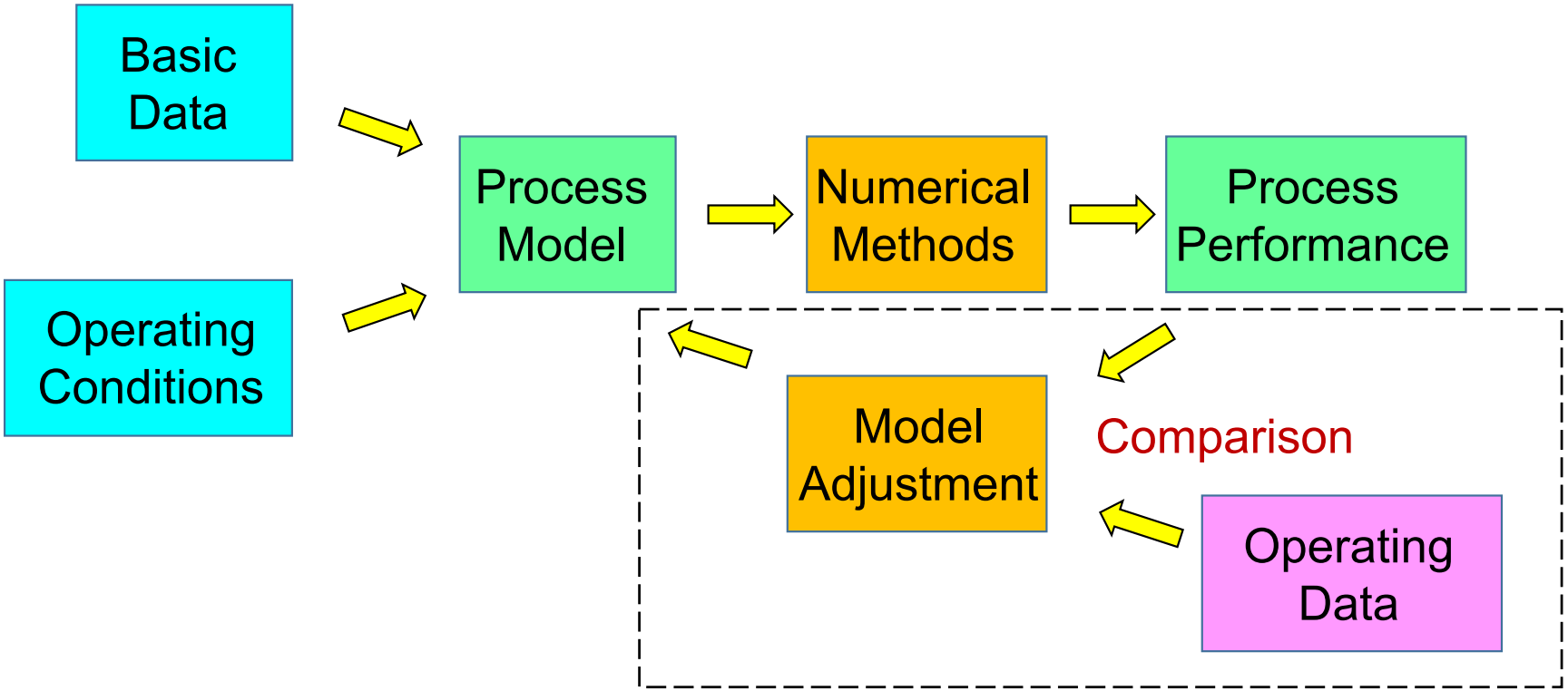
Python Bootcamps 1, 2 and 3

- ✓ 1: Getting up to speed with Python
- 2: Learning to use Python to solve typical problem scenarios
- 3: Detailed modeling of packed-bed and plug-flow reactors

Bootcamp 2 Outline

- Process modeling & numerical characteristics
- Algebraic models
  - Single, nonlinear, including polynomials
  - Linear sets
  - Nonlinear sets
- ODE models
  - Initial value problems
  - Split boundary problems
- Optimization
- Curve-fitting

Process Modeling



Often not possible at the design stage.

Process Modeling

Developing the Process Model

Conservation Balances

- Material
- Energy
  - Mechanical/Momentum
  - Thermal
- Thermodynamics
  - Equilibrium, Phase and Chemical
  - Heat Transfer
- Mass Transfer
- Separations
- Reaction Kinetics
- Monitoring and Control

Equipment

- Vessels
  - Tanks, Drums
  - Columns
  - Reactors
- Heat Exchangers
- Piping, Valves, Fittings
- Pumps, Compressors
- Columns
- Reactors
- Solids-handling
  - Crystallization
  - Filtration
- Instrumentation

Numerical Characteristics

- Algebraic equations
  - Nonlinear, one or more
  - Linear sets
- Differential equations
  - Ordinary (ODEs)
  - Partial (PDEs)
- Optimization
- Curve-fitting

when combined:
Differential-Algebraic
Systems (DAEs)

Examples Considered

- Single, nonlinear algebraic equation
  - water-gas shift equilibrium
- Set of linear algebraic equations
  - absorber column
- Set of nonlinear algebraic equations
  - steam/water equilibrium
- Single nonlinear ordinary differential equation
  - batch reactor, single reaction
- Set of nonlinear ordinary differential equations
  - batch reactor, multiple reactions
- Optimization
  - single factor
  - multiple factors with constraints
- Linear Regression
  - polynomial
  - general

Solving Single Algebraic Equations

- Bracketing methods
  - Bisection
  - False position
- Open methods
  - Newton-Raphson
  - Modified secant
- Hybrid
  - Brent's method

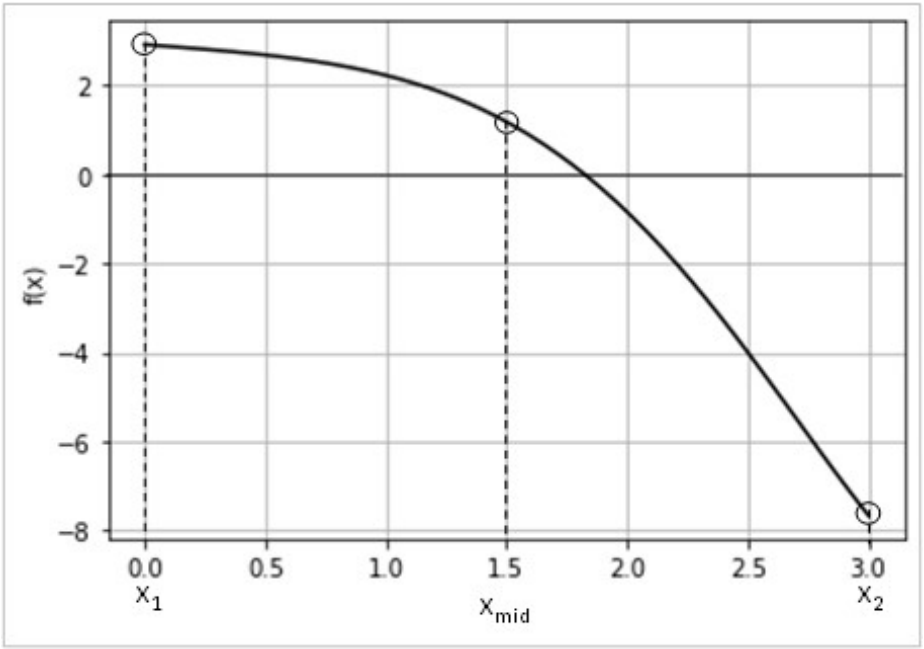
$$f(x) = 0$$

- Circular scenario
  - Substitution
  - Wegstein method

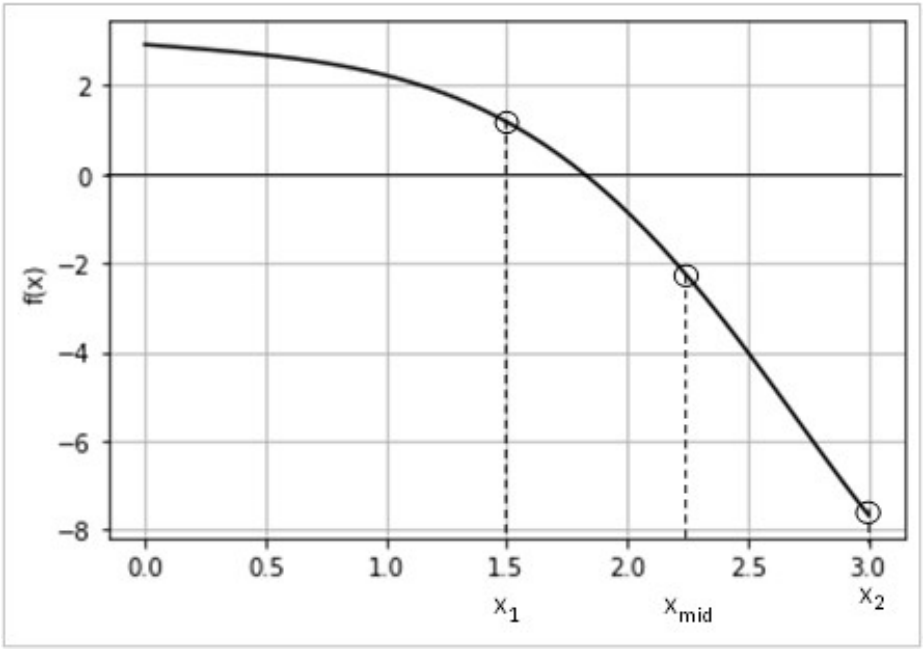
$$x = g(x)$$

For details and Python code on all these methods, see Chapra and Clough, *Applied Numerical Methods with Python for Engineers and Scientists*, McGraw-Hill, 2022.

Solving Single Algebraic Equations - Bisection



First iteration



Second iteration

Solving an Algebraic Equation with Bisection

```python
def bisect(func,x1,x2,maxit=20):
    """
    Uses the bisection method to estimate a root of func(x).
    The method is iterated maxit (default = 20) times.
    Input:
        func = name of the function
        x1 = lower guess
        x2 = upper guess
    Output:
        xmid = root estimate
        or
        error message if initial guesses do not bracket solution
    """
    if func(x1)*func(x2)>0:
        return 'initial estimates do not bracket solution'
    for i in range(maxit): # carry out maxit iterations
        xmid = (x1+x2)/2  # calculate midpoint
        if func(xmid)*func(x1)>0:  # check if f(xmid) same sign as f(x1)
            x1 = xmid   # if so, replace x1 with xmid
        else:
            x2 = xmid  # if not, replace x2 with xmid
    return xmid  # return the latest value of xmid as the solution
```

docustring

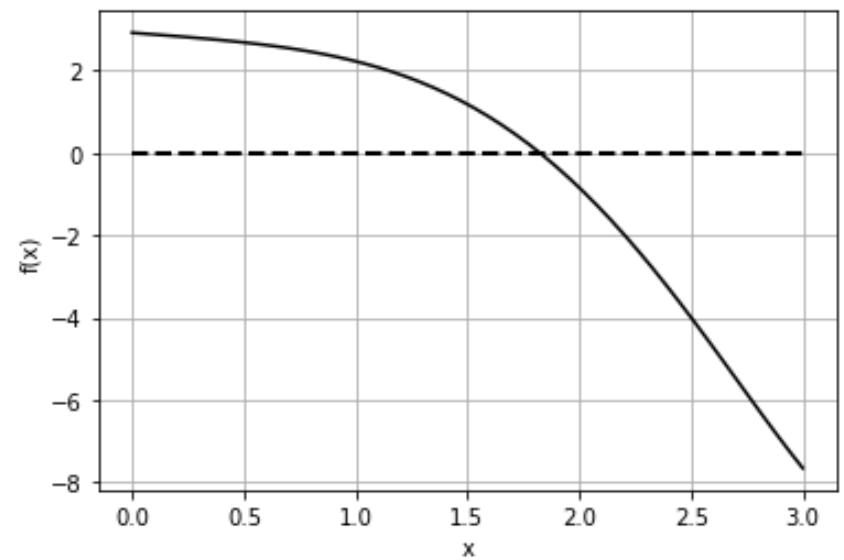
bisect1.py

Solving an Algebraic Equation with Bisection

Example $f(x) = \sin(x+2)\cdot\cosh(x)+2=0 \qquad x_1 = 0 \qquad x_2 = 3$

```
import numpy as np

def f(x):
    return np.sin(x+2)*np.cosh(x)+2.

x1 = 0. ; x2 = 3.
xsoln = bisect(f,x1,x2)
print('solution is {0:6.4f}'.format(xsoln))
```

```
solution is 1.8229
```



Solving an algebraic equation with SciPy brentq function from the optimize submodule

xs = brentq(f,x1,x2)

f : function name

x1,x2 : initial estimates that bracket the solution

Example $f(x) = \sin(x+2)\cdot\cosh(x)+2=0 \qquad x_1 = 0 \qquad x_2 = 3$

```
import numpy as np
from scipy.optimize import brentq

def f(x):
    return np.sin(x+2)*np.cosh(x)+2.

x1 = 0.
x2 = 3.

xsoln = brentq(f,x1,x2)

print('solution is {0:6.4f}'.format(xsoln))
```

```
solution is 1.8229
```

brent1.py

Solving an algebraic equation with SciPy brentq function from the optimize submodule

full output option

```
xsoln,results = brentq(f,x1,x2,full_output=True)

print('solution is {0:6.4f}'.format(xsoln))
print(results)
```

```
solution is 1.8229
      converged: True
           flag: 'converged'
 function_calls: 10
     iterations: 9
           root: 1.8228849971055754
```

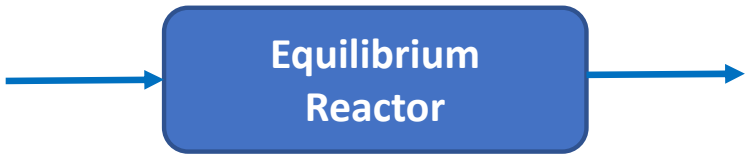
Example: water-gas shift equilibrium

$$CO + H_2O \Leftrightarrow H_2 + CO_2$$

$$\frac{[H_2]\cdot[CO_2]}{[H_2O]\cdot[CO]} = K_{eq}(T) \qquad ln\left[K_{eq}(T)\right] = -3.112 + \frac{3317}{T} \qquad T(K)$$

Feed

	kmol/hr
H_2	450
CO	500
CO_2	50
H_2O	1150



Product

T = 1200 °C
P = 1 atm

$$f(x) = \frac{\left[Feed_{H_2} + x\right]\cdot\left[Feed_{CO_2} + x\right]}{\left[Feed_{H_2O} - x\right]\cdot\left[Feed_{CO} - x\right]} - K_{eq}(T) = 0$$

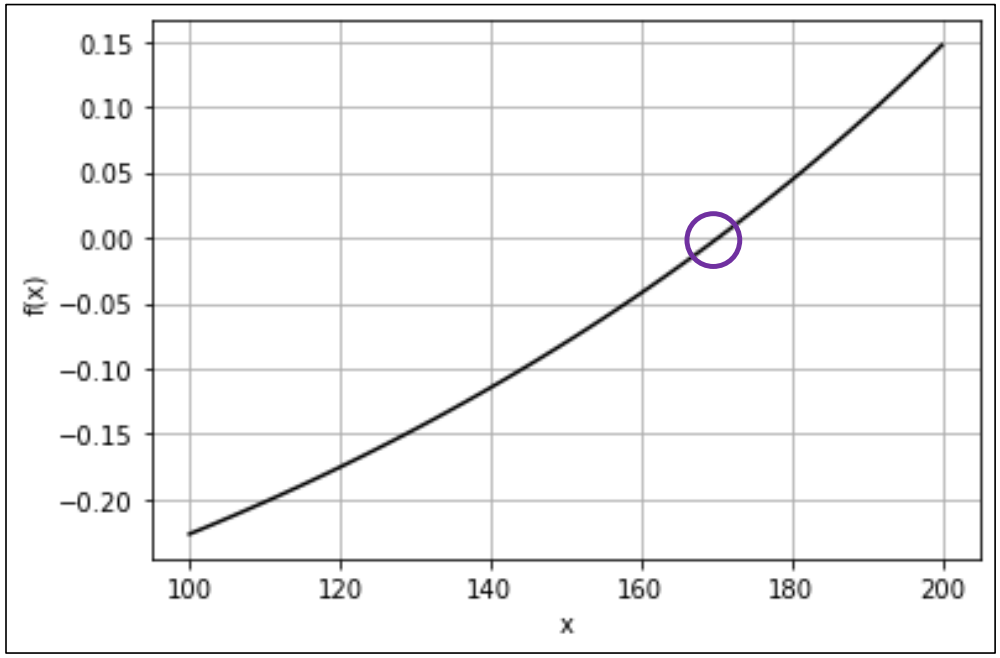
where x is the shift to equilibrium in kmol/hr.

Solve for x and the reactor product flow rates for the given temperature.

