

# Math 503 computing assignment 1

## FEM and FDM solution for $u''(x) + q u(x) = f$

by Nasser Abbasi

Project for Mathematics 503, Summer 2007. California State University, Fullerton, CA.

### (a) Purpose and design of project

In this project we are asked to implement the Finite Element Methods (FEM) and Finite difference method (FDM) to find the solution  $u(x)$  for the following differential equation, and also compare both methods.

$$-u''(x) + q u(x) = f \quad \text{where} \quad u(0) = 0, u(1) = 0 \quad (1)$$

The 2 methods are compared for speed of convergence to the exact solution. The exact analytical solution is known for this simple differential equation:

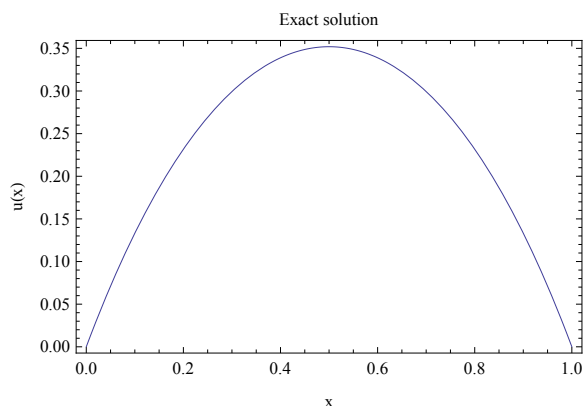
```
Clear[u, x, f, q];
sol = First@DSolve[{-u''[x] + q u[x] == f, u[0] == 0, u[1] == 0}, u[x], x]
```

$$\left\{ u[x] \rightarrow - \frac{e^{-\sqrt{q} x} \left( -1 + e^{\sqrt{q} x} \right) \left( -e^{\sqrt{q}} + e^{\sqrt{q} x} \right) f}{\left( 1 + e^{\sqrt{q}} \right) q} \right\}$$

```
sol = (u[x] /. %) /. {q -> 4, f -> 4}
```

$$- \frac{e^{-2x} \left( -1 + e^{2x} \right) \left( -e^2 + e^{2x} \right)}{1 + e^2}$$

```
Plot[sol, {x, 0, 1}, ImageSize -> 300, FrameLabel -> {"x", "u(x)", "Exact solution"},
Frame -> True]
```



The numerical solution for  $u(x)$  at each grid point is compared to the true solution, then the maximum error using each method is found. Also the RMSerror is calculated.

A brief overview of the FEM and FDM scheme used is now discussed.

### FEM method

For the FEM, the variational approach is used (as contrasted by the Galerkin method). In the variational method, we seek to find a solution  $y(x)$  to a functional  $J(y(x))$  defined by an integral such that this solution  $y(x)$  minimizes this functional. This solution will be the solution to the differential equation itself. However, we do not use nor try to find the differential equation at all in this method. We work directly on the first variation equation  $J'(y; \phi) = 0$  itself by finding  $y(x)$  such that  $J'(y; \phi) = 0$  for all the  $\phi_i$  permissible directions. In the Galerkin method, we are given the differential equation, and then we substitute equation (2) below into the differential equation itself.

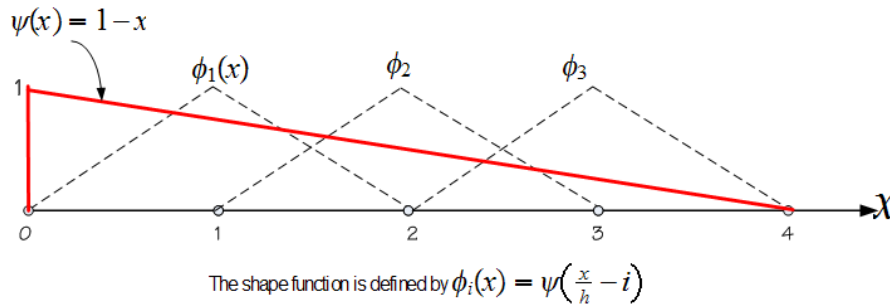
In this problem the  $\phi$  functions are also the basis for the vector space in which the solution  $y(x)$  defined in. In other words, these  $\phi_i$  represent a basis for the  $V$  space of  $y(x)$  and we seek a solution

$$u(x) = \sum_{i=1}^N c_i \phi_i(x) \quad (2)$$

Which satisfies  $J'(y; \phi_i) = 0$  for all the basis functions  $\phi_i$ .  $\phi_i$  is defined by

$$\phi_i(x) = \psi\left(\frac{x}{h} - i\right) \quad (3)$$

Where  $i$  is the shape function number  $i = 1, 2, 3 \dots n$  and  $\psi(x) = 1 - x$  for  $0 < x < 1$ , and  $\psi(x) = 0$  for  $x > 1$  and  $\psi(x) = \psi(-x)$ . The relationship between  $\phi_i(x)$  and  $\psi(x)$  is illustrated in this diagram



In this problem the functional we want to minimize is given

$$J(u) = \int_0^1 (u'(x))^2 + q u(x)^2 - 2 f u(x) dx \quad (4)$$

Where  $q > 0$  and  $f$  are given constants. In the first part of this project we found that  $J(u)$  achieves a minimum iff  $J'(y; \phi) = 0$  where  $J'(y; \phi)$  is given by

$$J'(y; \phi) = \int_0^1 u'(x) \phi'(x) + q u(x) \phi(x) - f \phi(x) dx \quad (5)$$

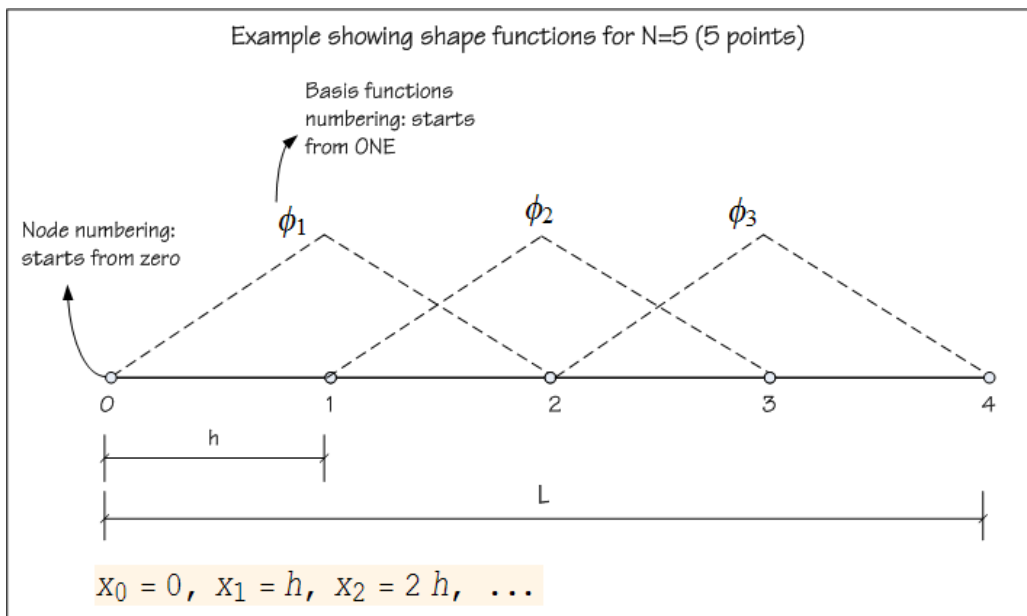
We start the FEM by setting up the equations  $J'(y; \phi_i) = 0$ , therefore, we will have  $N$  number of equations, where  $N$  is the number of the shape functions  $\phi$ . Hence for each  $j^{\text{th}}$  shape function  $\phi_j$  we would have

