COMPUTATIONAL METHODS & MODELLING 3

Study guide

Table of Contents

Contents

Guide to Using Python

For Loops

For (i = 1; I <=10; i+-):

    <loop conditions>

here i will start at 1 and will remain less than or equal to 10. The value of i increases by 1 in every step.

For I in range(5):

    <loop conditions>

here i will loop through every value from 0 to 5.

While Loops

While loops specify a continuing condition for the duration of a loop, such as:

```
i = 1

while i <6:

        print(i)

        if i ==3:

                break

        i +=1
```

Break and continue

"break" stops a loop and returns the first valid value, "continue" stops the current iteration and starts the next iteration of a loop.

Importing Data in Python

Text file

```
File = open("sample.txt")

Data = file.read()

Print(data)

File.close
```

Alternatively

```
With open("welcome.txt") as file:

        Object data = file.read()
```

This is more efficient as it closes the file after use.

Input Command

```
Value = input("Please enter a string: \n")

Print(f"You entered {value}")
```

Definition of Functions in Python

Defining Function

```
Def my_function(x):

        Return 5 * x

Print(my_function(6))
```

```
Print(x(5,6)
```

This method has more advantages as lambda can be used repeatedly to define numerous functions.

Lambda Function

```
X = lambda a,b: a*b
```

Symbolic Computation

```
From sympy import *

X = Symbol("x"_, y = symbol("y")
```

Algebraic Computation

```
Print(2*x + 3*x – y)
```

Differentiating wrt. X

```
Print(diff(x**2,x)
```

Integrating wrt. X

Print(integrate(cos(x),x)

Finding limit as x -> 0

Print(limit(sin(x)/x, x, 0))

Simplifies Expression

Print(simplify((x**2 + x**3)/x**2))

Solve

Print(solve(5*x – 15, x))

Solving Equations

Sympy solving multiple equations

Code

```
import sympy as sym
sym.init_printing()
x,y,z = sym.symbols('x,y,z')
c1 = sym.Symbol('c1')
f = sym.Eq(2*x**2+y+z,1)
g = sym.Eq(x+2*y+z,c1)
h = sym.Eq(-2*x+y,-z)

sym.solve([f,g,h],(x,y,z))
```

Result

$$x = -\frac{1}{2} + \frac{\sqrt{3}}{2}$$

$$y = c_1 - \frac{3\sqrt{3}}{2} + \frac{3}{2}$$

$$z = -c_1 - \frac{5}{2} + \frac{5\sqrt{3}}{2}$$

Objects to be solved

Solution outputs

Here comma indicates the '=' sign, i.e., $2x^2+y+z=1$

Fsolve for solving multiple equation systems (nonlinear)

fsolve for solving multiple equation systems (nonlinear equations)

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

def myFunction(z):
   x = z[0]
   y = z[1]
   w = z[2]

   F = np.empty((3))
   F[0] = x**2+y**2-20
   F[1] = y - x**2
   F[2] = w + 5 - x*y
   return F

zGuess = np.array([1,1,1])
z = fsolve(myFunction,zGuess)
print(z)
```

Imports **fsolve** from standard scipy.optimise package

Creates an empty row array of 3 elements

Populates guess array for solution with three guess values (not necessarily known roots).

Linalg for solving multiple equation systems (linear equations)

linalg for solving multiple equation systems (linear equations)

```
import numpy as np

A = np.array([ [3,-9], [2,4] ])
b = np.array([-42,2])
z = np.linalg.solve(A,b)
print(z)

M = np.array([ [1,-2,-1], [2,2,-
1], [-1,-1,2] ])
c = np.array([6,1,1])
y = np.linalg.solve(M,c)
print(y)
```

Creates a matrix row by row from the top.

linalg solves the equation immediately using a method such as Gauss Seidel or equivalent 'in the background'.

Two separate examples given here.

Standard Functions

Quad

General Purpose integration

Example

From scipy.integrate import quad

Def integrand(x, a, b):

Return a*x**2 + b

a = 2

b = 1

I = quad(integrand, 0, 1, args=(a,b))

Dblquad

Performs double integration on a lambda function

Example

$$I = \int_{y=0}^{1/2} \int_{x=0}^{1-2y} xy\,dx\,dy$$

From scipy.integrate import dblquad

Area = dblquad(lambda x, y: x*y, 0, 0.5, lambda x:0, lambda x: 1-2*y)

NOTE: The routine defines outer integral with fixed bounds first, then uses lambda to define the non-fixed bounds of the integral

Minimise (Optimisation)

For unconstrained multivariate optimisation

Rosenbrock Example

Import numpy as np

```
From scipy.optimize import minimize

Def rosen(x): “””The Rosenbrock function”””

Return(100.0*(x[1:]-x[:-1]**2.0)**2.0 +
(1-x[:-1])**2)

X0 = np.array([1.3, 0.7, 0.8, 1.9, 1.2])

Res = minimize(rosen, x0,
method=”nelder-mead”, options={“xatol”:
1e-8, “disp”:True})
```

Rosen hess Example

For unconstrained multivariate optimisation using gradients

```
Def rosen_hess(x):

        x = np.asarray(x)

        H = np.diag(-400*x[:-1],1) –
np.diag(400*x[:-1],-1)

        diagonal = np.zeros_like(x)

        diagonal[0] = 1200*x[0]**2 –
400*x[1] + 2

        diagonal[-1] = 200

        diagonal[1:-1] = 202 +
1200*x[1:-1]**2 – 400*x[2:]

        H = H + np.diag(diagonal

        Return H

Res = minimize(rosen, x0,
method=”Newton-CG”, jac=rosen_der,
hess=rosen_hess, options={“xtol”: 1e-8,
“disp”: True})

Res.x array([1., 1., 1., 1., 1.])
```

Nonlinear Constraint Example

For constrained multivariate optimisation using gradients

```
def cons_f(x):

        return [x[0]**2 + x[1], x[0]**2 - x[1]]

def cons_J(x):

        return [[2*x[0], 1], [2*x[0], -1]]

def cons_H(x, v):

        return v[0]*np.array([[2, 0], [0, 0]]) +
v[1]*np.array([[2, 0], [0, 0]])

from scipy.optimize import
NonlinearConstraint

nonlinear_constraint =
NonlinearConstraint(cons_f, - np.inf, 1,
jac=cons_J, hess=cons_H)
```

Constrained multivariate optimisation using gradients

for constrained multivariate optimisation using gradients

For the previous codes, boundaries and linear constraints for the optimisation routine are specified with commands such as these.

```
from scipy.optimize import Bounds >>>

bounds = Bounds([0, -0.5], [1.0, 2.0])

from scipy.optimize import LinearConstraint
>>>

linear_constraint = LinearConstraint([[1, 2],
[2, 1]], [-np.inf, 1], [1, 1])
```

All arguments of the minimise function have been defined

```
x0 = np.array([0.5, 0])

res = minimize(rosen, x0,
method='trust-constr',

jac=rosen_der, hess=rosen_hess,

constraints=[linear_constraint,nonlinear_const
raint],

options={'verbose': 1}, bounds=bounds)
```

Solution of ODEs by odeint

```
import numpy as np

from scipy.integrate import odeint

import matplotlib.pyplot as plt

        # function that returns dy/dt
```

```
    def model(y,t):
    k = 0.3
    dydt = -k * y
    return dydt
# initial condition
y0 = 5
# time points
t = np.linspace(0,20)
# solve ODE
y = odeint(model,y0,t)
# plot results
plt.plot(t,y)
plt.xlabel('time')
plt.ylabel('y(t)')
plt.show()
```

A family of solutions can be solved for different values of the parameter, k as below:

```
import numpy as np
from scipy.integrate import odeint
import matplotlib.pyplot as plt
# function that returns dy/dt
def model(y,t,k): dydt = -k * y
    return dydt
# initial condition
y0 = 5
# time points
t = np.linspace(0,20)
# solve ODEs
k = 0.1
y1 = odeint(model,y0,t,args=(k,))
k = 0.2
y2 = odeint(model,y0,t,args=(k,))
k = 0.5
y3 = odeint(model,y0,t,args=(k,))
# plot results
plt.plot(t,y1,'r-',linewidth=2,label='k=0.1')
plt.plot(t,y2,'b--',linewidth=2,label='k=0.2')
plt.plot(t,y3,'g:',linewidth=2,label='k=0.5')
plt.xlabel('time')
plt.ylabel('y(t)')
plt.legend()
plt.show()
```

Root Finding

Bracketing method

Root finding with graphs can be fast, but inaccurate without testing estimated roots against function value.

Evaluating root estimates based solely on function values can be laborious and haphazard without a reasonable algorithm to save computational steps.

Bisection Technique

Works relatively well but is comparatively less efficient. This is a brute force technique that relies on enough iterations to achieve a true answer, or one with an acceptable error margin.

Step 1: Choose lower x_l and upper x_u guesses for the root such that the function changes sign over the interval. This can be checked by ensuring that $f(x_l)f(x_u) < 0$.
Step 2: An estimate of the root x_r is determined by

$$x_r = \frac{x_l + x_u}{2}$$

Step 3: Make the following evaluations to determine in which subinterval the root lies:
(a) If $f(x_l)f(x_r) < 0$, the root lies in the lower subinterval. Therefore, set $x_u = x_r$ and return to step 2.
(b) If $f(x_l)f(x_r) > 0$, the root lies in the upper subinterval. Therefore, set $x_l = x_r$ and return to step 2.
(c) If $f(x_l)f(x_r) = 0$, the root equals x_r; terminate the computation.

This algorithm could proceed for large iterations in order to determine an accurate/exact answer.

In practice, a termination criteria should be used to prevent it going for infinity.

Step 1: Choose lower x_l and upper x_u guesses for the root such that the function changes sign over the interval. This can be checked by ensuring that $f(x_l)f(x_u) < 0$.
Step 2: An estimate of the root x_r is determined by

$$x_r = \frac{x_l + x_u}{2}$$

Step 3: Make the following evaluations to determine in which subinterval the root lies:
(a) If $f(x_l)f(x_r) < 0$, the root lies in the lower subinterval. Therefore, set $x_u = x_r$ and return to step 2.
(b) If $f(x_l)f(x_r) > 0$, the root lies in the upper subinterval. Therefore, set $x_l = x_r$ and return to step 2.
(c) If $f(x_l)f(x_r) = 0$, the root equals x_r; terminate the computation.

Acceptable Error

This code then needs an acceptable error which, when achieved, terminates the program

$$\varepsilon_a = \left| \frac{x_r^{new} - x_r^{old}}{x_r^{new}} \right| \times 100$$

Number of Iterations

To estimate the number of iterations based on the knowledge of the initial bracket size

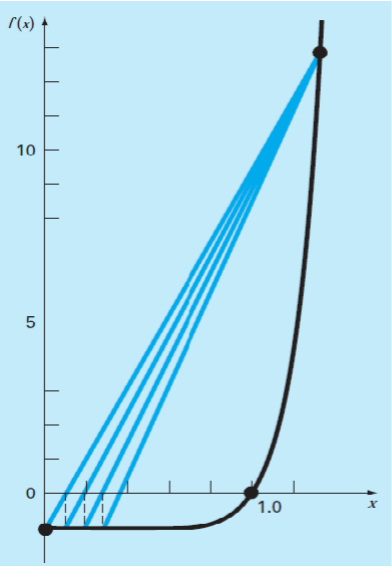
$$no.\ of\ evaluations = \frac{\log log\left(\frac{initial\ interval}{final\ relative\ error}\right)}{\log log\ (2)}$$

False Position Method

Takes into account the magnitude of the function values on either bound of the interval.

$$\frac{f(x_1)}{x_r - x_1} = \frac{f(x_u)}{x_r - x_u} \qquad x_r = x_u - \frac{f(x_u)(x_1 - x_u)}{f(x_1) - f(x_u)}$$

This technique is dependent on the shape of the function cure, and therefore won't always be the fastest way on convergence.



Example: The code cannot "see" past the long, flat section and therefore must crawl along the curve.

NOTE: it is essential to plot a graph of a function before choosing a suitable estimation technique.

Modified FPM

An unmodified FPM has the issue that one bound may remain completely unchanged through many iterations.

A modified FPM halves the function value at the "stuck" bound.

Modified FPM Example

