Chapter 4 ODE IVP

Euler Method

In [9]:

```
import numpy as np

def odeEuler( df, y0, h, start, end ):
    nSteps = int((end - start) / h)
    y = y0
    t = start
    for i in range(nSteps):
        y = y + h * df(t,y)
        t = t + h
    return y

def df(t,y):
    return t - 2*y*y

start  = 0
end    = 1
h      = 0.2
y0     = 1

print("Euler:y(1) =", odeEuler( df, y0, h, start, end ) )
```

```
odeEuler:y(1) = 0.5637970486321761
```

Euler variant

In [13]:

```
import numpy as np

def odeEulerVariant( df, y0, h, start, end ):
    nSteps = int((end - start) / h)
    y = y0
    t = start
    for i in range(nSteps):
        ybar = y + h     * df(t,y)
        y    = y + h/2.0 * ( df(t,y) + df(t+h,ybar) )
        t    = t + h
    return y

def df(t,y):
    return t - 2*y*y

start  = 0
end    = 1
h      = 0.2
y0     = 1

print("Eulear variant:y(1) =", odeEulerVariant( df, y0, h, start, end ) )
```

```
Eulear variant:y(1) = 0.6296597708927296
```

RK4

In [16]:

```
import numpy as np

def odeRK4( df, y0, h, start, end ):
    nSteps = int((end - start) / h)
    y = y0
    t = start
    for i in range(nSteps):
        k1 = df(t,        y           )
        k2 = df(t+0.5*h,  y+0.5*h*k1 )
        k3 = df(t+0.5*h,  y+0.5*h*k2 )
        k4 = df(t+h,      y+h*k3      )
        y = y + 1.0/6.0*h*( k1 + 2.0 * k2 + 2.0 * k3 + k4)
        t = t + h
    return y

def df(t,y):
    return t - 2*y*y

start  = 0
end    = 1
h      = 0.2
y0     = 1

print("RK4: y(1) =", odeRK4( df, y0, h, start, end ) )
```

```
RK4: y(1) = 0.6189811976695696
```

Chapter 5 Non-linear equations

Bisection method

In [28]:

```
def f(x):
    return math.pow(x,3) - x - 1

def bisect( f,a,b,tolerance ):
    k = 1
    if (f(a) * f(b) < 0):
        while (abs(a-b) / 2.0 > tolerance):
            k = k+1
            xnew=float(a+b) / 2.0
            if (f(a) * f(xnew) < 0):
                b = xnew
            else:
                a = xnew
    else:
        print("f(a)*f(b) is non-negative, find another pair of input")
    return float(a+b)/2.0 , k

xstar, n = bisect( f, 1.0, 2.0, 0.000001 )
print("xstar = ", xstar, ", No. iters = ", n)
```

```
xstar =  1.3247175216674805 , No. iters =  20
```

Newton's method

In [17]:

```
import math

def f(x):
    return math.pow(x,3) - x - 1

def df(x):
    return 3*math.pow(x,2)-1

def Newton(f, df, x0, maxIter, tol):
    for i in range(maxIter):
        x1 = x0 - f(x0)/ df(x0)
        if ( abs(x1-x0) < tol ):
            break
        x0 = x1
        print( "i = ", i, ", x1 = ", x1)
    return x1

xstar = Newton( f, df, 1.5, 100, 0.000001 )
print("xstar = ", xstar)
```

```
i =  0 , x1 =  1.3478260869565217
i =  1 , x1 =  1.325200398950907
i =  2 , x1 =  1.3247181739990537
xstar =  1.3247179572447898
```

