Computer Lab 2: Implicit Finite-Difference Schemes for the Diffusion Equation with Smooth Initial Conditions

Schemes Investigated

In this session we continue a comparsion the accuracy of various difference schemes for solving the diffusion equation. This is the equation that arises when the Black-Scholes differential equation is transformed into a form suitable for treatment by finite-difference methods. We compare (a was done last time)

(a) explicit finite-difference, with 400 time-steps (equivalent to the use of a binomial model, but on a grid rather than a tree);

(b) fully implicit, also with 400 time-steps;

(c) Crank-Nicholson, with 40 time-steps;

(d) Douglas, with 40 time-steps.

The solution method for type (a) is a simple updating rule, while (b), (c), (d) require the solution of tridiagonal systems of equations.

A Simple Test Problem with Smooth Initial Conditions

We consider the diffusion equation

$$\frac{\partial u}{\partial \tau} = \frac{\partial^2 u}{\partial x^2} \tag{1}$$

on the region defined by

$$-2 \le x \le 2 \qquad \tau \ge 0 \tag{2}$$

The initial condition is

$$u(x, 0) = \sin\left(\frac{\pi x}{2}\right) \tag{3}$$

and the boundary conditions are

$$u(2, \tau) = u(-2, \tau) = 0 \tag{4}$$

This has the exact solution

$$u(x, \tau) = \sin\left(\frac{\pi x}{2}\right) e^{-\frac{\pi^2 \tau}{4}} \tag{5}$$

So it is a simple matter to test various difference schemes by comparing with this known exact solution. It should be emphasized that this type of smooth initial data, which also joins continuously onto the boundary conditions, is rather atypical of option-pricing problems. Our purpose here is to simplify matters to get a general feel for the relative merits of explicit and implicit scheme.

A = LU Decomposition for Tridiagonal Systems

Our implicit schemes involve the solution of a matrix problem $A.x = r$ where A has the tridiagonal form :

```
A = {{b0, c0, 0, …, …, 0, 0}, {a1, b1, c1, 0, …, …, 0},
    {0, a2, b2, c2, …, …, 0}, {0, …, ⋱, ⋱, ⋱, ⋱, 0},
    {0, …, …, 0, a"N-2", b"N-2", c"N-2"},
    {0, …, …, …, 0, a"N-1", b"N-1"}};
```

```
MatrixForm[A]
```

$$\begin{pmatrix} b_0 & c_0 & 0 & \cdots & \cdots & 0 & 0 \\ a_1 & b_1 & c_1 & 0 & \cdots & \cdots & 0 \\ 0 & a_2 & b_2 & c_2 & \cdots & \cdots & 0 \\ 0 & \cdots & \ddots & \ddots & \ddots & \ddots & 0 \\ 0 & \cdots & \cdots & 0 & a_{N-2} & b_{N-2} & c_{N-2} \\ 0 & \cdots & \cdots & \cdots & 0 & a_{N-1} & b_{N-1} \end{pmatrix}$$

We can write down all sorts of fancy notation for solving this problem, but it is really very easy. We can perform row operations to reduce the problem to upper triangular form, then solve the resulting problem by back substitution starting from the bottom. This amounts to writing $A = L.U$ where L is lower triangular and U is upper triangular, or indeed doing Gaussian elimination. The L matrix is given by

```
TraditionalForm[
 MatrixForm[{{1, 0, 0, …, …, 0, 0}, {l₁, 1, 0, 0, …, …, 0},
   {0, l₂, 1, 0, …, …, 0}, {0, …, ⋱, ⋱, ⋱, ⋱, 0},
   {0, …, …, 0, l"N-2", 1, 0}, {0, …, …, …, 0, l"N-1", 1}}]]
```

$$\begin{pmatrix} 1 & 0 & 0 & \dots & \dots & 0 & 0 \\ l_1 & 1 & 0 & 0 & \dots & \dots & 0 \\ 0 & l_2 & 1 & 0 & \dots & \dots & 0 \\ 0 & \dots & \ddots & \ddots & \ddots & \ddots & 0 \\ 0 & \dots & \dots & 0 & l_{\mathrm{N}-2} & 1 & 0 \\ 0 & \dots & \dots & \dots & 0 & l_{\mathrm{N}-1} & 1 \end{pmatrix}$$

The U matrix is given by

```
TraditionalForm[
 MatrixForm[{{d₀, u₀, 0, …, …, 0, 0}, {0, d₁, u₁, 0, …, …, 0},
   {0, 0, d₂, u₂, …, …, 0}, {0, …, ⋱, ⋱, ⋱, ⋱, 0},
   {0, …, …, 0, 0, d"N-2", u"N-2"}, {0, …, …, …, 0, 0, d"N-1"}}]]
```

$$\begin{pmatrix} d_0 & u_0 & 0 & \dots & \dots & 0 & 0 \\ 0 & d_1 & u_1 & 0 & \dots & \dots & 0 \\ 0 & 0 & d_2 & u_2 & \dots & \dots & 0 \\ 0 & \dots & \ddots & \ddots & \ddots & \ddots & 0 \\ 0 & \dots & \dots & 0 & 0 & d_{\mathrm{N}-2} & u_{\mathrm{N}-2} \\ 0 & \dots & \dots & \dots & 0 & 0 & d_{\mathrm{N}-1} \end{pmatrix}$$

If we can solve first $L.z = r$, then solve $U.x = z$, we have solved the problem $A.x = r$, because

$$A.x = L.U.x = L.(U.x) = L.z = r \tag{6}$$

Finding L and U is easy (multiply them out to see how). Then solving $L.z = r$ is forward subsitutation. $U.x = z$ is then back substitution. In general we supply three vectors for the three diagonals and the RHS, solve for the LHS.