Water-gas shift equilibrium

Setting up a function to compute f(x)

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

def f(x):
    T = 1200 # degC
    TK = T + 273.15  # K
    Keq = np.exp(-3.112+3317/TK) # equil constant
    FeedH2 = 450 # kmol/hr
    FeedCO2 = 50
    FeedH2O = 1150
    FeedCO = 500
    ferr = (FeedH2+x)*(FeedCO2+x)/(FeedH2O-x)/(FeedCO-x) - Keq
    return ferr

x = np.linspace(100,200)
plt.plot(x,f(x),c='k')
plt.grid()
plt.xlabel('x')
plt.ylabel('f(x)')
```



Plot shows solution **x** = ~170

watergasplot.py

Water-gas shift equilibrium

Setting up a function to compute f(x)
including temperature and feed rates as arguments

```
def f(x,T,FeedH2,FeedCO2,FeedH2O,FeedCO):
    TK = T + 273.15  # K
    Keq = np.exp(-3.112+3317/TK) # equil constant
    ferr = (FeedH2+x)*(FeedCO2+x)/(FeedH2O-x)/(FeedCO-x) - Keq
```

Case study of equilibrium flow rates for a range of temperatures

Water-gas shift equilibrium

Passing extra arguments through a built-in function

Main script
Set T and Feed rates
Set x1 and x2

need to get these values “through” brentq to f using args=(...)

xsoln = brentq(f,x1,x2,args=(...))

function f with arguments x, T, Feed rates

Water-gas shift equilibrium

Case study for a range of temperatures with a plot

Use NumPy **append** function here to extend arrays instead of creating zero-filled arrays.

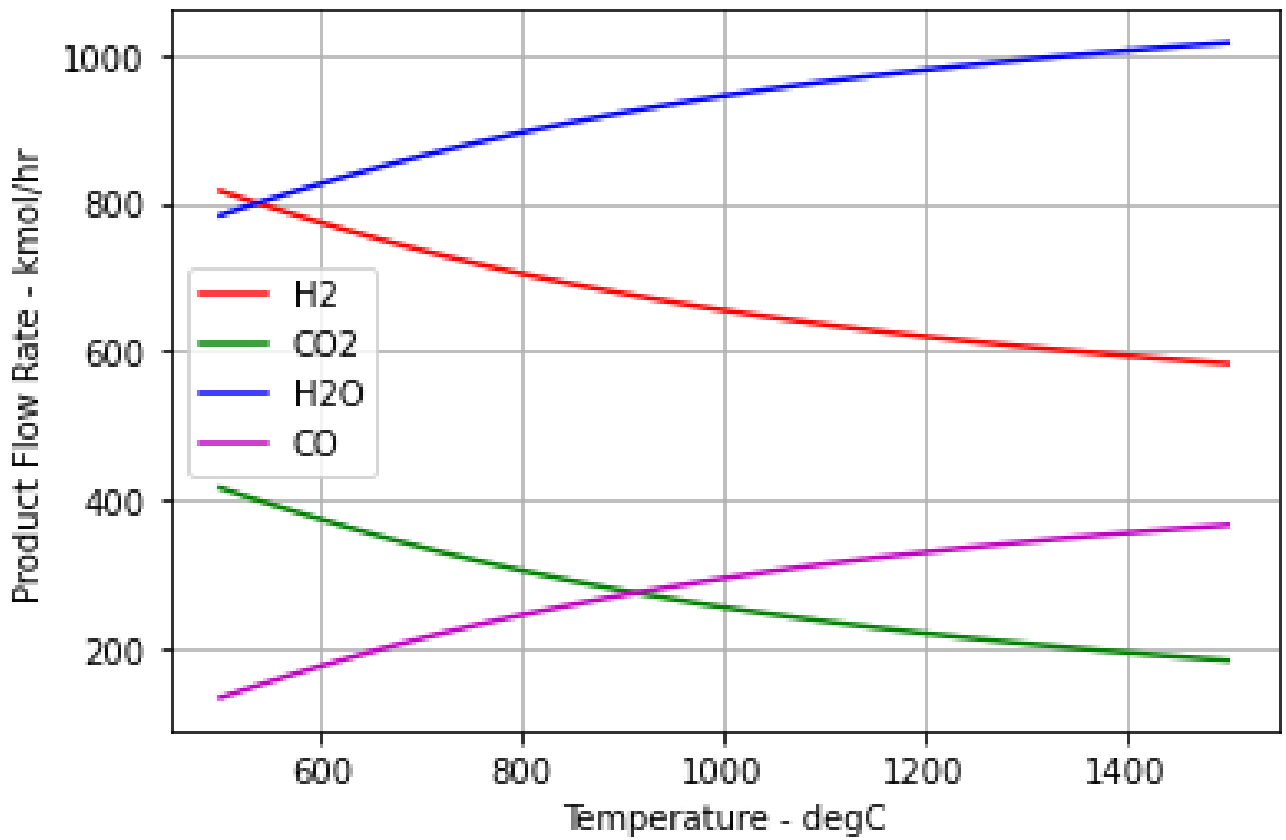
```
import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import brentq

def f(x,T,FeedH2,FeedCO2,FeedH2O,FeedCO):
    TK = T + 273.15  # K
    Keq = np.exp(-3.112+3317/TK) # equil constant
    ferr = (FeedH2+x)*(FeedCO2+x)/(FeedH2O-x)/(FeedCO-x) - Keq
    return ferr

FeedH2 = 450 # kmol/hr
FeedCO2 = 50
FeedH2O = 1150
FeedCO = 500

T = np.arange(500,1525,25)
n =len(T)
x = []
ProdH2 = []
ProdCO2 = []
ProdH2O = []
ProdCO = []
```

```
x1 = 50 ; x2 = 450
for i in range(n):
    xsoln = brentq(f,x1,x2,args=(T[i], \
           FeedH2,FeedCO2,FeedH2O,FeedCO))
    x = np.append(x,xsoln)
    ProdH2 = np.append(ProdH2,FeedH2+xsoln)
    ProdCO2 = np.append(ProdCO2,FeedCO2+xsoln)
    ProdH2O = np.append(ProdH2O,FeedH2O-xsoln)
    ProdCO = np.append(ProdCO,FeedCO-xsoln)

plt.plot(T,ProdH2,c='r',label='H2')
plt.plot(T,ProdCO2,c='g',label='CO2')
plt.plot(T,ProdH2O,c='b',label='H2O')
plt.plot(T,ProdCO,c='m',label='CO')
plt.grid()
plt.legend()
plt.xlabel('Temperature - degC')
plt.ylabel('Product Flow Rate - kmol/hr')
```

watergas_solve.py

Water-gas shift equilibrium

Case study for a range of temperatures with a plot



Solving for the Roots of Polynomials

General form for nth-order polynomial

$$a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0 = 0$$

will have *n* roots, either real or complex conjugate pairs, $\alpha + \beta j \quad \alpha - \beta j$

Example: $x^4 - 8x^3 - 3x^2 + 62x + 56 = 0$

NumPy function **roots**

```
import numpy as np

coeff = [1., -8., -3., 62., 56.]
r = np.roots(coeff)
print('roots are\n',r)
```

```
roots are
 [ 7.  4. -2. -1.]
```

Example: $x^2 + x + 1 = 0$

polynomial_example.py

```
coeff2 = [1., 1., 1.]
r2 = np.roots(coeff2)
print('\nroots are\n',r2)
```

```
roots are
 [-0.5+0.8660254j -0.5-0.8660254j]
```

Solving for the Roots of Polynomials

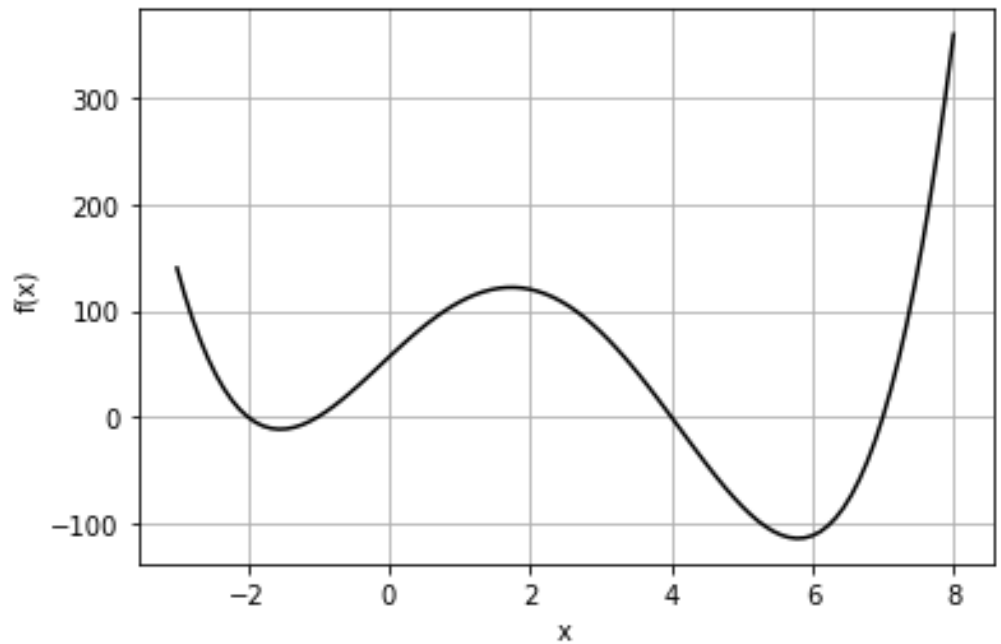
Composing the polynomial from its roots, NumPy **poly** function

```
coeff1 = np.poly([7., 4., -2., -1.])
print('\ncoefficients are\n',coeff1)
```

```
coefficients are
 [ 1. -8. -3. 62. 56.]
```

Evaluating a polynomial to create a plot

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

coeff = [1., -8., -3., 62., 56.]
x = np.linspace(-3.,8.,100)
y = np.polyval(coeff,x)

plt.plot(x,y,c='k')
plt.grid()
plt.xlabel('x')
plt.ylabel('f(x)')
```



polynomial_example2.py

Solving Sets of Linear Algebraic Equations

n equations in *n* unknowns

$$
\begin{aligned}
a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n &= b_1 \\
a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n &= b_2 \\
&\vdots \\
a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn}x_n &= b_n
\end{aligned}
\qquad
\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}
\qquad
\mathbf{A} \cdot \mathbf{x} = \mathbf{b}
$$

Solving the equations:

1. matrix algebra and computations

$$
\begin{aligned}
\mathbf{A}^{-1} \cdot \mathbf{A} \cdot \mathbf{x} &= \mathbf{A}^{-1} \cdot \mathbf{b} \\
\mathbf{I} \cdot \mathbf{x} &= \mathbf{A}^{-1} \cdot \mathbf{b} \\
\mathbf{x} &= \mathbf{A}^{-1} \cdot \mathbf{b}
\end{aligned}
$$

compute the inverse of **A** and multiply it by **b**

2. more efficient numerical method

- Gaussian elimination with enhancements
- LU decomposition

Solving Sets of Linear Algebraic Equations

Example

$$3x + 2y - z = 10$$
$$-x + 3y + 2z = 5$$
$$x - y - z = -1$$

$$\begin{bmatrix} 3 & 2 & -1 \\ -1 & 3 & 2 \\ 1 & -1 & -1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 10 \\ 5 \\ -1 \end{bmatrix}$$

A^{-1}*b ⇨ not recommended, inefficient

simple_linear_set_linalg_solve.py

```
import numpy as np

A = [[3, 2, -1],[-1, 3, 2],[1,-1,-1]]
b = [10, 5, -1]

x = np.linalg.solve(A,b)
print(x)

[-2.  5. -6.]
```

using the **solve** function
in the NumPy **linalg** submodule

The **solve** function is adapted from the LAPACK package **gesv** routine and uses LU decomposition with partial pivoting.

Solving Sets of Linear Algebraic Equations

Example problem: Six-stage absorber column

Equilibrium relationship on tray *i* $y_i = ax_i + b$

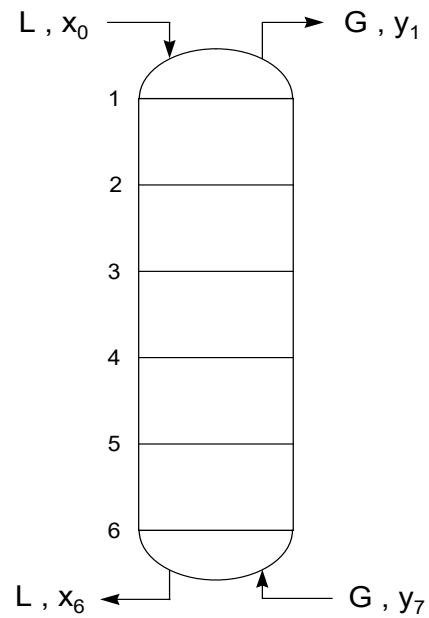
x_0 , y_7 , L and G specified

Component material balance on tray *i*

$$L \cdot x_{i-1} + G \cdot y_{i+1} = L \cdot x_i + G \cdot y_i$$

Incorporate equilibrium relationship

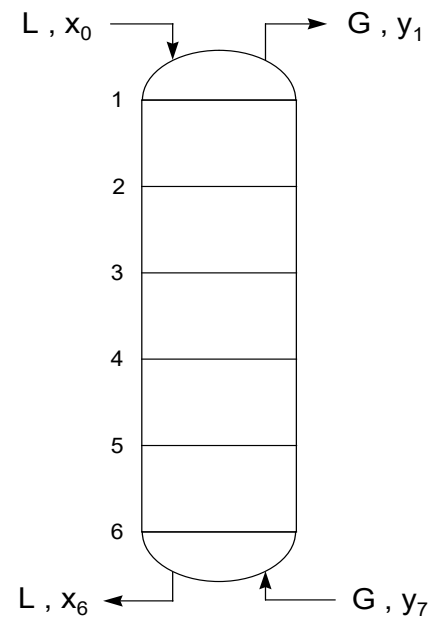
$$L \cdot x_{i-1} - \left(L + G \cdot a\right) \cdot x_i + G \cdot a \cdot x_{i+1} = 0$$



Solving Sets of Linear Algebraic Equations

Example problem: Six-stage absorber column

Write component material balances for each tray and rearrange with unknowns on the left and knowns on the right.



$$-(L+Ga)x_1 + Gax_2 = -Lx_0$$
$$Lx_1 - (L+Ga)x_2 + Gax_3 = 0$$
$$Lx_2 - (L+Ga)x_3 + Gax_4 = 0$$
$$Lx_3 - (L+Ga)x_4 + Gax_5 = 0$$
$$Lx_4 - (L+Ga)x_5 + Gax_6 = 0$$
$$Lx_5 - (L+Ga)x_6 = -G(y_7 - b)$$

This represents a set of six linear equations in the six unknown mass fractions.

Basic data: equilibrium model: a = 0.7, b = 0
Operating conditions: L = 20 mol/s, G = 12 mol/s
Inlet gas mole fraction: $y_7 = 0.1$
Inlet liquid mole fraction: $x_0 = 0$

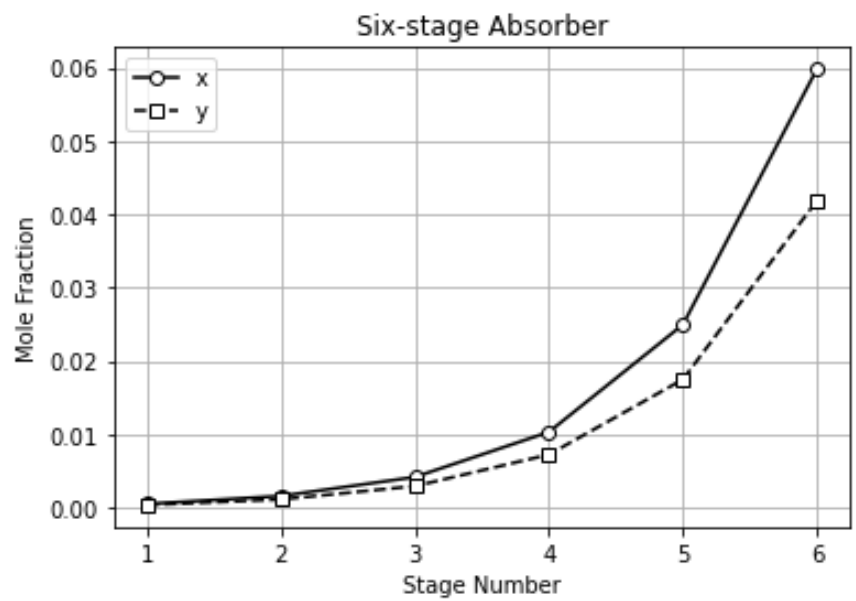
Solving Sets of Linear Algebraic Equations

Example problem: Six-stage absorber column

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

a = 0.7  # equilibrium factor
yin = 0.1  # vapor entry mole fraction
L = 20  # mol/s
G = 12  # mol/s
n = 6  # number of stages

A = np.zeros((n,n))
b = np.zeros(n)

for i in range(n):
    A[i,i] = -(L+G*a)
for i in range(n-1):
    A[i,i+1] = G*a
for i in range(1,n):
    A[i,i-1] = L
b[5] = -G*yin

x = np.linalg.solve(A,b)
y = a*x
```

```
stage = np.arange(1,7)

plt.plot(stage,x,c='k',marker='o',mec='k',mfc='w',label='x')
plt.plot(stage,y,c='k',ls='--',marker='s',mec='k',mfc='w',label='y')
plt.grid()
plt.legend()
plt.xlabel('Stage Number')
plt.ylabel('Mole Fraction')
plt.title('Six-stage Absorber')
```

Absorber.py



Solving Sets of Nonlinear Algebraic Equations

$$\begin{aligned} f_1\left(x_1, x_2, \ldots, x_n\right) &= 0 \\ f_2\left(x_1, x_2, \ldots, x_n\right) &= 0 \\ &\vdots \\ f_n\left(x_1, x_2, \ldots, x_n\right) &= 0 \end{aligned} \qquad \text{or} \qquad \mathbf{f}\left(\mathbf{x}\right) = \mathbf{0}$$

Common solution technique: Newton's Method

Start with an initial estimate of the solution: $\mathbf{x}^0$

Iterate with $\mathbf{x}^{i+1} = \mathbf{x}^i - \mathbf{J}^{-1}\left(\mathbf{x}^i\right) \cdot \mathbf{f}\left(\mathbf{x}^i\right)$ until a convergence criterion is met.

Jacobian matrix

$$\mathbf{J} = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \cdots & \frac{\partial f_2}{\partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_n}{\partial x_1} & \frac{\partial f_n}{\partial x_2} & \cdots & \frac{\partial f_n}{\partial x_n} \end{bmatrix}$$

or, where analytical derivatives are difficult:

$$\frac{\partial f_1}{\partial x_1}\left(\mathbf{x}^i\right) \cong \frac{f_1\left(x_1^i + \delta, x_2^i, \ldots, x_n^i\right) - f_1\left(x_1^i - \delta, x_2^i, \ldots, x_n^i\right)}{2 \cdot \delta}$$

and so forth.

Solving Sets of Nonlinear Algebraic Equations

A Python function for Newton's method: **multinewt**

