x1] = [3x1] matrix. Note the brackets in Bb; Bb is a column vector, which is needed for matrix multiplication. Note the difference between Bb and setting

```
>>>b=np.array([1, 3, 5])
>>>print(b)
[1 3 5]
```

F.3. FINDING AND FIXING ERRORS

One source of confusion is when you think you have defined a variable, but haven't really. Use print(x), Run to see the current value of x in the output window. The programs illustrated in this book are fairly simple, and they are easy to debug. All programmers make mistakes; good programmers learn to find them! Finding them is easy if you take the time. First consider a line in a code. Set all the parameters used in the line and execute the line in the console. Compare the answer with a calculation done with a calculator or one you did by hand. Be sure that all the parameters you use are different, and don't use zero or one, because some mistakes won't be picked up if the variable is zero or one. The parameters you use for testing can be single digit numbers that are easy to calculate in your head, and they can be completely unrelated to the problem you are solving. If you haven't specified one of the parameters the output will identify the missing parameter.

F.4. SOLVING LINEAR EQUATIONS AND FINDING EIGENVALUES OF A MATRIX

To solve the equation

$A * x' = Bb$, where A and Bb are

$$A = \begin{bmatrix} 2 & 6 & 6 \\ 8 & 10 & 12 \\ 14 & 10 & 18 \end{bmatrix}, \quad Bb = \begin{bmatrix} 1 \\ 3 \\ 5 \end{bmatrix}$$

we say

```
>>>x=LA.solve(AA,Bb)
>>>print (x)
```

and get

```
[[ -2.50000000e-01]
[ -9.71445147e-17]
[  4.16666667e-01]].
```

To get the inverse of A, say

```
>>>CC = LA.inv(A)
>>>print(CC)
[[-0.625      0.5       -0.125     ]
[-0.25       0.5       -0.25      ]
[ 0.625     -0.66666667  0.29166667]]
```

Multiplying A by CC gives a diagonal matrix.

```
>>>ans = np.dot(A,CC)
>>print(ans)

[[  1.00000000e+00   0.00000000e+00  -8.32667268e-17]
[  8.88178420e-16   1.00000000e+00   1.11022302e-16]
 [  1.77635684e-15   0.00000000e+00   1.00000000e+00]]
```

Eigenvalues are the polynomial roots to Eq. (F.2).

$$\left| A_{ij} - \lambda \delta_{ij} \right| = 0 \tag{F.2}$$

They are easy to calculate in Python using the eig command.

```
>>>import numpy
>>>import numpy . linalg as LA

>>>A = ([2, 4, 6],[8, 10, 12], [14, 16, 18])
>>>evalues , evectors = LA.eig (A)
>>>print ( evalues )
>>>print ( evectors )
```

gives the eigenvalues and eigenvectors.

```
[  3.22336879e+01  -2.23368794e+00  -2.60735545e-15]
[[-0.23197069 -0.78583024  0.40824829]
[-0.52532209 -0.08675134 -0.81649658]
[-0.8186735   0.61232756  0.40824829]]
```

In this example, the matrix is nearly singular, and one of the eigenvalues is very small.

F.5. EVALUATE AN INTEGRAL

To evaluate the integral, Eq. F.3, import the function *quad* from scipy.integrate.

$$area = \int_0^2 x^2 dx \tag{F.3}$$

```
from scipy.integrate import quad
# function we want to integrate
def f(x):
      return x * x

res, err = quad(f, 0, 2) # call quad to integrate f from 0 to 2
# print("The numerical result is {:f} (+-{g})".format(res,err))
print("The numerical result is {:f} ".format(res))
```