Details

To see how it works out, let $N = 4$

```
L = {{1, 0, 0, 0, 0}, {l1, 1, 0, 0, 0}, {0, l2, 1, 0, 0},
  {0, 0, l3, 1, 0}, {0, 0, 0, l4, 1}};
MatrixForm[L]
```

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ l_1 & 1 & 0 & 0 & 0 \\ 0 & l_2 & 1 & 0 & 0 \\ 0 & 0 & l_3 & 1 & 0 \\ 0 & 0 & 0 & l_4 & 1 \end{pmatrix}$$

```
U = {{d0, u0, 0, 0, 0}, {0, d1, u1, 0, 0}, {0, 0, d2, u2, 0},
  {0, 0, 0, d3, u3}, {0, 0, 0, 0, d4}};
MatrixForm[U]
```

$$\begin{pmatrix} d_0 & u_0 & 0 & 0 & 0 \\ 0 & d_1 & u_1 & 0 & 0 \\ 0 & 0 & d_2 & u_2 & 0 \\ 0 & 0 & 0 & d_3 & u_3 \\ 0 & 0 & 0 & 0 & d_4 \end{pmatrix}$$

```
MatrixForm[L.U]
```

$$\begin{pmatrix} d_0 & u_0 & 0 & 0 & 0 \\ d_0\, l_1 & d_1 + l_1\, u_0 & u_1 & 0 & 0 \\ 0 & d_1\, l_2 & d_2 + l_2\, u_1 & u_2 & 0 \\ 0 & 0 & d_2\, l_3 & d_3 + l_3\, u_2 & u_3 \\ 0 & 0 & 0 & d_3\, l_4 & d_4 + l_4\, u_3 \end{pmatrix}$$

and this must equal

```
A = {{b0, c0, 0, 0, 0}, {a1, b1, c1, 0, 0}, {0, a2, b2, c2, 0},
  {0, 0, a3, b3, c3}, {0, 0, 0, a4, b4}};
MatrixForm[A]
```

$$\begin{pmatrix} b_0 & c_0 & 0 & 0 & 0 \\ a_1 & b_1 & c_1 & 0 & 0 \\ 0 & a_2 & b_2 & c_2 & 0 \\ 0 & 0 & a_3 & b_3 & c_3 \\ 0 & 0 & 0 & a_4 & b_4 \end{pmatrix}$$

We can now read off that

$$u_i = c_i \ , \ i = 0, \ldots , N - 1$$

$$d_0 = b_0$$

$$l_i = a_i / d_{i-1} , \ i = 1, \ldots, N$$

$$d_i = b_i - l_i u_{i-1} , \ i = 1, \ldots, N$$

So having worked out those cofficients of L and U, we then solve the easy pair of linear systems. Solving $L.z = r$ gives us

$$z_0 = r_0$$

$$z_i = r_i - l_i z_{i-1}, \ i = 1, \ldots , N - 1$$

Solving $U.x = z$ gives us, working backwards

$$d_{N-1} x_{N-1} = z_{N-1}$$

$$x_j = \left(z_j - u_j x_{j+1}\right) / d_j , \ j = N - 2, \ldots, 1, 0$$

Compact versions of the algorithm can be given by reusing/overwriting the arrays. E.g. the *Mathematica* one below and tridiag in the NR class. We shall give a more explicit C++ code that follows the above very closely.

```
//Tridiagonal matrix solver
#include "nrutil_nr.h"
//Uses expanded algorithm in course notes and PD NR vectors.
//There are more compact versions
//This version could be modified to output L and U as well as soln

void tridiagsolve(NRVec<double> &a, NRVec<double> &b, NRVec<double> &c, NRVec<double> &r, NRVec<dou-
ble> &x)
{
    int j;
    int P=r.size();
    NRVec<double> u(P);
    NRVec<double> d(P);
    NRVec<double> l(P);
    NRVec<double> z(P);
    u[0]=c[0];
    d[0]=b[0];
    z[0]=r[0];
    // Do LU decomposition and forward subsititution as you go
    for (j=1;j<P;j++) {
        u[j]=c[j];
        l[j]=a[j]/d[j-1];
        d[j]=b[j]-l[j]*u[j-1];
        z[j] = r[j]-l[j]*z[j-1];
        }
    //follow with back substitution
    x[P-1] = z[P-1]/d[P-1];
    for (j=(P-2);j>=0;j--){
        x[j] = (z[j] - u[j]*x[j+1])/d[j];
        }
}
```

Implicit Scheme Tridiagonal Solver

Here is a compacted *Mathematica* version

```
CompTridiagSolve =
  Compile[{{a, _Real, 1}, {b, _Real, 1}, {c, _Real, 1},
    {r, _Real, 1}},
 Module[{len = Length[r], solution = r, aux = 1 / (b[[1]]),
     aux1 = r, a1 = Prepend[a, 0.0], iter},
        solution[[1]] = aux * r[[1]];
        Do[aux1[[iter]] = c[[iter - 1]] aux;

     aux = 1 / (b[[iter]] - a1[[iter]] * aux1[[iter]]);
                 solution[[iter]] =
                (r[[iter]] - a1[[iter]] solution[[iter - 1]])
       aux,
     {iter, 2, len}];

    Do[solution[[iter]] -= aux1[[iter + 1]] solution[[iter + 1]],
     {iter, len - 1, 1, -1}];
 solution]];
```

```
A = {{3, 1, 0, 0, 0}, {1, 3, 1, 0, 0}, {0, 1, 3, 1, 0},
   {0, 0, 1, 3, 1}, {0, 0, 0, 1, 3}};
MatrixForm[A]
```

$$\begin{pmatrix} 3 & 1 & 0 & 0 & 0 \\ 1 & 3 & 1 & 0 & 0 \\ 0 & 1 & 3 & 1 & 0 \\ 0 & 0 & 1 & 3 & 1 \\ 0 & 0 & 0 & 1 & 3 \end{pmatrix}$$

```
{v, w, x, y, z} /. Solve[A.{v, w, x, y, z} == {1, 2, 3, 4, 5},
   {v, w, x, y, z}] // N
```

```
{{0.208333, 0.375, 0.666667, 0.625, 1.45833}}
```

```
CompTridiagSolve[{1, 1, 1, 1}, {3, 3, 3, 3, 3}, {1, 1, 1, 1},
 {1, 2, 3, 4, 5}]
```

```
{0.208333, 0.375, 0.666667, 0.625, 1.45833}
```

```
A.%
```

```
{1., 2., 3., 4., 5.}
```

Comparison with C++ code

tridiagsolve is a function defined and tested with same example in testtridiag.cpp. Go check it!

