MTH 452 Final Project

Kewalin Samart

Solve the boundary value problem using Finite Difference method and Finite Element method with piecewise linear function, respectively. Compare the efficiency of the schemes.

$$-\frac{d}{dx}(xy') + 4y = 4x^2 - 8x + 1, \qquad 0 \le x \le 1, \tag{1}$$

with boundary condition $y(0) = y(1) = 0$. The exact solution is known as $y = x^2 - x$.

In [1]:
```python
# Importing libraries
import numpy as np
import matplotlib.pyplot as plt
```

In [2]:
```python
# Defining function for the exact solution
def exact_sol(x):
    exact = x**2 - x
    return exact
```

Linear Finite Difference Method

- To approximate the solution of the boundary-value problem
$$y'' = p(x)y' + q(x)y + r(x),$$
  - for $a \le x \le b$, with $y(a) = \alpha$ and $y(b) = \beta$
- Here we want to pproximate
$$y'' = (\frac{-1}{x})y' + (\frac{4}{x})y + \frac{-4x^2 + 8x - 1}{x}$$
  - for $0 \le x \le 1$ with $y(0) = y(1) = 0$
  - using $N = 9$ so $h = 0.1$

In [3]:

```
# Defining functions for polynomial p, q, and r
def p(x):
    p_ = -1/x
    return p_

def q(x):
    q_ = 4/x
    return q_

def r(x):
    r_ = (-4*x**2+8*x-1)/x
    return r_
```

Algorithm 11.3: Linear Finite Difference Method

In [4]:

```
# Defining variables

a = 0 # left endpoint
b = 1 # right endpoint

alpha = 0 # boundary condition at y(0)
beta = 0 # boundary condition at y(1)

N = 9
h = (b-a)/(N+1)
x = a + h
a1 = 2 + (h**2)*q(x)
b1 = -1 + (h/2)*p(x)
d1 = -h**2*r(x) + (1 + (h/2)*p(x))*alpha
```

In [5]:

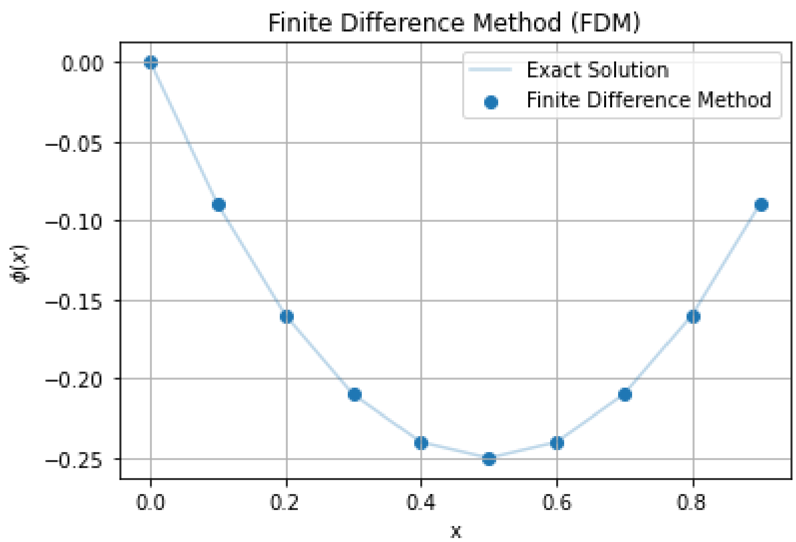
```python
## Step 1
a_list = [None, a1]
b_list = [None, b1]
c_list = [None, None]
d_list = [None, d1]

## Step 2
for i in range(2,N):
    x = a + i*h
    ai = 2 + (h**2)*q(x)
    bi = -1 + (h/2)*p(x)
    ci = -1 - (h/2)*p(x)
    di = (-h**2)*r(x)

    a_list.append(ai)
    b_list.append(bi)
    c_list.append(ci)
    d_list.append(di)

## Step 3
x = b - h
aN = 2 + (h**2)*q(x)
cN = -1 - (h/2)*p(x)
dN = (-h**2)*r(x) + (1-(h/2)*p(x))*beta

a_list.append(aN)
c_list.append(cN)
d_list.append(dN)

## Step 4
l1 = a1
u1 = b1/a1
z1 = d1/l1

## Step 5
l_list = [None, l1]
u_list = [None, u1]
z_list = [None, z1]

for i in range(2,N):
    li = a_list[i] - c_list[i]*u_list[i-1]
    l_list.append(li)

    ui = b_list[i]/l_list[i]
    u_list.append(ui)

    zi = (d_list[i] - c_list[i]*z_list[i-1])/l_list[i]
    z_list.append(zi)

## Step 6
lN = a_list[N] - c_list[N]*u_list[N-1]
l_list.append(lN)
zN = (d_list[N] - c_list[N]*z_list[N-1])/l_list[N]

## Step 7
w0 = alpha
```

```python
wNp1 = beta
wN = zN

w_list = [w0]
w_list.extend([None for i in range(N-1)])
w_list.append(wN)

## Step 8
for i in reversed(range(1,N)):
    wi = z_list[i] - u_list[i]*w_list[i+1]
    w_list[i] = wi

## Step 9
print("i, x, wi, yi, absolute error")
exact_sol_ = []
x_list = []
error_ = []
for i in range(N+1):
    x = a + i*h
    print(i, x, w_list[i], exact_sol(x), abs(w_list[i]-exact_sol(x)))
    exact_sol_.append(exact_sol(x))
    x_list.append(x)
    error_.append(abs(w_list[i]-exact_sol(x)))
```