```
MAX_ITER = 1000000

# The function is x^3 - x^2 + 2

def func( x ):

return (x**3 - 4*(x*2) + 10)

# Prints root of func(x) in interval [a, b

]def regulaFalsi( a , b):

if func(a) * func(b) >= 0:

print("You have not assumed right a and b")

return -1c = a

# Initialize result

for i in range(MAX_ITER):

# Find the point that touches x axis

c = (a * func(b) - b * func(a))/ (func(b)
-func(a))

# Check if the above found point is root

if func(c) == 0:

break

# Decide the side to repeat the steps

elif func(c) * func(a) < 0:

b = c

else:

a = c

print("The value of root is : " , '%.4f' %c)

# Test the function by setting a and b and
calling the code by the filename

# Initial values assumed

a =-200


b = 300

regulaFalsi(a, b)
```

Incremental Search Techniques

These techniques choose one point at either end of an interval of interest around a root of a function. Then at arbitrary increments (moving in one direction), the function is evaluated until the value is within the required error tolerance.

In cases where there are very close roots, if too big of a step interval is chosen then the roots may be skipped all together.

Example

```
def naive_root(f, x_guess, tolerance, step_size):

    steps_taken = 0

    while abs(f(x_guess)) > tolerance:

        if f(x_guess) > 0:
            x_guess -= step_size
        elif f(x_guess) < 0:
            x_guess += step_size
        else:
            return x_guess

        steps_taken += 1

    return x_guess, steps_taken

f = lambda x: x**2 - 20
root, steps = naive_root(f, x_guess=4.5, tolerance=.01, step_size=.001)
print ("root is:", root)
print ("steps taken:", steps)
```

Open Root Finding Methods

This technique only requires 1 starting point.

Differences in Open and Closed Root Finding Methods

Closed methods will always converge to a solution, provided that the root or roots actually lie in the domain of x chosen for investigation.

Open methods, by contrast, may not converge towards a root but if they do they tend to be far more efficient in reaching a solution in fewer steps than bisection

Single Fixed Point Iteration Example

```
def f(x):
    return x*x*x + x*x -1

# Re-writing f(x)=0 to x = g(x)
def g(x):
    return 1/math.sqrt(1+x)

# Implementing Fixed Point Iteration Method
def fixedPointIteration(x0, e, N):
    print('\n\n*** FIXED POINT ITERATION ***')
    step = 1
    flag = 1
    condition = True
    while condition:
        x1 = g(x0)
        print('Iteration-%d, x1 = %0.6f and f(x1) = %0.6f' % (step, x1,
f(x1)))
        x0 = x1
        step = step + 1

    if step > N:
            flag=0
            break
        condition = abs(f(x1)) > e

    if flag==1:
        print('\nRequired root is: %0.8f' % x1)
    else:
        print('\nNot Convergent.')

# Input Section
x0 = input('Enter Guess: ')
e = input('Tolerable Error: ')
N = input('Maximum Step: ')

# Converting x0 and e to float
x0 = float(x0)
e = float(e)

# Converting N to integer
N = int(N)

#Note: You can combine above three section like this
# x0 = float(input('Enter Guess: '))
# e = float(input('Tolerable Error: '))
# N = int(input('Maximum Step: '))
```

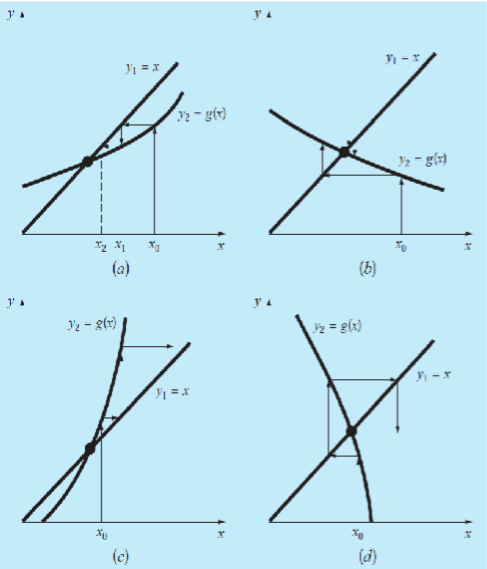
Two Point Graphical Method

$$e^{-x} - x = 0$$

The function above can easily be split into two further equations, as shown

$$y - e^{-x} \qquad y = x$$

The root of the main function is then the x-value of where these functions intersect. This technique cannot be used uncritically in all situations.



1. Four generic examples of function are given in the graph (left).
2. In each graph the progress of an iteration towards the intersection of each pair is shown.
3. a) and b) show converging solutions, while c) and d) show diverging non-solutions where the simple fixed iteration method fails.
4. Secondly, a) and c) represent monotone iteration patterns, while b) and d) represent spiral iteration patterns.
5. A solution is possible for the condition

$$|g'(x)| < 1$$

Newton Raphson Method

An initial root value is used to evaluate a function value from which a tangent line is drawn to the x-axis to produce a next estimate for the root. This continues until the function approximates to 0 within an error. This has a much faster convergence and a far lower error than the single point fixed method.

Newton Raphson Example

```
def newton(f,Df,x0,epsilon,max_iter):
    '''Solution of f(x)=0 by Newton's method.

    Parameters
    ----------
    f : function for which we are searching for a solution
f(x)=0.
    Df : Derivative of f(x).
    x0 : Initial guess for a solution f(x)=0.
    epsilon : number
        Stopping criteria is abs(f(x)) < epsilon.
    max_iter : integer
        Maximum number of iterations
    Examples
    --------
    >>> f = lambda x: x**2 - x - 1
    >>> Df = lambda x: 2*x - 1
    >>> newton(f,Df,1,1e-8,10)
    Found solution after 5 iterations.
    1.618033988749989
    '''
    xn = x0
    for n in range(0,max_iter):
        fxn = f(xn)
        if abs(fxn) < epsilon:
            print('Found solution after',n,'iterations.')
            return xn
        Dfxn = Df(xn)
        if Dfxn == 0:
            print('Zero derivative. No solution found.')
            return None
        xn = xn - fxn/Dfxn
    print('Exceeded maximum iterations. No solution found.')
    return None

f = lambda x: x**4 - x - 1
df= lambda x: 4*x**3 - 1
x0=1
epsilon=0.001
max_iter=100
solution = newton(f,df,x0,epsilon,max_iter)
print(solution)
```

Functions that are not solveable

These functions include asymptotic functions, functions with multiple minima, cyclic/periodic functions or functions with no real root.

Measures to Mitigate

- Graph the function to see the shape
    - Establish whether a root exists in the domain of interest
- Check each solution to establish its closeness to 0
- Include an upper limit in the number of iterations to prevent infinite cycling

Complications

One of the complications of this technique is that it requires the calculation of an exact derivative. For known functions, this is fine as they are analytically differentiable but for those that are this method is problematic. The secant method may then be used instead (this calculates an approximation to the point derivative).

Secant Technique Example

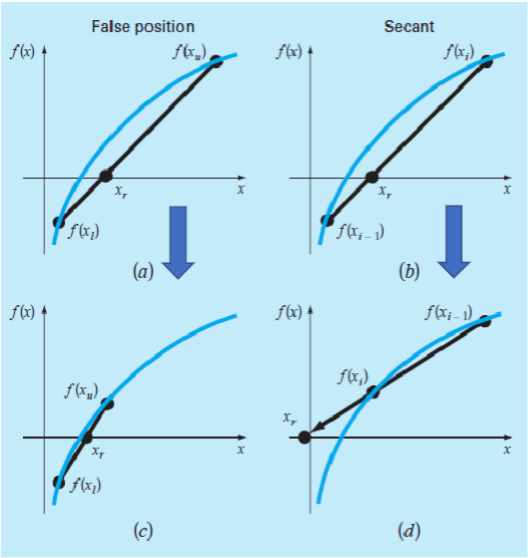
```
def secant(f,a,b,N):

'Examples -------- >>> f = lambda x: x**2 -
x - 1 >>> secant(f,1,2,5) 1.6180257510729614
if f(a)*f(b) >= 0:
          print("Secant method fails.")
return None
          a_n = a b_n = b
for n in range(1,N+1):
          m_n = a_n - f(a_n)*(b_n -
          a_n)/(f(b_n) - f(a_n)) f_m_n =
          f(m_n)
if f(a_n)*f_m_n < 0:
          a_n = a_n b_n = m_n
elif f(b_n)*f_m_n < 0:
          a_n = m_n b_n = b_n
elif f_m_n == 0:
          print("Found exact solution.")
return m_n
else: print("Secant method fails.")
          return None
return a_n - f(a_n)*(b_n - a_n)/(f(b_n) -
f(a_n))
```

Implement code by defining function, f, and calling secant. The result is allocated to variable 'solution'.