Fully Implicit Scheme

Here is the matrix involved in the computations:

```
FullyImpCMatrix[alpha_, nminus_, nplus_] :=
Sequence[Table[-alpha, {nplus+nminus-2}],
Table[1+2*alpha, {nplus+nminus-1}],
Table[-alpha, {nplus+nminus-2}]]
```

Here are the parameters of this particular model:

```
M=400;
nminus = 80;
nplus = 80;
dx = 0.025;
dtau = 0.00025;
alpha = dtau/dx^2
```

```
0.4
```

Initial and boundary conditions, and problem initialization:

```
initial = Table[N[Sin[Pi * (k - 1 - nminus) / nminus] ],
   {k, 1, nminus + nplus + 1}];
lower = Table[0, {m, 1, M + 1}];
upper = Table[0, {m, 1, M + 1}];
wold = initial;
wvold = wold;
wnew = wold;
```

The solution:

```
CMat = FullyImpCMatrix[alpha,nminus,nplus];
```

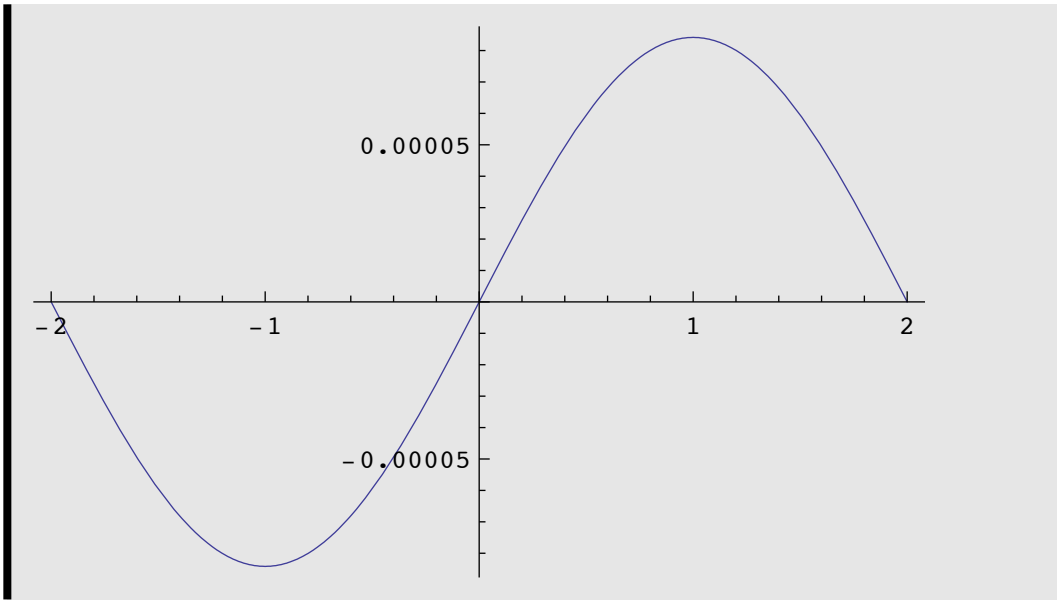
```
For[m=2, m<=M+1, m++,
(wvold = wold; wold = wnew;
rhs = Take[wold, {2, -2}]+
Table[Which[k==1, alpha*lower[[m]],
k== nplus + nminus-1,  alpha*upper[[m]], True, 0],
{k, 1, nplus + nminus-1}];
temp = CompTridiagSolve[CMat, rhs];
wnew = Join[{lower[[m]]}, temp, {upper[[m]]}])]
```

Interpolating the answer:

```
interpoldata =
Table[{(k-nminus-1)*dx ,wnew[[k]]}, {k, 1, nminus+nplus+1}];
ufunc = Interpolation[interpoldata, InterpolationOrder -> 3];
```

Now we plot the error:

```
Plot[ufunc[x] - Sin[π x / 2] e^(1/4 (-π^2) 0.1`), {x, -2, 2}, PlotPoints → 50]
```



Finally we tabulate the numerical result, the exact result and the error in the numerical scheme.

```
samples = TableForm[Join[{{"x", "Implicit FD", "Exact", "Error"}},
Table[Map[PaddedForm[N[Chop[#1]],{5,6}]&,
N[{x, ufunc[x], Sin[Pi*x/2]*Exp[-(Pi^2 0.1)/4],
ufunc[x]- Sin[Pi*x/2]*Exp[-(Pi^2 0.1)/4]},5]],
{x, -2, 2, 0.25}]]]
```

```
x          Implicit FD  Exact      Error
-2.000000   0.000000     0.000000   0.000000
-1.750000  -0.299040    -0.299010  -0.000032
-1.500000  -0.552550    -0.552490  -0.000060
-1.250000  -0.721950    -0.721870  -0.000078
-1.000000  -0.781430    -0.781340  -0.000084
-0.750000  -0.721950    -0.721870  -0.000078
-0.500000  -0.552550    -0.552490  -0.000060
-0.250000  -0.299040    -0.299010  -0.000032
 0.000000   0.000000     0.000000   0.000000
 0.250000   0.299040     0.299010   0.000032
 0.500000   0.552550     0.552490   0.000060
 0.750000   0.721950     0.721870   0.000078
 1.000000   0.781430     0.781340   0.000084
 1.250000   0.721950     0.721870   0.000078
 1.500000   0.552550     0.552490   0.000060
 1.750000   0.299040     0.299010   0.000032
 2.000000   0.000000     0.000000   0.000000
```

Note that we obtain no improvement in accuracy over the explicit scheme (the maximum error is about twice in fact). The only advantage of the fully implicit scheme over the explicit scheme is the fact that we can increase α, that is, the time-step for a given price-step, without the system going unstable.