$$J'(y; \phi_j) = 0$$

$$\int_0^1 \left( \sum_{i=1}^N c_i \phi'_i(x) \right) \phi'_j(x) + q \left( \sum_{i=1}^N c_i \phi_i(x) \right) \phi_j(x) - f \phi_j(x) dx = 0 \quad j = 1 \dots N$$

The above will generate  $N$  algebraic equations which we will then solve for the  $c_i$  coefficients. Once the  $c_i$  is found then we can determine the FEM solution  $u(x)$  from (2) above at any  $x$ .

It is important to agree on the numbering scheme of nodes, elements, and shape functions. The following diagram illustrates the numbering used.



This below plots the shape functions for illustration.

```

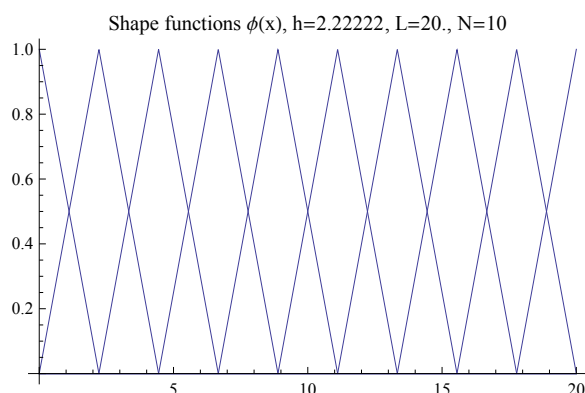
L = 20.; nPoints = 10; nElements = nPoints - 1; q = 4; f = 4;

nShapeFunctions = nPoints;

h =  $\frac{L}{nElements}$ ;

Plot[Table[ $\phi[i, x, h]$ , {i, 1, nShapeFunctions}], {x, 0, L}, PlotRange -> All,
  AxesOrigin -> {0, 0},
  PlotLabel -> "Shape functions  $\phi(x)$ , h=" <> ToString[N[h]] <> ", L=" <>
    ToString[L] <> ", N=" <> ToString[nPoints], ImageSize -> 300]

```



When generating each equation  $J'(y; \phi_j) = 0$  for each basis  $\phi_j$  we take advantage of the domain of influence of each specific  $\phi_j$ , we see from above that  $\phi_j$  is nonzero only over the range  $x_{j-1} \dots x_{j+1}$ . The program calls a function to generate one equation for each basis. This function performs the integration shown in equation (4) but will only do the integration over  $x_{j-1} \dots x_{j+1}$  since  $\phi_j=0$  elsewhere.

Once the N equations are computed, We solve  $Ax=b$  to find  $x$  where  $x$  here is the vector which contain  $c_j$  coefficients as shown in equation (2) above. The matrix A will be tridiagonal in this problem since for each basis  $\phi_j$  (other than the first and the last basis) we see that it overlaps with 2 other basis  $\phi_{j-1}$  and  $\phi_{j+1}$  hence this causes an equation with 3 unknowns to be generated ( $c_{j-1}, c_j, c_{j+1}$ ). This below is an small function will shows the equations for  $N=8$  and shows the A matrix to make this point more clear.

```

L = 20.; nPoints = 8; nElements = nPoints - 1; h =  $\frac{L}{nElements}$ ;
nShapeFunctions = nPoints; q = 4; f = 4; leftBC = 0; rightBC = 0;
coeffShapeFunctions = Array["c", nShapeFunctions];
grid = N[Range[0, L, h]];
(FEMeqs = Table[makeEquation[i, h, coeffShapeFunctions, q, f, grid, nPoints] == 0,
  {i, 1, nShapeFunctions}]) // TableForm

-5.714285714 + 4.159523809 c[1] + 1.554761906 c[2] == 0
-11.42857136 + 1.554761906 c[1] + 8.31904755 c[2] + 1.554761908 c[3] == 0
-11.4285713 + 1.554761908 c[2] + 8.31904748 c[3] + 1.55476191 c[4] == 0
-11.42857123 + 1.55476191 c[3] + 8.319047411 c[4] + 1.554761912 c[5] == 0
-11.42857117 + 1.554761912 c[4] + 8.319047342 c[5] + 1.554761914 c[6] == 0
-11.4285711 + 1.554761914 c[5] + 8.319047273 c[6] + 1.554761916 c[7] == 0
-11.42857104 + 1.554761916 c[6] + 8.319047203 c[7] + 1.554761918 c[8] == 0
-5.714285486 + 1.554761918 c[7] + 4.159523568 c[8] == 0

```

Which in  $Ax = b$  format, the above becomes

```

{b, A} = CoefficientArrays[FEMeqs, coeffShapeFunctions];
A[[1, 2 ;; -1]] = 0;
A[[-1, 1 ;; -2]] = 0;
b[[1]] = A[[1, 1]] * leftBC;
b[[-1]] = A[[-1, -1]] * rightBC;
For[i = 2, i ≤ nPoints - 1, i++,
{
  b[[i]] = b[[i]] - A[[i, 1]] * leftBC - A[[i, -1]] * rightBC;
  A[[i, 1]] = 0;
  A[[i, -1]] = 0;
}
];
MatrixForm[A]

```

```

( 4.159523809      0.      0.      0.      0.      0.      0.
   0.      8.31904755  1.554761908  0.      0.      0.      0.
   0.      1.554761908  8.31904748  1.55476191  0.      0.      0.
   0.      0.      1.55476191  8.319047411  1.554761912  0.      0.
   0.      0.      0.      1.554761912  8.319047342  1.554761914  0.
   0.      0.      0.      0.      1.554761914  8.319047273  1.554761916
   0.      0.      0.      0.      0.      1.554761916  8.319047273
   0.      0.      0.      0.      0.      0.      0.      0. )

```

The above shows that A is diagonally dominant, and tridiagonal. Hence a tridiagonal solver is used since it is much faster than Gaussian elimination in this case. In FEM, the generate A matrix will always contain diagonal bands as the above and these matrices are sparse in nature.