The output is

```
The numerical result is 2.666667
```

F.6. INTERPOLATION: SPLINES AND POLYNOMIALS

Spline Interpolation

Make a cubic B-spline pass through all the data points: [$x(i)$, $y(i)$, $i = 1,...,m$]. A cubic spline is a cubic function of position, defined on small regions between data points. It is constructed so the function and its first and second derivatives are continuous from one region to another. It usually makes a nice smooth curve through the points. The following commands create Figure F.2.

```
from scipy.interpolate import CubicSpline              # import the functions
from math import cos
import matplotlib.pyplot as plt
import numpy as np

x = np.arange (0, 11, 1)      # use points from 0 to 10 with an interval of 1
y = np.cos(x)                 # calculate the cos of each x in the vector x
print (x)
print (y)

#Cubic Spline
f = CubicSpline(x, y)         # compute the cubic spline
xnew = np.arange(0, 9, 0.1)# create a list of points closer together
ynew = f(xnew)                # calculate the value at xnew using the interpolation
                              # function computed by Cubic Spline
plt.axis([0, 10, -1.5, 1])    # set the x- and y- axis
plt.plot(x, y, 'o', xnew, ynew, '-') # plot
plt.show()
```

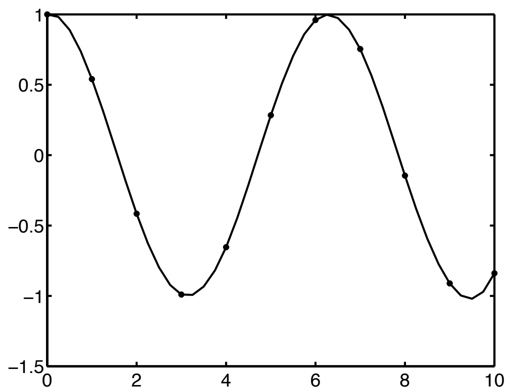


Figure F.2. Spline fit of a set of data points

Polynomial Interpolation

To use a piecewise polynomial through the points, not a Cubic Spline, simply replace the Cubic Spline code with the following.

```
# Linear (default)
y = x * x
f = interpolate.interp1d (x, y)
xnew = np.arange(0, 9, 0.1)
ynew = f(xnew)
# use interpolation function returned by `interp1d`
plt.plot(x, y, '+', xnew, ynew, '-')
plt.show()
```

To use a quadratic polynomial, use the command

```
f = interpolate.interp1d (x, y, 'quadratic')
```

Polynomials of degree *n*

A polynomial of degree *n* can be created using poly1d.

$$y = p_1 x^n + p_2 x^{n-1} + ... + p_n x + p_{n+1} \qquad \text{(B.8)}$$

```
p = poly1d([3,4,5])
```

In this case the polynomial is $3x^2 + 4x + 5$ and that is printed out. The coefficients can be obtained using

```
print (p.coeffs).
```

The polynomial can be differentiated

```
print(p.deriv())
```

to give $6x + 4$, and multiplied

```
print(p*p)
```

to give $9x^4 + 24x^3 + 46x^2 + 40x + 25$. To evaluate a polynomial use print(p(2)) to give 25.

Fit a Function to Data and Plot the Result

For data points $x(i), yi(i), i = 1,...,n$ fit a function that minimizes the least squares error.

```
import numpy as np
from scipy . optimize import curve_fit
import pylab as plt
```

```
def f(x, a, b, c):
    # Fit function y=f(x,p) with parameters p=(a,b,c).
     return a * np. exp (- b * x) + c

# create fake data with n = 50 points
x = np. linspace (0, 4, 50)
y = f(x, a=2.5 , b=1.3 , c =0.5)
# add noise
yi = y + 0.2 * np. random . normal ( size =len (x))
# call curve fit function to get the best values of a, b, and c
popt , pcov = curve_fit (f, x, yi)
a, b, c = popt
print (" Optimal parameters are a=%g, b=%g, and c=%g" % (a, b, c))

# plotting
yfitted = f(x, * popt ) # equivalent to f(x, popt [0] , popt [1] , popt [2])
plt . plot (x, yi , 'o', label ='data $y_i$ ')
plt . plot (x, yfitted , '-', label ='fit $f(x_i)$')
plt . xlabel ('x')
plt . legend ()
plt . show ()
```