C++ Code for Fully Implicit Method

```
M=400;
nminus = 80;
nplus = 80;
dx = 0.025;
dtau = 0.00025;
alpha = dtau/dx^2
```

```
#include <cmath>

#include <iostream>

#include <fstream>

#include "nrutil_nr.h" //allow access to public domain Numerical
Recipes matrix routines
```

```
using namespace std;
void tridiagsolve(NRVec<double> &a, NRVec<double> &b, NRVec<double>
&c, NRVec<double> &r, NRVec<double> &x)
{
    int j;
    int P=r.size();
    NRVec<double> u(P);
    NRVec<double> d(P);
    NRVec<double> l(P);
    NRVec<double> z(P);
    u[0]=c[0];
    d[0]=b[0];
    z[0]=r[0];
    // Do LU decomposition and forward subsititution as you go
    for (j=1;j<P;j++) {
        u[j]=c[j];
        l[j]=a[j]/d[j-1];
        d[j]=b[j]-l[j]*u[j-1];
        z[j] = r[j]-l[j]*z[j-1];
        }
    //follow with back substitution
    x[P-1] = z[P-1]/d[P-1];
    for (j=(P-2);j>=0;j--){
        x[j] = (z[j] - u[j]*x[j+1])/d[j];
        }
}
```

```
void ImplicitDiffusion(int N, int M)
{
	int i, j;
	double dt = 0.1/M ;
	double dx = 2.0/N;;
	const double PI = 3.141592653589793;
	NRMat<double> solvals(0.0,M+1,2*N+1); //NR dynamic matrix
	double alpha = 0.0; // key diffusion parameter
	NRVec<double> r(0.0,2*N-1); // create vectors for implicit
solver
	NRVec<double> x(0.0,2*N-1);
	NRVec<double> a(0.0,2*N-1);
	NRVec<double> b(0.0,2*N-1);
	NRVec<double> c(0.0,2*N-1);
	alpha = dt/(dx*dx);
	//output dt
	cout << "dt = " << dt << "\n";
	    //output dt
	cout << "dx = " << dx << "\n";
	// output alpha
	cout << "alpha = " << alpha << "\n";

	// initialize inititial data
	for (j = 0; j <= 2*N; j++)
	{
	    solvals[0][j] = sin(PI*(j-N)*dx/2);
```

```
}
// initialize boundary conditions

for (i = 1; i< M; i++)
{
    solvals[i][0] = 0;
    solvals[i][2*N] = 0;
    }
// Run the fully implicit algorithm
// First fill the arrays to feed to implicit solver
// Note form for fully implicit scheme
 for (j=0; j<=2*N-2;j++)
{
    a[j] = -alpha;
    b[j] = 1.0+2.0*alpha;
    c[j] = -alpha;
}

for (i= 1; i<= M; i++)
{
    for (j = 1; j<= 2*N-1; j++)
    {
        r[j-1] = solvals[i-1][j];
        }// The boundary conditions are zero so no end effects
    tridiagsolve(a, b, c, r, x);
```

```
            for (j = 1; j<= 2*N-1; j++)
            {
            solvals[i][j] = x[j-1];
            }
        }

    ofstream out("fdoutput3.txt");
    for (j=0; j<= 2*N; j++)
    {   out.precision(15);
        out << solvals[M][j] << "\n";
        }
       return ;
}

#include <iostream>
#include <cmath>
using namespace std;
int main()
{
    double result;
    char name[20];
    int N;
    int M;
    cout << "Implicit Diffusion Example \n ";
```

```
    cout << "Enter number of time steps (M) \n ";
    cin >> M;
    cout << M  << " time steps \n" ;
    cout << "Enter space step parameter (N) \n ";
    cin >> N;
    cout << 2*N  << " space steps \n" ;
    cout << " Result being written to fdoutput3.txt \n " << endl;
    ImplicitDiffusion(N,M);
    cout << "Hit any key+<RET> to finish \n ";
    cin >> name;
    return(0);
}
```

```
SetDirectory[
 "C:\Documents and Settings\William Shaw\My
   Documents\LGS0708\codes"]
```

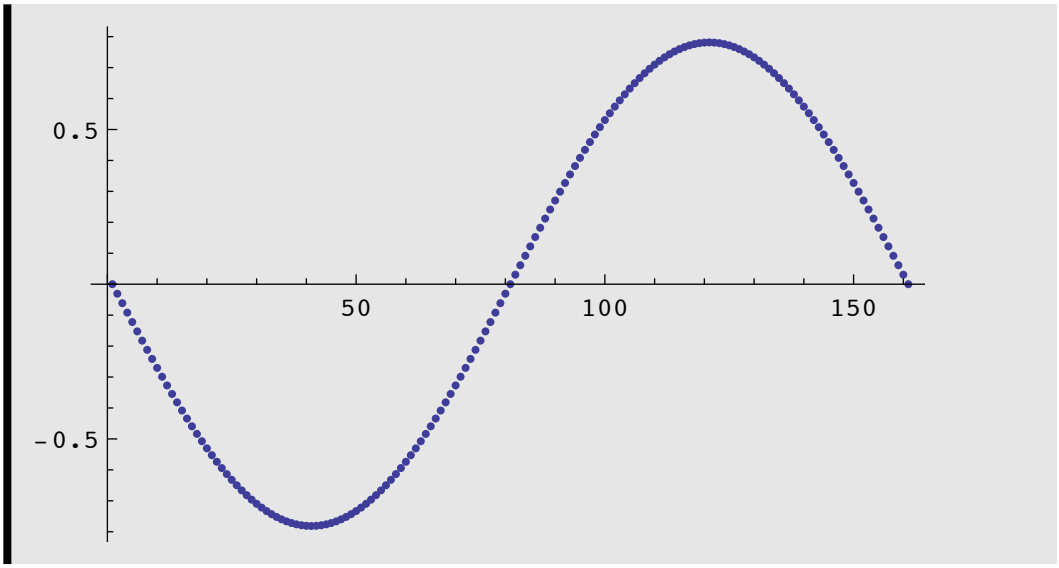
```
C:\Documents and Settings\William
  Shaw\My Documents\LGS0708\codes
```

```
FileNames["*.txt"]
```

```
{fdoutput2.txt, fdoutput3.txt, testtridiag_cpp.txt}
```

```
cppdata3 = ReadList["fdoutput3.txt", Number];
```

```
ListPlot[cppdata3]
```