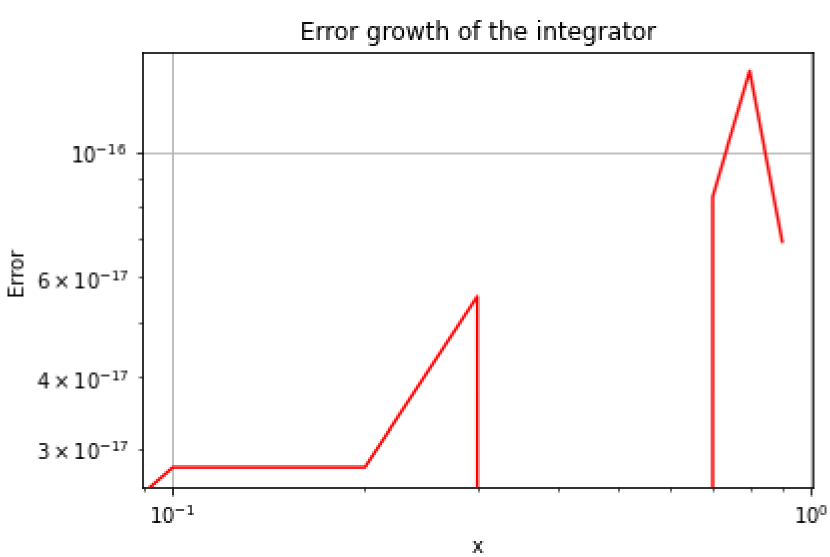
```
i, x, wi, yi, absolute error
0 0.0 0 0.0 0.0
1 0.1 -0.08999999999999997 -0.09 2.7755575615628914e-17
2 0.2 -0.15999999999999998 -0.16 2.7755575615628914e-17
3 0.30000000000000004 -0.20999999999999996 -0.21000000000000002 5.551115123125783e-17
4 0.4 -0.24 -0.24 0.0
5 0.5 -0.25 -0.25 0.0
6 0.6000000000000001 -0.24 -0.24 0.0
7 0.7000000000000001 -0.21000000000000005 -0.20999999999999996 8.326672684688674e-17
8 0.8 -0.16000000000000006 -0.15999999999999992 1.3877787807814457e-16
9 0.9 -0.09000000000000004 -0.08999999999999997 6.938893903907228e-17
```

In [6]:
```
# Plotting the approx soultion using algorithm 11.3 vs the exact solutio
n
plt.grid()
plt.scatter(x_list,w_list,label='Finite Difference Method')
plt.plot(x_list,exact_sol_,alpha=0.3,label='Exact Solution')
plt.xlabel(r'x')
plt.ylabel(r'$\phi(x)$')
plt.title('Finite Difference Method (FDM)')
plt.legend()
```