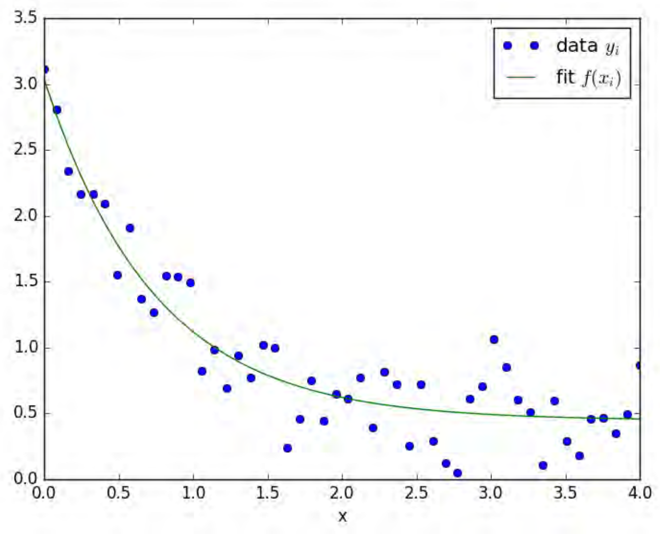


Figure F.3. Least Squares fit of data

Fit a polynomial to Data

To also get a polynomial giving the best fit to the same data, add the following code:

```
# fit with polynomial of order 3
pp = np.polyfit(x, yi, 3)
```

```
# print polynomial coefficient
print(pp)

# evaluate the polynomial at all x points
yfittedp = np.polyval(pp,x)

# output
print(x)
print(yfittedp)
plt . plot (x, yi, 'o', label ='data $y_i$ ')
plt . plot (x, yfittedp , '-', label ='fit $p(x_i)$')
plt . xlabel ('x')
plt . legend ()
plt . show ()
```

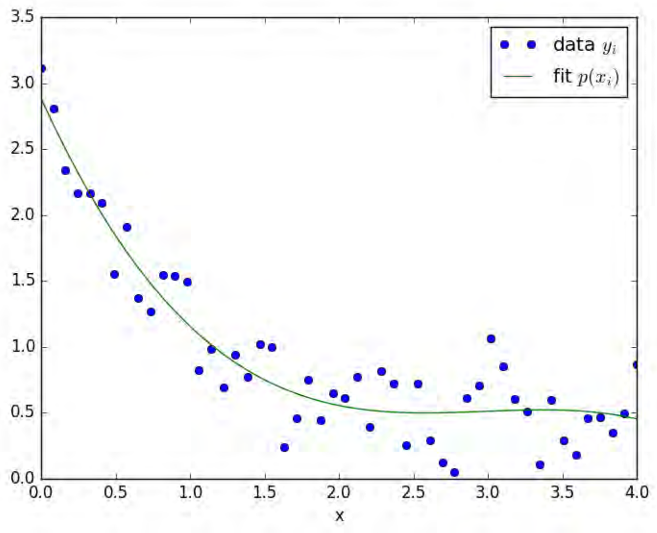


Figure F.4. Polynomial Least Squares Fit of Data

F.7. SOLVE ALGEBRAIC EQUATIONS

Solve $f_i(\{y_j\}) = 0, i = 1,...,n$ for the vector $\{y_j\}$. There are three steps.