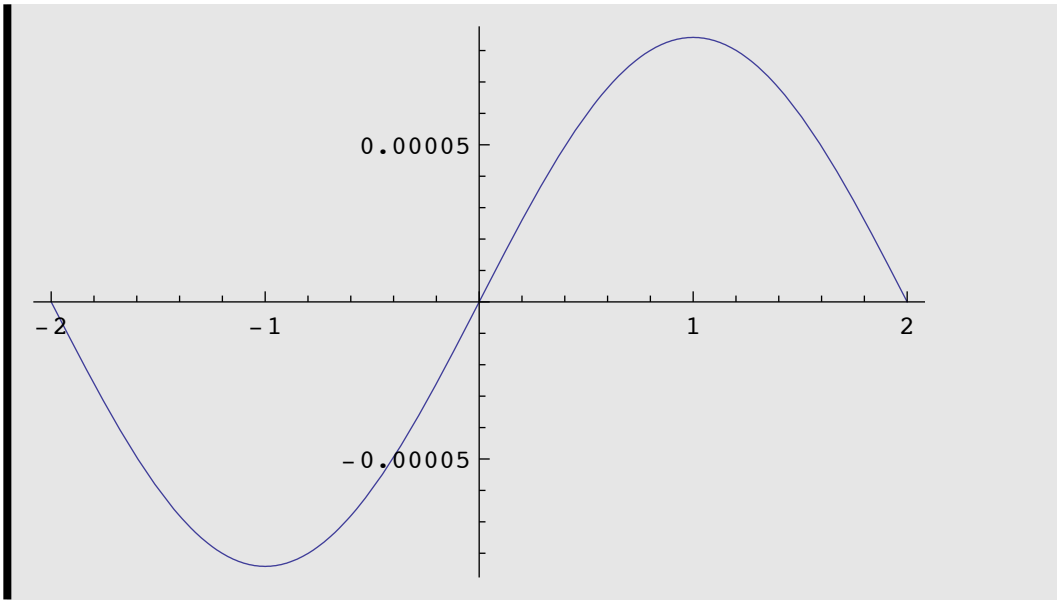

```
cppinterpoldata =
Table[{(k-nminus-1)*dx ,cppdata3[[k]]}, {k, 1, nminus+nplus+1}];
```

```
cppfunc = Interpolation[cppinterpoldata, InterpolationOrder -> 3]
```

```
InterpolatingFunction[{{-2., 2.}}, <>]
```

Now we plot the error in the answer:

$$\text{Plot}\left[\text{cppfunc}[x] - \text{Sin}\left[\frac{\pi x}{2}\right] e^{\frac{1}{4}\left(-\pi^2\right) 0.1`}, \{x, -2, 2\}, \text{PlotPoints} \to 50\right]$$




```
mmadata = Transpose[interpoldata][[2]];
```

```
mmacppdiff = mmadata - cppdata3;
```

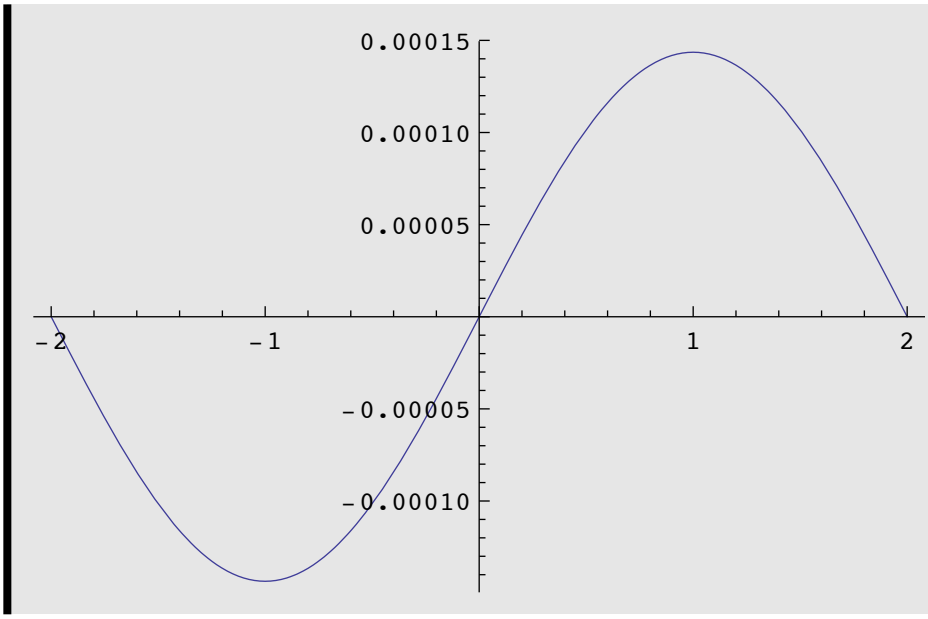
```
Max[Abs[mmacppdiff]]
```

5.44009×10^{-14}

Which is close enough. Now rerun in C++ with half the number of time steps. (M=200, N=80, alpha = 0.8).

```
cppdata3 = ReadList["fdoutput3.txt", Number];
```

```
cppinterpoldata = Table[{(k - nminus - 1) dx, cppdata3[[k]]},
  {k, 1, nminus + nplus + 1}];
cppfunc = Interpolation[cppinterpoldata, InterpolationOrder → 3];
Plot[cppfunc[x] - Sin[π x / 2] e^(1/4 (-π^2) 0.1`), {x, -2, 2}, PlotPoints → 50]
```



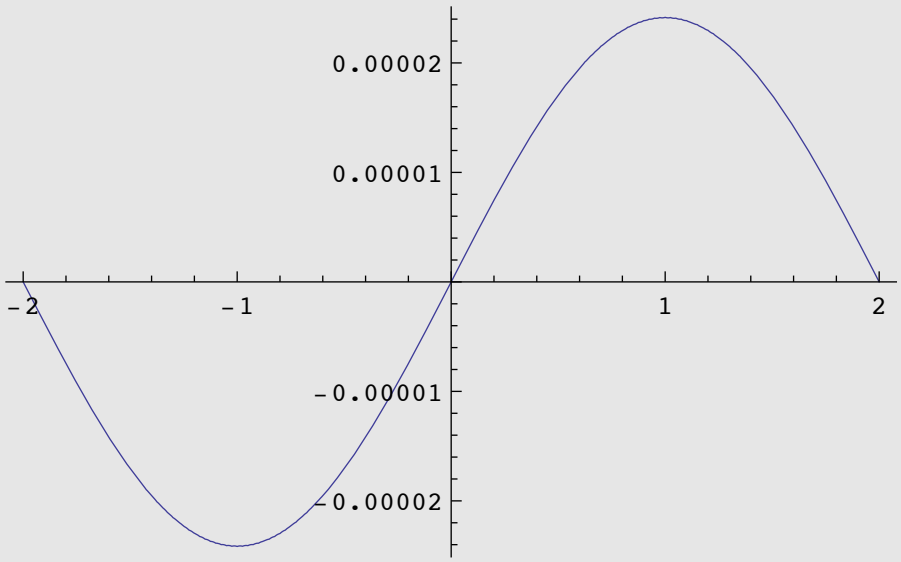
The error has grown, but not to order 10^100!!