```
f = lambda x: x**4 - x - 1
solution =
secant(f,1,2,25)
print(solution)
```

Comparing Secant Technique and False Position

These techniques are very similar, however False Position always sets the new right bound of the new line to be at the new estimate for the root. The secant method uses the new estimate to cut the curve above it. False position is guaranteed to find the root, however secant is not.



Modified Secant Technique

This involves the alteration of how derivatives are evaluated. In the secant, two different points are used to compute the derivative for the given step. However, in the modified secant, the new derivative is calculated by retaining one old point and taking a small increment on this value as the second reference point.

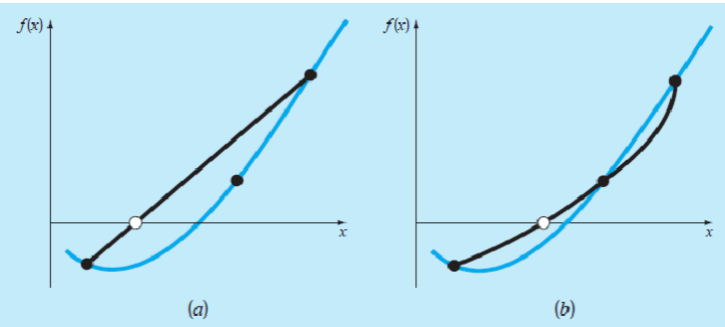
Modified Secant Technique Example

```
def secant_method(func, x0, alpha=1.0, tol=1E-9,
maxit=200):
    """
    Uses the secant method to find f(x)=0.

    INPUTS

        * f     : function f(x)
        * x0    : initial guess for x
        * alpha : relaxation coefficient:
modifies Secant step size
        * tol   : convergence tolerance
        * maxit : maximum number of iteration, default=200
    """

x, xprev = x0, 1.001*x0
    f, fprev = x**4 - x - 1, xprev**4 - xprev - 1

    rel_step = 2.0 *tol
    k = 0
    print('{0:12} {1:12} {2:12} {3:12} {4:12}'\
    .format('Iteration', 'x', 'f(x)', 'Rel step', 'alpha *
Delta x'))
```

```
while (abs(f) > tol) and (rel_step) > tol
and (k<maxit):
        rel_step = abs(x-xprev)/abs(x)

        # Full secant step
        dx = -f/(f - fprev)*(x - xprev)

        # Update `xprev` and `x`
simultaneously
        xprev, x = x, x + alpha*dx

        # Update `fprev` and `f`:
        fprev, f = f, func(x)

        k += 1

        print('{0:10d} {1:12.5f} {2:12.5f}
{3:12.5f} {4:12.5f}'\
        .format(k, xprev, fprev, rel_step,
alpha*dx))

    return x

    func = lambda x: x**4 - x - 1
    solution =
secant_method(func,0,alpha=1.0,tol=1E-
9,maxit=200)
    print(solution)
```

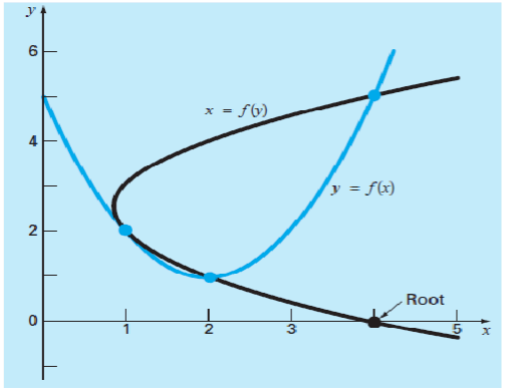
Inverse Quadratic

It is more effective to use a second order curve to model a curve that is third order or higher than it is to use the linear secant method. (GOOD FOR IMAGINARY)



Inverse quadratic interpolation uses a parabola x = f(y) (a parabola on its side) to model a function

Interpolation Without Real Roots



In this example, an attempted interpolation function gives a parabola with no real roots.

This method crashes immediately.

Inverse Quadratic Interpolation Example

```
def inverse_quadratic_interpolation(f, x0, x1, x2, max_iter=20,
tolerance=1e-5):
    steps_taken = 0
    while steps_taken < max_iter and abs(x1-x0) > tolerance: # last
guess and new guess are v close
        fx0 = f(x0)
        fx1 = f(x1)
        fx2 = f(x2)
        L0 = (x0 * fx1 * fx2) / ((fx0 - fx1) * (fx0 - fx2))
        L1 = (x1 * fx0 * fx2) / ((fx1 - fx0) * (fx1 - fx2))
        L2 = (x2 * fx1 * fx0) / ((fx2 - fx0) * (fx2 - fx1))
        new = L0 + L1 + L2
        x0, x1, x2 = new, x0, x1
        steps_taken += 1
    return x0, steps_taken

f = lambda x: x**2 - 20

root, steps = inverse_quadratic_interpolation(f, 4.3, 4.4, 4.5)
print ("root is:", root)
print ("steps taken:", steps)
```

Multiple Repeated Root Technique

Multiple roots exist for the condition that a point on the function touches the x-axis.

This can be solved with using open search methods. (Not bracketing as they do not change sign). It is easiest to use the Ralston-Rabinowitz method

Ralston-Rabinowitz Method

The first step is to define a function as follows:

$$u(x) = \frac{f(x)}{f'(x)}$$

Then substituting u(x) and u'(x) for f(x) and f'(x)

$$x_{i+1} = x_i - \frac{u(x_i)}{u'(x_i)}$$

Then, to find u'(x) in terms of f'(x) and f(x)

$$u'(x) = \frac{(f'(x)f'(x) - f(x)f''(x))}{f'(x)^2}$$

Substituting into Newton-Raphson we get:

$$x_{i+1} = x_i - \frac{f(x_i)f'(x_i)}{\left(f'(x_i)^2 - f(x_i)f''(x_i)\right)}$$

Supplement & Root Finding Examples

Engineering Examples

We are given function:

$$f = \frac{1.325}{\left[\ln ln\left(\frac{\varepsilon}{3.7D} + \frac{5.74}{Re^{0.9}}\right)\right]^2}$$

Solution

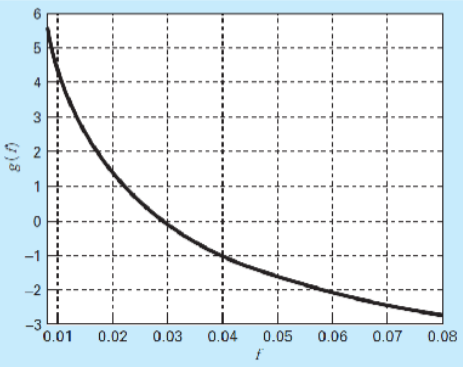
To solve this problem, we must calculate as many variables as possible. The first obvious variable to calculate is Re:

$$Re = \frac{\rho VD}{\mu}$$

We can then substitute this into Colebrook[1] to get the following:

$$\frac{1}{\sqrt{f}} = -1.0\left[\frac{\frac{\varepsilon}{D}}{3.7} + \frac{2.51}{Re\sqrt{f}}\right]$$

STEP ONE: graph the function



This gives us the following information:

- A solution actually exists for g(f) = 0
- The solution is in the region of f = 0.03
- The function is continuous and differentiable
- Can use bracketing or open method to solve this
- Known initial values to use

[1] Used to calculate Darcy friction Factor.

Bisection to solve Colebrook

```
def bisection(f,a,b,N):
    if f(a)*f(b) >= 0:
        print("Bisection method fails.")
        return None
    a_n = a
    b_n = b
    for n in range(1,N+1):
        m_n = (a_n + b_n)/2
        f_m_n = f(m_n)
        if f(a_n)*f_m_n < 0:
            a_n = a_n
            b_n = m_n
        elif f(b_n)*f_m_n < 0:
            a_n = m_n
            b_n = b_n
        elif f_m_n == 0:
            print("Found exact solution.")
            return m_n
        else:
            print("Bisection method fails.")
            return None
    return (a_n + b_n)/2
f = lambda x: x**3 - x - 1
approx_phi = bisection(f,1,2,25)
print(approx_phi)
```

Exercise 1 Input and test this code for any equation and compare with classic exact q solution. **Then solve the Colebrook equati**

False Position Code

```
import math

def f(x):
    return x**10-2*(x**2)+5

def root(a, b):
    return b - (f(b)*((a-b)/(f(a)-f(b))))

def regF(a, b):
    itr = 0
    maxItr = 100
    with open('rfValues.csv', 'w') as f:
        f.write('#iteration, f(a), f(b), currentRoot\n')

        while (itr < maxItr):
            r = root(a,b)
            if (r < 0):
                b = r
            else:
                a = r

f.write('str(itr)'+','+'str(f(a))'+','+'str((f(b)))' + ',' +
'str(r)' + '\n')
            itr = itr + 1
    return r
rootVal = regF(0,1)
print ("Value of root is: " + str(rootVal))
```

Newton Raphson