Out[6]: <matplotlib.legend.Legend at 0x7ff0e8d388b0>



In [7]:
```
# Plotting Error Growth
plt.grid()
plt.loglog(x_list,error_, 'red')
plt.title('Error growth of the integrator')
plt.xlabel('x')
plt.ylabel('Error')
plt.show()
```



Finite Element Method with piecewise linear function

- To approximate the solution to the boundary-value problem
$$-\frac{d}{dx}(p(x)\frac{dy}{dx}) + q(x)y = f(x)$$
  - for $0 \leq x \leq 1$ with $y(0) = 0$ and $y(1) = 0$
  - with piecewise linear function
$$\phi(x) = \sum_{i=1}^{n} c_i \phi_i(x)$$
- Here we want to approximate
$$-\frac{d}{dx}(xy') + 4y = 4x^2 - 8x + 1$$
  - for $0 \leq x \leq 1$ with $y(0) = y(1) = 0$
  - using $n = 9$ so $h = 0.1$

In [8]:
```
# Defining functions for polynomial p, q, and f
def p(x):
    return x

def q(x):
    return 4

def f(x):
    f_ = 4*(x**2) -8*x +1
    return f_
```

Algorithm 11.5: Finite Element method with piecewise linear function

In [9]:
```
# Defining x_list and h_list
n = 9 # partition into 9 subinterval
h = 0.1 # with mesh = 0.1
x_list = list(np.round_(np.arange(0.1,1.1,h), decimals=1))
h_list = [0.1 for i in range(10)]
print("xi:", x_list)
print("hi:", h_list)
```

```
xi: [0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0]
hi: [0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1]
```

In [10]:

```
# Defining the piecewise linear basis phi i
def phi(x_list, h_list, n):
    phi = np.zeros([n+1, n+1], dtype='float')
    for i in range(n-1):
        for j in range(n):
            if (0<=x_list[j]<=x_list[i-1]):
                phi[i, j] = 0
            elif (x_list[i-1]<x_list[j]<=x_list[i]):
                phi[i, j] = (x_list[j] - x_list[i-1])/h_list[i-1]
            elif (x_list[i]<x_list[j]<=x_list[i+1]):
                phi[i, j] = ((x_list[i+1] - x_list[j])/h_list[i])
            elif (x_list[i+1]<x_list[j]<=1):
                phi[i, j] = 0
    return phi
```

In [11]:

```
# Initializing vectors for computing integrals
Q1 = []
Q2 = []
Q3 = []
Q4 = []
Q5 = []
Q6 = []

# Approximating the 6 integrals
for i in range(n-1):
    q1 = (h_list[i]/12)*(p(x_list[i]) + q(x_list[i+1]))
    q2 = (h_list[i-1]/12)*(3*q(x_list[i]) + q(x_list[i-1]))
    q3 = (h_list[i]/12)*(3*q(x_list[i]) + q(x_list[i+1]))
    q4 = (h_list[i-1]/2)*(p(x_list[i]) + p(x_list[i-1]))
    q5 = (h_list[i-1]/6)*(2*f(x_list[i]) + f(x_list[i-1]))
    q6 = (h_list[i]/6)*(2*f(x_list[i]) + f(x_list[i+1]))

    Q1.append(q1)
    Q2.append(q2)
    Q3.append(q3)
    Q4.append(q4)
    Q5.append(q5)
    Q6.append(q6)
```