Step 1 Import 'fsolve' from the scipy, the SCIentific Python package. Scipy must be imported to your computer to add it to Pycharm.

```
from scipy . optimize import fsolve
```

Step2 Define the function.

```
def prob2 (y):
    u1, u2 = y
return (u1 ** 3 - 2 * u1 **2, u2 ** 2 - 2 * u2 + 1)
```

Step 3 Call 'fsolve' with an initial guess and print the results.

```
u1, u2 = fsolve (prob2, (3 , 1.5))
print ('\nNow solving two equations with fsolve\n')
print (u1, u2)
```

The result is

```
Now solving two equations with fsolve

2.0 1.00000001516
```

You need to make several checks. The number of unknowns is set by the number of elements in the initial guess, (in parentheses in the fsolve command). The defined function 'prob2' calculates $f_i, i = 1,...,n,$ given $\{y_j\}$, and there have to be as many elements in *f* as there are in *y*. The function needs to be checked, of course. The only way you can make Python find the solution to your problem is to make sure the function gives the correct set of *f*'s when given a set of *y*'s. If this 'fsolve' does not work well, try making an initial value problem and solving it using implicit methods, integrating to a long time.

$$\frac{dy_i}{dt} = f_i(\{y_j\}), \quad y_i(0) = \text{initial guess}$$

Alternatively, try a different initial guess.

There are other options for fsolve, too. These can be found in the document scipy-ref-0.18.pdf; search on scipy.optimize.fsolve.

F.8. INTEGRATE ORDINARY DIFFERENTIAL EQUATIONS THAT ARE INITIAL VALUE PROBLEMS

To solve a single ordinary differential equation

$$\frac{dy}{dt} = f(t,y), \quad y(0) = y_0$$

we use the 'odeint' functions in Python. There are four steps.

Step 1 Import odeint, and matplotlib.pyplot and numpy

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

Step 2 Define the function.

```
def f(y, t):
# this is the rhs of the ODE to integrate , i.e. dy/dt=f(y,t)
    return -2 * y * t
```

Step 3 Set the initial condition, y0 and time span for the integration as well as the values of *t* for which we want the solution from 0 to 10 at intervals of 0.1.

```
y0 = 3 # initial value
a = 0 # integration limits for t
b = 10
t = N. arange (a, b, 0.1)
```

Step 4 Call odeint.

```
y = odeint(f, y0, t) or y = odeint(f, 3, (0,10)) # actual computation of y(t)
```

Since *odeint* will integrate whatever equation you give it, and the function 'f' is going to be used many times, you must insure that it is correct: given a *t* and *y* it computes the correct *f(t,y)* and returns the variable which is *ydot*. Once you have checked the program and run it, it is easy to plot the solution.

```
plt . plot (t, y)
plt . xlabel ('t'); plt . ylabel ('y(t)')
plt . show ()
```

Numerical values can be obtained from

```
print(t, y)
```

to give

```
[ 0.   0.1  0.2  ......
[  3.00000000e+00]
[  2.45619230e+00]
[  2.01096019e+00]...
```

To solve a set of ordinary differential equations

$$\frac{dy_i}{dt} = f_i(\{y_j\}), \quad y_i(0) = y_{i0}$$

we do the same steps except that the initial condition is now a vector with N elements, the function 'rhs' must compute N functions using the vector y with N elements, and we have to return the result for each derivative.

```
# define the function evaluating the right-hand side
def ydot(y, t):
    return -y[0], -y[1], -y[2]

# set the initial conditions
y0 = np.zeros(3,dtype=float)
y0[0] = 1.
y0[1] = 2.
y0[2] = 3.

# set the integration limits and intervals for obtaining the solution
a = 0.
b = 11.
tspan = np. arange (a, b, 0.2)

#  call odeint to integrate the equations
y = odeint(ydot,y0,tspan)

# print the solution
print(tspan, y)

# plot the solution
plt . plot(tspan, y)
plt .show()
```

Examples with multiple parameters passed to the function are shown on pages 23 and 24. If your system is stiff (the eigenvalues of the linearized system are significantly different) the integration may taking an interminable amount of time. With MATLAB you had to use ode15s, which is designed for stiff systems. With Python, though, the odeint is apparently a compiled program, and it runs very fast so this usually won't be a problem.