```
Def newton(f,Df,x0,epsilon,max_iter):
xn = x0
    for n in range(0,max_iter):
        fxn = f(xn)
        if abs(fxn) < epsilon:
            print('Found solution after',n,'iterations.')
            return xn
        Dfxn = Df(xn)
        if Dfxn == 0:
            print('Zero derivative. No solution found.')
            return None
        xn = xn - fxn/Dfxn
    print('Exceeded maximum iterations. No solution found.')
    return None

f = lambda x: x**4 - x - 1
df= lambda x: 4*x**3 - 1
x0=1
epsilon=0.001
max_iter=100
solution = newton(f,df,x0,epsilon,max_iter)
print(solution)
```

Secant Technique

```
def secant(f,a,b,N):
    if f(a)*f(b) >= 0:
        print("Secant method fails.")
        return None
    a_n = a
    b_n = b
    for n in range(1,N+1):
        m_n = a_n - f(a_n)*(b_n - a_n)/(f(b_n) - f(a_n))
        f_m_n = f(m_n)
        if f(a_n)*f_m_n < 0:
            a_n = a_n
            b_n = m_n
        elif f(b_n)*f_m_n < 0:
            a_n = m_n
            b_n = b_n
        elif f_m_n == 0:
            print("Found exact solution.")
            return m_n
        else:
            print("Secant method fails.")
            return None
    return a_n - f(a_n)*(b_n - a_n)/(f(b_n) - f(a_n))