Chapter 6 Solution of linear equations

Jacobi iteration

In [19]:

```
import numpy as np
import math
def Jacobi( A, b, xguess, tol, maxIter ):
    n  = len(b)
    x0 = xguess
    nIter = 0
    residual = 10000.0
    while ( residual > tol and nIter < maxIter):
        xold = np.copy(x0)
        x1   = np.copy(xold)
        for i in range(n):
                                # 0:i -> 0, 1, 2, ... , i-1              #i+1:n -> i+1, i+2, ... ,n-1
            x1[i]=(  b[i]-np.dot(A[i,0:i], x0[0:i])-np.dot(A[i,i+1:n],x0[i+1:n])   )/A[i,i]
        x0 = x1
        residual = math.sqrt( (xold-x1).dot(xold-x1) )
        nIter    = nIter + 1
    return x1, residual, nIter

A = np.array([  [ 4, -2, 1 ], [ -2, 17, 1 ], [ 1, 1, 9 ] ])
b = np.array( [ -1, 33, 10 ] )
xguess = np.zeros(3) #np.array([0, 0, 0])
x, residual, nIters = Jacobi(A, b, xguess, 0.005, 100)
print ("final x = ", x, ", nIters = ", nIters, ", final tolerance = ", residual)
print ("check A*x = ", np.dot(A,x), "original b = ", b )
```

```
final x =  [0.51646879 1.9525629  0.836952  ] , nIters =  7 , final tolerance =  0.0
013621417296167519
check A*x =  [-1.00229865 32.99758376 10.00159972] original b =  [-1 33 10]
```

Gauss-Seidel

In [20]:

```
import numpy as np
import math
def GaussSeidelIteration( A, b, xguess, tol, maxIter ):
    n  = len(b)
    x0 = xguess
    nIter = 0
    residual = 10000.0
    while ( residual > tol and nIter < maxIter):
        xold = np.copy(x0)
        for i in range(n):
            x0[i]=(b[i]-(A[i,0:i]).dot(x0[0:i])-(A[i,i+1:n]).dot(x0[i+1:n]))/A[i,i]
        residual = math.sqrt( (xold-x0).dot(xold-x0) )
        nIter    = nIter + 1
    return x0, residual, nIter
A = np.array([  [ 4, -2, 1 ],[ -2, 17, 1 ], [ 1, 1, 9 ] ])
b = np.array( [ -1, 33, 10 ] )

xguess = np.zeros(3)
x, residual, nIters = GaussSeidelIteration(A, b, xguess, 0.005, 100)
print ("x = ", x, ", nIters = ", nIters, ", final tolerance = ", residual)
print ("check A*x = ", A.dot( x ), "original b = ", b )
```

```
x =  [0.5171443  1.9527956  0.83667334] , nIters =  5 , final tolerance =  0.0007016
825394534449
check A*x =  [-1.00034068 32.99991002 10.        ] original b =  [-1 33 10]
```

Gauss Elimination

In [23]:

```
import numpy as np
def GaussElimination( A, b ):
    n = len(b)
    M = np.zeros(shape=(n,n+1))
    M[0:n,0:n] = A
    M[0:n,n]   = b
    print ("M before Gauss Elimination")

    print ("Start Elimination")
    #elimination
    for col in range(0,n-1):
        for row in range(col+1,n):
            #print ("col = ", col, "row = ", row)
            coef            = M[row,col] / M[col,col]
            M[row,col:n+1] = M[row,col:n+1] - coef * M[col,col:n+1]
        print (M)

    print ("Start back substitution")
    #back substitution
    x = np.zeros(3)
    x[n-1] = M[n-1,n] / M[n-1,n-1]
    for row in range(n-2,-1,-1):
        print (row)
        x[row] = (M[row,n]-M[row,row+1:n].dot(x[row+1:n]))/M[row,row]
    return x

A = np.array([  [ 8.0, -3.0, 2.0 ],
                [ 4.0,  11.0, -1.0 ],
                [ 6.0,  3.0, 12.0 ]  ])
b = np.array( [ 20.0, 33.0, 36.0 ] )
x = GaussElimination(A, b)
print ("x = ", x)
print ("check A*x = ", A.dot( x ), "original b = ", b)
```

```
M before Gauss Elimination
Start Elimination
[[ 8.   -3.    2.    20.   ]
 [ 0.   12.5  -2.    23.   ]
 [ 0.    5.25 10.5   21.   ]]
[[ 8.   -3.    2.    20.   ]
 [ 0.   12.5  -2.    23.   ]
 [ 0.    0.   11.34 11.34]]
Start back substitution
1
0
x =  [3. 2. 1.]
check A*x =  [20. 33. 36.] original b =  [20. 33. 36.]
```