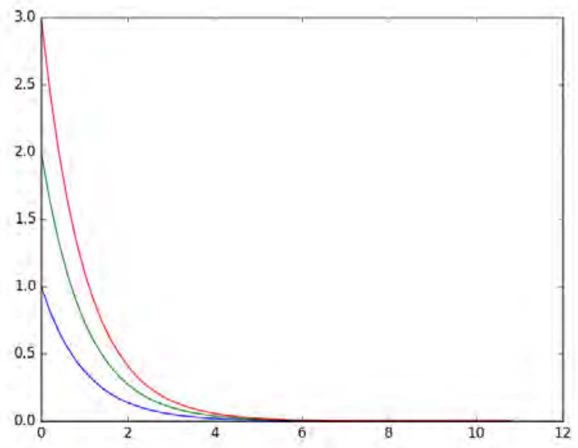


Figure F.5. Plot of Solution of Three ODEs

Differential-Algebraic Equations

Some problems may have ordinary differential equations and algebraic equations, too. There are add-ons to Python that will do this. See, for example, Assimulo at https://pypi.python.org/pypi/Assimulo.

Checklist for Using 'odeint'

When using 'odeint', your function for the right-hand side must meet these conditions:

- The name in the calling command must be the same as the function name.
- The variable *tspan* (or whatever it is called) must have at least two values.
- The number of entries in the vector for the initial conditions must be the same as the number of right-hand sides calculated in the function.
- The number of right-hand sides computed has to be equal to the number of differential equations. They are written in the return statement, with commas between them. Alternatively, the calculations can be put into a vector and the return statement can refer to the vector.
- Variables can be used in the function. Then can be part of a vector of numbers, or specific entries and are listed as arguments in the def function.
- The function will be called many times by the odeint function. However, you only have to check the calculation once. Check the function by giving it *t* and all the *y(i)*; compute what you expect the right-hand sides to be and see that the computer gives those values. This is the only way to ensure that Python is solving the problem you want solved.

F.9. PLOTTING

To plot in Python you need to have imported matplotlib, as discussed above. Many options are available at http://matplotlib.org/gallery.html.

Simple Plots

```
import numpy as np
import pylab as plt

def f(x):
    return 2. + 3.*x + 0.5*x*x

# create the data
x = np. linspace (0, 4, 50)
y = f(x)

# plotting
plt . plot (x,y,'o-g')
plt . xlabel ('x')
plt . ylabel ('y')
plt . title ('Test Plot')
plt . show ()
plt.savefig ('Figure_F6.png')
```

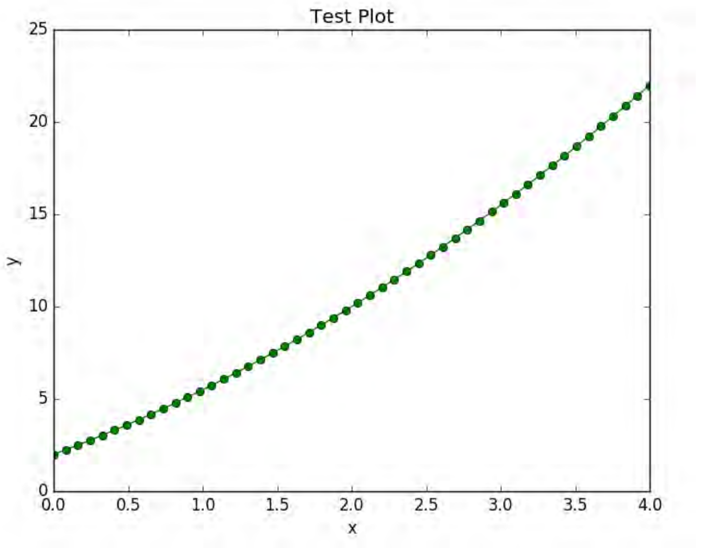


Figure F.6. Simple Plot

In the call to plot, the ‘o-g’ means plot the data with a circle, use a line between data points, and color it green. The plot appears on the screen. Press the image below the figure:

and you can save the figure to your computer.