f = lambda x: x**4 - x - 1
solution = secant(f,1,2,25)
print(solution)
```

Common Roots of Equation Systems (Lecture 3, Root Finding 2)

In scenarios where it is necessary to determine the common roots of multiple equations simultaneously.

For the following quadratics:

$$u(x, y) = x^2 + xy - 10 = 0 \qquad v(x, y) = y + 3xy^2 - 57 = 0$$

We can use fixed point iteration or the newton-raphson technique

Solution

Take initial guesses of x & y values (x0 and y0), in this case x0 = 1.5 and y0 = 3.5 and rearrange the equations as follows to get a better x(i+1) estimate

$$x_{i+1} = \frac{10 - x_i^2}{y_i}$$

The same is done simultaneously for the y-coordinate, as follows:

$$y_{i+1} = 57 - 3x_i y_i^2$$

Then, substituting the guesses an improved value of xi, xi+1 is given

If this value is off, then the equations can be rearranged differently, as follows to give more accurate results:

$$x = \sqrt{10 - xy} \qquad y = \sqrt{\frac{57 - y}{3x}}$$

Roots of Equation Systems

The condition for fixed point iteration to converge on a solution for two non-linear equations is:

$$\left|\frac{\partial u}{\partial x}\right| + \left|\frac{\partial u}{\partial y}\right| < 1 \qquad \left|\frac{\partial v}{\partial x}\right| + \left|\frac{\partial v}{\partial y}\right| < 1$$

NOTE: test the conditions above prior to using this technique

An alternative is to use Newton-Raphson method. This is difficult to implement, and therefore isn't used.

Roots of Polynomial Equations

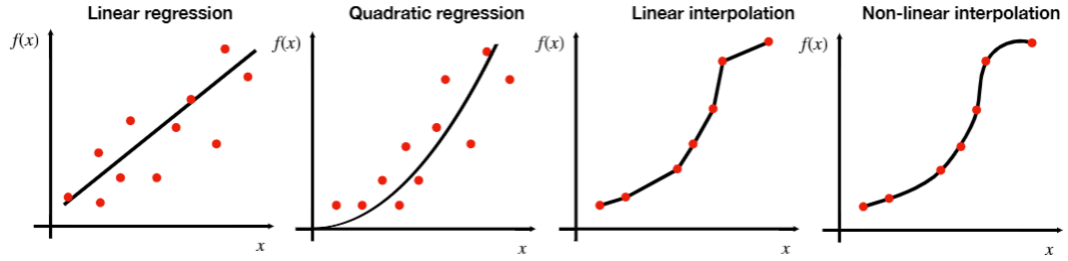
Number of computations for an nth order polynomial is:

$$\frac{n(n+1)}{2}$$

NOTE: To reduce the number of computations, employ a nested version of the polynomial, as follows

$$f_3(x) = \left(\left(a_3 x + a_2\right)x + a_1\right)x + a_0$$

Regression and Interpolation (Lecture 4)



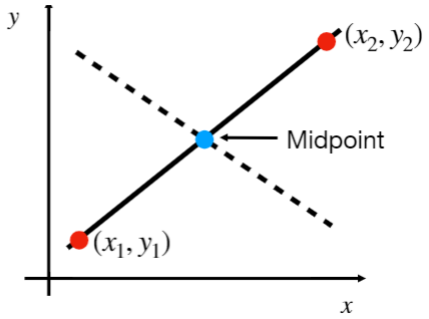
The mathematical expression for a straight line is:

$$y = a_0 + a_1 x + e$$

Where a0 and a1 are coefficients for the intercept and gradient and e is the error. We want to minimize the error

Minimizing the sum of errors for all available data

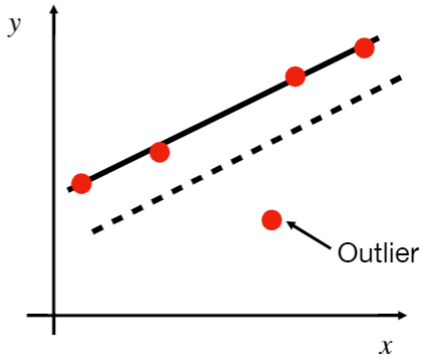
This is inadequate



As shown, obviously the best fit is the solid line between points. However, any straight line passing through the midpoint results in a minimum error of 0 because the errors cancel.

Minimax Criterion

This chooses the line that minimizes the max distance among points. This isn't effective as it gives undue influence on an outlier



To overcome this, we can minimize the sum of squares of the residuals

Minimizing the sum of squares of the residuals

$$S_r = \sum_{i=1}^{n} e_i^2 = \sum_{i=1}^{n} \left(y_i - a_0 - a_1 x_i\right)^2$$

This has many advantages, including that it yields a unique line for a given set of data

Example

To use the following model to fit data:

$$y = a_0 + a_1 x$$

We need to determine the a0 and a1 so that the least-square error is minimized.

To do this:

Differentiate Sr with respect to each coefficient

Setting these equal to 0 will result in a minimum Sr

Realizing that $\sum_{i=1}^{n} a_0 = na_0$ and grouping the terms, we obtain the two equations for the two unknowns (a_0 and a_1):

$$na_0 + \left(\sum_{i=1}^{n} x_i\right) a_i = \sum_{i=1}^{n} y_i \tag{10}$$

$$\left(\sum_{i=1}^{n} x_i\right) a_0 + \left(\sum_{i=1}^{n} x_i^2\right) a_1 = \sum_{i=1}^{n} x_i y_i \tag{11}$$

which give the solution for a_0 and a_1:

$$a_1 = \frac{n\sum_{i=1}^{n} x_i y_i - \sum_{i=1}^{n} x_i \sum_{i=1}^{n} y_i}{n\sum_{i=1}^{n} x_i^2 - \left(\sum_{i=1}^{n} x_i\right)^2} \tag{12}$$

$$a_0 = \bar{y} - a_1 \bar{x} \tag{13}$$

where $\bar{y} = \left(\sum_{i=1}^{n} y_i\right)/n$ and $\bar{x} = \left(\sum_{i=1}^{n} x_i\right)/n$ are the means of y and x, respectively.

Polynomial Regression

It is possible to fit polynomials of any order m to data using polynomial regression

Quadratic Regression

To get a minimum Sr, you must set the respective derivatives equal to 0. Then you have three linear equations and three unknowns.

General nonlinear regression

In many engineering cases, nonlinear models must be fit to data. These models depend on their parameters, for example an exponential:

$$f(x) = a_0\left(1 - e^{-a_1 x}\right) + error$$

Here we use a numerical optimisation method.

Linear interpolation with Newton polynomials

The first order interpolation with a Newton polynomial is:

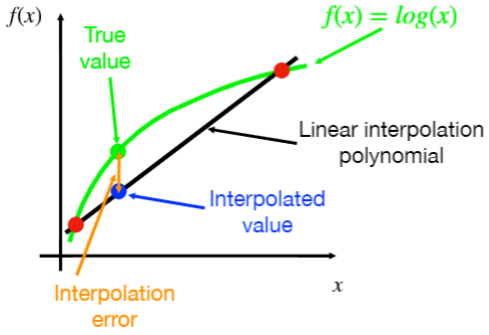
$$f(x) = f(x_0) + \frac{f(x_1) - f(x_0)}{x_1 - x_0}(x - x_0)$$

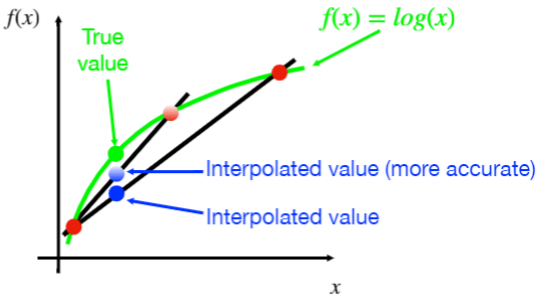
In general, the smaller the interval between points – the better the approximation.

Using Interpolation to approximate a function

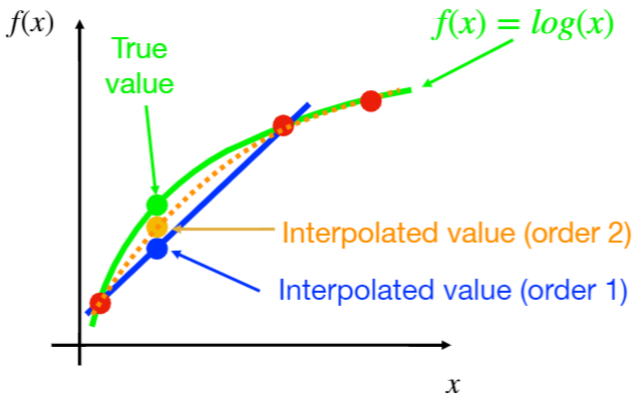
To approximate a function, we sample the function at two locations and use linear interpolation to approximate the function between these locations.

If two points are close, we obtain a better approximation.





First and Second order Interpolation



First order, we approximate the underlying function with a straight line.

Second order, we approximate the underlying function with a parabola.

For second order we need 3 points to construct a parabola that goes through them.

Second order is usually more accurate

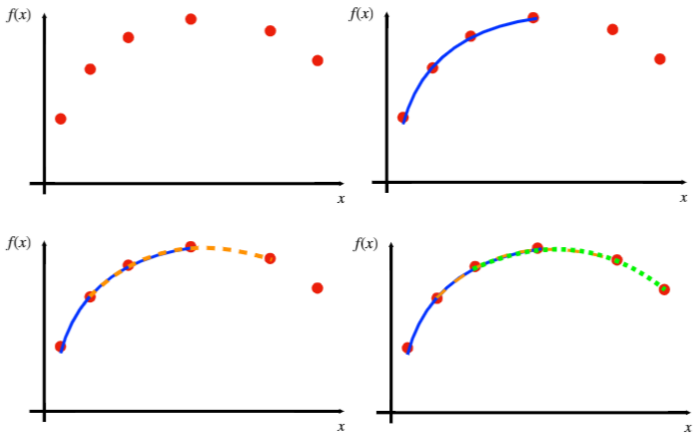
Splines

Spline interpolation is very flexible and powerful.

The interpolant is a special kind of piecewise polynomial called a spline.

Spline interpolation fits low-degree polynomials to small subsets of the values. The cubic spline is most common. This can be made very accurate.

NOTE: avoid the oscillatory behaviour that occurs if we fit a single high-order polynomial using many data points



Linear Algebra

Solutions of Systems of Linear Equations

Usually systems of three or less equations can be solved graphically by Cramer's Rule or elimination of unknown variables.

If there are common roots of any two equations, they can be evaluated at the intersection point of the two functions when they are plotted.

Consider:

$$a_{11}x_1 + a_{12}x_2 = b_1 \qquad a_{21}x_1 + a_{22}x_2 = b\_2$$

With two equations and two unknowns, so that it will be possible to solve for both unknowns x1 and x2

Solution of Systems of Linear Equations

- To progress the solution further, we isolate a different one of each of the variables of interest in both equations as follows:

$$a_{11}x_1 + a_{12}x_2 = b_1$$
$$a_{21}x_1 + a_{22}x_2 = b_2$$

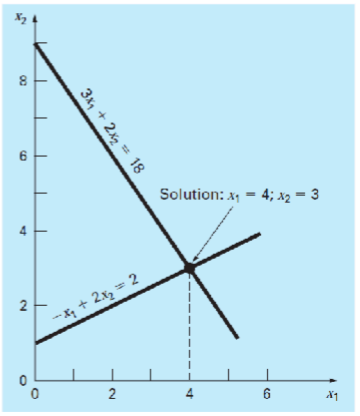
$$x_2 = -\left(\frac{a_{11}}{a_{12}}\right)x_1 + \frac{b_1}{a_{12}}$$
$$x_2 = -\left(\frac{a_{21}}{a_{22}}\right)x_1 + \frac{b_2}{a_{22}}$$

- Note that these equations can now be plotted as two straight lines on a common plot, and if they possess solutions, then the solution will be visually inspectable by recording the co-ordinates of their intersection point (right).

- A numerical example is given as follows:

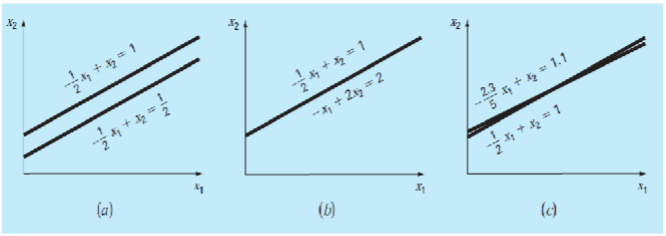
$$3x_1 + 2x_2 = 18$$
$$-x_1 + 2x_2 = 2$$

$$x_2 = -\frac{3}{2}x_1 + 9 \qquad x_2 = \frac{1}{2}x_1 + 1$$



Graphical solution of two simultaneous equations

Not all two-equation systems behave this way, as shown:



a) No solution (no intersection); b) infinite solutions (concurring equations); c) ill-conditioned solution (slopes are too close to be efficiently detected visually)

A and B are not resolvable as they are intrinsic properties of the equation system. C, however, is solvable and there needs to be found a suitable non-graphical technique to determines the solution.

Cramer's Rule

Given a set of linear equations expressed by: $[A]\{X\} = \{B\}$

[A] is the coefficient matrix

$$[A] = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}$$

The determinant is not a matrix but one number generated by the following operation:

$$D = a_{11}\begin{vmatrix} a_{22} & a_{23} \\ a_{32} & a_{33} \end{vmatrix} - a_{12}\begin{vmatrix} a_{21} & a_{23} \\ a_{31} & a_{33} \end{vmatrix} + a_{13}\begin{vmatrix} a_{21} & a_{22} \\ a_{31} & a_{32} \end{vmatrix}$$

In this case, the determinant is a function of various elements of the matrix and determinants of minors of matrix A.

A representative calculation of a determinant for one of the minors of A is:

$$D = \begin{vmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{vmatrix} \qquad D = a_{11}a_{22} - a_{12}a_{21}$$

If the determinant is 0, the system is singular, and if it is very close to 0, the problem is said to be ill-conditioned and almost singular.

Cramer's Rule for up to Three Equations

Cramer's rule allows the solution of each unknown to be written as follows for the example of x_1:

$$x_1 = \frac{\begin{vmatrix} b_1 & a_{12} & a_{13} \\ b_2 & a_{22} & a_{23} \\ b_3 & a_{32} & a_{33} \end{vmatrix}}{D}$$

Example: Take the following system of three equations:

$$0.3x_1 + 0.52x_2 + x_3 = -0.01$$
$$0.5x_1 + x_2 + 1.9x_3 = 0.67$$
$$0.1x_1 + 0.3x_2 + 0.5x_3 = -0.44$$

First, the determinant of the system is written from the nine coefficients of the three equations as follows:

$$D = \begin{vmatrix} 0.3 & 0.52 & 1 \\ 0.5 & 1 & 1.9 \\ 0.1 & 0.3 & 0.5 \end{vmatrix}$$

Then, the three minors of the system can be written down as follows:

$$A_1 = \begin{vmatrix} 1 & 1.9 \\ 0.3 & 0.5 \end{vmatrix} = 1(0.5) - 1.9(0.3) = -0.07$$

$$A_2 = \begin{vmatrix} 0.5 & 1.9 \\ 0.1 & 0.5 \end{vmatrix} = 0.5(0.5) - 1.9(0.1) = 0.06$$

$$A_3 = \begin{vmatrix} 0.5 & 1 \\ 0.1 & 0.3 \end{vmatrix} = 0.5(0.3) - 1(0.1) = 0.05$$

The determinant is now calculated as follows from the coefficients and the minors:

$$D = 0.3(-0.07) - 0.52(0.06) + 1(0.05) = -0.0022$$

Now, Cramer's Rule is used to calculate the three solutions for x_1, x_2, and x_3 as follows:

$$x_1 = \frac{\begin{vmatrix} -0.01 & 0.52 & 1 \\ 0.67 & 1 & 1.9 \\ 0.44 & 0.3 & 0.5 \end{vmatrix}}{-0.0022} = \frac{0.03278}{-0.0022} = -14.9$$

$$x_2 = \frac{\begin{vmatrix} 0.3 & -0.01 & 1 \\ 0.5 & 0.67 & 1.9 \\ 0.1 & -0.44 & 0.5 \end{vmatrix}}{-0.0022} = \frac{0.0649}{-0.0022} = -29.5$$

$$x_3 = \frac{\begin{vmatrix} 0.3 & 0.52 & -0.01 \\ 0.5 & 1 & 0.67 \\ 0.1 & 0.3 & -0.44 \end{vmatrix}}{-0.0022} = \frac{-0.04356}{-0.0022} = 19.8$$

Elimination of Unknowns

Cramer's Rule is 100% effective in cases with solutions. (Non-singular system). It is not computationally efficient for systems involving more than 3 equations.

Here, we use the elimination of unknowns. Example below.

$$a_{11}x_1 + a_{12}x_2 = b_1 \qquad a_{21}x_1 + a_{22}x_2 = b_2$$

Here we can times the first equation by a21 and the second equation by a11 to eliminate x1 and solve for x2.

Naïve (Simple) Gauss Elimination

This systemizes the process of forward elimination and reverse substitution.

Consider a system of linear equations as follows:

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + a_{13}x_3 + \cdots + a_{1n}x_n &= b_1 \\ a_{21}x_1 + a_{22}x_2 + a_{23}x_3 + \cdots + a_{2n}x_n &= b_2 \\ &\vdots \\ a_{n1}x_1 + a_{n2}x_2 + a_{n3}x_3 + \cdots + a_{nn}x_n &= b_n \end{aligned}$$

The first stage in the Gauss method is to implement the following manipulation of equation 1 in this series as follows:

$$a_{21}x_1 + \frac{a_{21}}{a_{11}}a_{12}x_2 + \cdots + \frac{a_{21}}{a_{11}}a_{1n}x_n = \frac{a_{21}}{a_{11}}b_1$$

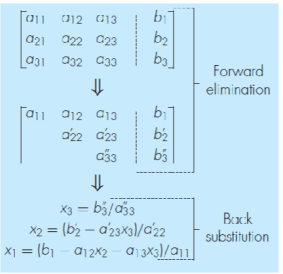
Now this equation is subtracted from the second in the series to give

$$\left(a_{22} - \frac{a_{21}}{a_{11}}a_{12}\right)x_2 + \cdots + \left(a_{2n} - \frac{a_{21}}{a_{11}}a_{1n}\right)x_n = b_2 - \frac{a_{21}}{a_{11}}b_1$$

The coefficients in the original entries of the coefficient matrix are now replaced so that the new equation above is labelled

$$a'_{22}x_2 + \cdots + a'_{2n}x_n = b'_2$$

The first equation remains unchanged in the revised matrix after this and subsequent operations.



After the forward eliminations are complete, the unknown variable in the last modified equation can be solved for directly. (In this case x3) Once this is solved, the remaining unknown in the previous equation is solved etc etc. This same process applies to any system of n linear equations that one solves using NGE. This can be computerised, unlike crammer's as it has a recognisable repeatable algorithm.

NGE Disadvantages

- NGE and all elimination techniques will crash as they cannot handle divisions by 0
- NGE can accumulate inaccuracies due to round-off error
    - Every decimal operation incurs some intrinsic relative error, and these accumulate to high total error over multiple calculations.
- Ill-Conditioned Systems are not suited to solution using NGE (and elimination methods) because of round off error
    - This is where small changes in coefficients induce large changes in solution.

Techniques for improving solutions to LESs

- Use of more significant figures
    - Reduces round-off errors
- Use of partial or full pivoting
    - This is the practice of identifying zero or near-zero coefficients of elements in a given row and switching the row with one containing much larger coefficients in the same position.

Gauss Elimination with Partial Pivoting