## FDM

In the FDM method, the central difference method is used to approximate  $u''(x)$ . Hence in this method we already know the differential equation and we work directly on the differential equation, while in the FEM method above, we do not know necessarily what the differential equation is and work directly on the first variational term.

The central difference scheme is given by

$$u_i'' = \frac{u_{i+1} - 2u_i + u_{i-1}}{h^2} \quad (6)$$

Where  $u_i$  means  $u$  at grid point  $x_i$ . Now we substitute the above equation directly into the differential equation  $-u''(x) + q u(x) = f$  and obtain

$$-\left(\frac{u_{i+1} - 2u_i + u_{i-1}}{h^2}\right) + q u_i = f$$

$$u_{i-1} + (2 + qh^2)u_i - u_{i+1} = fh^2$$

Hence for each point (we start from the first internal grid point  $i = 1$  and not from the boundary point  $i = 0$ ) and hence for each such internal point, we see that we have 3 unknown. Hence the general  $Ax=b$  equations will also be tridiagonal. This is illustrate by this short example below. Notice that 2 equations are inserted manually into the set of the generated equations since  $u_0 = 0$  and  $u_n = 0$ , since these are boundary conditions given.

```

L = 1.; nPoints = 8; nElements = nPoints - 1; q = 4; f = 4;
init[ nPoints, q, f, L, 0, 0];
u = Array["u", nPoints];

FDMeqs = Table[ $-\left(\frac{u[[i+1]] - 2 u[[i]] + u[[i-1]]}{h^2}\right) + q u[[i]] = f, \{i, 2, nPoints - 1\}$ ];

eq = u[[1]] == leftBC; FDMeqs = Append[FDMeqs, eq];
eq = u[[nPoints]] == rightBC; FDMeqs = Append[FDMeqs, eq];
FDMeqs // TableForm

4 u[2] - 0.1225 (u[1] - 2 u[2] + u[3]) == 4
4 u[3] - 0.1225 (u[2] - 2 u[3] + u[4]) == 4
4 u[4] - 0.1225 (u[3] - 2 u[4] + u[5]) == 4
4 u[5] - 0.1225 (u[4] - 2 u[5] + u[6]) == 4
4 u[6] - 0.1225 (u[5] - 2 u[6] + u[7]) == 4
4 u[7] - 0.1225 (u[6] - 2 u[7] + u[8]) == 4
u[1] == 0
u[8] == 0

```

This shows the A matrix from the above set of equations. We see it is a tridiagonal, The last 2 rows are for the 2 equations for the boundary conditions

```

{b, A} = CoefficientArrays[FDMeqs, u];
A // MatrixForm

```

$$\begin{pmatrix} -0.1225 & 4.245 & -0.1225 & 0 & 0 & 0 & 0 & 0 \\ 0 & -0.1225 & 4.245 & -0.1225 & 0 & 0 & 0 & 0 \\ 0 & 0 & -0.1225 & 4.245 & -0.1225 & 0 & 0 & 0 \\ 0 & 0 & 0 & -0.1225 & 4.245 & -0.1225 & 0 & 0 \\ 0 & 0 & 0 & 0 & -0.1225 & 4.245 & -0.1225 & 0 \\ 0 & 0 & 0 & 0 & 0 & -0.1225 & 4.245 & -0.1225 \\ 1. & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1. \end{pmatrix}$$

## (b) Summary of numerical results

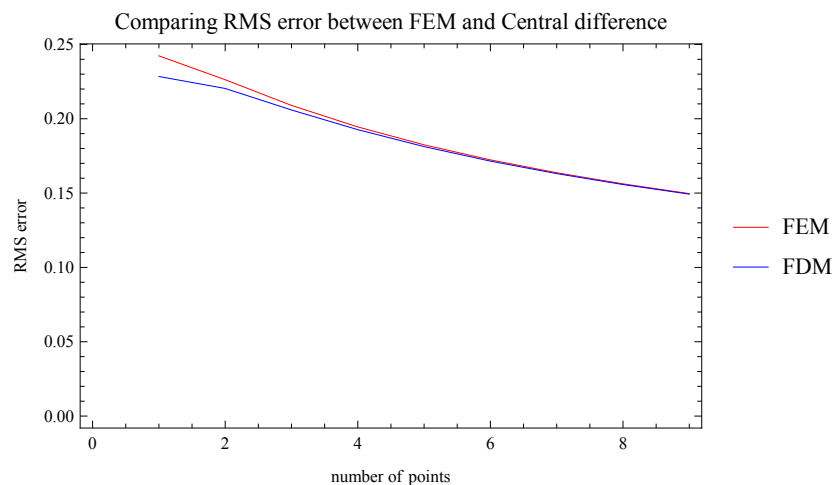
The error in approximation between numerical and exact solution for  $u(x)$  as a function of increasing number of points is generated for up to  $n = 200$  and the complete table is shown in the appendix at the end of the report. Both max error and RMS error is calculated for each  $n$ . Below is partial listing of the table for number of points  $n = 4..21$

```
q = 4; f = 4; L = 1; leftBC = 0; rightBC = 0; nElements = 20;
p = analysis[nElements, q, f, L, leftBC, rightBC];
Grid[p, Frame → All]
```