```
def multinewt(f,J,x0,Ea=1.e-7,maxiter=30,decel=1.0):
    xold = x0
    for iter in range(maxiter):
        Jinv = np.linalg.inv(J(xold))
        xnew = xold - decel*Jinv.dot(f(xold))
        xdev = xnew - xold
        xerr = xdev.dot(xdev)
        if xerr < Ea:
            break
        xold = xnew
    return xnew,iter+1
```

f : function to evaluate equations' errors
J: function to provide the Jacobian
Ea: relative error criterion for convergence
maxiter: maximum number of iterations
decel: decelerator, if needed for stability

- **xnew** computed with the Newton formula using **dot** method
- **xdev** is a vector of x differences from last iteration to this one
- **xerr** is sum of squares of **xdev** elements (inner product with **dot**)
- function returns the latest **xnew** and number of iterations

Solving Sets of Nonlinear Algebraic Equations

Solution with function **multinewt**

Example

$$x^2 + y^2 - 4 = 0$$
$$x \cdot y - 1 = 0$$

$$\mathbf{J} = \begin{bmatrix} 2x & 2y \\ y & x \end{bmatrix}$$

```
def f(x):
    fn = np.zeros(2)
    fn[0] = x[0]**2 + x[1]**2 - 4
    fn[1] = x[0]*x[1] -1
    return fn

def J(x):
    Jac = np.zeros((2,2))
    Jac[0,0] = 2*x[0]
    Jac[0,1] = 2*x[1]
    Jac[1,0] = x[1]
    Jac[1,1] = x[0]
    return Jac

x0 = np.array([0.5, 1.4])
xsoln,niter = multinewt(f,J,x0)
print(xsoln)
print(niter)
```

MultiNewton.py

```
[0.51763809 1.93185165]
4
```

Solving Sets of Nonlinear Algebraic Equations

Solution with the **root** function from the SciPy **optimize** submodule

Example $$x^2 + y^2 - 4 = 0$$
$$x \cdot y - 1 = 0$$

There is an option to allow the Jacobian matrix to be supplied.

x = root(f,x0) without supplying J (Jacobian approximated numerically)

```
import numpy as np
from scipy.optimize import root

def f(x):
    fn = np.zeros(2)
    fn[0] = x[0]**2 + x[1]**2 - 4
    fn[1] = x[0]*x[1]-1
    return fn

x0 = np.array([0.5, 1.4])
result = root(f,x0)

print('\nsolution is',result.x)
print('\n',result)
```

root returns a solution object, **result**

x is the solution – **result.x** is the property of **result**

```
solution is [0.51763809 1.93185165]

 message: The solution converged.
 success: True
  status: 1
     fun: [ 4.859e-12  1.982e-11]
       x: [ 5.176e-01  1.932e+00]
    nfev: 11
    fjac: [[-4.791e-01 -8.778e-01]
           [ 8.778e-01 -4.791e-01]]
       r: [-2.129e+00 -1.971e+00  3.232e+00]
     qtf: [ 2.564e-09  6.572e-10]
```

root_function_example.py

Solving Sets of Nonlinear Algebraic Equations

Example problem: steam/water equilibrium

$$P \cdot V = \frac{m}{MW} \cdot R \cdot (T + 273.15)$$

ideal gas law

$$log_{10} P = A - \frac{B}{T + C}$$

Antoine equation

P : absolute pressure, Pa

V : vapor volume, m^3

m : mass of vapor, kg

MW: H_2O molecular weight, ≅ 18.02 kg/kgmol

R : gas law constant, 8314 (Pa•m^3)/(kgmol•K)

T : temperature, °C

A, B, C : Antoine constants for H_2O

$A = 11.21 \quad B = 2354.7 \quad C = 280.71$

Operating conditions: $m = 3.755\ kg \qquad V = 3.142\ m^3$

Solve for P and T .

Solving Sets of Nonlinear Algebraic Equations

Example problem: steam/water equilibrium

Formulating the problem for solution

$$f_1(T,P) = P \cdot V - \frac{m}{MW} \cdot R \cdot (T + 273.15)$$

$$f_2(T,P) = log_{10} P - A + \frac{B}{T + C}$$

$$\mathbf{J}\left(\begin{bmatrix} P \\ T \end{bmatrix}\right) = \begin{bmatrix} V & -\frac{m}{MW} \cdot R \\ \frac{1}{ln(10) \cdot P} & -\frac{B}{(C+T)^2} \end{bmatrix}$$

analytical Jacobian practical in this case

A possible issue here is the comparative scaling of the two equations. Typical values for the PV term could be of magnitude 10^6; whereas, terms in the second equation are closer to unity. A practical approach to this is to scale the first equation by dividing it by, e.g., 100,000.

$$f_1(T,P) = \left(P \cdot V - \frac{m}{MW} \cdot R \cdot (T + 273.15) \right) \Big/ 100000$$

$$f_2(T,P) = log_{10} P - A + \frac{B}{T + C}$$

$$\mathbf{J}\left(\begin{bmatrix} P \\ T \end{bmatrix}\right) = \begin{bmatrix} V/1e5 & -\frac{m}{MW} \cdot R \Big/ 1e5 \\ \frac{1}{ln(10) \cdot P} & -\frac{B}{(C+T)^2} \end{bmatrix}$$

Solving Sets of Nonlinear Algebraic Equations

Example problem: steam/water equilibrium

```
import numpy as np
from scipy.optimize import root

R = 8314.  # Pa*m3/(kmol*K)
MW = 18.02  # kg/kmol
# Antoine coefficients
A = 11.21 ; B = 2354.7 ; C = 280.7

def SteamEq(x,m,V):
    P = x[0]
    T = x[1]
    ferr = np.zeros(2)
    ferr[0] = (P*V - m/MW*R*(T+273.15))/1e5
    ferr[1] = np.log10(P) - A + B/(T+C)
    return ferr

m = 3.755  # kg
V = 3.142  # m3

x0 = [ 2.e5, 110.]

result = root(SteamEq,x0,args=(m,V))
print('\nsolution is:')
print('P = {0:5.1f} kPa'.format(result.x[0]/1000))
print('T = {0:6.1f} degC'.format(result.x[1]))
```

SteamEquilbrium.py

```
solution is:
P = 216.9 kPa
T =  120.2 degC
```

Solving Single Differential Equations

Two types

generally, analytical solutions are not feasible

$$\frac{dy}{dt} = f(t) \quad \Longrightarrow \quad \int_{y_0}^{y_f} dy = y_f - y_0 = \underline{\int_{t_0}^{t_f} f(t)\,dt}$$

finding the area under the curve or quadrature

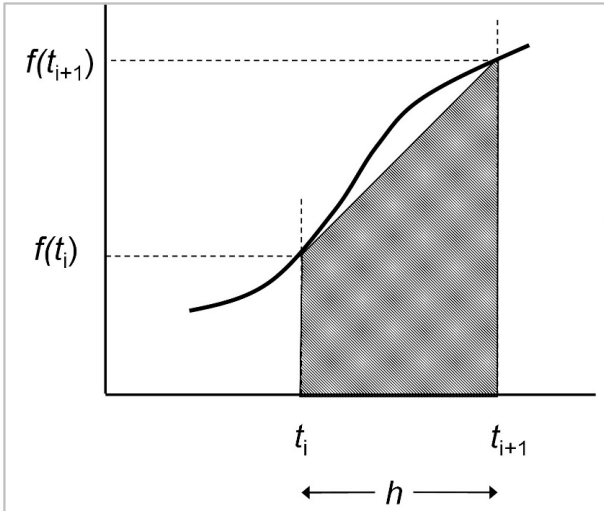
$$\frac{dy}{dt} = f(t, y)$$

numerical methods used to solve

Solving Single Differential Equations

Quadrature – Analytical function

Trapezoidal rule



$$Area = \frac{f\left(t_{i+1}\right) + f\left(t_i\right)}{2} \cdot h$$

$$\int_a^b f\left(t\right) \cdot dt \cong \left(f\left(a\right) + 2f\left(a + \Delta t\right) + 2f\left(a + 2\Delta t\right) + \cdots + 2f\left(a + \left(n-1\right)\Delta t\right) + f\left(b\right)\right)\frac{h}{2}$$

Solving Single Differential Equations

Quadrature – Analytical function

$$\frac{dy}{dt} = t \cdot \cos(t) \quad \Rightarrow \quad y = \int_0^{\pi/2} t \cdot \cos(t) \cdot dt$$

Python function for trapezoidal rule, **trap**

```
def trap(f,a,b,n=100):
    x = a  # set x to left side a
    h = (b-a)/n  # compute interval width
    sm = f(a)  # first term of sum
    for i in range(n-1):
        x = x + h  # advance x
        sm = sm + 2*f(x)  # add 2 * f(x) to sum
    sm = sm + f(b)  # add last term to sum
    ar = sm*h/2  # complete integral formula
    return ar

import numpy as np

def f(t):
    return t*np.cos(t)

y = trap(f,0.,np.pi/2)
print(y)
```

```
0.5707434665276926
```

Solving Single Differential Equations

quad_example.py

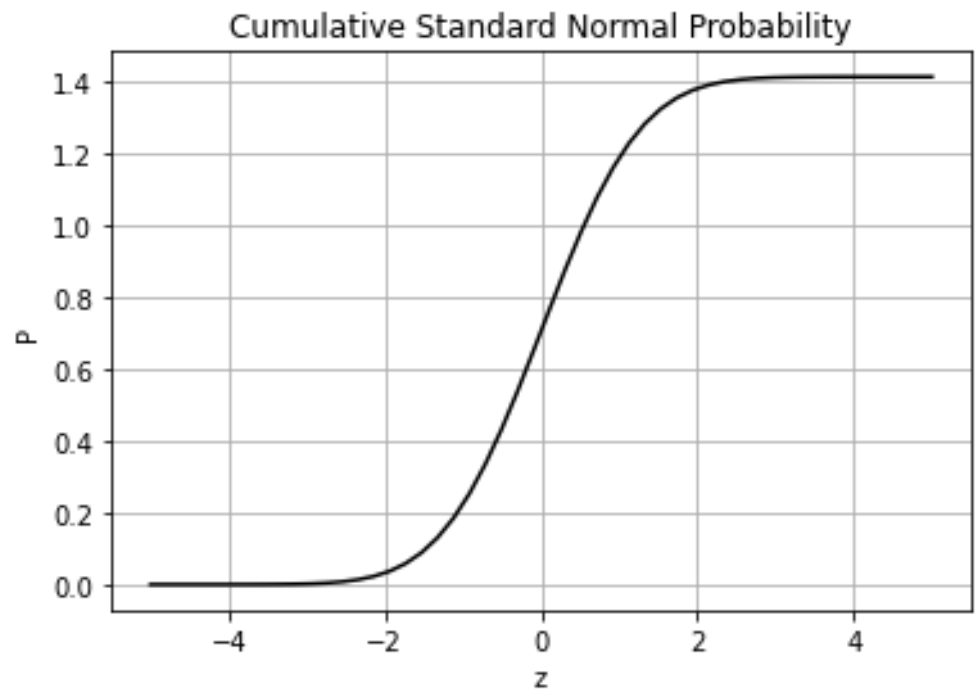
Quadrature – Analytical function

– **quad** function from the SciPy **integral** submodule

Standard normal distribution – cumulative probability

$$f(z) = \frac{1}{\sqrt{2\pi}} e^{-\frac{1}{2}z^2} \quad \Rightarrow \quad \frac{dP}{dz} = f(z) \quad \Rightarrow \quad P[-\infty \le z \le a] = \int_{-\infty}^{a} f(z)\,dz$$

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

stdnormdens = lambda z: 1/np.sqrt(np.pi)*np.exp(-z**2/2)

a = np.linspace(-5,5)
n = len(a)
P = np.zeros(n)

for i in range(n):
    P[i],ep = quad(stdnormdens,-np.inf,a[i])

plt.plot(a,P,c='k')
plt.grid()
plt.xlabel('z')
plt.ylabel('P')
plt.title('Cumulative Standard Normal Probability')
```



Solving Single Differential Equations

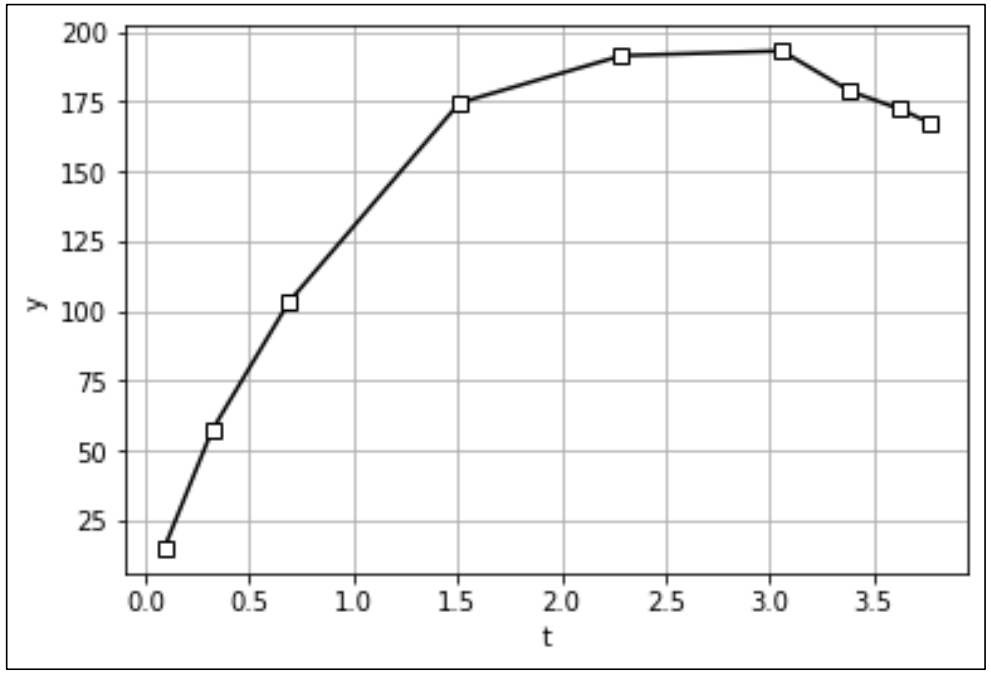
trapz_example.py

Quadrature – Data

SciPy function **trapz** from the **integrate** submodule

```
import numpy as np
from scipy.integrate import trapz

t = np.array([0.09,0.32,0.69,1.51,2.29,3.06,3.39,3.63,3.77])
y = np.array([15.1,57.3,103.3,174.6,191.5,193.2,178.7,172.3,167.5])

A = trapz(y,t)
print('\narea = {0:5.1f}'.format(A))

plt.plot(t,y,c='k',marker='s',mec='k',mfc='w')
plt.grid()
plt.xlabel('t')
plt.ylabel('y')
```

```
area = 570.1
```



Solving Single Differential Equations

Initial Value Problem

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

Example

$$\frac{dy}{dt} = 5\left(y - t^2\right) \qquad y\left(0\right) = 0.08 \qquad 0 \le t \le 5$$