LU decomposition

In [22]:

```
import numpy as np

def LUdecomposition( A, b ):
    n = len(b)
    L = np.zeros(shape=[n,n])
    U = np.zeros(shape=[n,n])

    #set diagonals of L
    for i in range(n):
        L[i,i] = 1.0

    # set first row of U
    for j in range(n):
        U[0,j] = A[0,j]

    # set first column of L
    for i in range(1,n):
        L[i,0] = A[i,0] / U[0,0]

    # interatively compute k-th row of U and k-th col of L
    for k in range(1,n):
        for j in range(k,n):
            U[k,j] = A[k,j] - (L[k,0:k]).dot( U[0:k,j] )

        for i in range(k+1,n):
            L[i,k] = ( A[i,k] - (L[i,0:k]).dot(U[0:k,k]) ) / U[k,k]

    # now that L and U are computed, do back substitution
    # solve Ly = b
    y = np.zeros(n)
    y[0] = b[0] / L[0,0]
    for i in range(1,n):
        y[i] = ( b[i] - (L[i,0:i]).dot( y[0:i] ) ) / L[i,i]

    x = np.zeros(n)

    x[n-1] = y[n-1] / U[n-1,n-1]
    for i in range(n-2,-1,-1):
        x[i] = (y[i] - U[i,i+1:n].dot(x[i+1:n] ) ) / U[i, i]

    return L, U, y, x


A = np.array([  [ -2.0, 4.0, 8.0   ],
                [ -4.0, 18.0, -16.0],
                [ -6.0, 2.0, -20.0 ],
              ])
b = np.array( [ 5.0, 8.0, 7.0 ] )

L, U, y, x = LUdecomposition(A, b)

print ("L = ", L)
print (" ")
print ("U = ", U)
print (" ")
print ("y = ", y)
print ("")
print ("x = ", x)
print (" ")
```

```
print ("check A*x = ", A.dot( x ), "original b = ", b)
```

```
L =  [[ 1.  0.  0.]
 [ 2.  1.  0.]
 [ 3. -1.  1.]]

U =  [[ -2.   4.   8.]
 [  0.  10. -32.]
 [  0.   0. -76.]]

y =  [  5.  -2. -10.]

x =  [-1.53157895  0.22105263  0.13157895]

check A*x =  [5. 8. 7.] original b =  [5. 8. 7.]
```

Chapter 8 Matrix eigenvalue problem

power iteration

In [29]:

```
import numpy as np
import math

def EigPow(A):
    ep = 1e-12
    n = len(A)
    u = np.ones([n,1])
    k = 0
    residual = 10000
    while (k<6 and residual > ep):
        v = A.dot(u)
        m = max(v,key=abs)
        v = v/max(abs(v))

        residual = np.linalg.norm(u-v,ord=2)
        u = v
        k = k+1
    return m,v, k

[m,v,k] = EigPow(A)
print("Power Iteration: MAX EigenValue =",m,"\nEigenVector =\n",v, "total iteration:",k)
print(" ")

A = np.array([[8.0 , 4.0 , 2.0],
              [4.0 , 1.0 , 3.0],
              [2.0 , 3.0 , 7]])

w,v = np.linalg.eig(A)
print("actrual eignvalues =", w)
print("actrual eigenvectors =",v)
```

```
Power Iteration: MAX EigenValue = [13.57158336]
EigenVector =
 [[0.21814475]
 [0.11102957]
 [1.        ]] total iteration: 6

actrual eignvalues = [11.86249379  5.43934373 -1.30183752]
actrual eigenvectors = [[-0.71987459 -0.60651638  0.33751807]
 [-0.41810335 -0.00923866 -0.90835249]
 [-0.55404888  0.79501731  0.24693585]]
```