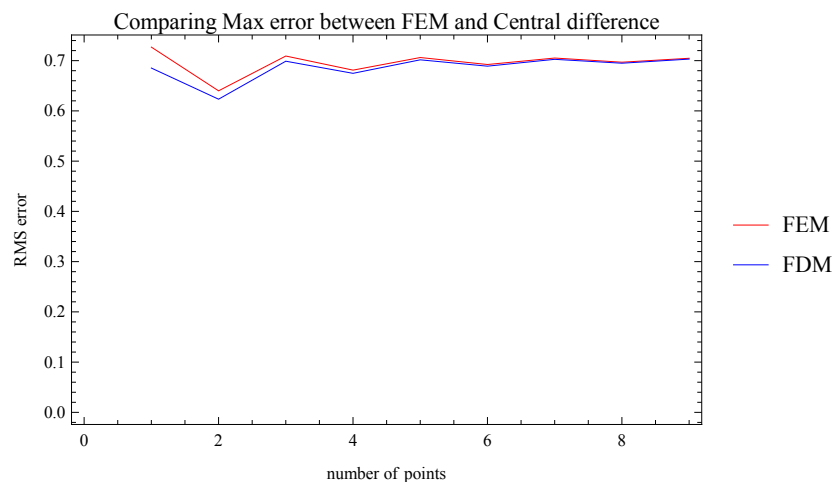
number of elements	FEM rms error	FDM rms error	FEM max error	FDM max error
2	0.2423152421	0.2284263532	0.7269457264	0.6852790597
3	0.2262502829	0.220369977	0.6399324371	0.6233004204
4	0.2089802182	0.2059041154	0.709175374	0.6988845018
5	0.1944929825	0.1926552141	0.6811213574	0.6747543445
6	0.1824593741	0.1812602056	0.7062042017	0.7016335051
7	0.1723355501	0.1715026324	0.6923088451	0.689006997
8	0.1636906816	0.1630846053	0.7051854934	0.7026147643
9	0.156207374	0.1557502056	0.6968932722	0.6948825608
10	0.1496520314	0.1492971628	0.7047176159	0.7030723997
11	0.1438502382	0.1435682687	0.6992096169	0.6978590958
12	0.1386694627	0.1384410264	0.7044644168	0.7033219182
13	0.1340072442	0.133819121	0.7005405594	0.6995717615
14	0.1297830918	0.1296259812	0.7043120698	0.703472687
15	0.125932847	0.1258000358	0.7013751198	0.7006465698
16	0.1224046941	0.1222912285	0.7042133209	0.7035706699
17	0.1191562888	0.1190584416	0.7019326612	0.7013649956
18	0.1161526555	0.1160675744	0.704145678	0.703637905
19	0.1133646248	0.113290094	0.7023234977	0.7018687951
20	0.1107676559	0.1107019306	0.7040973229	0.7036860271

The numerical results in the table is also plotted. The following 2 plots compare both methods accuracy for max error and rms error as function of increasing n for n up to 200. The full table is in the appendix.

```
ListLinePlot[ {p[[2 ;; 10, 2]], p[[2 ;; 10, 3]]},
  PlotRange → All,
  PlotStyle → {Red, Blue},
  AxesOrigin → {0, 0},
  FrameLabel → {"number of points", "RMS error"},
  PlotLabel → "Comparing RMS error between FEM and Central difference",
  Frame → True,
  PlotLegends → {"FEM", "FDM"}]
```



```
ListLinePlot[ {p[[2 ;; 10, 4]], p[[2 ;; 10, 5]]},
  PlotRange → All,
  PlotStyle → {Red, Blue},
  AxesOrigin → {0, 0},
  FrameLabel → {"number of points", "RMS error"},
  PlotLabel → "Comparing Max error between FEM and Central difference",
  Frame → True,
  PlotLegends → {"FEM", "FDM"}]
```



### (c) Discussion of numerical results

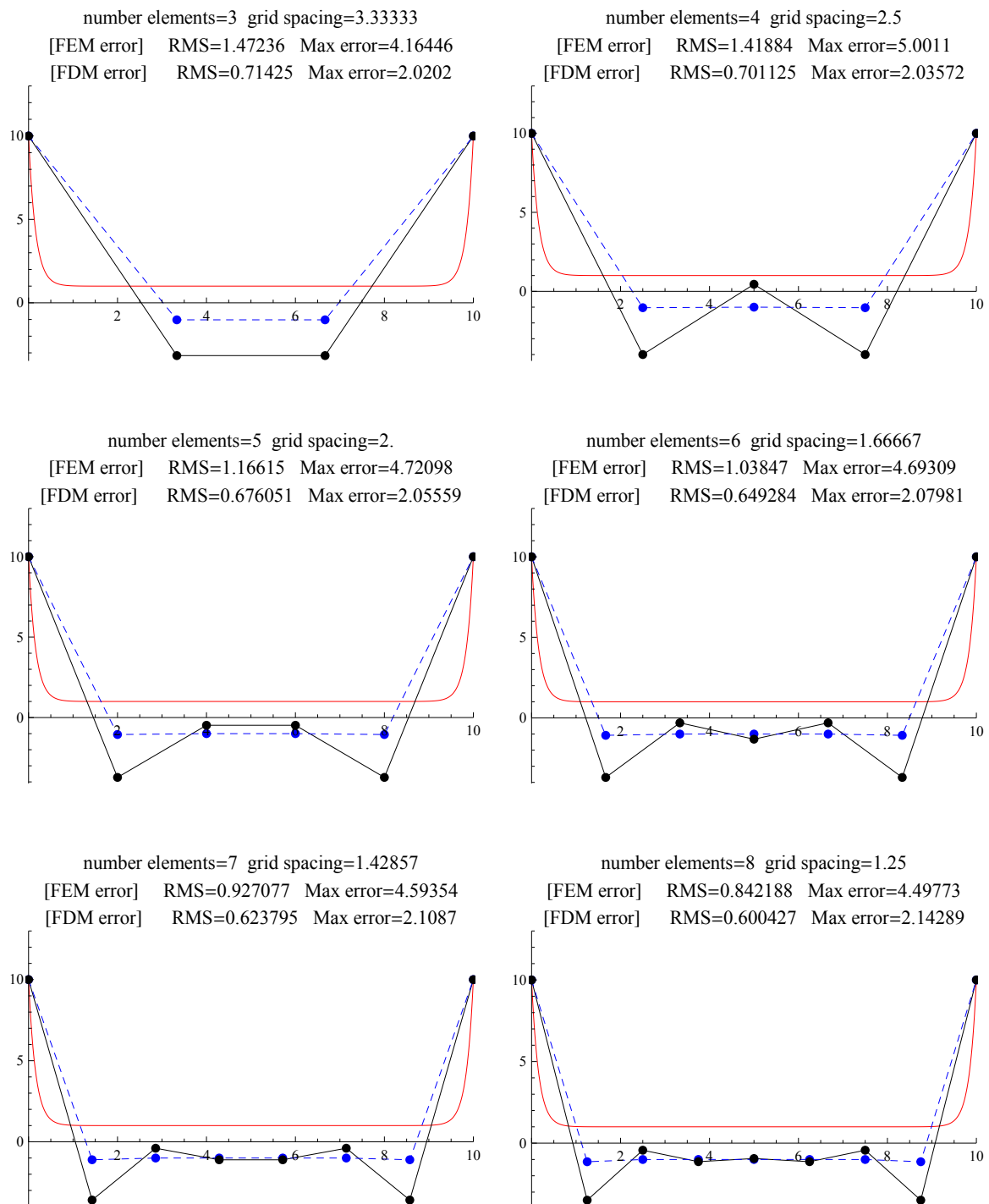
The FDM is more accurate than the FEM for low  $N$ . Two error measurements were used: RMS error and Max error. For low  $N$ , FDM is more accurate than FEM, even more so when looking at the RMS error as seen by the plot above.