There is an analytical solution:

$$y = t^2 + 0.4t + 0.08$$

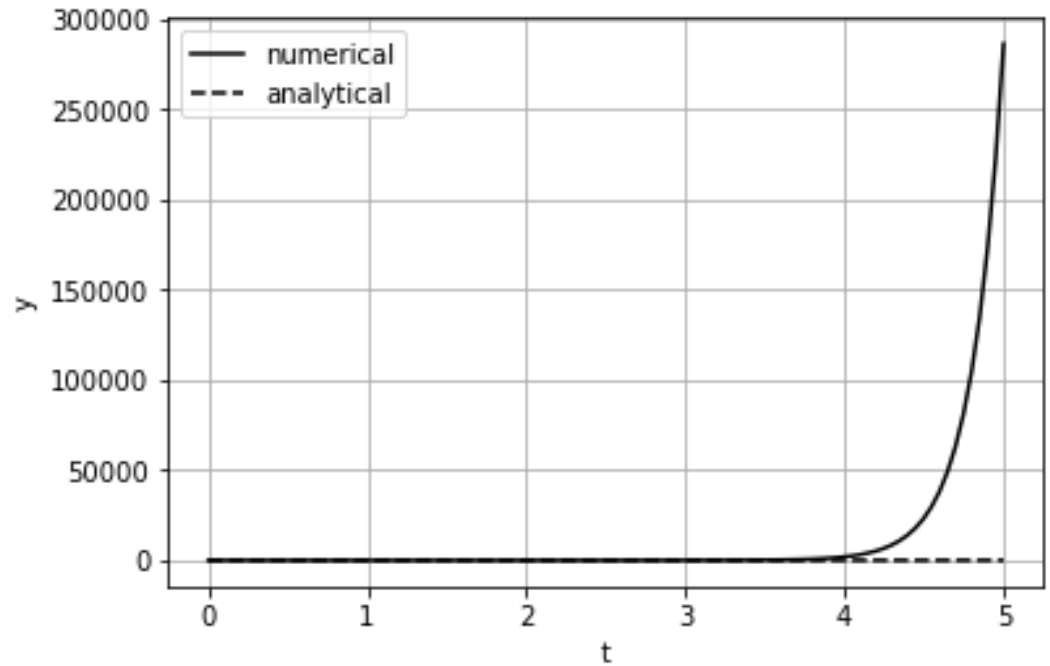
Solving Single Differential Equations

Initial Value Problem

Example using **solve_ivp**

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

f = lambda t,y: 5*(y-t**2)

fa = lambda t: t**2 + 0.4*t + 0.08

tspan = [0., 5.]
teval = np.linspace(0.,5.,100)
y0 = [0.08]

result = solve_ivp(f,tspan,y0,t_eval=teval)
tm = result.t
yr = result.y[0,:]

plt.plot(tm,yr,c='k',label='numerical')
plt.plot(tm,fa(tm),c='k',ls='--',label='analytical')
plt.grid()
plt.xlabel('t')
plt.ylabel('y')
plt.legend()
```

Numerical solution blows up!



parasite.py

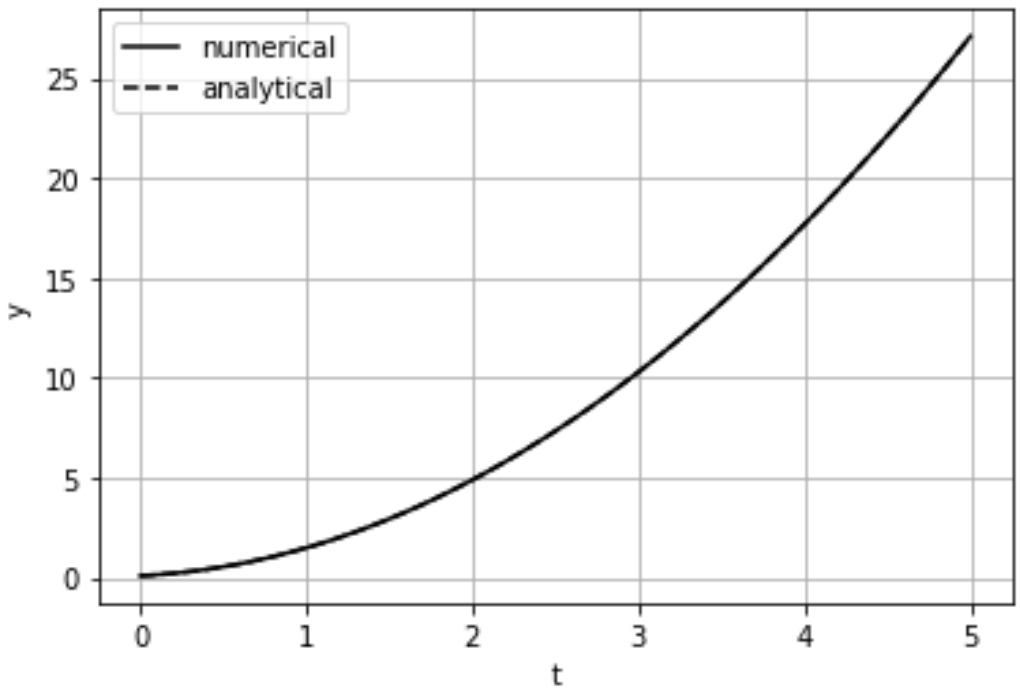
Solving Single Differential Equations

parasite1.py

Initial Value Problem

Example using **solve_ivp** with tightened tolerances

```
result = solve_ivp(f,tspan,y0,t_eval=teval \
                   , atol=1.e-12, rtol=1.e-12)
```



numerical and analytical solutions coincide

Solving Single Differential Equations

Single Equation Example – Isothermal Batch Reactor $A + B \overset{k}{\Rightarrow} C$

Rate of disappearance of A: $$\frac{dC_A}{dt} = -k \cdot C_A \cdot C_B$$

Initial conditions: $C_A(0) = C_{A0}$ $C_B(0) = C_{B0}$ $C_C(0) = C_{C0}$

Basic data: $k = 14.7 \dfrac{1}{mol/L} \cdot \dfrac{1}{min}$

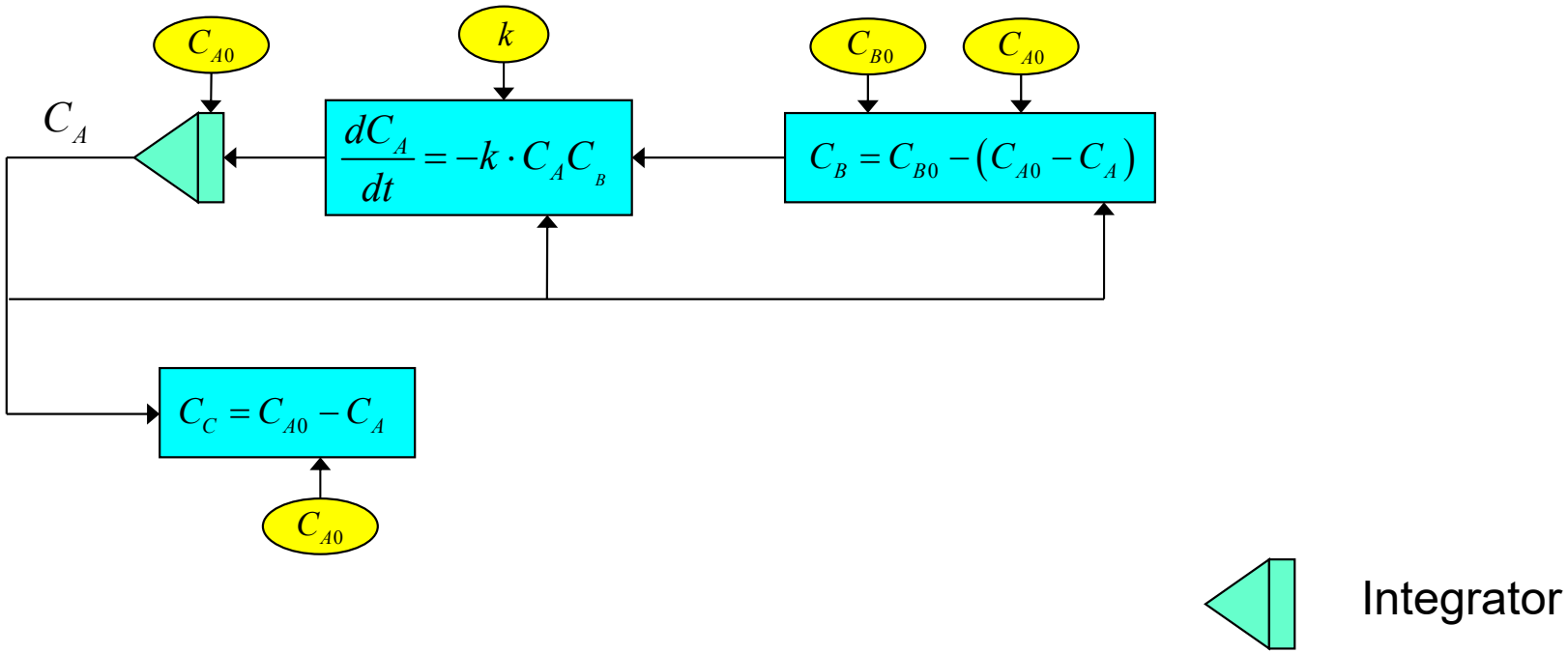
Initial conditions: $C_{A0} = 0.0209 \dfrac{mol}{L}$ $C_{B0} = C_{A0}/3$ $C_{C0} = 0$

Stoichiometric relationships: $$C_B(t) = C_{B0} - \left(C_{A0} - C_A(t)\right)$$
$$C_C(t) = C_{C0} + \left(C_{A0} - C_A(t)\right)$$

Solving Single Differential Equations

Single Equation Example – Isothermal Batch Reactor

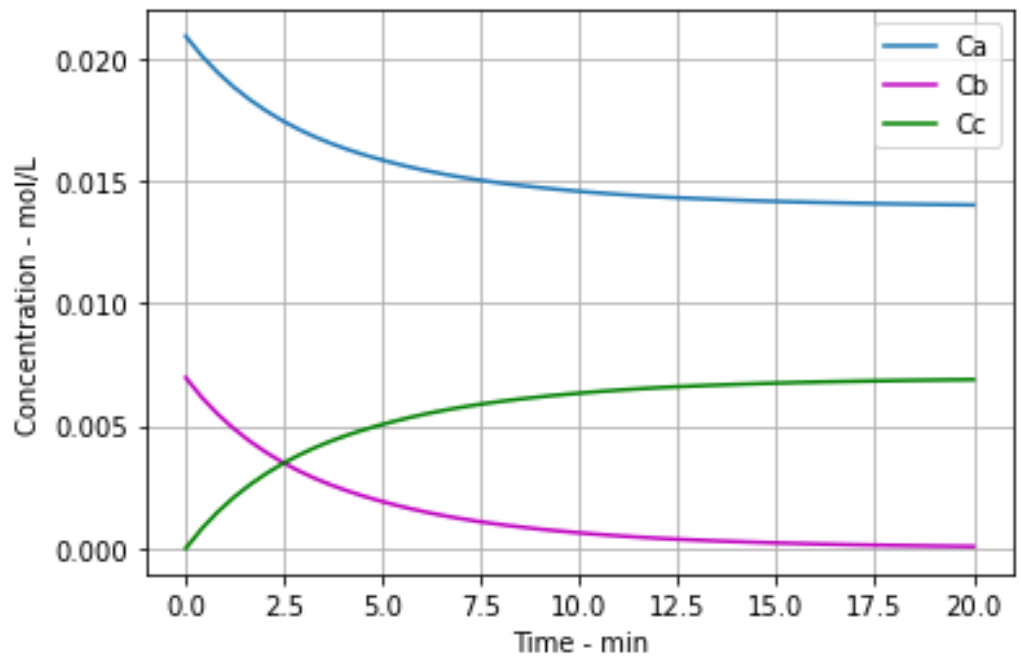
Information Flow Diagram



Solving Single Differential Equations

Single Equation Example – Isothermal Batch Reactor

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

k = 14.7
Ca0 = np.array([0.0209])
Cb0 = Ca0/3

def batch(t,Ca,k,Ca0,Cb0):
    Cb = Cb0 - (Ca0-Ca)
    return -k*Ca*Cb

tspan = [0., 20.]
teval = np.linspace(0., 20.)
result = solve_ivp(batch,tspan,Ca0,t_eval=teval,args=(k,Ca0,Cb0))

tm = result.t
Ca = result.y[0,:]
Cb = Cb0 - (Ca0-Ca)
Cc = Ca0 - Ca

plt.plot(tm,Ca,label='Ca')
plt.plot(tm,Cb,c='m',label='Cb')
plt.plot(tm,Cc,c='g',label='Cc')
plt.grid()
plt.xlabel('Time - min')
plt.ylabel('Concentration - mol/L')
plt.legend()
```



SimpleBatch.py

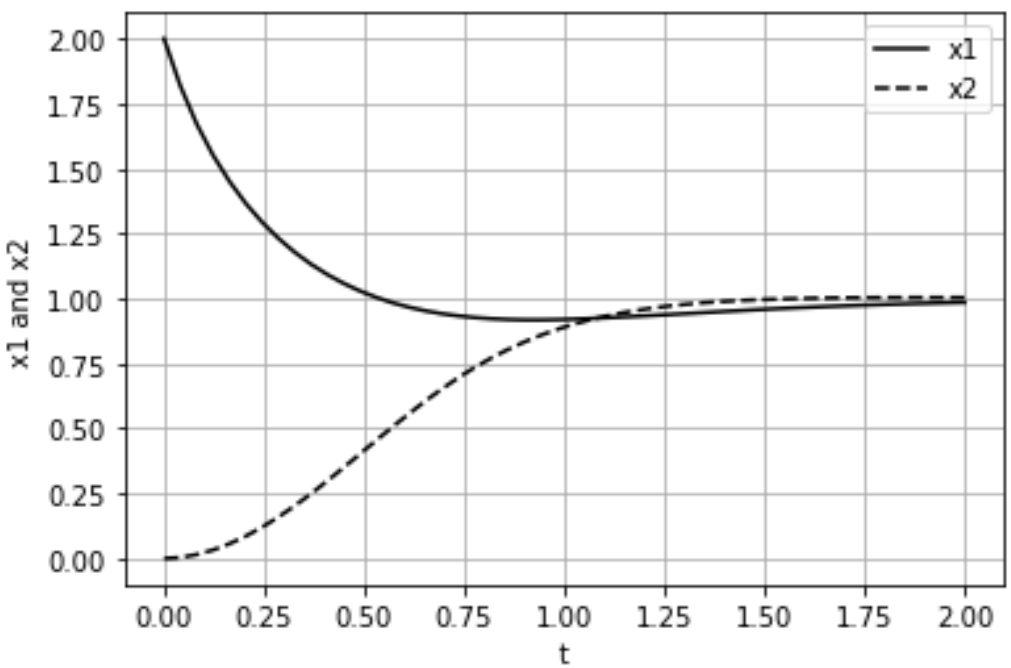
Solving Multiple Differential Equations

twoODEs.py

Example

$$\frac{dx_1}{dt} = -2x_1^2 + 2x_1 + x_2 - 1 \qquad x_1(0) = 2$$

$$\frac{dx_2}{dt} = -x_1 - 3x_2^2 + 2x_2 + 2 \qquad x_2(0) = 0$$




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

def f(t,y):
    dy = np.zeros(2)
    dy[0] = -2*y[0]**2 + 2*y[0] + y[1] -1
    dy[1] = -y[0] -3*y[1]**2 +2*y[1] + 2
    return dy

tf = 2.
tspan = [0., tf]
teval = np.linspace(0., tf)
y0 = np.array([2., 0.])
result = solve_ivp(f,tspan,y0,t_eval=teval)
tm = result.t
x1 = result.y[0,:]
x2 = result.y[1,:]

plt.plot(tm,x1,c='k',label='x1')
plt.plot(tm,x2,c='k',ls='--',label='x2')
plt.grid()
plt.xlabel('t')
plt.ylabel('x1 and x2')
plt.legend()
```

Solving Multiple Differential Equations

Multiple Equation Models – Isothermal Batch Reactor

$$A + B \xrightarrow{k_1} C + F$$
$$A + C \xrightarrow{k_2} D + F$$
$$A + D \xrightarrow{k_3} E + F$$

$$\frac{dA}{dt} = -k_1 AB - k_2 AC - k_3 AD \qquad A(0) = 0.0209\ \frac{mol}{L}$$

$$\frac{dB}{dt} = -k_1 AB \qquad B(0) = \frac{A(0)}{3}$$

$$\frac{dC}{dt} = k_1 AB - k_2 AC \qquad C(0) = 0$$

$$\frac{dD}{dt} = k_2 AC - k_3 AD \qquad D(0) = 0$$

$$k_1 = 14.7\ \frac{1}{mol/L} \cdot \frac{1}{min}$$

$$k_2 = 1.53\ \frac{1}{mol/L} \cdot \frac{1}{min}$$

$$k_3 = 0.294\ \frac{1}{mol/L} \cdot \frac{1}{min}$$

From stoichiometry: $E = \dfrac{A(0) - A - C - 2D}{3}$ and $F = A(0) - A$

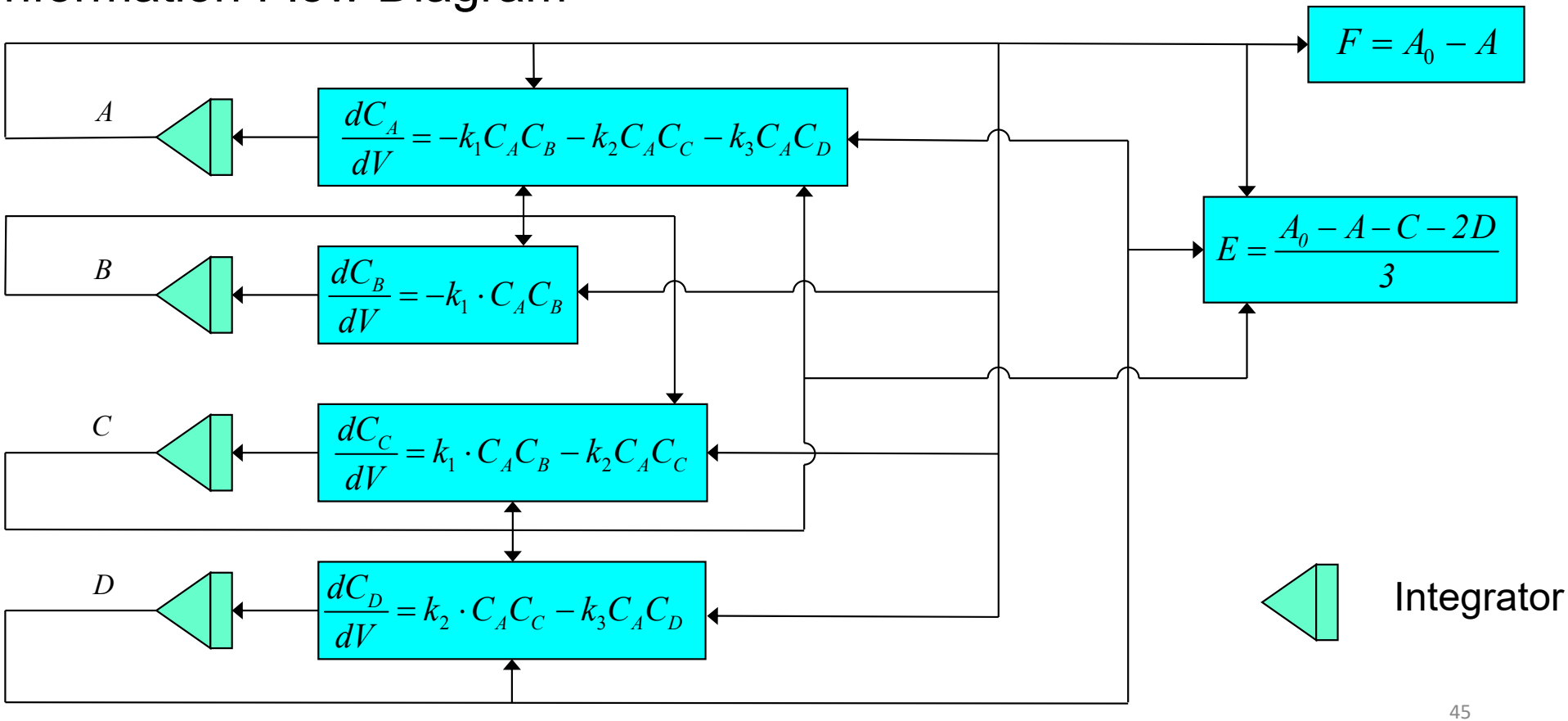
Svirbely, W.J., and J.A. Blauer, *The Kinetics of Three-step Competitive Consecutive Second-order Reactions*, **J. Amer. Chem. Soc., 83**, 4115, 1961.

Svirbely, W.J., and J.A. Blauer, *The Kinetics of the Alkaline Hydrolysis of 1,3,5,Tricarbomethoxybenzene*, **J. Amer. Chem. Soc., 83**, 4118, 1961.

Solving Multiple Differential Equations

Multiple Equation Models – Isothermal Batch Reactor

Information Flow Diagram



Solving Multiple Differential Equations

Multiple Equation Models – Isothermal Batch Reactor

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

def multibatch(t,y,k1,k2,k3):
    A = y[0] ; B = y[1] ; C = y[2] ; D = y[3]
    dA = -k1*A*B - k2*A*C - k3*A*D
    dB = -k1*A*B
    dC = k1*A*B - k2*A*C
    dD = k2*A*C - k3*A*D
    return [dA, dB, dC, dD]
```

additional arguments

unpack y into familiar variables

compute derivatives

Solving Multiple Differential Equations

Multiple Equation Models – Isothermal Batch Reactor

```
k1 = 14.7
k2 = 1.53
k3 = 0.294
A0 = 0.0209
B0 = A0/3

tspan = [ 0., 500.]
teval = np.linspace(0.,500.,100)
y0 = [A0, B0, 0, 0]

result = solve_ivp(multibatch,tspan,y0,t_eval=teval,args=(k1,k2,k3))

tm = result.t
A = result.y[0,:]
B = result.y[1,:]
C = result.y[2,:]
D = result.y[3,:]

E = (A0 - A - C - 2*D)/3
F = A0 - A
```