More Complicated Plots

Possible symbols are (there are others):

+ plus	^ triangle (up)
o circle	v triangle (down)
* star	> triangle (right)
. point	< triangle (left)
x cross	p pentagram
s square	h hexagram
	none no marker (default)

The colors are in the default order for multiple plots (Pylab cycles through the first 5), but white is not used.

Colors		Line types	
b	blue –	___	solid
g	green	_ _	dashed
r	red	:	dotted
c	cyan	_.	dashdot
m	magenta		

y	yellow	<u>Line width</u>
k	black	lw = 2, etc.
w	white	

For a vector x with n entries, and a vector y with n entries, plot them in one of four ways:

```
1. plt(x,y)    2. plt.loglog(x,y)    3. plt.semilogx(x,y)    4. plt.semilogy(x,y)
```

To plot more than one variable:

```
plt . plot (x,y1,'o-g',x,y2,'x-r')   # if the y1 and y2 are known at the same values of x
plt(x1,y1,x2,y2)                     # if y1 is known at x1 and y2 is known at x2
```

You can also issue two plot commands, one for each line (y1 and y2), so that the line width can be different for the two lines.

Limit the axes:

```
v = [0., 1., 0., 1.5]
plt . axis(v)
```

Add a legend:

```
plt . plot (tspan, y, label='First curve')
plt . legend(loc='best')
```

Add a grid with plt.grid().

Plot only one column of a matrix, here the second row:

```
plt.plot (x,mat[1,:],'o-g')
plt.show()
```

Multiple plots

The command: subplot(2,1,1) says there will be two plots and the next plot command will be shown in the top half of the window. For the second figure, say subplot(2,1,2) and the next plot will be in below the first one.

```
>>>plt.subplot(2,1,1)
>>>plt.plot (x,y,'o-g')
>>>plt.subplot(2,1,2)
>>>plt. plot (x,y2,'x-r')
>>>plt.show()
```

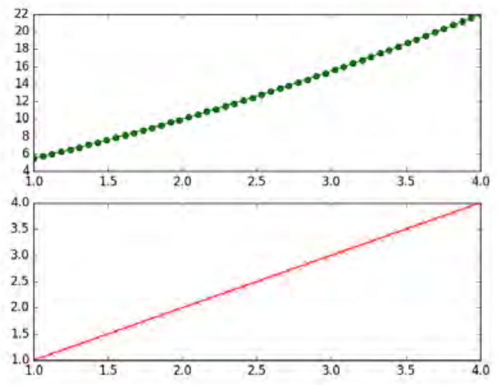


Figure B.7. Multiple Plots

The numbering system says that there are 2 rows, 1 figure across, and the last number refers to the plots in the order they are defined. If you want four figures, two in the top row, then use subplot(2,2,1), subplot(2,2,2), subplot(2,2,3), subplot(2,2,4). The first two numbers say that there will be 2 x 2 plots. The last number represents the plot as defined in order.

Bold, Italics, and Subscripts

Other options are to change the fontstyle, fontsize, fontweight, linewidth, and Greek letters.

```
plt . xlabel ('t', fontstyle='italic',fontsize=24)
plt . ylabel ('y', fontweight='bold')
plt . plot (tspan, y, linewidth = 2)
```

To add a subscript use plt . title(r'Q_1') to make Q_1.
To add a superscript use plt . title(r'Q^1') to make Q^1.
To use Greek letters use

```
plt . title(r'$\alpha$')
```

The **Greek letters** are (These are TEX commands.)

α	\ alpha	Γ	\ Gamma	∞	\ infty
β	\ beta	Δ	\ Delta	≥	\ geq
γ	\ gamma	Θ	\ Theta	≤	\ leq
.	.	.	.	∂	\ partial
.	.	.	.	±	\ pm
ω	\ omega	Ω	\ Omega	ℜ	\ Re
				ℑ	\ Im

Additional options can be found at
http://matplotlib.org/api/pyplot_api.html#matplotlib.pyplot.text.