```
def column(m, c):
  return [m[i][c] for i in range(len(m))]

def row(m, r):
  return m[r][:]

def height(m):
  return len(m)

def width(m):
  return len(m[0])

def print_matrix(m):
  for i in range(len(m)):
    print(m[i])

def gaussian_elimination_with_pivot(m):

# forward elimination
  n = height(m)
  for i in range(n):
    pivot(m, n, i)
    for j in range(i+1, n):
      m[j] = [m[j][k] - m[i][k]*m[j][i]/m[i][i] for
k in range(n+1)]
```

```
if m[n-1][n-1] == 0:
raise ValueError('No unique solution')

  # backward substitution
  x = [0] * n
  for i in range(n-1, -1, -1):
    s = sum(m[i][j] * x[j] for j in
range(i, n))
    x[i] = (m[i][n] - s) / m[i][i]
  return x

def pivot(m, n, i):
    max = -1e100
    for r in range(i, n):
        if max < abs(m[r][i]):
            max_row = r
            max = abs(m[r][i])
    m[i], m[max_row] = m[max_row], m[i]

if __name__ == '__main__':
  #m = [[0,-2,6,-10], [-1,3,-6,5], [4,-
12,8,12]]
  #m = [[1,-1,3,2], [3,-3,1,-1], [1,1,0,3]]
  m = [[1,-1,3,2], [6,-6,2,-2], [1,1,0,3]]
  print(gaussian_elimination_with_pivot(m))
```

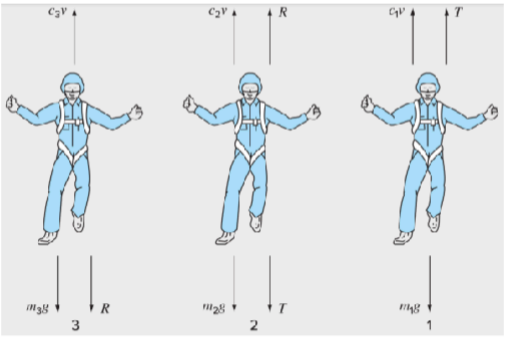
Gauss Elimination Applied to a Physics Problem

Calculate Cord Tensions in a Tandem Team of Parachutists

- Three parachutists are connected by a weightless cord while free-falling at a velocity of 5 m/s.
- Calculate the tension in each cord, and the acceleration of the team given the following data

Parachutist	Mass, kg	Drag Coefficient, kg/s
1	70	10
2	60	14
3	40	17

We firstly write Newton's Second Law expressions for each of the three parachutists:



Here, the accelerations of the three parachutists are each expressed as a balance of gravitational force (m_ig), and the tension force on each parachutist is R or/and T.

The drag force exerted by air against direction of fall is cv, where c_i are the drag coefficients, v_i are the velocities.

These three equations are linear and together comprise a linear equation system that can be solved using Gauss:

$$\begin{aligned} m_1 g - T - c_1 \nu &= m_1 a \\ m_2 g + T - c_2 \nu - R &= m_2 a \\ m_3 g - c_3 \nu + R &= m_3 a \end{aligned}$$

Filling in all the values we were provided with we get the following.

$$\begin{bmatrix} 70 & 1 & 0 \\ 60 & -1 & 1 \\ 40 & 0 & -1 \end{bmatrix} \begin{Bmatrix} a \\ T \\ R \end{Bmatrix} = \begin{Bmatrix} 636.7 \\ 518.6 \\ 307.4 \end{Bmatrix}$$

This is equivalent to a system **A.x = b**. We solve for x.

```
import numpy as np

def linearsolver(A,b):
    n = len(A)

    #Initialise solution vector as an empty array
    x = np.zeros(n)

    #Join A and use concatenate to form an
augmented coefficient matrix
    M = np.concatenate((A,b.T), axis=1)

    for k in range(n):
        for i in range(k,n):
            if abs(M[i][k]) > abs(M[k][k]):
                M[[k,i]] = M[[i,k]]
            else:
                pass
                for j in range(k+1,n):
                    q = M[j][k] / M[k][k]
                    for m in range(n+1):
                        M[j][m] +=  -q * M[k][m]
    #Python starts indexing with 0, so the last
element is n-1
    x[n-1] =M[n-1][n]/M[n-1][n-1]

    #We need to start at n-2, because of
Python indexing
    for i in range (n-2,-1,-1):
        z = M[i][n]
        for j in range(i+1,n):
            z = z  - M[i][j]*x[j]
        x[i] = z/M[i][i]

    return x

#Initialise the matrices to be solved.

A=np.array([[71., 1., 0],[60., -1., 1.],
[40, 0, -1]])
b=np.array([[636.7., 518.6, 307.4]])