Crank-Nicolson

The fully implicit analysis can be repeated with a Crank-Nicolson scheme by making some minor changes to the difference algorithm. **Note number of time steps is now 40:**

```
CNCMatrix[alpha_, nminus_, nplus_] :=
Sequence[Table[-alpha/2, {nplus+nminus-2}],
Table[1+alpha, {nplus+nminus-1}],
Table[-alpha/2, {nplus+nminus-2}]];

CNDMatrix[alpha_, vec_List] := Module[{temp},
temp = (1 - alpha)*vec + (alpha/2)*(RotateRight[vec] + RotateLeft[vec]
temp[[1]] = Simplify[First[temp] - alpha*Last[vec]/2];
temp[[-1]] = Simplify[Last[temp] - alpha*First[vec]/2];
temp];

M=40; nminus = 80; nplus = 80;
dx = 0.025; dtau = 0.0025; alpha = dtau/dx^2;
initial=Table[N[Sin[Pi*(k-1-nminus)/nminus] ], {k,1, nminus+nplus+1}];
lower=Table[0, {m, 1, M+1}]; upper=Table[0, {m, 1, M+1}];
wold = initial; wvold = wold; wnew = wold;
CMat = CNCMatrix[alpha,nminus,nplus];

For[m=2, m<=M+1, m++,
(wvold = wold; wold = wnew;
rhs = CNDMatrix[alpha, Take[wold, {2, -2}]]+
Table[Which[k==1, alpha*(lower[[m-1]] + lower[[m]])/2,
k== nplus + nminus-1,  alpha*(upper[[m-1]] + upper[[m]])/2,
True, 0],
{k, 1, nplus + nminus-1}];
temp = CompTridiagSolve[CMat, rhs];
wnew = Join[{lower[[m]]}, temp, {upper[[m]]}])
];

interpoldata =
Table[{(k-nminus-1)*dx ,wnew[[k]]}, {k, 1, nminus+nplus+1}];

ufunca = Interpolation[interpoldata, InterpolationOrder -> 3];
```

```
Plot[ufunca[x] - Sin[π x / 2] e^(1/4 (-π^2))^0.1` , {x, -2, 2}, PlotPoints → 50]
```




```
samples = TableForm[Join[{{"x", "Crank-Nic", "Exact", "Error"}},
Table[Map[PaddedForm[N[Chop[#1]],{5,6}]&,
N[{x, ufunca[x], Sin[Pi*x/2]*Exp[-(Pi^2 0.1)/4],
ufunca[x]- Sin[Pi*x/2]*Exp[-(Pi^2 0.1)/4]},5]],{x, -2, 2, 0.25}]]]
```

x	Crank-Nic	Exact	Error
-2.000000	0.000000	0.000000	0.000000
-1.750000	-0.299020	-0.299010	-9.246900×10^{-6}
-1.500000	-0.552510	-0.552490	-0.000017
-1.250000	-0.721890	-0.721870	-0.000022
-1.000000	-0.781370	-0.781340	-0.000024
-0.750000	-0.721890	-0.721870	-0.000022
-0.500000	-0.552510	-0.552490	-0.000017
-0.250000	-0.299020	-0.299010	-9.246900×10^{-6}
0.000000	0.000000	0.000000	0.000000
0.250000	0.299020	0.299010	9.246900×10^{-6}
0.500000	0.552510	0.552490	0.000017
0.750000	0.721890	0.721870	0.000022
1.000000	0.781370	0.781340	0.000024
1.250000	0.721890	0.721870	0.000022
1.500000	0.552510	0.552490	0.000017
1.750000	0.299020	0.299010	9.246900×10^{-6}
2.000000	0.000000	0.000000	0.000000

Note that we obtain comparable accuracy or better with a tenth the number of time-steps that where used for the explicit case. This example nicely illustrates the power of the implicit approach and the Crank-Nicholson scheme in particular. As we shall see, we can do rather better.