In [12]:
```
# Computing  Q1,n , Q2,n , Q3,n , Q4,n Q4,n+1, Q5n, Q6,n
q1n = (h_list[n]/12)*(p(x_list[n-1]) + q(x_list[n]))
q2n = (h_list[n-2]/12)*(3*q(x_list[n-1]) + q(x_list[n-2]))
q3n = (h_list[n]/12)*(3*q(x_list[n-1]) + q(x_list[n]))
q4n_last = (h_list[n-1]/2)*(p(x_list[n]) + p(x_list[n-1]))
q4n_second_last = (h_list[n-2]/2)*(p(x_list[n-1]) + p(x_list[n-2]))
q5n = (h_list[n-1]/6)*(2*r(x_list[n-1]) + r(x_list[n-2]))
q6n = (h_list[n]/6)*(2*r(x_list[n-1]) + r(x_list[n]))

Q1.append(q1n)
Q2.append(q2n)
Q3.append(q3n)
Q4.append(q4n_last)
Q4.append(q4n_second_last)
Q5.append(q5n)
Q6.append(q6n)
```

In [13]:
```
# Solving a symmetric tridiagonal linear system

alpha = np.zeros(n+1, dtype='float')
beta = np.zeros(n+1, dtype='float')
b = np.zeros(n+1, dtype='float')
a = np.zeros(n+1, dtype='float')

for i in range(n-1):
    alpha[i] = Q4[i] + Q4[i+1] + Q2[i] + Q3[i]
    beta[i] = Q1[i] - Q4[i+1]
    b[i] = Q5[i] + Q6[i]

alpha[n] = Q4[n-1] + Q4[n] + Q2[n-1] + Q3[n-1]
b[n] = Q5[n-1] + Q6[n-1]
a[0] = alpha[0]
zeta = np.zeros(n+1, dtype='float')
z = np.zeros(n+1, dtype='float')
c = np.zeros(n+1, dtype='float')
zeta[0] = beta[0]/alpha[0]
z[0] = b[0]/a[0]

for i in range(1, n-1):
    a[i] = alpha[i] - beta[i-1]*zeta[i-1]
    zeta[i] = beta[i]/a[i]
    z[i] = (b[i] - beta[i-1]*z[i-1])/a[i]

a[n] = alpha[n] - beta[n-1]*zeta[n-1]
z[n] = (b[n] - beta[n-1]*z[n-1])/a[n]
```

In [14]:

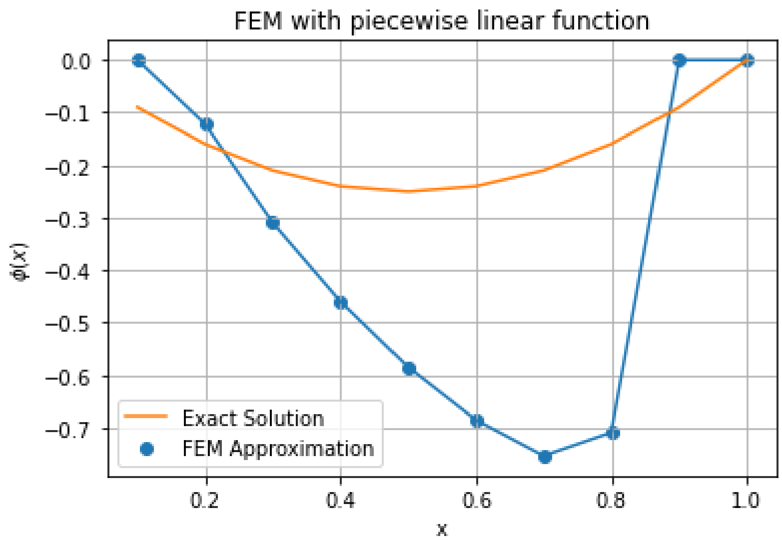
```
# Indices for the weights
indices = []
for i in range(n):
    a =  n-1-i
    indices.append(a)

c[n] = z[n]

for i in indices:
    c[i] = z[i] - zeta[i]*c[i+1]
u = phi(x_list, h_list, n)
```

In [15]:

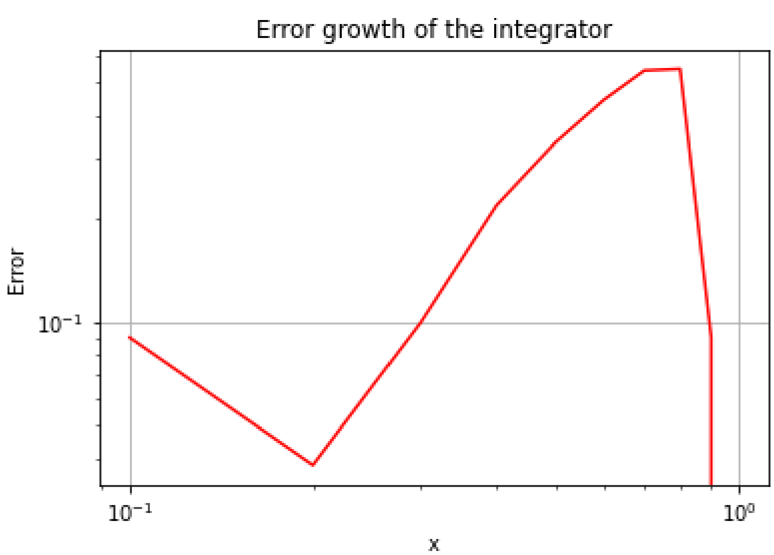
```
# Calculating the absolute error
error = np.zeros(n+1, dtype='float')
phi_new = np.dot(u, c)
exact_sol_ = []
print('i, xi, phi(xi), y(xi), absolute error')
for i in range(len(x_list)):
    error[i] = (abs(phi_new[i]-exact_sol(x_list[i])))
    print(i, x_list[i], phi_new[i], exact_sol(x_list[i]), error[i])
    exact_sol_.append(exact_sol(x_list[i]))
```

```
i, xi, phi(xi), y(xi), absolute error
0 0.1 0.0 -0.09 0.09
1 0.2 -0.12180191912276547 -0.16 0.03819808087723453
2 0.3 -0.3093849619216495 -0.21 0.0993849619216495
3 0.4 -0.45947189724021054 -0.24 0.21947189724021055
4 0.5 -0.5849694213849195 -0.25 0.3349694213849195
5 0.6 -0.68532506842679 -0.24 0.44532506842679
6 0.7 -0.7533273094046016 -0.21000000000000002 0.5433273094046016
7 0.8 -0.7091470883671723 -0.15999999999999992 0.5491470883671724
8 0.9 0.0 -0.08999999999999997 0.08999999999999997
9 1.0 0.0 0.0 0.0
```

In [16]:
```
# Plotting the approx solution by Algorithm 11.5 vs the exact solution
plt.grid()
plt.plot(x_list, phi_new)
plt.scatter(x_list, phi_new, label='FEM Approximation')
plt.plot(x_list, exact_sol_, label='Exact Solution')
plt.legend()
plt.xlabel(r'x')
plt.ylabel(r'$\phi(x)$')
plt.title('FEM with piecewise linear function')
plt.show()
```



In [17]:
```
# Plotting error growth
plt.grid()
plt.loglog(x_list,error, 'red')
plt.title('Error growth of the integrator')
plt.xlabel('x')
plt.ylabel('Error')
plt.show()
```



Discussion

- Theorically, the finite element method (FEM) with piecewise linear function is supposed to perform better than the finite difference method (FDM) especially when performing with higher dimension problems.
- According to the results of the algorithm 11.3 and 11.5 above, the finite difference method performs better in approximating the solution to the given boundary condition problem compared to the finite element method with piecewise linear function (as we can observe the absolute errors). The huge errors might come from the formulas for Q that were used to approximate the actual integrations instead of computing the integrals directly.
- To further test the accuracy of the written algorithms, trying to apply the algorithms with more problems with different dimensions is a good idea to investigate the dimension effects influencing the accuracy of the results.

References

- Burden, Richard L., and J. Douglas Faires. Numerical Analysis. Brooks/Cole Pub. Co., 2011.
- Lecture video MTH 452 April 19^{th}, 2021. https://www.youtube.com/watch?v=y-cr-j0RD6c (https://www.youtube.com/watch?v=y-cr-j0RD6c)
- Some pieces of code from https://github.com/readikole/Picewise-linear-Rayleigh-Ritz/blob/main/Rayleigh-Ritz%20Method.py (https://github.com/readikole/Picewise-linear-Rayleigh-Ritz/blob/main/Rayleigh-Ritz%20Method.py)