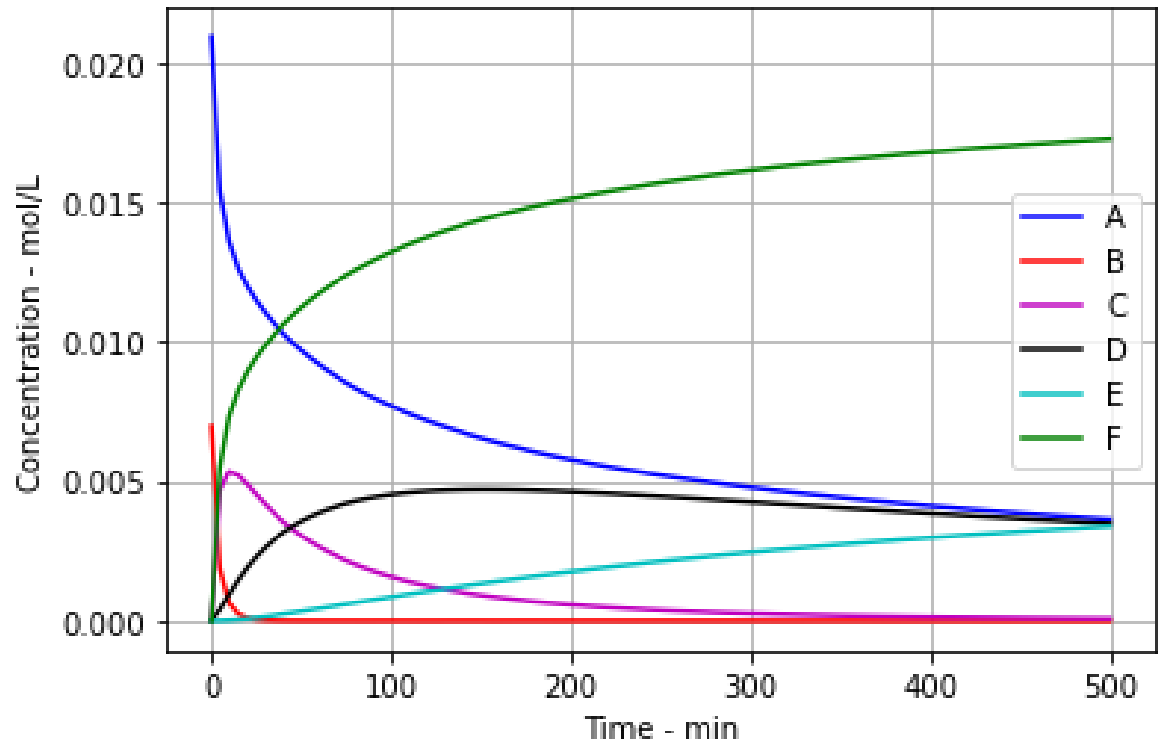
unpack results

compute E and F from A, C and D

Solving Multiple Differential Equations

Multiple Equation Models – Isothermal Batch Reactor

```
plt.plot(tm,A,c='b',label='A')
plt.plot(tm,B,c='r',label='B')
plt.plot(tm,C,c='m',label='C')
plt.plot(tm,D,c='k',label='D')
plt.plot(tm,E,c='c',label='E')
plt.plot(tm,F,c='g',label='F')
plt.grid()
plt.xlabel('Time - min')
plt.ylabel('Concentration - mol/L')
plt.legend()
```



multibatch_reactor.py

Solving Ordinary Differential Equations

Second-order differential equation with split boundary conditions

$$\frac{d^2y}{dt^2} = \frac{1}{4}\frac{dy}{dt} + y \qquad y(0) = 5 \qquad y(10) = 8 \qquad 0 \le t \le 10$$

Decompose into two first-order ODEs

$$\frac{dy}{dt} = y_1 \qquad y(0) = 5 \qquad y(10) = 8$$

$$\frac{dy_1}{dt} = \frac{1}{4}y_1 + y$$

"Shooting" Strategy

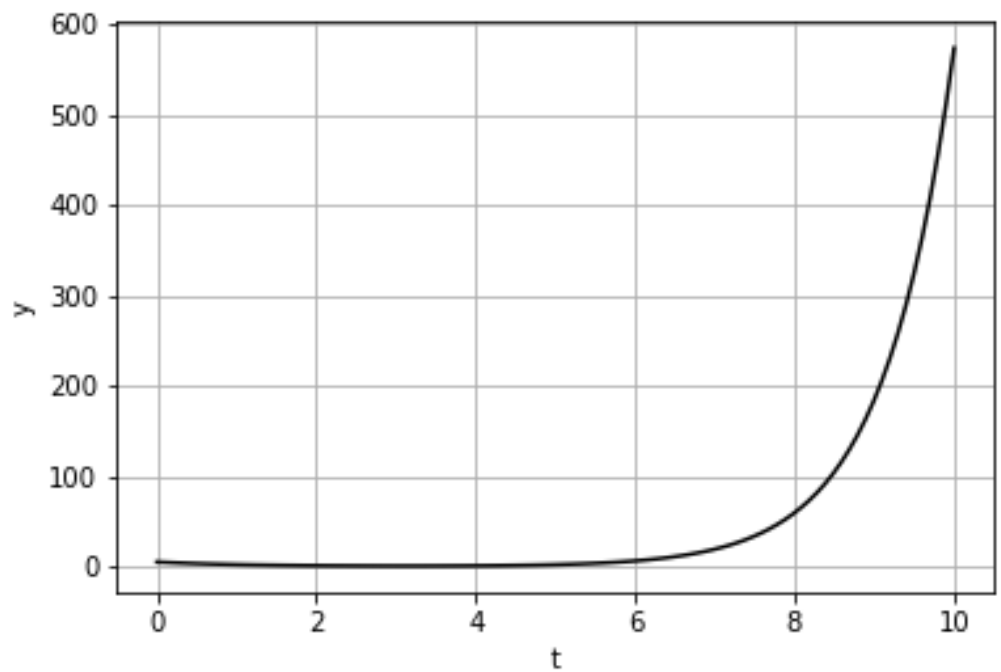
1. Estimate a value for y1 (dy/dt) at t = 0.
2. Solve the ODEs to t = 10
3. Check y(10) versus the required value, 8.
4. Adjust the y1(0) value and repeat steps 2 and 3 until the desired y(10)=8 value is obtained.

Solving Ordinary Differential Equations

Second-order differential equation with split boundary conditions

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

def diffeqs(t,y):
    dy1 = y[1]
    dy2 = y[1]/4. + y[0]
    return [ dy1, dy2]

y0 = [5., -4.4]
tspan = [ 0., 10.]
teval = np.linspace(0.,10.,100)

result = solve_ivp(diffeqs,tspan,y0,t_eval=teval)

tm = result.t
y1 = result.y[0,:]
y2 = result.y[1,:]

plt.plot(tm,y1,c='k')
plt.grid()
plt.xlabel('t')
plt.ylabel('y')
```

TwoPointBC.py



Equations solved with an estimate for y1(0). Clearly doesn't meet the required final condition y(10)=8.

Solving Ordinary Differential Equations

Second-order differential equation with split boundary conditions

Employ **brentq** to satisfy the final boundary condition

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import solve_ivp
from scipy.optimize import brentq

def diffeqs(t,y):
    dy1 = y[1]
    dy2 = y[1]/4. + y[0]
    return [ dy1, dy2]

def splitboundary(y01):
    y0 = [5., y01]
    tspan = [ 0., 10.]
    teval = np.linspace(0.,10.,100)
    result = solve_ivp(diffeqs,tspan,y0,t_eval=teval)
    tm = result.t
    n = len(tm)
    return result.y[0,n-1]-8.
```

```
y01 = -4.4
y1_soln = brentq(splitboundary,-5.,-4.)
print('initial derivative value =',y1_soln)

y0 = [5., y1_soln]
tspan = [ 0., 10.]
teval = np.linspace(0.,10.,100)
result = solve_ivp(diffeqs,tspan,y0,t_eval=teval)
tm = result.t
y1 = result.y[0,:]

plt.plot(tm,y1,c='k')
plt.grid()
plt.xlabel('t')
plt.ylabel('y')
```

TwoPointBC1.py

Solving Ordinary Differential Equations

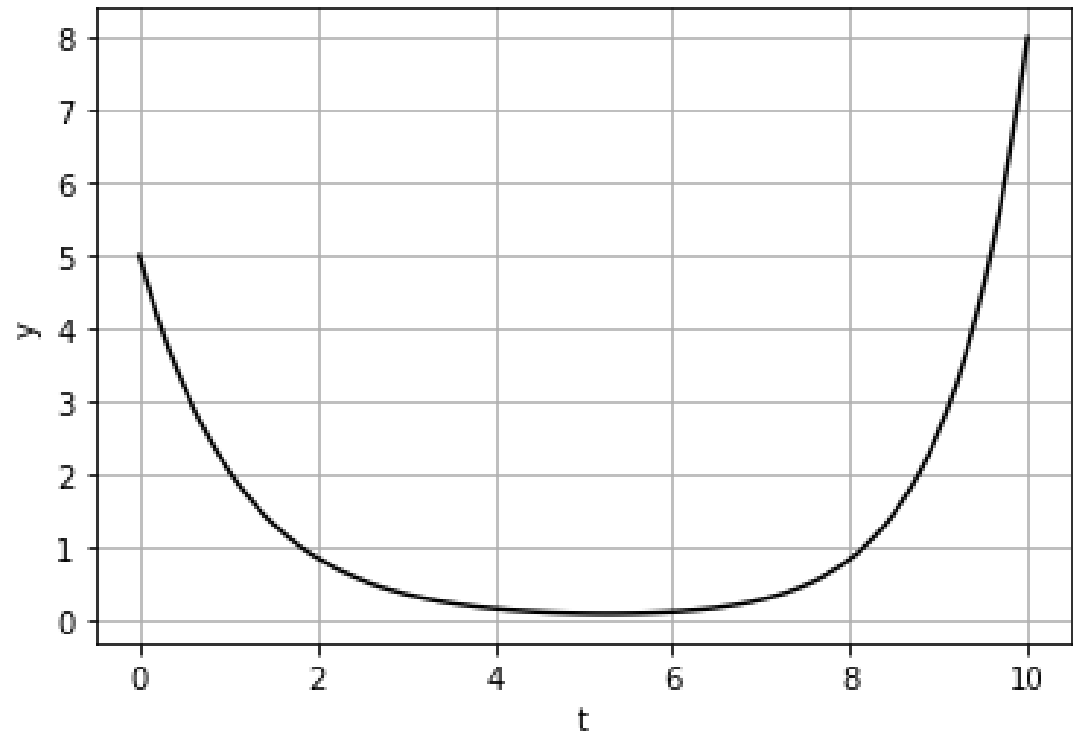
Second-order differential equation with split boundary conditions

Employ **brentq** to satisfy the final boundary condition

Final condition, y = 8.,
is now met.

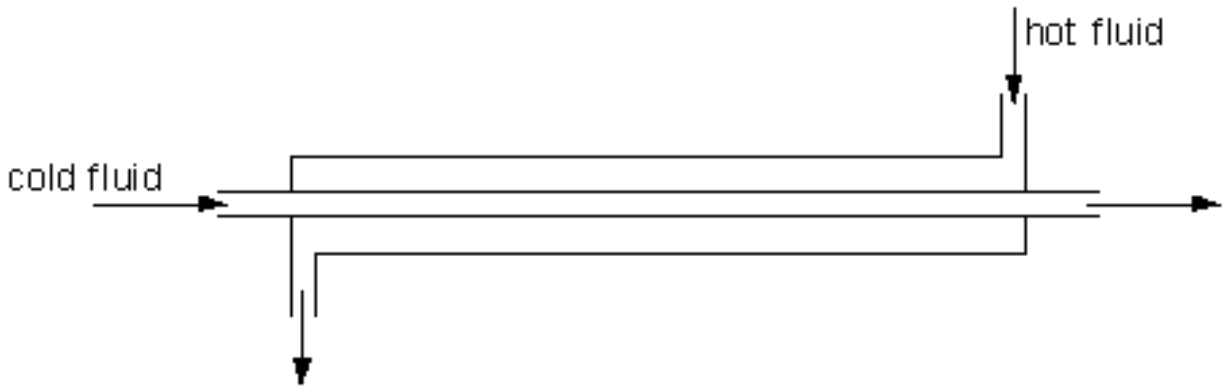
```
initial derivative value = -4.413717066724333
```

Final condition is <u>very</u> sensitive
to the initial derivative value.



Solving Ordinary Differential Equations

Example: tube-in-tube, countercurrent heat exchanger



$$\frac{dT_c}{dz} = \frac{h_i A_i}{w_c C_c}\left(T_h - T_c\right) \qquad T_c\left(0\right) = T_{ci}$$

$$\frac{dT_h}{dz} = \frac{h_o A_o}{w_h C_h}\left(T_h - T_c\right) \qquad T_h\left(L\right) = T_{hi}$$

$$h_o = \frac{h_i \cdot D_i}{D_o}$$

Solving Ordinary Differential Equations

Example: tube-in-tube, countercurrent heat exchanger

z: distance down the heat exchanger from the cold fluid inlet (on the left)

L: length of the heat exchanger

T_c: temperature of the cold fluid, a function of z

T_{ci}: cold water inlet temperature, at $z=0$

T_{hi}: hot water inlet temperature, at $z=L$

T_h: temperature of the hot fluid, a function of z

w_c: mass flow rate of cold fluid

w_h: mass flow rate of hot fluid

C_c: heat capacity of cold fluid

C_h: heat capacity of hot fluid

A_i: inside area for heat transfer (cold fluid) per unit length

A_o: outside area for heat transfer (hot fluid) per unit length

h_i: inside heat transfer coefficient (cold fluid)

h_o: outside heat transfer coefficient (hot fluid)

Solving Ordinary Differential Equations

Example: tube-in-tube, countercurrent heat exchanger

$$\frac{dT_c}{dz} = \frac{h_i A_i}{w_c C_c}\left(T_h - T_c\right) \qquad T_c\left(0\right) = T_{ci}$$

$$\frac{dT_h}{dz} = \frac{h_o A_o}{w_h C_h}\left(T_h - T_c\right) \qquad T_h\left(L\right) = T_{hi}$$

The issue we have with solving these equations is that the cold stream boundary condition is at z = 0 and the hot stream boundary condition is at z = L, the other end of the heat exchanger. A practical way to handle this is to estimate the hot stream temperature at z = 0, proceed with the solution, and adjust that estimate later on to meet the condition at z = L.