In [30]:

```
def EigJacobi(A):
    n = len(A)
    R = np.eye(n)
    Amax = 500
    tol = 1e-5
    nIters = 0

    while (Amax > tol and nIters < 500):
        Amax = 0
        nIters = nIters + 1

        #find the largest element on the off-diagonal line of the matrix
        for l in range(n):
            for k in range(l+1,n):
                if (abs(A[l,k])>Amax):
                    Amax = abs(A[l,k])
                    i=l
                    j=k

        #compute rotate angle
        y = abs(A[i,i]-A[j,j])
        x = np.sign(A[i,i]-A[j,j])*2*A[i,j]
        c = math.sqrt(0.5*(1+y/math.sqrt(pow(x,2)+pow(y,2))))
        s = x/(2*c*math.sqrt(pow(x,2)+pow(y,2)))

        #sort eigenvalues&compute eigenvectors
        for l in range(n):
            if (l==i):
                Aii = A[i,i]*pow(c,2)+A[j,j]*pow(s,2)+2*A[i,j]*s*c
                Ajj = A[i,i]*pow(s,2)+A[j,j]*pow(c,2)-2*A[i,j]*s*c
                A[i,j] = (A[j,j]-A[i,i])*s*c+A[i,j]*(pow(c,2)-pow(s,2))
                A[j,i] = A[i,j]
                A[i,i] = Aii
                A[j,j] = Ajj
            elif (l!=j):
                Ail =  A[l,i]*c+A[l,j]*s
                Ajl = -A[l,i]*s+A[l,j]*c
                A[i,l]=Ail
                A[l,i]=Ail
                A[j,l]=Ajl
                A[l,j]=Ajl
            Rli = R[l,i]*c+R[l,j]*s
            Rlj = -R[l,i]*s+R[l,j]*c
            R[l,i] = Rli
            R[l,j] = Rlj
    D = np.diag(A)

    return D,R


A = np.array([[8.0 , 4.0 , 2.0],
              [4.0 , 1.0 , 3.0],
              [2.0 , 3.0 , 7]])

[D,R]=EigJacobi(A)
print("Jacobi Eigenvalues =\n",D,"\nEigenvectors =\n",R)
print(' ')
```

```
Jacobi Eigenvalues =
 [11.86249379 -1.30183752  5.43934373]
Eigenvectors =
 [[ 0.71987459 -0.33751807 -0.60651638]
 [ 0.41810335  0.90835249 -0.00923866]
 [ 0.55404888 -0.24693585  0.79501731]]
```

In [31]:

```
def EigQR(A):
    q,r = np.linalg.qr(A)
    A0 = np.copy(A)

    for i in range(9):
        A   = r.dot(q)
        q, r=np.linalg.qr(A)

    gamma0 = np.diag(A)
    gamma  = np.copy(gamma0)
    n = len(gamma0)
    vectors = np.zeros((n,n))
    #calculate the eigenvectors by inverse power method
    for i in range(n):
        Ai = A0-gamma0[i]*np.eye(n)
        invAi = np.linalg.inv(Ai)
        residual = 10000
        u = np.ones([n,1])
        k = 0
        ep = 1e-12
        while (k<500 and residual > ep):
            v = invAi.dot(u)
            m = max(v,key=abs)
            v = v/max(abs(v))
            residual = np.linalg.norm(u-v,ord=2)
            u = v
            k = k+1
        gamma[i] = gamma0[i]+(1/m)
        vectors[:,i]=v[:,0]

    return gamma,vectors

A = np.array([[8.0 , 4.0 , 2.0],
              [4.0 , 1.0 , 3.0],
              [2.0 , 3.0 , 7]])

[gamma,v] = EigQR(A)
print("QR: eigenvalues =\n",gamma,"\n","eigenvectors =\n",v)
print(' ')
```

```
QR: eigenvalues =
 [11.86249379  5.43935105 -1.30183752]
 eigenvectors =
 [[ 1.         -0.76289708 -0.37157169]
 [ 0.58080025 -0.01162071  1.        ]
 [ 0.76964638  1.         -0.27185025]]
```

In [ ]:

```
```