print(linearsolver(A,b))
```

The answer received (as an array of three elements for x, is:
[8.55380117 29.38011696 34.75204678]

The first number is the acceleration of the team, while the second two numbers are the tensions.

Ordinary Differential Equations 1 (Lecture 6)

Taylor's Theorem and Taylor Series

If the function of independent variable x and its n+1 derivatives are continuous in an interval containing both points xi and xi+1 = xi + h, f(x) can be expanded in the following series:

$$\begin{aligned} f(x_{i+1}) = & f(x_i) + \frac{f'(x_i)}{1!}(x_{i+1} - x_i) + \\ & + \frac{f''(x_i)}{2!}(x_{i+1} - x_i)^2 + \frac{f'''(x_i)}{3!}(x_{i+1} - x_i)^3 + \\ & + \ldots + \\ & + \frac{f^{(n)}(x_i)}{n!}(x_{i+1} - x_i)^n + R_n \end{aligned}$$

where

$$R_n = \int_{x_i}^{x_{i+1}} \frac{(x_i - t)^n}{n!} f^{(n+1)}(t)dt$$

Rn can also be expressed in Lagrangian form, and is often called the Truncation Error.

$$R_n = \frac{f^{(n+1)}(\xi)}{(n+1)!}(x_{i+1} - x_i)^{n+1}$$

Taylor series term-by-term

Defining h = xi+1 – xi the Taylor series can be written as:

$$f(x_{i+1}) = f(x_i) + \frac{f'(x_i)}{1!}h + \frac{f''(x_i)}{2!}h^2 + \frac{f'''(x_i)}{3!}h^3 + \ldots + \frac{f^{(n)}(x_i)}{n!}h^n + R_n$$

Depending on the number of terms kept in the series, we have different levels of approximation.

- Zero-order approximation
  - F(xi+1) = f(xi)
    - If f(x) is constant, this is a perfect estimate
- First order approximation
  - F(xi+1) = f(xi) + f'(xi)h
    - This can predict a change in the function but is exact only if the function is linear
- Order n approximation

$$f(x_{i+1}) = f(x_i) + f'(x_i)h + \frac{f''(x_i)}{2!}h^2 + \ldots + R_n \quad \text{with} \quad R_n = \frac{f^{(n+1)}(\xi)}{(n+1)!}h^{n+1}$$

  - o

Truncation Error

$$R_n = \frac{f^{(n+1)}(\xi)}{(n+1)!}h^{n+1}$$

Cannot be determined since ξ is not known. We only know that it lies between xi and xi+1

We have control over h. For different orders, the error decreases in different ways if we decrease h.

We often write $R_n = \mathcal{O}(h^{n+1})$.
The expression $\mathcal{O}(h^{n+1})$ states that the error is of order of h^{n+1}, which means that the error is proportional to h^{n+1}.

Numerical Derivative

Considering,

$$f(x_{i+1}) = f(x_i) + f'(x_i)(x_{i+1} - x_i) + R_1$$

This equation can be solved for $f'(x_i) = \frac{f(x_{i+1}) - f(x_i)}{x_{i+1} - x_i} - \frac{R_1}{x_{i+1} - x_i}$

With an estimate of error:

$$\frac{R_1}{x_{i+1} - x_i} = \frac{f''(\xi)(x_{i+1} - x_i)^2}{2!}\frac{1}{x_{i+1} - x_i} = \mathcal{O}(x_{i+1} - x_i)$$

in the usual more compact form $f'(x_i) = \frac{\Delta f_i}{h} + \mathcal{O}(h)$

where :

- Δfi is the first forward difference

- h is the step size
- Δfi/h is the first forward divided difference
    - This is one of many ways to approximate the derivative using the taylor series.

One-Step Methods for ODEs

To solve equations in the form:

$$\frac{dy}{dx} = f(x, y)$$

With a given initial condition y0 = y(x0)

To solve this equation with a numerical method on a set of discrete points, we need to be able to extrapolate from a value yi to a new value yi+1 over a step h:

$$y_{i+1} = y_i + \emptyset h$$

Where ∅ is an estimate of an appropriate slope of the function y over the step h.

The simplest approach is to estimate the slope from the differential equation itself as the first derivative of y at the point xi, which is the Euler Method

Euler Method

A new value of y is computed extrapolating linearly over the step h using a slope approximated with the derivative in the original point xi, where the solution and its derivatives are known.

Runge-Kutta Method

Achieves high accuracy without the use of higher order derivatives like with the Taylor series.

Many versions exist but it can be cast as:

$$y_{i+1} = y_i + \emptyset(x_i, y_i, h)h$$

Where $\emptyset(x_i, y_i, h_i)$ is an increment function

NOTE: a runge-kutta method with n =1 is the Euler method

Systems of ODEs and Stiff Equations (Lecture 7)

Stiff ODEs

ODE or a system of ODEs where fast and slow components exist.

- Slow component: we need to solve the equation over a large interval
- Fast component: we usually need a small step h to capture the fast component
- Long interval with small steps means many steps.

Implicit Euler

Implicit methods employ information at locations that have not yet been computed. For the implicit euler method, we use the derivative in the point x(i+1) to estimate the slope. To compute y(i+1) we must find the root of the function F(yi+1)

Implicit vs explicit

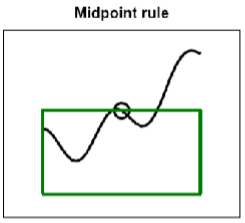
The implicit Euler method requires the solution of the (in general, non-linear) equation F(yi+1) = 0. This requires a root finding method, for example the Secant method in this case.

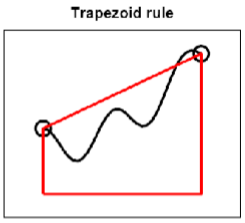
Numerical Integration (Lecture 9)

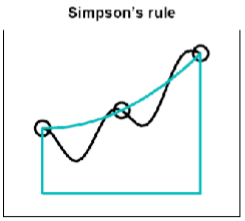
Quadrature Techniques

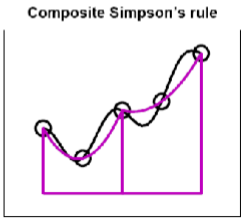
Quadrature is the process of evaluating the integral of the curve – it is not exact, it is approximate.

M – midpoint rule T = Trapezodial Rule S – true value of integral









Simpsons Third Rule

$$S = \frac{2}{3}M + \frac{1}{3}T$$

Gives an exact answer for a third order integral. (Not exact for powers > 4)

Simpsons Third Rule quadratic interpolation

Between limits a, b and their midpoint c

$$c = \frac{a+b}{2} \qquad h = b - a$$

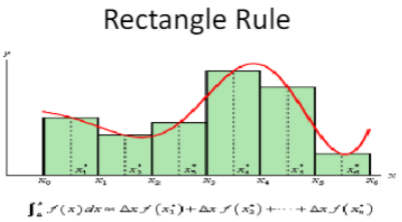
$$S = \frac{h}{6}(f(a) + 4f(c) + f(b))$$

Three Main Quadrature Rules

Rectangle/Midpoint Rule

$$\int_a^b f(x)dx \sim h\sum_{n=0}^{N-1} f(x_n)$$

$$h = \frac{b-a}{N} \qquad x_n = a + nh$$

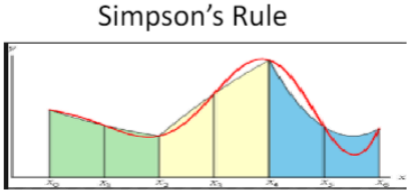


Trapezoidal Rule

$$\int_a^b f(x)dx \sim h\left[\frac{1}{2}f(x_0) + \sum_{i=1}^{n-1} f(x_i) + \frac{1}{2}f(x_n)\right]$$

Composite Simpson's Rule

$$\int_a^b f(x)dx = h/3\left[f(x_0) + 2\sum_{j=1}^{\frac{n}{2}-1} f(x_{2j}) + 4\sum_{j=1}^{\frac{n}{2}} f(x_{2j-1}) + f(x_n)\right]$$



Three Eighths Simpson's Rule

A more sophisticated model of Simpson third rule. Resulting in

$$\int_a^b f(x)dx = \frac{b-a}{8}\left[f(a) + 3f\left(\frac{2a+b}{3}\right)3f\left(\frac{a+2b}{3}\right) + f(b)\right]$$

Adaptive Algorithm Numerical Methods

Quadrature techniques work well for certain functions and less well for others. Adaptive algorithms allow intervals to be subdivided during the numerical integration if accuracy is not achieved after n steps. The adaptive algorithm approach can be applied dynamically within any of the three quadrature techniques. The "basic" equation is shown below.

$$\int_a^b f(x)dx \sim \frac{b-a}{6}\left[f(a) + 4f\left(\frac{a+b}{2}\right) + f(b)\right]$$

Which we can refine by halving the step size to give:

$$\int_a^b f(x) \sim \frac{b-a}{12}[f(a) + 4f(d) + 2f(c) + 4f(e) + f(b)]$$

$$c = \frac{a+b}{2} \quad d = \frac{a+c}{2} \quad e = \frac{c+b}{2} \quad h_1 = \frac{b-a}{6} \quad h_2 = \frac{(b-a)}{12}$$

The error can be defined as the difference between integral estimates

$$E = I(h_2) - I(h_1)$$

Optimisation (Lecture 10)

Two types of optimisations:

- One-dimensional optimisation – a curve in a plane (2D)
- Two-dimensional optimisation (3D functional surface)

If both the function and constraints are linear then the optimisation is an example of linear programming. If f(x) is a quadratic but its constraints are linear, we have quadratic programming. If both f(x) and constraints are non-linear, we have non-linear programming.

The DoF of a system in an optimisation problem is calculated by the term n-p-m. Where:

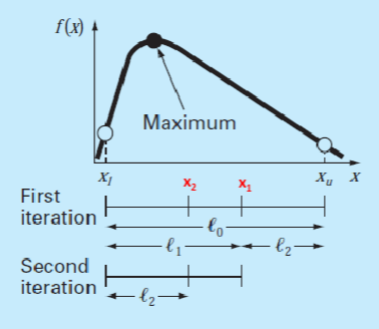
- N – the number of dimensions in the x vector
- P – number of equality constraints
- M – number of inequality constraints

To obtain a solution, m + p < n. If m + p >n, the optimisation is overconstrained.

NOTE: Pre-assess if a problem is over-constrained

Golden Search Technique

The most basic unconstrained one-dimensional search technique. Modelled closely on the bisection method used to find a function root (except this time for minimum/maximum)



GS relies on selecting two estimate points either side of a maximum or minimum. An effective strategy for selection is by using the golden ratio

$$l_o = l_1 + l_2 \qquad \frac{l_1}{l_0} = \frac{l_2}{l_1} \qquad \frac{l_1}{l_1 + l_2} = \frac{l_2}{l_1}$$

$$R = \frac{l_2}{l_1} \quad 1 + R = \frac{1}{R} \quad R^2 + R - 1 = 0 \quad R = \frac{\sqrt{5}-1}{2}$$

Then using the quadratic equation to solve for R gives our upper and lower estimates.

If f(x1)>f(x2), then all points left of x2 can be eliminated as no maximum will occur here. X2 becomes the new xl. If f(x1) < f(x2), then all points right of x1 can be eliminated from the region of interest. Xl becomes the new xu for the next iteration. Then we can calculate a new x1.

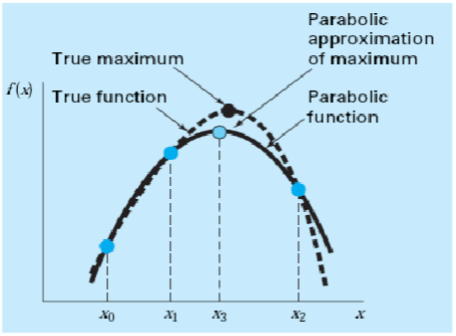
$$x_1 = x_1 + R(x_u - x_l) \qquad d = R(x_u - x_1) \qquad x_1 = x_1 + d \qquad x_2 = x_u - d$$

This golden ration halves the number of necessary function evaluations needed to complete the algorithm. Convergence is guaranteed, although the rate of convergence is infinite.

The approximation error for the optimal x value.

$$\varepsilon_a = (1 - R)\left|\frac{x_u - x_l}{x_{opt}}\right|100\%$$

Parabolic Interpolation



$$x_3 = \frac{f(x_0)(x_1^2 - x_2^2) + f(x_1)(x_2^2 - x_0^2) + f(x_2)(x_0^2 - x_1^2)}{2f(x_0)(x_1 - x_2) + 2f(x_1)(x_2 - x_0) + 2f(x_2)(x_0 - x_1)}$$

Where x0, x1 & x2 are the initial guesses and x3 is calculated approximation to the point xopt. The new points can then be done by reassigning sequentially.

Newtons Method