From the plots above we observe that as  $N$  increases both methods become as accurate as each others in approximating the exact solution. This happens for both max error and for RMS error.

The following diagram below shows this more clearly, where both FEM and FDM solution are plotted for  $n = 2, 3, 4, 5, 6, 7, 8$ . We see that FDM for low  $N$  is more accurate, but FEM catches up very quickly, and about  $N=8$  FEM max error was the same as FDM error for up to 6 decimal places. The following was done for  $L = 10$  instead of  $L = 1$  to make the plots a little more interesting.



```
analysis2[] := Module[{},
  L = 10; nPoints = 10; q = 40; f = 40;
  leftBC = 10; rightBC = 10; plotType = 2;
  Table[process[i, q, f, L, leftBC, rightBC, plotType], {i, 3, 8}]
]
p = analysis2[]; GraphicsGrid[{{p[[1]], p[[2]]}, {p[[3]], p[[4]]}, {p[[5]], p[[6]]}},
  ImageSize -> 600]
```



FEM method is more complicated to implement than FDM. And from the above, we see that it does not give more accurate results than the simpler FDM method using central difference scheme. One might ask then why use FEM over FDM? Although this did not come up in this project, one can argue the following advantages of one method over the other:

**Advantage of FEM over FDM**

- FEM solution  $u(x)$  can be used at point  $x$  in the domain and not just at the grid points  $x_i$  as with FDM
- FEM can handle complicated boundary geometry, while FDM requires simple boundary geometry.
- With FEM, one can add more elements in the vicinity where the solution changes rapidly to obtain more accuracy there. With FDM this is normally not done. Although one can also implement adaptive grid sizing as well in FDM, it is normally done in the time domain and not in the space domain.
- Does not require knowing the differential equation to find the solution when working from the Variational approach as in this project. FDM requires knowing the differential equation.

**Advantages of FDM over FEM**

- More accurate than FEM for low  $N$ . This is in particular when the exact solution is not too smooth. When the solution is not too smooth, more points are needed to make FEM as accurate as FDM.
- Simpler to code and implement.

## (d) Source code listing

Global parameters

```
gPlotTypeFEM = 0;
gPlotTypeFDM = 1;
gPlotTypeBoth = 2;
gPlotTypeFEMdata = 3;
gPlotTypeFDMdata = 4;
```

**Function name:** makeEquation and its helpers

**Input:** the shape function number. Also access variable  $q$ ,  $f$ ,  $\phi_i$ ,  $c_i$  to generate the weak form equation for the  $i^{\text{th}}$  shape function

**Purpose:** generate the weak for equation for  $i^{\text{th}}$  shape function. To make it simpler, I have 2 special functions to handle the first and last shape elements since these are special conditions.

**Returns:** The weak for equation for the  $i^{\text{th}}$  shape function

```
makeEquation[i_, h_, c_, q_, f_, grid_, nShapeFunctions_] := Module[{g},
  If[i == 1, g = makeFirstEquation[h, c, q, f, grid],
  If[i == Length[c], g = makeLastEquation[h, c, q, f, grid, nShapeFunctions],
  g = makeInnerEquation[i, h, c, q, f, grid]]]
]
makeFirstEquation[h_, c_, q_, f_, grid_] :=
Module[{},
  
$$\int_{\text{grid}[[1]]}^{\text{grid}[[2]]} \left( \left( c[[1]] \left( \frac{-1}{h} \right) + c[[2]] \left( \frac{1}{h} \right) \right) \left( \frac{-1}{h} \right) + \right.$$


$$\left. (q(c[[1]] \text{Evaluate}[\phi[1, x, h]] + c[[2]] \text{Evaluate}[\phi[2, x, h]]) - f) \right.$$


$$\left. \text{Evaluate}[\phi[1, x, h]] \right) dx$$

]
makeLastEquation[h_, c_, q_, f_, grid_, nMax_] :=
Module[{},
  
$$\int_{\text{grid}[[2]]}^{\text{grid}[[1]]} \left( \left( c[[-2]] \left( \frac{-1}{h} \right) + c[[-1]] \left( \frac{1}{h} \right) \right) \left( \frac{1}{h} \right) + \right.$$


$$\left. (q(c[[-2]] \text{Evaluate}[\phi[nMax - 1, x, h]] + c[[-1]] \text{Evaluate}[\phi[nMax, x, h]]) - f) \right.$$


$$\left. \text{Evaluate}[\phi[nMax, x, h]] \right) dx$$

]
makeInnerEquation[i_, h_, c_, q_, f_, grid_] := Module[{r1, r2},
  
$$\int_{\text{grid}[[i-1]]}^{\text{grid}[[i]]} \left( \left( c[[i-1]] \left( \frac{-1}{h} \right) + c[[i]] \left( \frac{1}{h} \right) \right) \left( \frac{1}{h} \right) + \right.$$


$$\left. (q(c[[i-1]] \phi[i-1, x, h] + c[[i]] \phi[i, x, h]) - f) \phi[i, x, h] \right) dx +$$


$$\int_{\text{grid}[[i]]}^{\text{grid}[[i+1]]} \left( \left( c[[i]] \left( \frac{-1}{h} \right) + c[[i+1]] \left( \frac{1}{h} \right) \right) \left( \frac{-1}{h} \right) + \right.$$


$$\left. (q(c[[i]] \phi[i, x, h] + c[[i+1]] \phi[i+1, x, h]) - f) \phi[i, x, h] \right) dx$$

]
```

**Function name:**  $\phi$

**Input:** i, the shape function number. z, the distance along the domain to find  $\phi$  at

**Purpose:** generate the  $\phi$  function.

**Returns:** the value of  $\phi$  function at this z value.

```
 $\psi[x_] := Module[{}, If[x > 1, 0, If[x \ge; 0, 1 - x, \psi[-x]]]]$ 
```

```

 $\phi[i\_ , x\_ , h\_ ] := \text{Module}[\{\},$ 
 $\psi\left[\frac{x}{h} - (i - 1)\right]$ 
 $]$ 

```

**Function name:** yApprox

**Input:** x, distance along x direction. coeff, the  $c_i$  coefficient found for the FEM approximation from the equation  $y = \sum_{i=1}^n c(i) \phi[i, x]$ , h is the grid spacing, nPoints is number of points.