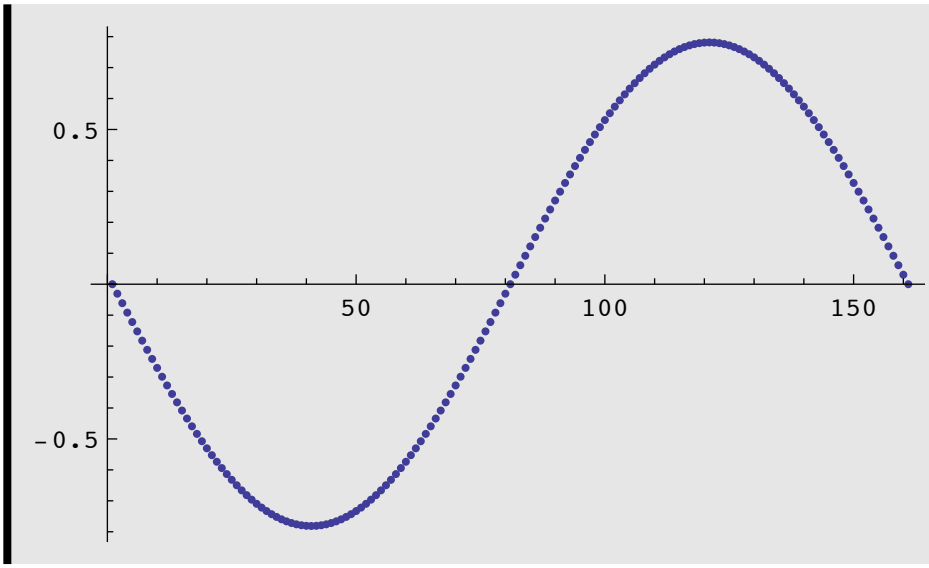
C++ Code

The C++ code is the same apart from the use of a different tridiagonal matrix and RHS:

```
for (j=0; j<=2*N-2;j++)
      {
          a[j] = -0.5*alpha;
          b[j] = 1.0+alpha;
          c[j] = -0.5*alpha;
      }

      for (i= 1; i<= M; i++)
      {
          for (j = 1; j<= 2*N-1; j++)
          {
              r[j-1]=(1.0-alpha)*solvals[i-1][j]+
              0.5*alpha*(solvals[i-1][j+1]+solval-
s[i-1][j-1]);
              }// The boundary conditions are zero so no
end effects
          tridiagsolve(a, b, c, r, x);
```

```
cppdata4 = ReadList["fdoutput4.txt", Number];
```

```
ListPlot[cppdata4]
```

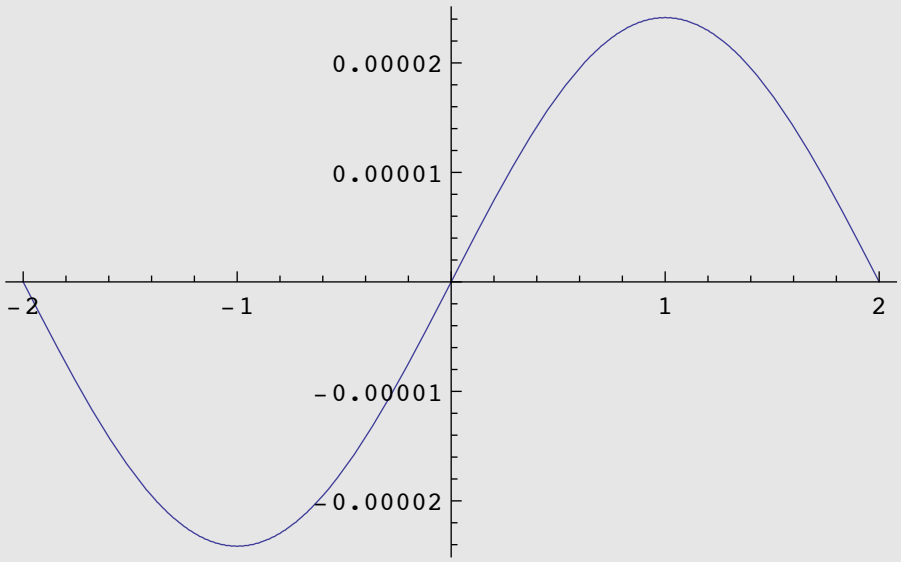



```
cppinterpoldata =
Table[{(k-nminus-1)*dx ,cppdata4[[k]]}, {k, 1, nminus+nplus+1}];
cppfunc = Interpolation[cppinterpoldata, InterpolationOrder -> 3]
```

```
InterpolatingFunction[{{-2., 2.}}, <>]
```

Now we plot the error in the answer:

```
Plot[cppfunc[x] - Sin[π x / 2] e^(1/4 (-π^2) 0.1`), {x, -2, 2}, PlotPoints → 50]
```




```
mmadata = Transpose[interpoldata][[2]];
mmacppdiff = mmadata - cppdata4;
Max[Abs[mmacppdiff]]
```

1.05471×10^{-14}

Douglas

A further small change to the matrices takes us to the Douglas algorithm.

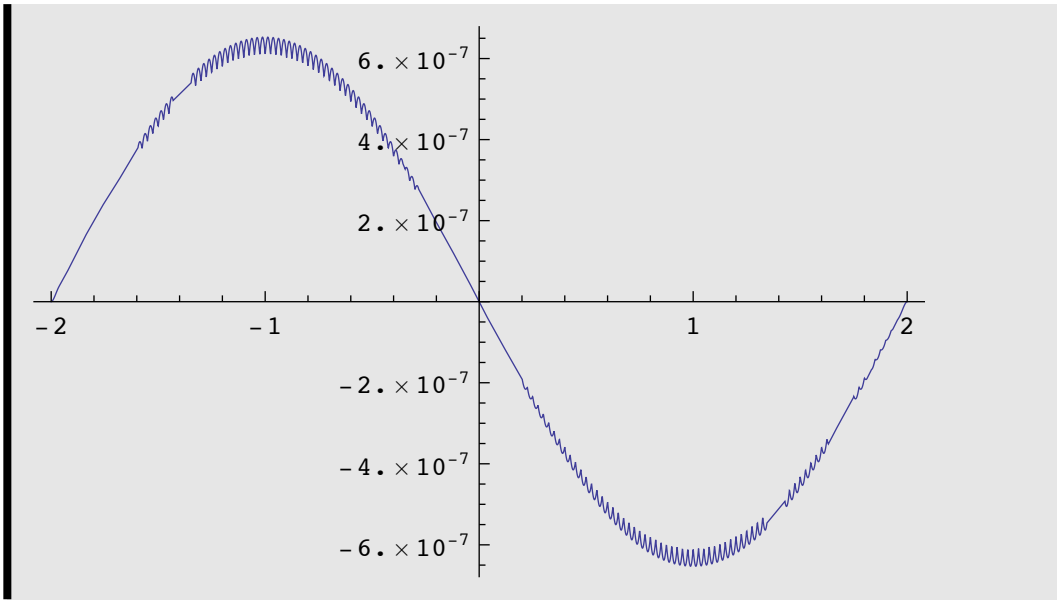
■ Douglas Algorithm

```
DougCMatrix[alpha_, nminus_, nplus_] :=
Sequence[Table[1-6*alpha, {nplus+nminus-2}], Table[10+12*alpha, {nplus-
Table[1-6*alpha, {nplus+nminus-2}]];

DougDMatrix[alpha_, vec_List] := Module[{temp},
temp = (10 - 12*alpha)*vec + (1+6*alpha)*(RotateRight[vec] + RotateLeft
temp[[1]] = Simplify[First[temp] - (1+6*alpha)*Last[vec]];
temp[[-1]] = Simplify[Last[temp] - (1 + 6*alpha)*First[vec]];
temp];

M=40;nminus = 80;nplus = 80;dx = 0.025;
dtau = 0.0025;alpha = dtau/dx^2;
Print[alpha];
CMat = DougCMatrix[alpha,nminus,nplus];
initial = Table[N[Sin[Pi*(k-1-nminus)/nminus] ], {k,1, nminus+nplus+1}]
lower=Table[0, {m, 1, M+1}];
upper=Table[0, {m, 1, M+1}];
wold = initial; wvold = wold; wnew = wold;

For[m=2, m<=M+1, m++,
(wvold = wold;
wold = wnew;
rhs = DougDMatrix[alpha, Take[wold, {2, -2}]]+
Table[
Which[
k==1, (6*alpha+1)*lower[[m-1]] +(6 alpha - 1)*lower[[m]],
k== nplus + nminus-1,
(6*alpha+1)*upper[[m-1]] + (6 alpha - 1)*upper[[m]],
True, 0],
{k, 1, nplus + nminus-1}];
temp = CompTridiagSolve[CMat, rhs];
wnew = Join[{lower[[m]]}, temp, {upper[[m]]}])];
interpoldata =
Table[{(k-nminus-1)*dx ,wnew[[k]]}, {k, 1, nminus+nplus+1}];
ufunc = Interpolation[interpoldata, InterpolationOrder -> 3];
```

```
4.
```