Contour Plots

The following example comes from
http://matplotlib.org/examples/pylab_examples/contour_demo.html

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.mlab as mlab  #MATLAB compatibility command

plt.rcParams['xtick.direction'] = 'out'
plt.rcParams['ytick.direction'] = 'out'

delta = 0.025
x = np.arange(-3.0, 3.0, delta) # sets the x-values
y = np.arange(-2.0, 2.0, delta) # sets the y-values
X, Y = np.meshgrid(x, y) # MATLAB command for generating the grid
Z1 = mlab.bivariate_normal(X, Y, 1.0, 1.0, 0.0, 0.0)
Z2 = mlab.bivariate_normal(X, Y, 1.5, 0.5, 1, 1)
# difference of Gaussians
Z = 10.0 * (Z2 - Z1)
plt.figure()
CS = plt.contour(X, Y, Z)
plt.clabel(CS, inline=1, fontsize=10)
plt.title('Simplest default with labels')
plt . show ( )
```

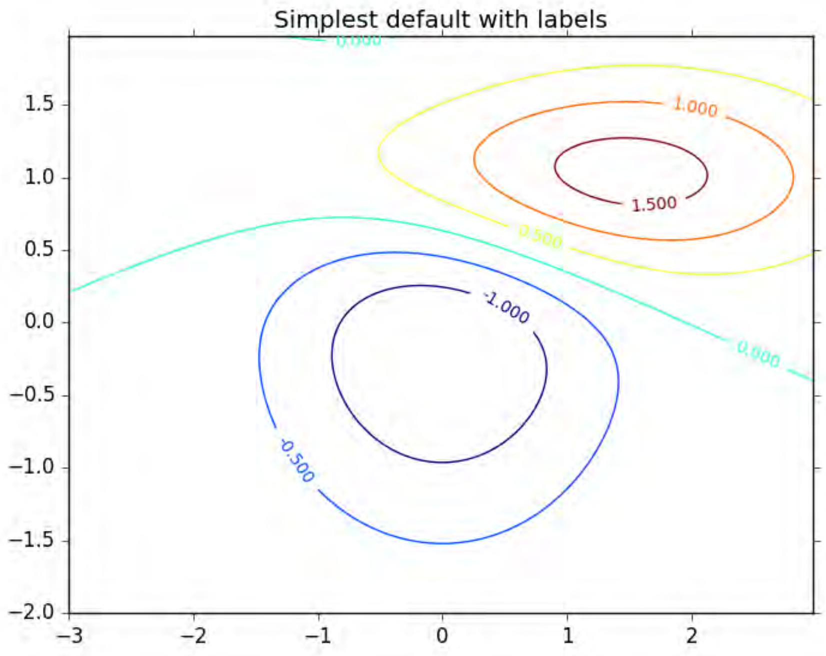


Figure F.8. Contour Plot of 2D Function

3D plots

To plot a function $z(x,y)$, create an $x-y$ grid (rectangular), evaluate the function at each grid point, and plot. The program to generate Figure F.9 is:

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.mlab as mlab
import mpl_toolkits.mplot3d.axes3d as p3 # note the new import command

delta = 0.025
x = np.arange(-2.0, 2.0, delta) #create the x-grid from -2 to 2, increment 0.025
y = np.arange(-2.0, 2.0, delta) #create the y-grid from -2 to 2, increment 0.025
X, Y = np.meshgrid(x, y)
Z = X*X + Y*Y                        # evaluate the function
fig = plt.figure()
ax = p3.Axes3D(fig)
ax.plot_surface(X, Y, Z)
ax.set_xlabel('X')
ax.set_ylabel('Y')
ax.set_zlabel('Z')
plt.show()
```

Addition information can be found at
https://scipy.github.io/old-wiki/pages/Cookbook/Matplotlib/mplot3D.html