Solving Ordinary Differential Equations

Example: tube-in-tube, countercurrent heat exchanger

Basic data and operating conditions

Outer tube	Inner tube	
11 BWG	11 BWG	Length 5 m
OD 2 in, ID 1.76 in	OD 1 in, ID 0.76 in	

Inlet temperatures	Fluid density (H_2O)	Heat capacity (H_2O)
Hot stream 50 °C	988 kg/m^3	4187 J/(kg·°C)
Cold stream 10 °C		

Hot stream flow rate 1 L/s	Heat transfer coefficient
Cold stream 0.3 L/s	h_i = 14,000 W/(m^2·°C)

Solving Ordinary Differential Equations

Example: tube-in-tube, countercurrent heat exchanger

countercurrent_heatexchanger1.py

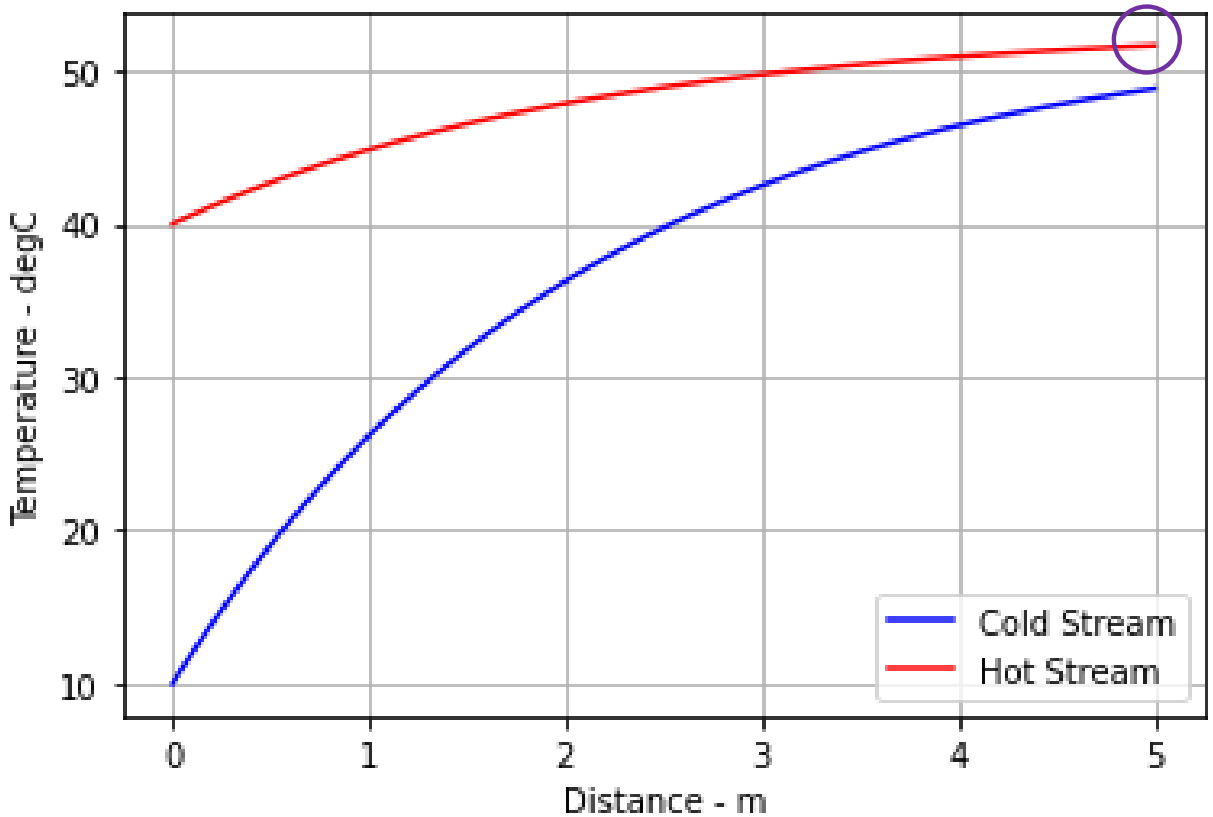
```
import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import solve_ivp

def htxr(z,T,hi,ho,Ai,Ao,wc,wh,cP):
    Tc = T[0]
    Th = T[1]
    dTc = hi*Ai/wc/cP*(Th-Tc)
    dTh = ho*Ao/wh/cP*(Th-Tc)
    return [dTc, dTh]

doin = 1.  # in
do = doin * 0.0254  # m
Ao = np.pi*do  # m2/m
diin = 0.76  # in
di = diin * 0.0254  # m
Ai = np.pi*di  # m2/m
L = 5.  # m

den = 998.  # kg/m3
cP = 4187.  # J/(kg*degC)

hi = 14000.  # W/(m2*degC)
ho = hi*di/do
```

```
qcL = 0.3  # L/s
qc = qcL/1000.  # m3/s
wc = qc * den  # kg/s
qhL = 1.  # L/s
qh = qhL/1000.  # m3/s
wh = qh * den  # kg/s

Tci = 10  # degC
Thi = 50  # degC

zspan = [0., L]
zeval = np.linspace(0.,L,100)

Tho = 40  #  degC

T0 = [ Tci, Tho]

result = solve_ivp(htxr,zspan,T0,t_eval=zeval \
                   ,args=(hi,ho,Ai,Ao,wc,wh,cP))

zs = result.t
Tc = result.y[0,:]
Th = result.y[1,:]

plt.plot(zs,Tc,c='b',label='Cold Stream')
plt.plot(zs,Th,c='r',label='Hot Stream')
plt.grid()
plt.xlabel('Distance - m')
plt.ylabel('Temperature - degC')
plt.legend()
```

Solve the model first with an estimate for the hot stream outlet temperature.

Solving Ordinary Differential Equations

Example: tube-in-tube, countercurrent heat exchanger



Hot stream inlet condition, 50°C, not met.

Solving Ordinary Differential Equations

Example: tube-in-tube, countercurrent heat exchanger

Use the **brentq** function to adjust the hot stream outlet temperature until the inlet hot stream condition is met.

countercurrent_heatexchanger2.py

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import solve_ivp
from scipy.optimize import brentq

def htxr(z,T,hi,ho,Ai,Ao,wc,wh,cP):
    Tc = T[0]
    Th = T[1]
    dTc = hi*Ai/wc/cP*(Th-Tc)
    dTh = ho*Ao/wh/cP*(Th-Tc)
    return [dTc, dTh]

def findTho(Tho,Thi_spec):
    zspan = [0., L]
    T0 = [ Tci, Tho]
    result = solve_ivp(htxr,zspan,T0 \
                       ,args=(hi,ho,Ai,Ao,wc,wh,cP))
    zs = result.t
    n = len(zs)
    return result.y[1,n-1] - Thi_spec
```

Solving Ordinary Differential Equations

Example: tube-in-tube, countercurrent heat exchanger

```
doin = 1.  # in
do = doin * 0.0254  # m
Ao = np.pi*do  # m2/m
diin = 0.76  # in
di = diin * 0.0254  # m
Ai = np.pi*di  # m2/m
L = 5.  # m

den = 998.  # kg/m3
cP = 4187.  # J/(kg*degC)

hi = 14000.  # W/(m2*degC)
ho = hi*di/do

qcL = 0.3  # L/s
qc = qcL/1000.  # m3/s
wc = qc * den  # kg/s
qhL = 1.  # L/s
qh = qhL/1000.  # m3/s
wh = qh * den  # kg/s
```

Solving Ordinary Differential Equations

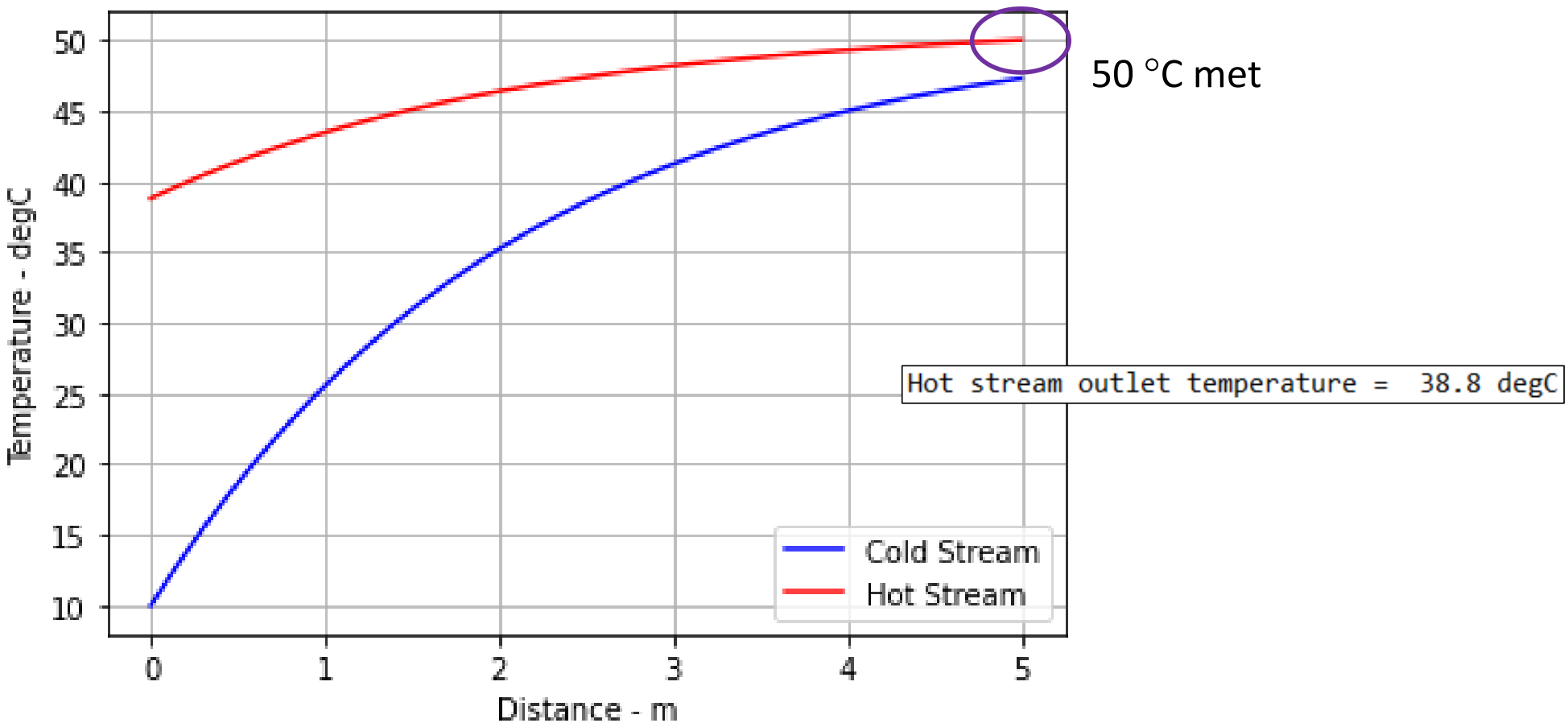
Example: tube-in-tube, countercurrent heat exchanger

```
Tci = 10  # degC
Thi_spec = 50  # degC

Tho_soln = brentq(findTho,35.,45.,args=(Thi_spec))

zspan = [0., L]
zeval = np.linspace(0.,L,100)
T0 = [Tci, Tho_soln]
result = solve_ivp(htxr,zspan,T0,t_eval=zeval \
                   ,args=(hi,ho,Ai,Ao,wc,wh,cP))
zs = result.t
Tc = result.y[0,:]
Th = result.y[1,:]

print('Hot stream outlet temperature = {0:5.1f} degC'.format(Tho_soln))

plt.plot(zs,Tc,c='b',label='Cold Stream')
plt.plot(zs,Th,c='r',label='Hot Stream')
plt.grid()
plt.xlabel('Distance - m')
plt.ylabel('Temperature - degC')
plt.legend()
```

brentq adjusts **Tho** until **Thi** computed meets the spec, **Thi_spec**

Solving Ordinary Differential Equations

Example: tube-in-tube, countercurrent heat exchanger

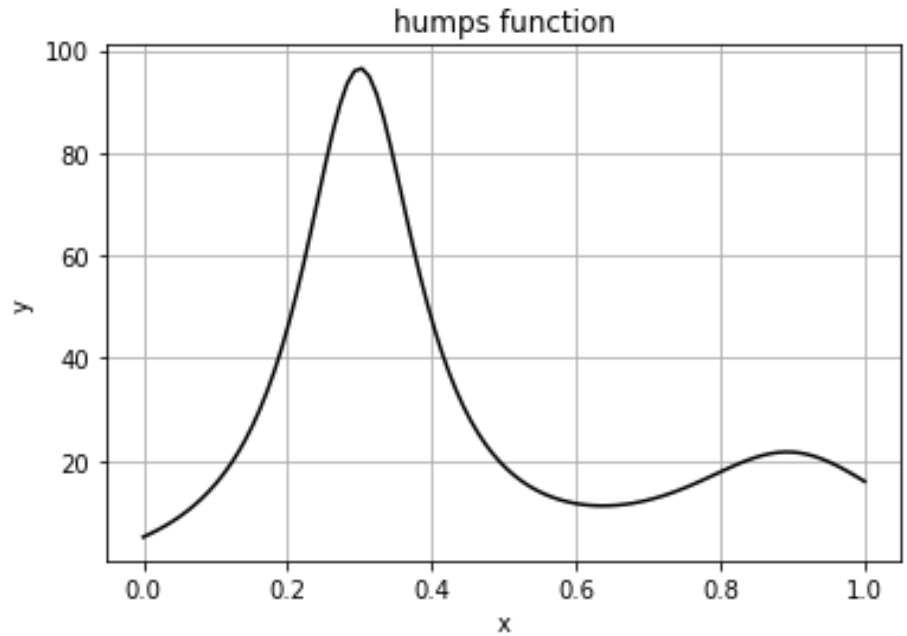


Optimization

Finding a maximum or minimum of a function with a single adjustable variable

Example $$y = \frac{1}{(x-0.3)^2+0.01} + \frac{1}{(x-0.9)^2+0.04} - 6$$

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

def humps(x):
    y = 1./((x-0.3)**2+0.01) + 1./((x-0.9)**2+0.04) - 6.
    return y

x = np.linspace(0.,1.,100)
plt.plot(x,humps(x),c='k')
plt.grid()
plt.xlabel('x')
plt.ylabel('y')
plt.title('humps function')
```



plothumps.py

Optimization

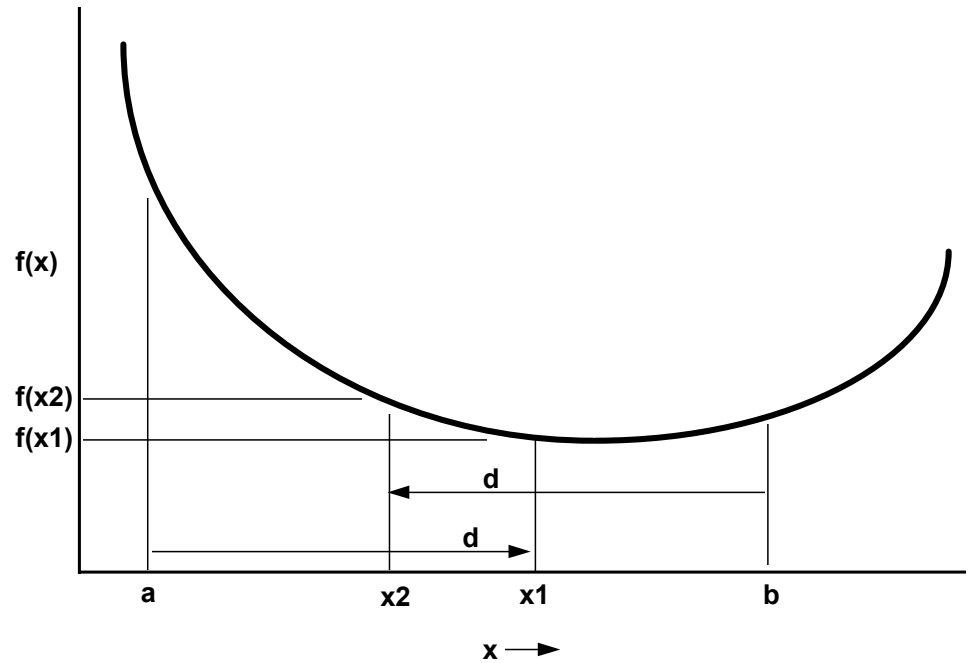
Finding a maximum or minimum of a function with a single adjustable variable - the Golden Section search

This is a bracketing method, similar to bisection. The figure shows a curve *f(x)* with a minimum between two initial estimates, *a* and *b*. Instead of using the midpoint between *a* and *b*, an overlapping interval *d* is used to compute x_1 and x_2. The interval is given by

$$d = \frac{\sqrt{5}-1}{2} \cdot (b-a)$$

where $\frac{\sqrt{5}-1}{2}$ is the Golden Ratio (*GR*)

with the unique property $GR = \frac{1}{1+GR}$



Optimization

Finding a maximum or minimum of a function with a single adjustable variable - the Golden Section search

For the figure to the right, we can see that

$$f(x_2) > f(x_1)$$

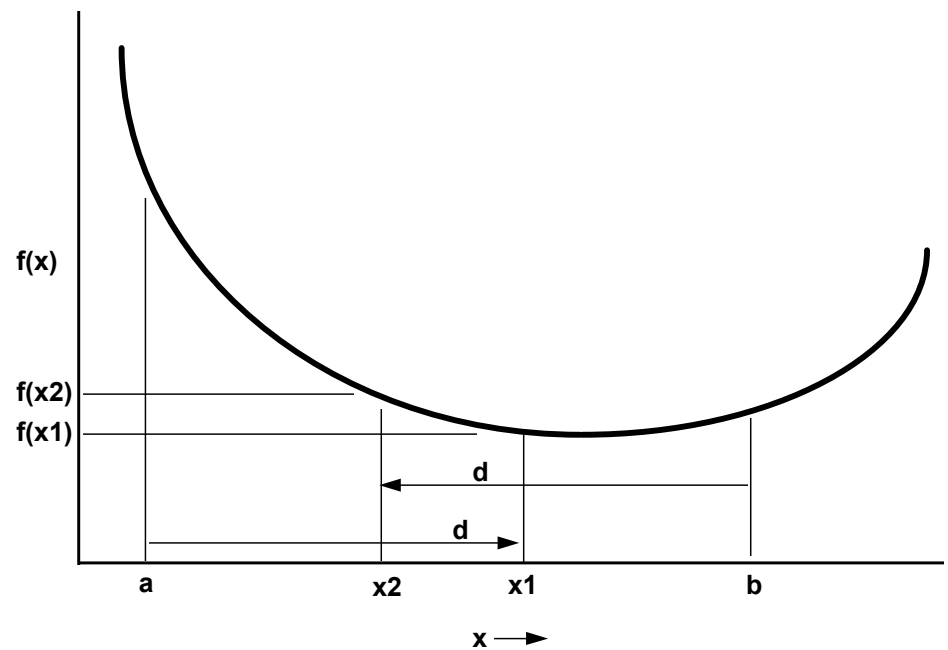
and that leads to the conclusion that the minimum must be between x_2 and b, and the interval $[a,x_2]$ can be excluded. This implies that x_2 becomes the new a for the next iteration of the method.