**Purpose:** calculate the solution from the FEM calculation after the  $c_i$  has been found.

**Returns:** u(x) based on FEM approximation

```

yApprox[x_, coeff_, h_, nPoints_] := Module[{i},
  Sum[coeff[[i]]  $\phi[i, x, h]$ , {i, 1, nPoints}]
]

```

**Function name:** getUCentralMethod

**Input:** access to number of points

**Purpose:** performs finite difference calculation using the central difference scheme for the second derivative.

**Returns:** data, which is a matrix of 2 columns. The first column is the x value, second column is the u value at this x.

```
getUCentralMethod[rightBC_, leftBC_, h_, q_, f_, nPoints_, grid_] :=
Module[{data, i, eq, A, b, u, coeff, nRow, nCol, FDMeqs},
  u = Array["c", nPoints - 1];
  (*The equation uses central difference method for the u''[x]. We just
  need to be carfull with the first last grid points,
  since these need boundary conditions on this*)

  FDMeqs = Table[If[nPoints == 3, -  $\left(\frac{\text{rightBC} - 2 u[[i]] + \text{leftBC}}{h^2}\right) + q u[[i]] == f,$ 
    If[i == 2, -  $\left(\frac{u[[i + 1]] - 2 u[[i]] + \text{leftBC}}{h^2}\right) + q u[[i]] == f,$ 
    If[i == nPoints - 1, -  $\left(\frac{\text{rightBC} - 2 u[[i]] + u[[i - 1]]}{h^2}\right) + q u[[i]] == f,$ 
    -  $\left(\frac{u[[i + 1]] - 2 u[[i]] + u[[i - 1]]}{h^2}\right) + q u[[i]] == f]], {i, 2, nPoints - 1}];

  {b, A} = CoefficientArrays[FDMeqs, u[[2 ;;]]];
  {nRow, nCol} = Dimensions[A];

  (*Now solve Ax=b. Use triDiagonal for speed for large matrices*)
  coeff = If[nRow < 5, LinearSolve[A, b], triDiagonalSolve[A, b]];

  data = Table[{grid[[i]], If[i == 1, leftBC, If[i == nPoints, rightBC, coeff[[i - 1]]]}],
    {i, 1, nPoints}];
  {data, FDMeqs, A, b}
];$ 
```

**Function name:** getErrorsInApproximation

**Input:** access FEM matrix, grid points, FDM data.

**Purpose:** called to calculate the RMS error and Max error for the FEM and FDM methods.

**Returns:** 4 numbers. rmseErrorFEM, maxErrorFEM, rmseErrorDiff, maxErrorFDM

```
getErrorsInApproximation[plotType_, grid_, nPoints_, FEMcoeff_, xnumeric_, sol_, h_] :=
Module[{k, i, rmseErrorFEM = 0, maxErrorFEM = 0, rmseErrorDiff = 0, maxErrorFDM = 0,
  femSolAtPoints, exactSolAtPoints},

exactSolAtPoints = Table[sol /. {z → grid[[i]]}, {i, 1, nPoints}];
If[plotType == gPlotTypeFEM || plotType == gPlotTypeBoth,
{
  femSolAtPoints = Table[yApprox[grid[[i]], FEMcoeff, h, nPoints], {i, 1, nPoints}];
  rmseErrorFEM = Sum[N[(exactSolAtPoints[[k]] - femSolAtPoints[[k]])^2],
    {k, 1, nPoints}];
  rmseErrorFEM = Sqrt[rmseErrorFEM] / nPoints;
  maxErrorFEM = Max[Abs[exactSolAtPoints - femSolAtPoints]];
}
];

If[plotType == gPlotTypeFDM || plotType == gPlotTypeBoth,
{
  rmseErrorDiff = Sum[N[(exactSolAtPoints[[k]] - xnumeric[[k, 2]])^2], {k, 1, nPoints}];
  rmseErrorDiff = Sqrt[rmseErrorDiff] / nPoints;
  maxErrorFDM = Max[Abs[exactSolAtPoints - xnumeric[[All, 2]]]];
}
];

N[{rmseErrorFEM, maxErrorFEM, rmseErrorDiff, maxErrorFDM}]
]
```

**Function name:** triDiagonalSolve

**Input:** A,d

**Purpose:** called to solve for x in  $Ax=d$  for use on tridiagonal matrices A. This only works correctly if A is diagonally dominant.

**Returns:** vector x, the solution from  $Ax=d$

```

triDiagonalSolve[A_, d_] := Module[{nRow, nCol,  $\beta$ ,  $\alpha$ , b, a, c, z, i, j, n, x},
  {n, nCol} = Dimensions[A];
  If[n  $\neq$  nCol, {Print["Matrix must be square. Matrix is" <> ToString[N[A]]],
    Abort[]}] ;
   $\beta$  = Table[0, {n}];
  c = Table[0, {n - 1}];
  z = Table[0, {n}];
   $\alpha$  =  $\beta$ ; b =  $\alpha$ ; x = z;
  a = Diagonal[A];

  For[{i = 2; j = 1}, i  $\leq$  n, {i++; j++}, b[[i]] = A[[i, j]];
  For[{i = 1; j = 2}, i < n, {i++; j++}, c[[i]] = A[[i, j]];

   $\alpha$ [[1]] = a[[1]];
  For[i = 2, i  $\leq$  n, i++, { $\beta$ [[i]] =  $\frac{b[[i]]}{\alpha[[i - 1]]}$ ;  $\alpha$ [[i]] = a[[i]] -  $\beta$ [[i]] c[[i - 1]]}];
  z[[1]] = d[[1]];
  For[i = 2, i  $\leq$  n, i++, z[[i]] = d[[i]] -  $\beta$ [[i]] z[[i - 1]]];

  x[[n]] =  $\frac{z[[n]]}{\alpha[[n]]}$ ;
  For[i = n - 1, i  $\geq$  1, i--, x[[i]] =  $\frac{1}{\alpha[[i]]}$  (z[[i]] - c[[i]] x[[i + 1]])];
  x
]

```

**Function name:** process

**Input:** access to all the input parameters from the GUI. See init[] function above.

**Purpose:** called by the GUI after the call to init[] is made. This function is the driver which calls all the other functions above. It find the FEM and FDM solutions and plot them.