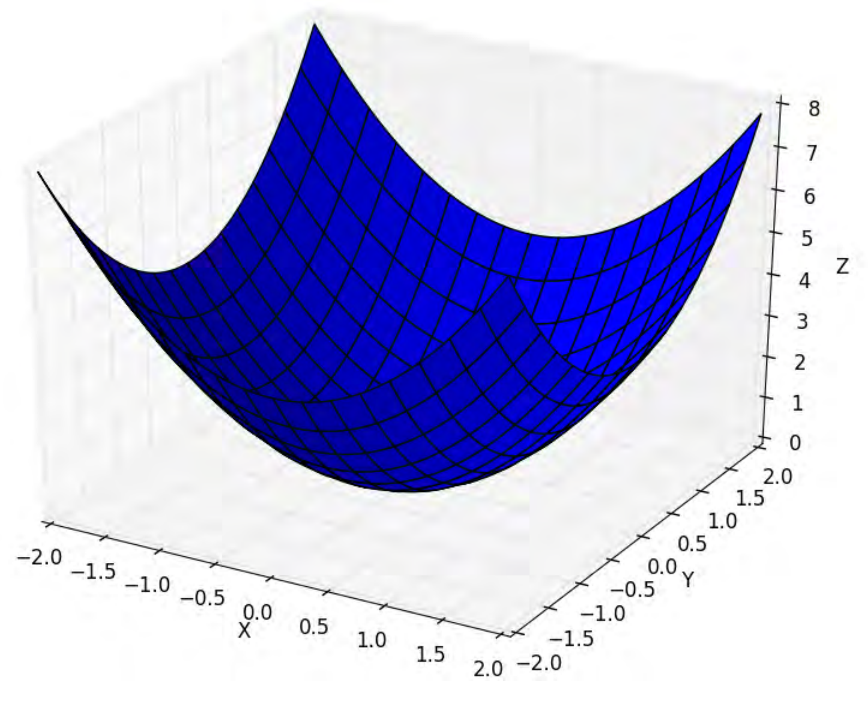


Figure F.9. 3D Plot of 2D Function using Python

F.10. PYTHON HELP

For detailed information and a complete list of all commands in the SciPy file, go to https://docs.python.org/3/library/index.html. You can also get the latest (and earlier) User Guides and Reference Guides for Numpy and Scipy at https://docs.scipy.org/doc/ : numpy-user-1.11.0.pdf and scipy-ref-0.18.1.pdf.

If you want to see what modules you can import, go to the terminal window and say `help('modules')`. If you want to see what scipy contains, `import scipy` and say `help(scipy)`. To see the details in the math section, go to the console and say `print(dir(math))`.

A source with scientific examples is *Introduction to Python for Computational Science and Engineering*, by Hans Fangohr of the University of Southhampton, available as a pdf from https://www.southampton.ac.uk/~fangohr/training/python/pdfs/Python-for-Computational-Science-and-Engineering.pdf

Additional useful websites include:
http://docs.python.org/2/tutorial
http://matplotlib.org/api/pyplot_summary.html
http://matplotlib.org/gallery.html
http://www.scipy.org/NumPy_for_Matlab_Users

There are also specialized programs submitted by others. See them at the Python Package Index: http://pypi.python.org/pypi. These require a compiler for another language and are not treated here, but they include methods to solve boundary value problems and partial differential equations (like in Chapter 9). If you want to do serious numerical analysis using Python, these programs should be considered. Modules there include methods using the finite element method, methods for problems with strong convection, spectral methods, and collocation methods.

F.11. APPLICATIONS OF PYTHON

There are many chemical engineering examples in the book that can use Python.

- Solving a single nonlinear equation, Ch. 2, p. 15, 16-20; Ch. 3, p. 34-35, Ch. 4., p. 53-56.
- Plotting, Ch. 2, p. 19.
- Multiple equations, few unknowns, Chapter 4, p. 58-59.

The Python programs applied to these problems are in the document *Introduction to Chemical Engineering Computing; Extension to Python*.pdf