If

$$f(x_1) > f(x_2)$$

the interval $[x_1,b]$ would be excluded and x_1 would become the next b.

For each iteration, the interval containing the minimum is reduced by a factor of *GR*.



Any ratio other than *GR* that is greater than 0.5 and less than 1 could be used. The advantage of using the *GR* is that, for the first scenario above, x_1 is the x_2 value for the next iteration, avoiding the need to compute $f(x_2)$ then.

Optimization

Python function to implement Golden Section search: **goldmin_1.py**

```
def mingold(f,a,b,maxit=30):
    GR = (np.sqrt(5)-1)/2
    d = GR*(b-a)
    x1 = a + d
    x2 = b - d
    f1 = f(x1)
    f2 = f(x2)
    for i in range(maxit):
        d = GR*d
        if f2 > f1:
            a = x2
            x2 = x1
            f2 = f1
            x1 = a + d
            f1 = f(x1)
        else:
            b = x1
            x1 = x2
            f1 = f2
            x2 = b - d
            f2 = f(x2)
    return (x1+x2)/2
```

Additional script to solve for minimum of humps function:

```
def humps(x):
    hmps = 1/((x-0.3)**2+0.01)+1/((x-0.9)**2+0.04)-6
    return hmps

a = 0.5
b = 0.8
xmin = mingold(humps,a,b)
print('minimum x = {0:6.4f}'.format(xmin))
```

```
minimum x = 0.6370
```

Because of the overlap coincidence, only one function evaluation is needed for each iteration.
The *d* value is reduced by *GR* each iteration. For 30 iterations, the original *b* - *a* is reduced by $GR^{30} \cong 5\text{x}10^{-7}$.

Optimization

Finding the minimum of a function with a single adjustable variable

Using the **minimize_scalar** function from the SciPy **optimize** submodule

```
from scipy.optimize import minimize_scalar

def humps(x):
    y = 1./((x-0.3)**2+0.01) + 1./((x-0.9)**2+0.04) - 6.
    return y

result = minimize_scalar(humps)
print(result)
```

```
message:
         Optimization terminated successfully;
         The returned value satisfies the termination criteria
         (using xtol = 1.48e-08 )
success: True
    fun: -6.0
      x: -111097479.90657699
    nit: 35
   nfev: 110
```

determines a minimum well out of range expected for x

```
from scipy.optimize import minimize_scalar

def humps(x):
    y = 1./((x-0.3)**2+0.01) + 1./((x-0.9)**2+0.04) - 6.
    return y

result = minimize_scalar(humps,bounds=(0.4,0.8))
print(result)
```

the interval is bounded

```
  message: Solution found.
  success: True
   status: 0
      fun: 11.252754126374835
        x: 0.6370104788470293
      nit: 9
     nfev: 9
```

the local minimum is determined

minimize_humps.py

Optimization

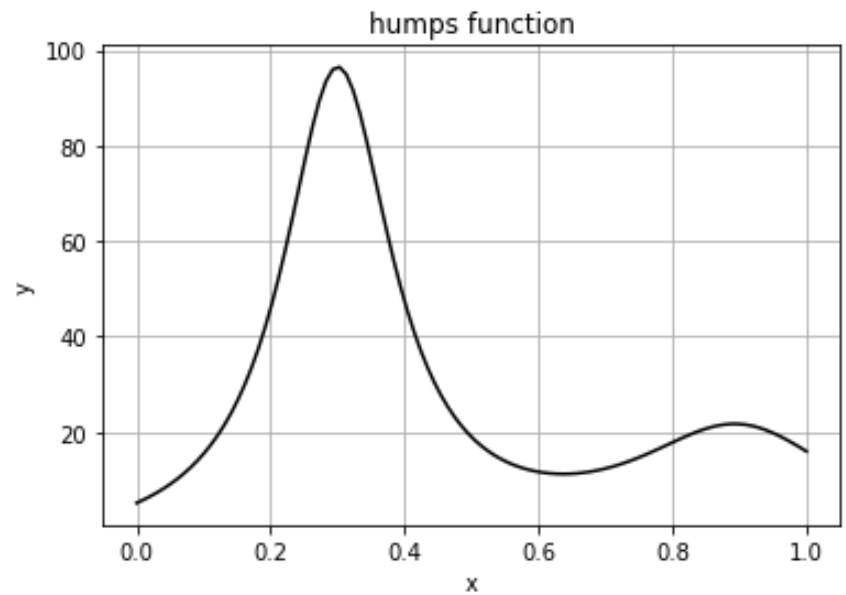
Finding the maximum of a function with a single adjustable variable

Negate the function to determine a maximum

maximize_humps.py

```
from scipy.optimize import minimize_scalar

def humps(x):
    y = 1./((x-0.3)**2+0.01) + 1./((x-0.9)**2+0.04) - 6.
    return -y

result = minimize_scalar(humps,bounds=(0.0,1.0))
print(result)
```

```
 message: Solution found.
 success: True
  status: 0
     fun: -96.50140855598092
       x: 0.30037628482912704
     nit: 11
    nfev: 11
```



Could restrict bounds to 0.8 – 1.0 to find local maximum there.

Optimization

Finding a maximum or minimum of a function with multiple adjustable variables and one or more constraints

Using **minimize** function from SciPy **optimize** submodule

result = minimize(f,x0)

f : function of x, vector of variables
x0 : initial estimates of x values

options include different algorithms, bounds, and constraints

Example: $f(x_1, x_2) = 2 + x_1 - x_2 + 2x_1^2 + 2x_1x_2 + x_2^2$ $\quad -2 \le x_1 \le 0 \quad 0 \le x_2 \le 3$

```
from scipy.optimize import minimize

def f(x):
    x1 = x[0]
    x2 = x[1]
    return 2+x1-x2+2*x1**2+2*x1*x2+x2**2

x0 = [-0.5, 0.5]

result = minimize(f,x0,bounds=((-2, 0),(0, 3)))
xmin = result.x
print(xmin)
```

minimize_example.py

```
[-1.00000001  1.50000003]
```

Optimization

Finding a maximum or minimum of a function with multiple adjustable variables and one or more constraints

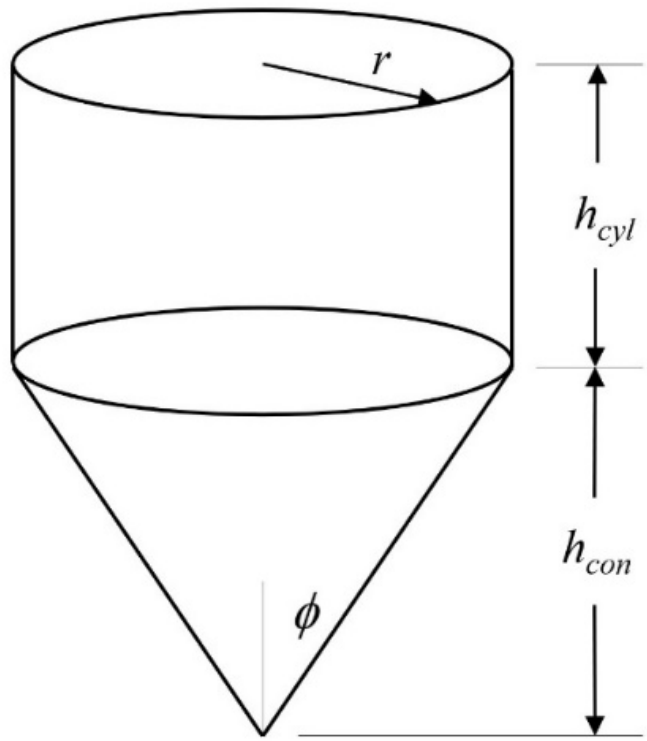
Example:
Optimal grain bin design

Minimize surface area not including the top

Constraints

$$V = V_{cyl} + V_{con} = 10 \text{ m}^3$$

$$\phi_{\max} = 20.4°$$



$$V_{cyl} = \pi r^2 h_{cyl}$$

$$S_{cyl} = 2\pi r h_{cyl}$$

$$V_{con} = \frac{1}{3}\pi r^2 h_{con}$$

$$S_{con} = \pi r \sqrt{r^2 + h_{con}^2}$$

$$\phi = \tan^{-1}\left(\frac{r}{h_{con}}\right)$$

Optimization

Example: Optimal grain bin design

Function definitions

grainbin.py

```python
import numpy as np
from scipy.optimize import minimize

def S(x):  # bin surface area
    r = x[0]
    hcyl = x[1]
    hcon = x[2]
    Scyl = 2*np.pi*r*hcyl
    Scon = np.pi*r*np.sqrt(r**2+hcon**2)
    return Scyl+Scon

def V(x):  # bin volume
    r = x[0]
    hcyl = x[1]
    hcon = x[2]
    Vcyl = np.pi*r**2*hcyl
    Vcon = np.pi*r**2*hcon/3
    return Vcyl+Vcon
```

```python
def VolCon(x):  # volume equality constraint
    return V(x)-Volspec

def AngCon(x):  # cone angle inequality constraint
    r = x[0]
    hcon = x[2]
    Angrad = np.arctan(r/hcon)
    Angdeg = np.rad2deg(Angrad)
    return Angspec - Angdeg  # must be >= 0
```

Optimization

Example: Optimal grain bin design

Main script

```
Volspec = 10  # m
Angspec = 20.4  # deg
x0 = [1., 1., 1.]  # m
con = ({'type':'eq','fun':VolCon},
       {'type':'ineq','fun':AngCon})

result = minimize(S,x0,constraints=con)

r = result.x[0]
hcyl = result.x[1]
hcon = result.x[2]
print('radius = {0:5.2f} m'.format(r))
print('cylinder height = {0:5.2f} m'.format(hcyl))
print('cone height = {0:5.2f} m'.format(hcon))
phi = np.arctan(r/hcon)
phid = np.rad2deg(phi)
print('cone angle = {0:5.1f} deg'.format(phid))
print('total surface area = {0:6.1f} m2'.format(S(result.x)))
print('bin volume = {0:5.1f} m3'.format(V(result.x)))
```

Results

```
radius =  1.44 m
cylinder height =  0.26 m
cone height =  3.86 m
cone angle =  20.4 deg
total surface area =   20.9 m2
bin volume =  10.0 m3
```

Most of the bin is the conical part

Angle is at the constraint
Volume constraint met

Curve-Fitting

Polynomial regression

coeff = polyfit(x,y,order)

y = polyval(coeff,x)

Example:

Density of Methanol-Water Solutions at 20 degC	
Wt% Methanol	Density (kg/m^3)
0	998.2
10	981.5
20	966.6
30	951.5
40	934.5
50	915.6
60	894.6
70	871.5
80	846.9
90	820.2
100	791.7

polynomialfit.py

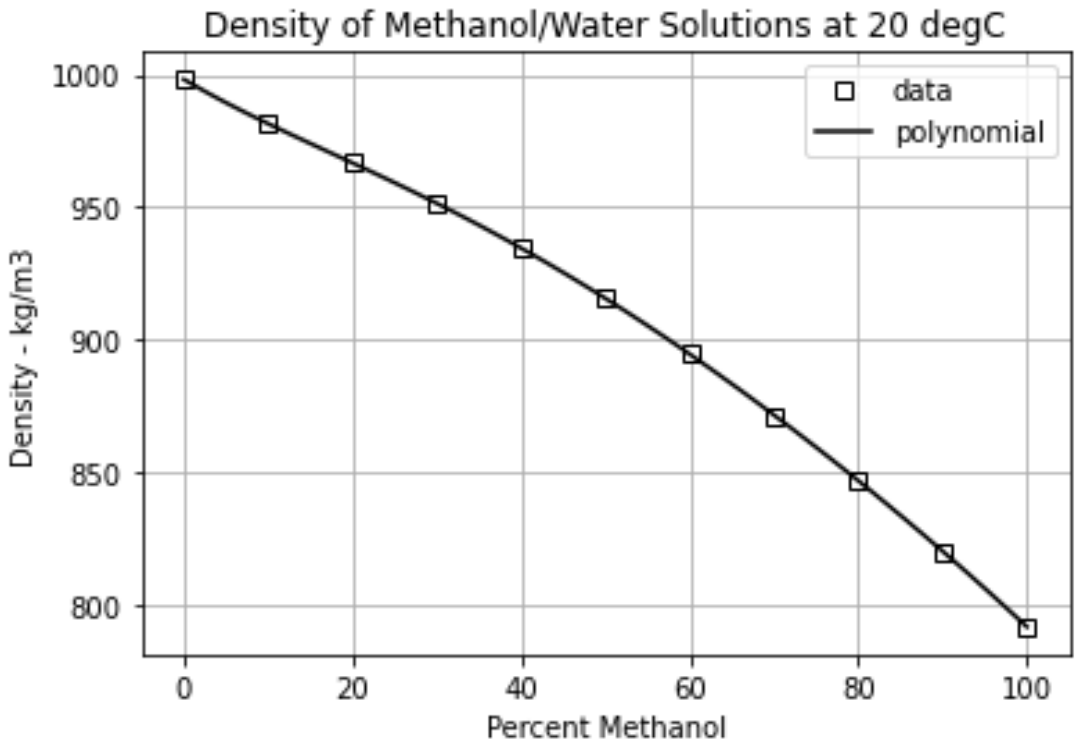
```
import numpy as np
import matplotlib.pyplot as plt

pct = np.arange(0,110,10)
Den = np.array([998.2, 981.5, 966.6, 951.5, 934.5,
                915.6, 894.6, 871.5, 846.9, 820.2, 791.7])

coeff = np.polyfit(pct,Den,6)
print(coeff)

pctp = np.linspace(0,100,100)
DenP = np.polyval(coeff,pctp)

plt.plot(pct,Den,c='w',marker='s',mec='k',mfc='w',label='data')
plt.plot(pctp,DenP,c='k',label='polynomial')
plt.grid()
plt.xlabel('Percent Methanol')
plt.ylabel('Density - kg/m3')
plt.legend()
plt.title('Density of Methanol/Water Solutions at 20 degC')
```

Curve-Fitting

Polynomial regression

Coefficients from high to low order

```
[ 1.07843137e-10 -5.09426848e-08  9.31033183e-06 -8.37157206e-04
  2.90083656e-02 -1.88888510e+00  9.98205759e+02]
```



Curve-Fitting

Multilinear regression

Model $$y = \beta_0 + \beta_1 f_1\left(x_j, j = 1,\ldots,m\right) + \beta_2 f_2\left(x_j, j = 1,\ldots,m\right) + \cdots + \beta_k f_k\left(x_j, j = 1,\ldots,m\right)$$

measurement responses

$$\mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix}$$

realization matrix of input levels

$$\mathbf{X} = \begin{bmatrix} 1 & f_{11} & f_{21} & \cdots & f_{m1} \\ 1 & f_{12} & f_{22} & \cdots & f_{m2} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & f_{1n} & f_{2n} & \cdots & f_{mn} \end{bmatrix}$$

intercept

parameter estimates

$$\hat{\boldsymbol{\beta}} = \begin{bmatrix} \hat{\beta}_1 \\ \hat{\beta}_2 \\ \vdots \\ \hat{\beta}_k \end{bmatrix}$$

model predictions

$$\hat{\mathbf{y}} = \mathbf{X}\hat{\boldsymbol{\beta}}$$

residuals

$$\mathbf{e} = \begin{bmatrix} e_1 \\ e_2 \\ \vdots \\ e_n \end{bmatrix} \qquad \mathbf{e} = \mathbf{y} - \hat{\mathbf{y}}$$

sum of squares criterion

$$V = \sum_{i=1}^{n} e_i^2 = \mathbf{e}^T \cdot \mathbf{e}$$

Dataset

$$\left\{y_i, x_{1i}, \ldots, x_{mi}, i = 1,\ldots,n\right\}$$

Minimize V by choice of $\hat{\boldsymbol{\beta}}$ via calculus

Normal equations

$$\left(\mathbf{X}^T\mathbf{X}\right)\mathbf{b} = \mathbf{X}^T\mathbf{y}$$

Fitted model parameters

$$\mathbf{b} = \left(\mathbf{X}^T\mathbf{X}\right)^{-1}\mathbf{X}^T\mathbf{y}$$

Curve-Fitting

Multilinear regression

Example

Density of NaCl Aqueous Solutions

		Temperature			
		0 °C	10 °C	25 °C	40 °C
Wt % NaCl	1	1.00747	1.00707	1.00409	0.99908
	2	1.01509	1.01442	1.01112	1.00593
	4	1.03038	1.02920	1.02530	1.01977
	8	1.06121	1.05907	1.05412	1.04798
	12	1.09244	1.08946	1.08365	1.07699
	16	1.12419	1.12056	1.11401	1.10688
	20	1.15663	1.15254	1.14533	1.13774
	24	1.18999	1.18557	1.17776	1.16971
	26	1.20709	1.20254	1.19443	1.18614

from *Perry's Chemical Engineer's Handbook*, Green and Southard, Ed., 9th Ed., p. 2-103.