For finding a root approximation:

$$x_{i+1} = x_i - \frac{f'(x_i)}{f''(x_i)}$$

When f'(x) = 0, there is a minimum/maximum

Brent's Method

Combines parabolic interpolation and golden search techniques in one code.

Newtons Method (two – variable optimisation)

The absolute value of the Hessian is a test point for maximum or minimum.

$$|H| = \frac{\partial^2 f}{\partial x^2}\frac{\partial^2 f}{\partial y^2} - \left(\frac{\partial^2 f}{\partial x \partial y}\right)^2$$

If $|H| > 0$ and $\partial^2 f/\partial x^2 > 0$, then $f(x, y)$ has a local minimum.
If $|H| > 0$ and $\partial^2 f/\partial x^2 < 0$, then $f(x, y)$ has a local maximum.
If $|H| < 0$, then $f(x, y)$ has a saddle point.

Optimisation (Lecture 11)

Steepest Ascent (Hill Climb)

The solution to a multivariative problem is equivalent to finding the shortest route to the top of a high mountain.

The first step is to identify where you are on the surface, then the curve of travel in any direction is expressible by a function. At the peak of the optimisation path function, g(h), the derivative is 0. The value of h at this point is used to calculate the next point.

$$g'(h) = \frac{\partial f}{\partial x} cos\theta + \frac{\partial f}{\partial y} sin\theta = 0$$

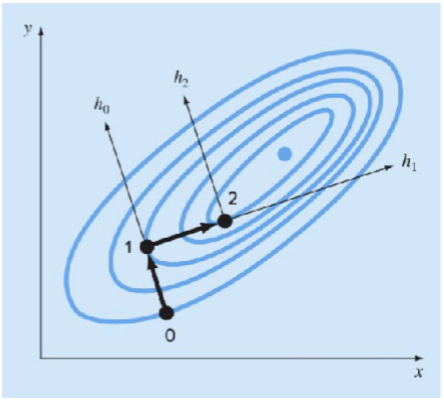
The path of steepest ascent is shown below.

$$x = x_0 + \frac{\partial f}{\partial x} h \qquad (A)$$

$$y = y_0 + \frac{\partial f}{\partial y} h$$

$$x = 1 + 3h \qquad (B)$$

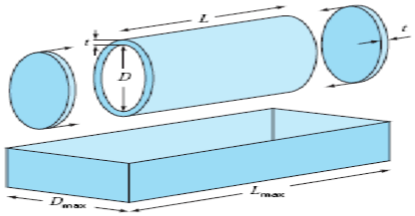
$$y = 2 + 4h$$



Constrained Non-Linear Optimisation (finding min & maxima)

Tank Example

You are asked to design a cylindrical tank as shown with geometrical variables that have not yet been fully optimised (you will do this).



Parameter	Symbol	Value	Units
Required volume	V_o	0.8	m^3
Thickness	t	3	cm
Density	ρ	8000	kg/m^3
Bed length	L_{max}	2	m
Bed width	D_{max}	1	m
Material cost	c_m	4.5	$/kg
Welding cost	c_w	20	$/m

The **volume of the tank is fixed** at 0.8 m^3, and **the tank thickness is defined** by waste handling regulations (3 cm).

Your constraints are that the tank must be transportable on a truck bed with **fixed, pre-defined dimensions (2 x 1 m)**

Note that these represent maximum value constraints; you may design a tank with l and d lower than these maxima.

The tank expense consists of 1) material expense (steel), and 2) welding expense, which is proportional to the length of welding seams in the tank.

Solution: The first step is to **define an objective function for cost** that you will minimise via an optimisation routine.

$$C = c_m m + c_w \ell_w$$

Here, C is total cost, c_m is material cost per unit mass of material, c_w is welding cost per m of weld, and l_w is weld length.

Now we need to define the volume of the cylinder and understand the contributions of cylinder length and diameter to its volume.

The volume of its cylindrical side wall can be computed as follows

$$V_{\text{cylinder}} = L\pi\left[\left(\frac{D}{2}+t\right)^2 - \left(\frac{D}{2}\right)^2\right]$$

The volume of each of its end plates is:

$$V_{\text{plate}} = \pi\left(\frac{D}{2}+t\right)^2 t$$

Thus, the mass of the total tank is given by:

$$m = \rho\left\{L\pi\left[\left(\frac{D}{2}+t\right)^2 - \left(\frac{D}{2}\right)^2\right] + 2\pi\left(\frac{D}{2}+t\right)^2 t\right\}$$

Here, we have multiplied the tank material volume by the steel density, rho.

Weld length is defined by: $\ell_w = 2\left[2\pi\left(\frac{D}{2}+t\right) + 2\pi\frac{D}{2}\right] = 4\pi(D+t)$

Now we calculate the constraints on the problem. Firstly, the volume, V_0, of the tank (i.e., its hollow volume) is fixed.

$$V = \frac{\pi D^2}{4} L$$

The remaining constraints are the bed dimensions of the truck as follows:

$$L \leq L_{\max}$$
$$D \leq D_{\max}$$

Our task is now expressible as follows (mathematically), where 4.5 and 20 are the cost coefficients of the material and the welding, respectively

Minimise $C = 4.5m + 20\ell_w$

Subject to the following conditions:

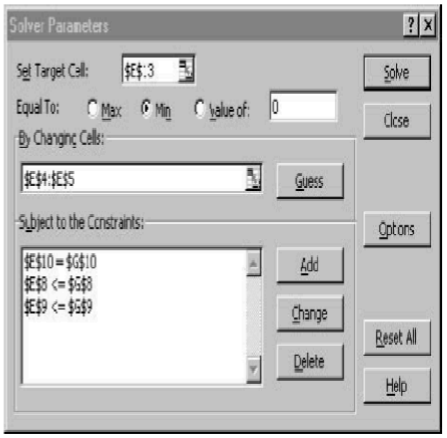
$$\frac{\pi D^2 L}{4} = 0.8$$
$$L \leq 2$$
$$D \leq 1$$

To solve this we can use the in-built solver of Microsoft Excel

Inputs and formulae for this problem in an Excel spreadsheet

	A	B	C	D	E	F	G
1	**Optimum tank design**						
2							
3	**Parameters:**			**Design variables:**			
4	V0	0.8		D	1		
5	t	0.03		L	2		
6	rho	8000					
7	Lmax	2		**Constraints:**			
8	Dmax	1		D	1	<=	1
9	cm	4.5		L	2	<=	2
10	cw	20		Vol	1.570796	=	0.8
11							
12	**Computed values:**			**Objective function:**			
13	m	1976.791		C	9154.425		
14	lw	12.94336					
15							
16	Vshell	0.19415					
17	Vends	0.052948					

Solver Panel where we enter target, cells to be varied (inputs) and constraints.



Now, the result of the optimisation appears in the spreadsheet itself (right panel) where all values have been updated in line with the specifications made in the Solver dialogue box before it ran. The original sheet is on the left below: the new sheet is on the right.

Here, we see that the mass of the tank has been reduced from 1976 kg to 1215 kg, while the weld seam has only been reduced slightly in length from 12.94 m to 12.73 m.

The cost has dropped significantly! (See Cell E13 in both panels). This is an effective way of solving many such constrained multivariate problems relevant to engineering.

	A	B	C	D	E	F	G
	Optimum tank design						
	Parameters:			**Design variables:**			INPUT VALUES
	V0	0.8		D	1		
	t	0.03		L	2		
	rho	8000					
	Lmax	2		**Constraints:**			
	Dmax	1		D	1	<=	1
	cm	4.5		L	2	<=	2
	cw	20		Vol	1.570796	=	0.8
	Computed values:			**Objective function:**			
	m	1976.791		C	9154.425		
	lw	12.94336					
	Vshell	0.19415					
	Vends	0.052948					

	A	B	C	D	E	F	G
1	**Optimum tank design**						
2							
3	**Parameters:**			**Design variables:**			OUTPUT VALUES
4	V0	0.8		D	0.98351		
5	t	0.03		L	1.053033		
6	rho	8000					
7	Lmax	2		**Constraints:**			
8	Dmax	1		D	0.98351	<=	1
9	cm	4.5		L	1.053033	<=	2
10	cw	20		Vo	0.799999	=	0.8
11							
12	**Computed values:**			**Objective function:**			
13	m	1215.206		C	5723.149		
14	lw	12.73614					
15							
16	Vshell	0.100587					
17	Vends	0.051314					

Example 1 Lagrange Multiplier Technique for Multivariate Constrained Optimisation

The use of a Lagrange Multiplier solves a constrained multivariate optimisation problem more efficiently by incorporating the constraint function into the objective function.

A compound expression for the objective and constrain functions gives the following:

$$L(y) = f(x) + \gamma g(x) \qquad f(x) = objective\ function \qquad g(x) = equality\ constraint$$

In this example, the expression for the compound equation contains the objective function plus the equality constraint in brackets multiplied by a constant λ, (the Lagrange multiplier) that needs to be calculated to solve the problem.

$$F^*(x, y, \lambda) = (x-5)^2 + (y-8)^2 + \lambda(xy-5)$$

The optimum point will be found when the components of the main objective equation are differentiated and the differentiated expressions are solved for zero. In this case, the relevant equations of the system are:

$$\frac{\partial F^*}{\partial x} = 2(x-5) + \lambda y = 0$$

$$\frac{\partial F^*}{\partial y} = 2(y-8) + \lambda x = 0$$

$$g(x) = xy - 5 = 0$$

The three equations define the identity of a common optimum point that satisfies each, therefore **we can solve for the optimum point by solving all three equations as a system of simultaneous equations.**

However, we could not use Gauss Elimination in this case, because g(x) uniquely contains a term in xy, whereas the other two equations do not. That is, **it is not a system of linear equations.**

In this case, the Lagrange Multiplier technique can be used: (next page)

When the algorithm is finished in this case, the result of the optimisation gives the co-ordinate values of the point required:

X = [0.6556 (x) 7.6265 (y) 1.1393 (λ)]^T

Example 2 Maximising Revenue of an Engineering Project

The revenue of an engineering project can be expressed by the following equation, where s is the mass of steel used and h is the number of hours of labour employed on the project:

$$R(h, s) = 160\ h^{2/3} s^{1/3}$$

The project has a limited budget of £20,000, and the price of steel is £0.15 per kg, while the labour rate is: £20 per hour.

Use an appropriate optimisation technique to maximise revenue while maintaining costs within the allowable budget.

It is clear that the objective equation is the one given to calculate revenue, and that we need to maximise the value of R while not allowing total cost to exceed £20,000

$$R(h, s) = 160\ h^{2/3} s^{1/3}$$

We now need to express what the constraint is mathematically. It is clear that we need an equation for total cost in terms of the individual costs of steel and labour as follows:

$$20h + 0.15s = 20{,}000$$

This is an equality constraint. We could choose to aim for a budget below £20,000 to save cost, but it makes more sense to maximise revenue by using up all of our budget. This means we have an equality constraint rather than an inequality constraint (less than).

To solve this we can use the Langrange Multiplier Technique.

To do this we need to combine the objective function and the equality constraint into a combined equation as follows:

$$L(y) = f(x) + \gamma g(x)$$

This allows us to write for this case that:

$$R(h, s) = 160\ h^{2/3} s^{1/3} - \lambda(20h + 0.15h - 20{,}000)$$

Now that we have the form of the equation we can use the code we previously used to solve the problem as per the next slide.