**Returns:** Graphics plot for the GUI to display

```
process[nElements_, q_, f_, L_, leftBC_, rightBC_, plotType_] := Module[
  {i, A, b, k, p, p2, v, y, maxu, minu, nRow, nCol, nPoints, pExact, pFEM, h,
   grid, nShapeFunctions, coeffShapeFunctions, pFEMpoints, pFDMpoints,
   exactSolAtPoints, femSolAtPoints, maxFEM, minFEM, FEMeqs, sol, FEMcoeff,
   FDMeqs, FDMmatrix, xnumeric, FDMbMatrix, rmseerrorFEM, maxErrorFEM,
   rmseerrorDiff, maxErrorFDM, nRowFDM, nColFDM, g, statsFDM, statsFEM, stats,
   imageSize},

  imageSize = {300, 250};
  nPoints = nElements + 1;
  h = L / nElements; (*distance between 2 points*)
  grid = N[Range[0, L, h]]; (*the GRID itself*)
  nShapeFunctions = nPoints;
  coeffShapeFunctions = Array["c", nShapeFunctions];

  sol =
    y[z] /. First@DSolve[{-y'[z] + q y[z] == f, y[0] == leftBC, y[L] == rightBC}, y[z], z];
  exactSolAtPoints = Table[sol /. {z -> grid[[i]]}, {i, 1, nPoints}];

  (*Find max/min limits for plotting *)
  maxu = Max[exactSolAtPoints];
  maxu = maxu + 0.3 Abs[maxu];

  minu = Min[exactSolAtPoints];
  minu = minu - 0.3 Abs[minu];

  (*Now based on plot type requested, do the needed calculations*)
  If[plotType == gPlotTypeFEM || plotType == gPlotTypeBoth ||
    plotType == gPlotTypeFEMdata,
    {
      (*Now generate the FEM set of equations from the weak form*)

      FEMeqs =
        Table[makeEquation[i, h, coeffShapeFunctions, q, f, grid, nShapeFunctions] == 0,
              {i, 1, nShapeFunctions}];

      {b, A} = CoefficientArrays[FEMeqs, coeffShapeFunctions];

      (*Now we need to fix up the A matrix due to the initial conditions.*)
      (*This below is a standard method I learned from my other courses,
      and it works, so I use it*)

      A[[1, 2 ;; -1]] = 0;
      A[[-1, 1 ;; -2]] = 0;
      b[[1]] = A[[1, 1]] * leftBC;
      b[[-1]] = A[[-1, -1]] * rightBC;
      For[i = 2, i < nPoints, i++,
        {
          b[[i]] = b[[i]] - A[[i, 1]] * leftBC - A[[i, -1]] * rightBC;
          A[[i, 1]] = 0;
          A[[i, -1]] = 0;
        }
      ];
    }
  ];
```



```

(*Now solve Ax=b. *)
{nRow, nCol} = Dimensions[A];
FEMcoeff = LinearSolve[A, b];
femSolAtPoints = Table[yApprox[grid[[i]], FEMcoeff, h, nPoints], {i, 1, nPoints}];
maxFEM = Max[femSolAtPoints];
If[ maxFEM > maxu, {maxu = maxFEM; maxu = maxu + 0.1 Abs[maxu]}];

minFEM = Min[femSolAtPoints];
If[ minFEM < minu, {minu = minFEM; minu = minu - 0.1 Abs[minu]}];

}
];

If[plotType == gPlotTypeFDM || plotType == gPlotTypeBoth ||
  plotType == gPlotTypeFDMdata,
{
  {xnumeric, FDMeqs, FDMmatrix, FDMbMatrix} =
    getUCentralMethod[rightBC, leftBC, h, q, f, nPoints, grid];
  {nRowFDM, nColFDM} = Dimensions[FDMmatrix]
}
];

(*rest is just plotting stuff *)

g = 0;
Which[
  plotType == gPlotTypeFDMdata,
  g = {Text[MatrixForm[N[FDMmatrix]], {0, 0}],
    Text[MatrixForm[N[FDMbMatrix]], {0.4 + 0.03 nRowFDM, 0}],
    Text[TableForm[N[FDMeqs]], {0, .5}]},

  plotType == gPlotTypeFEMdata,
  g = {Text[MatrixForm[N[A[[2 ;; -2, 2 ;; -2]]]], {0, 0}],
    Text[MatrixForm[N[b[[2 ;; -2]]]], {0.4 + 0.05 nRow, 0}],
    Text[TableForm[FEMeqs], {0, .5}]
},

True,
{
  {rmSErrorFEM, maxErrorFEM, rmSErrorDiff, maxErrorFDM} =
    getErrorsInApproximation[plotType, grid, nPoints, FEMcoeff, xnumeric, sol, h];

  statsFEM = ""; statsFDM = "";
  stats = "number elements=" <> ToString[nElements] <> " grid spacing=" <>
    ToString[N[h]];
  If[plotType == 0 || plotType == 2,
    statsFEM = "[FEM error] RMS=" <> ToString[AccountingForm[rmSErrorFEM]] <>
      " Max error=" <> ToString[AccountingForm[maxErrorFEM]]
  ];

  If[plotType == gPlotTypeFDM || plotType == gPlotTypeBoth,
    statsFDM = "[FDM error] RMS=" <> ToString[AccountingForm[rmSErrorDiff]] <>
      " Max error=" <> ToString[AccountingForm[maxErrorFDM]]
  ];

  stats = stats <> "\n" <> statsFEM <> "\n" <> statsFDM;
  pExact = Plot[sol /. z -> x, {x, 0, L}, PlotRange -> {{0, L}, {maxu, minu}},

```

```

    ImageSize → imageSize, PlotStyle → Red, PlotLabel → stats];

If[plotType == gPlotTypeFEM || plotType == gPlotTypeBoth,
{
  pFEM = Plot[yApprox[x, FEMcoeff, h, nPoints], {x, 0, L},
    PlotRange → {{0, L}, {maxu, minu}}, AxesOrigin → {0, 0}, PlotStyle → Black,
    ImageSize → imageSize];
  pFEMpoints =
    Table[Graphics[{PointSize[0.02], Black,
      Point[{i h, yApprox[i h, FEMcoeff, h, nPoints]}]}], {i, 0, nPoints - 1}];
}
];

If[plotType == gPlotTypeFDM || plotType == gPlotTypeBoth,
{
  p2 = ListLinePlot[xnumeric, PlotRange → {{0, L}, {maxu, minu}},
    AxesOrigin → {0, 0}, PlotStyle → {Blue, Dashed}, ImageSize → imageSize];
  pFDMpoints =
    Table[Graphics[{PointSize[0.02], Blue,
      Point[{xnumeric[[i, 1], xnumeric[[i, 2]}]}], {i, 1, nPoints}];
}
];

Which[plotType == gPlotTypeFEM,
  g = Show[{pExact, pFEM, pFEMpoints}],

  plotType == gPlotTypeFDM,
  g = Show[{pExact, p2, pFDMpoints}],

  plotType == gPlotTypeBoth,
  g = Show[{pExact, pFEM, p2}, pFDMpoints, pFEMpoints]
]
}
];

Graphics[g, ImageSize → imageSize]
]