Model

$$\rho = \beta_0 + \beta_1 w + \beta_2 T + \beta_3 w^2 + \beta_4 T^2 + \beta_5 wT$$

Curve-Fitting

Multilinear regression using vector-matrix calculations

Prepare the dataset

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

wtpct = np.array([1.,2.,4.,8.,12.,16.,20.,24.,26.])
T = np.array([0.,10.,25.,40.,60.,80.,100.])
Den = np.loadtxt('SaltDensity.csv',delimiter=',',unpack=True)

n1 = len(wtpct)
n2 = len(T)

wn = np.zeros(n1*n2)
Tn = np.zeros(n1*n2)
y = np.reshape(Den,(n1*n2,1),order='F')

for i in range(n1):
    for j in range(n2):
        wn[i*n2+j] = wtpct[i]
        Tn[i*n2+j] = T[j]
```

MultilinearRegression.py

Curve-Fitting

Multilinear regression using vector-matrix calculations

Form the X matrix and carry out the regression calculations

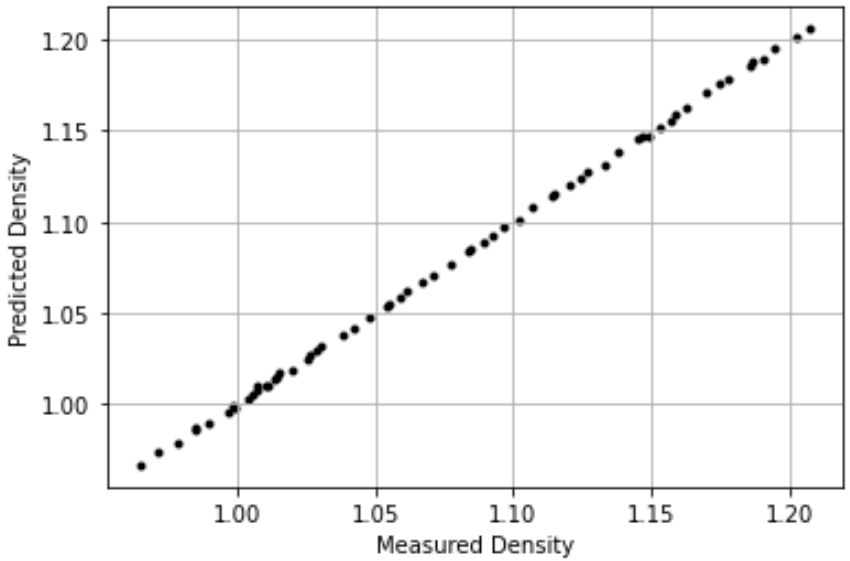
```
X = np.column_stack((np.ones(n), wn, Tn, wn**2, Tn**2, wn*Tn))

Xt = np.transpose(X)
XtX = Xt.dot(X)
Xty = Xt.dot(y)
b = np.linalg.solve(XtX,Xty)
print(b)
```

Model coefficients from left to right

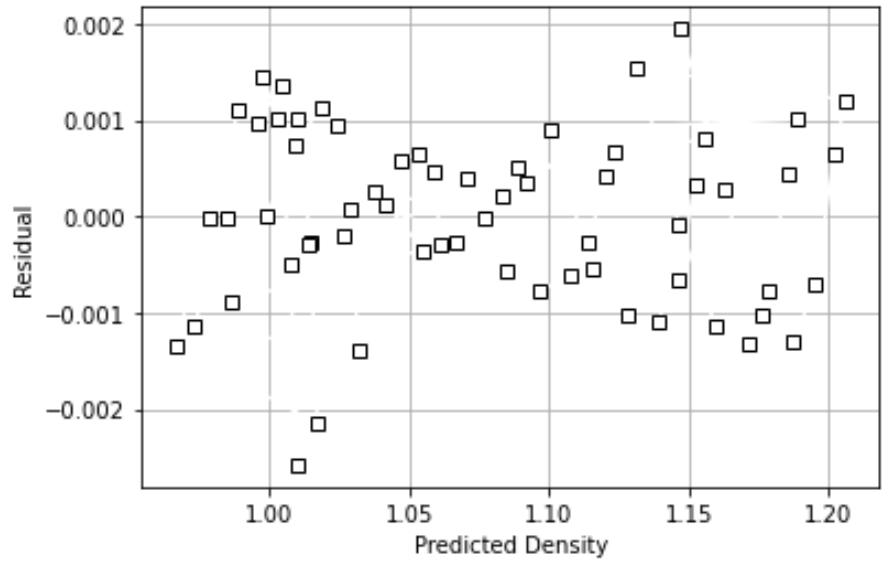
```
[[ 1.00291293e+00]
 [ 7.10909423e-03]
 [-2.21137527e-04]
 [ 2.68240071e-05]
 [-2.08788409e-06]
 [-6.01447754e-06]]
```

Curve-Fitting

Multilinear regression using vector-matrix calculations



Very close to perfect agreement



Model appears to be adequate

Curve-Fitting

Multilinear regression using the **statsmodels** module

Anaconda Prompt (Anaconda3)

```
conda install -c conda-forge statsmodels
```

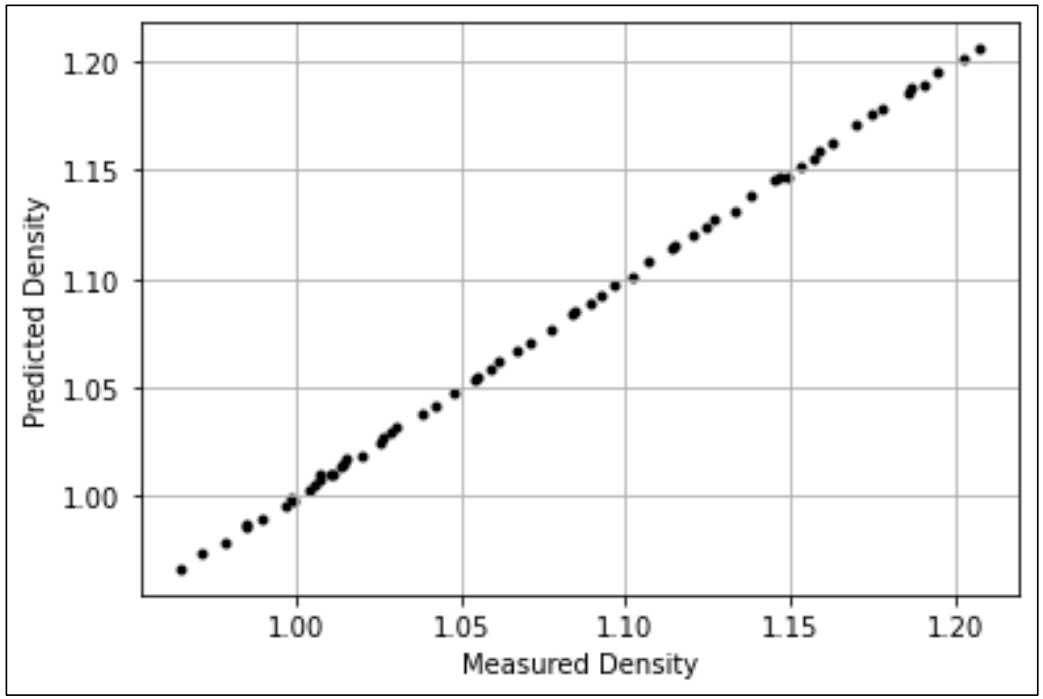
```
import numpy as np
import matplotlib.pyplot as plt
import statsmodels.api as ssm

wtpct = np.array([1.,2.,4.,8.,12.,16.,20.,24.,26.])
T = np.array([0.,10.,25.,40.,60.,80.,100.])
Den = np.loadtxt('SaltDensity.csv',delimiter=',',unpack=True)

n1 = len(wtpct)
n2 = len(T)
n = n1*n2

wn = np.zeros(n)
Tn = np.zeros(n)
y = np.reshape(Den,(n,1),order='F')
y = y.flatten()

for i in range(n1):
    for j in range(n2):
        wn[i*n2+j] = wtpct[i]
        Tn[i*n2+j] = T[j]
```

MultiLinearRegressionStatsModels.py

Curve-Fitting

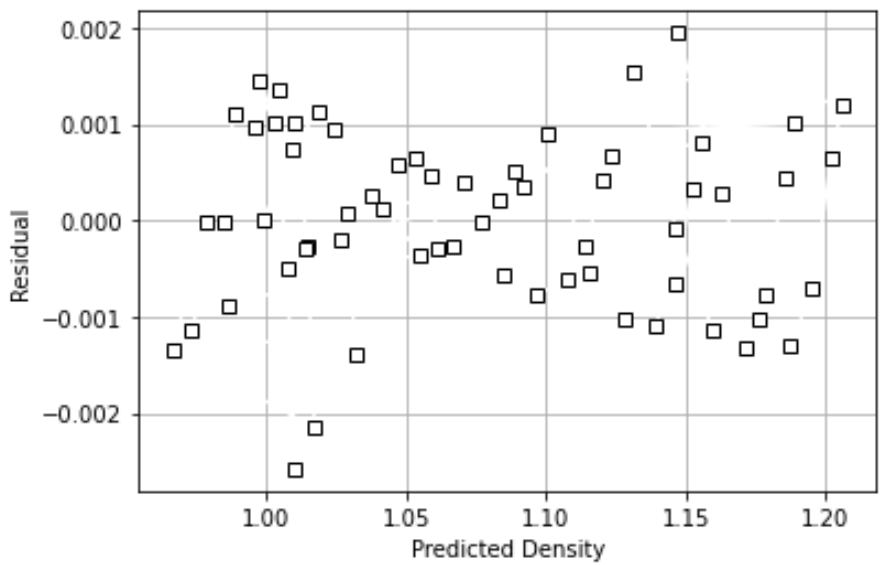
Multilinear regression using the **statsmodels** module

```
X = np.column_stack((np.ones(n), wn, Tn, wn**2, Tn**2, wn*Tn))

mod = ssm.OLS(y,X)
result = mod.fit()

print(result.params)
print('\n',result.summary())


b = result.params
yp = X.dot(b)
resid = y - yp

plt.scatter(y,yp,marker='.',c='k')
plt.grid()
plt.xlabel('Measured Density')
plt.ylabel('Predicted Density')

plt.figure()
plt.plot(yp,resid,c='w',marker='s',mec='k',mfc='w')
plt.grid()
plt.xlabel('Predicted Density')
plt.ylabel('Residual')
```



Curve-Fitting

Multilinear regression using the **statsmodels** module




```
[ 1.00291293e+00  7.10909423e-03 -2.21137527e-04  2.68240071e-05
 -2.08788409e-06 -6.01447754e-06]

                            OLS Regression Results
==============================================================================
Dep. Variable:                      y   R-squared:                       1.000
Model:                            OLS   Adj. R-squared:                  1.000
Method:                 Least Squares   F-statistic:                 6.543e+04
Date:                Thu, 01 Jun 2023   Prob (F-statistic):          9.06e-106
Time:                        16:22:12   Log-Likelihood:                 351.10
No. Observations:                  63   AIC:                            -690.2
Df Residuals:                      57   BIC:                            -677.3
Df Model:                           5
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
const          1.0029      0.000   2330.561      0.000       1.002       1.004
x1             0.0071   5.97e-05    119.130      0.000       0.007       0.007
x2            -0.0002    1.4e-05    -15.774      0.000      -0.000      -0.000
x3           2.682e-05   2.09e-06     12.829      0.000    2.26e-05     3.1e-05
x4          -2.088e-06   1.28e-07    -16.352      0.000   -2.34e-06   -1.83e-06
x5          -6.014e-06   3.99e-07    -15.057      0.000   -6.81e-06   -5.21e-06
==============================================================================
Omnibus:                        1.424   Durbin-Watson:                   0.825
Prob(Omnibus):                  0.491   Jarque-Bera (JB):                1.287
Skew:                          -0.342   Prob(JB):                        0.525
Kurtosis:                       2.850   Cond. No.                     1.70e+04
==============================================================================

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.
[2] The condition number is large, 1.7e+04. This might indicate that there are
strong multicollinearity or other numerical problems.
```

Curve-Fitting

Nonlinear regression

Model: $y = f\left(\mathbf{x}, \boldsymbol{\beta}\right)$ Dataset: $\left\{y_i, x_{1i}, \ldots, x_{mi}, i = 1, \ldots, n\right\}$

$$\mathbf{f}\left(\mathbf{x}, \boldsymbol{\beta}\right) = \begin{bmatrix} f\left(\mathbf{x}_1\right) \\ f\left(\mathbf{x}_2\right) \\ \vdots \\ f\left(\mathbf{x}_n\right) \end{bmatrix}$$

$$\mathbf{e} = \mathbf{y} - \mathbf{f}\left(\mathbf{x}, \hat{\boldsymbol{\beta}}\right) \qquad \min_{\hat{\boldsymbol{\beta}}} \mathbf{e}^T \mathbf{e}$$

using an optimization routine

Curve-Fitting Nonlinear regression

Example: fitting the Antoine equation to vapor pressure data

$$\log_{10} P_V = A - \frac{B}{C+T}$$

Vapor Pressure of 95%(wt) Sulfuric Acid Aqueous Solution

Temperature (degC)	Vapor Pressure (torr)
35	0.0015
40	0.00235
45	0.0037
50	0.0058
55	0.00877
60	0.0133
65	0.0196
70	0.0288
75	0.0415
80	0.0606
85	0.0879
90	0.123
95	0.172
100	0.237
105	0.321
110	0.437
115	0.59
120	0.788
125	1.07
130	1.42
135	1.87
140	2.4
145	3.11
150	4.02
155	5.13
160	6.47
165	8.39
170	10.3
175	12.9
180	15.9
185	20.2
190	24.8
195	30.7
200	36.7
205	45.3
210	55
215	66.9
220	79.8
225	95.5
230	115
235	137
240	164
245	193
250	229
255	268
260	314
265	363
270	430
275	500
280	580
285	682
290	790

Curve-Fitting Nonlinear regression

Example: fitting the Antoine equation to vapor pressure data

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import minimize

def SSE(x,T,LVP):
    A = x[0]
    B = x[1]
    C = x[2]
    LVPM = A - B / (C + T)
    VPerr = LVP - LVPM
    return VPerr.dot(VPerr)

T,VP = np.loadtxt('H2SO4VaporPressure.txt',unpack=True)

LVP = np.log10(VP)

A = 10 ; B = 2000 ; C = 250
x0 = [ A, B, C ]
```

```
result = minimize(SSE,x0,args=(T,LVP))
print(result)

A = result.x[0]
B = result.x[1]
C = result.x[2]

Tplot = np.linspace(np.min(T),np.max(T),100)
LVplot = A - B / (C + Tplot)

plt.plot(T,LVP,c='w',marker='s',mec='k',mfc='w',label='data')
plt.plot(Tplot,LVplot,c='k',label='Antoine Eqn')
plt.grid()
plt.xlabel('Temperature - degC')
plt.ylabel('log10(Vapor Pressure)')
plt.legend()
```

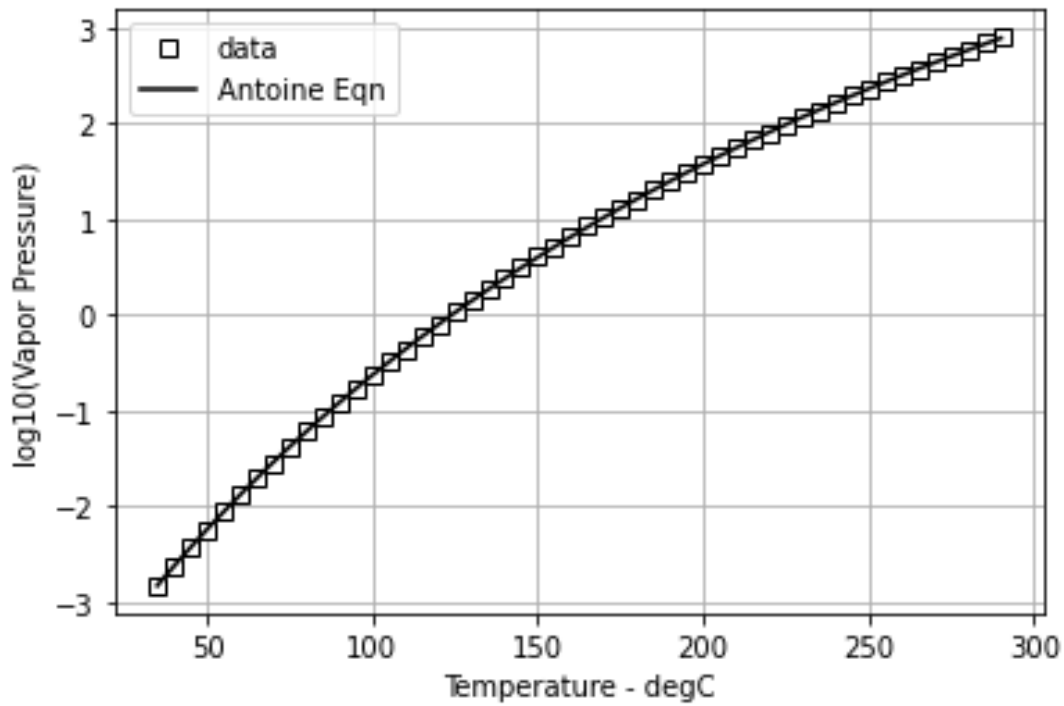
NonlinearRegression.py

Curve-Fitting

Nonlinear regression

Example: fitting the Antoine equation to vapor pressure data

```
 message: Optimization terminated successfully
 success: True
  status: 0
     fun: 0.0009705402527890349
       x: [ 9.806e+00  3.902e+03  2.739e+02]
     nit: 51
     jac: [-4.518e-06  5.178e-08 -9.486e-07]
hess_inv: [[ 1.222e+01  1.022e+04  5.318e+02]
           [ 1.022e+04  8.589e+06  4.488e+05]
           [ 5.318e+02  4.488e+05  2.354e+04]]
    nfev: 232
    njev: 58
```



A = 9.806

B = 3902

C = 273.9

Reference:
Applied Numerical Methods with Python
Steven C. Chapra
David E. Clough
McGraw-Hill, 2022

"Prof. Clough, may I be excused? My brain is full."

What's Next?

Python Bootcamps 1, 2 and 3

- ✓ 1: Getting up to speed with Python
- ✓ 2: Learning to use Python to solve typical problem scenarios
- • 3: Detailed modeling of packed-bed and plug-flow reactors