```

This is the Manipulate function. This is the GUI interface for the program. It display the GUI and called the Init and Process functions above to do the actual calculations

```

Manipulate[
  process[numberOfElements, q, f, L, leftBC, rightBC, plotType],
  {{q, 4, "q"}, 1, 100, 1, Appearance → "Labeled"},
  {{f, 4, "f"}, 1, 100, 1, Appearance → "Labeled"},
  {{L, 1, "Length"}, 1, 200, 1, Appearance → "Labeled"},
  {{leftBC, 0, "Left boundary condition: u(0)"}, -50, 50, 1,
    Appearance → "Labeled"},
  {{rightBC, 0, "Right boundary condition: u(L)"}, -50, 50, 1,
    Appearance → "Labeled"},
  {{plotType, 0, "display type"},
    {gPlotTypeFEM → "PLOT: Finite Elements",
     gPlotTypeFDM → "PLOT: Central Difference",
     gPlotTypeBoth → "PLOT: Both",
     gPlotTypeFEMdata → "DATA: Finite element",
     gPlotTypeFDMdata → "DATA: central difference"}
  },
  {{numberOfElements, 2, "number of Elements"}, 2, 50, 1, Appearance → "Labeled"},
  FrameLabel → "Solution to  $-y''+q y==f$  using FEM and divided difference methods",
  SynchronousUpdating → True]

```

q  4  
 f  4  
 Length  1  
 Left boundary condition: u(0)  0  
 Right boundary condition: u(L)  0  
 display type PLOT: Finite Elements PLOT: Central Difference PLOT: Both DATA: Finite element  
 number of Elements  2

process[2, 4, 4, 1, 0, 0, 0]

Solution to  $-y''+q y==f$  using FEM and divided difference methods

## Appendix

This is an analysis function. Not part of the main program. It is used to generate a table that shows the error (RMS and Max) between FEM and FDM methods as  $n$ , the number of points, is increased.

```
analysis[maxN_, q_, f_, L_, leftBC_, rightBC_] := Module[
  {i, j, A, b, k, p, p2, v, y, maxu, minu, nRow, nCol, nPoints, pExact, pFEM,
   h, grid, nShapeFunctions, coeffShapeFunctions, pFEMpoints, pFDMpoints,
   exactSolAtPoints, femSolAtPoints, maxFEM, minFEM, startingAt, nEntries,
   sol, nElements, FEMeqs, xnumeric, FEMcoeff, rmseerrorFEM, FDMeqs, FDMmatrix,
   FDMbMatrix, maxErrorFEM, rmseerrorDiff, maxErrorFDM, nRowFDM, nColFDM},

  startingAt = 2;
  nEntries = maxN - startingAt + 1;
  p = Table[{0, 0, 0, 0, 0}, {nEntries + 1}];
  p[[1, 1]] = "number of elements"; p[[1, 2]] = "FEM rms error";
  p[[1, 3]] = "FDM rms error";
  p[[1, 4]] = "FEM max error"; ; p[[1, 5]] = "FDM max error";

  j = 1;
  For[nElements = startingAt, nElements ≤ maxN, nElements++,
    {
      nPoints = nElements + 1;
      h = L / nElements; (*distance between 2 points*)
      grid = N[Range[0, L, h]]; (*the GRID itself*)
      nShapeFunctions = nPoints;
      coeffShapeFunctions = Array["c", nShapeFunctions];

      sol =
        y[z] /. First@DSolve[{-y'[z] + q y[z] == f, y[0] == leftBC, y[L] == rightBC},
          y[z], z];
      exactSolAtPoints = Table[sol /. {z → grid[[i]]}, {i, 1, nPoints}];

      (*Find max/min limits for plotting *)
      maxu = Max[exactSolAtPoints];
      maxu = maxu + 0.3 Abs[maxu];

      minu = Min[exactSolAtPoints];
      minu = minu - 0.3 Abs[minu];

      FEMeqs =
        Table[makeEquation[i, h, coeffShapeFunctions, q, f, grid, nShapeFunctions] == 0,
          {i, 1, nShapeFunctions}];

      FEMeqs = Simplify[FEMeqs];

      {b, A} = CoefficientArrays[FEMeqs, coeffShapeFunctions];

      (*Now we need to fix up the A matrix due to the initial conditions.*)
      (*This below is a standard method I learned from my other courses,
      and it works, so I use it*)

      A[[1, 2 ;; -1]] = 0;
      A[[-1, 1 ;; -2]] = 0;
      b[[1]] = A[[1, 1]] * leftBC;
      b[[-1]] = A[[-1, -1]] * rightBC;
      For[i = 2, i < nPoints, i++,
        {
          b[[i]] = b[[i]] - A[[i, 1]] * leftBC - A[[i, -1]] * rightBC;
```

```

    A[[i, 1]] = 0;
    A[[i, -1]] = 0;
  }
];

(*Now solve Ax=b. Use triDiagonal for speed for large matrices*)
{nRow, nCol} = Dimensions[A];
FEMcoeff = LinearSolve[A, b];

{xnumeric, FDMeqs, FDMmatrix, FDMbMatrix} =
  getUCentralMethod[rightBC, leftBC, h, q, f, nPoints, grid];
{nRowFDM, nColFDM} = Dimensions[FDMmatrix];

{rmerrorFEM, maxErrorFEM, rmerrorDiff, maxErrorFDM} =
  getErrorsInApproximation[2, grid, nPoints, FEMcoeff, xnumeric, sol, h];

j = j + 1;
p[[j, 1]] = nElements;
(*p[[j, 2]] = AccountingForm[rmerrorFEM];
p[[j, 3]] = AccountingForm[rmerrorDiff];
p[[j, 4]] = AccountingForm[maxErrorFEM];
p[[j, 5]] = AccountingForm[maxErrorFDM]; *)

p[[j, 2]] = rmerrorFEM;
p[[j, 3]] = rmerrorDiff;
p[[j, 4]] = maxErrorFEM;
p[[j, 5]] = maxErrorFDM;
}

];
p
]

```

```

L = 10; nPoints = 10; q = 40; f = 40;
leftBC = 10; rightBC = 10; plotType = 2;
p = analysis[9, q, f, L, leftBC, rightBC];
Grid[p, Frame → All]

```

number of elements	FEM rms error	FDM rms error	FEM max error	FDM max error
2	2.483549379	0.6726546906	7.450648138	2.017964072
3	1.472357539	0.714250167	4.164456016	2.020204546
4	1.418838555	0.7011253334	5.001097349	2.035716634
5	1.166152855	0.6760513056	4.720983299	2.055586579
6	1.038474961	0.649283981	4.693088351	2.079811891
7	0.9270771087	0.6237946859	4.593536486	2.108701262
8	0.8421881382	0.6004271042	4.497725424	2.142886186
9	0.7729730024	0.5793160904	4.392476013	2.183208617