Now we plot the error - note the vertical scale!

```
Plot[ufunc[x] - Sin[π x / 2] e^(1/4 (-π^2) 0.1`), {x, -2, 2}, PlotPoints → 50]
```



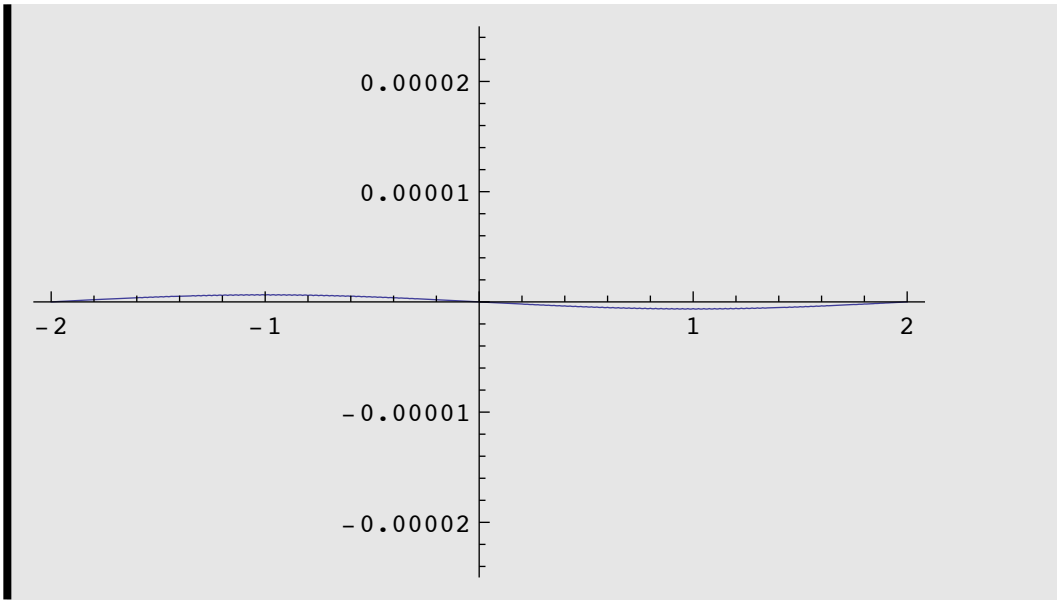
Finally we tabulate the *x*-value, the numerical result, the exact result and the error in the numerical scheme.

```
samples = TableForm[Join[{{"x", "Douglas", "Exact", "Error"}},
Table[Map[PaddedForm[N[Chop[#1]],{5,6}]&,
N[{x, ufunc[x], Sin[Pi*x/2]*Exp[-(Pi^2 0.1)/4],
ufunc[x]- Sin[Pi*x/2]*Exp[-(Pi^2 0.1)/4]},5]],
{x, -2, 2, 0.25}]]]
```

x	Douglas	Exact	Error
-2.000000	0.000000	0.000000	0.000000
-1.750000	-0.299010	-0.299010	2.332100×10^{-7}
-1.500000	-0.552490	-0.552490	4.309100×10^{-7}
-1.250000	-0.721870	-0.721870	5.630100×10^{-7}
-1.000000	-0.781340	-0.781340	6.094000×10^{-7}
-0.750000	-0.721870	-0.721870	5.630100×10^{-7}
-0.500000	-0.552490	-0.552490	4.309100×10^{-7}
-0.250000	-0.299010	-0.299010	2.332100×10^{-7}
0.000000	0.000000	0.000000	0.000000
0.250000	0.299010	0.299010	-2.332100×10^{-7}
0.500000	0.552490	0.552490	-4.309100×10^{-7}
0.750000	0.721870	0.721870	-5.630100×10^{-7}
1.000000	0.781340	0.781340	-6.094000×10^{-7}
1.250000	0.721870	0.721870	-5.630100×10^{-7}
1.500000	0.552490	0.552490	-4.309100×10^{-7}
1.750000	0.299010	0.299010	-2.332100×10^{-7}
2.000000	0.000000	0.000000	0.000000

Note that we obtain errors of about 1/40 *of those obtained with Crank-Nicholson, with identical time-step parameters*. Some oscillations have started to appear in our error plot, though it should be appreciated that our plot is now given on a much finer scale than was the case for CN - if we plot the Douglas errors on the same scale as the CN error plot the difference is clearer:

```
Plot[ufunc[x] - Sin[π x / 2] e^(1/4 (-π^2) 0.1`), {x, -2, 2}, PlotPoints → 50,
 PlotRange → {-0.000025`, 0.000025`}]
```



C++ Version

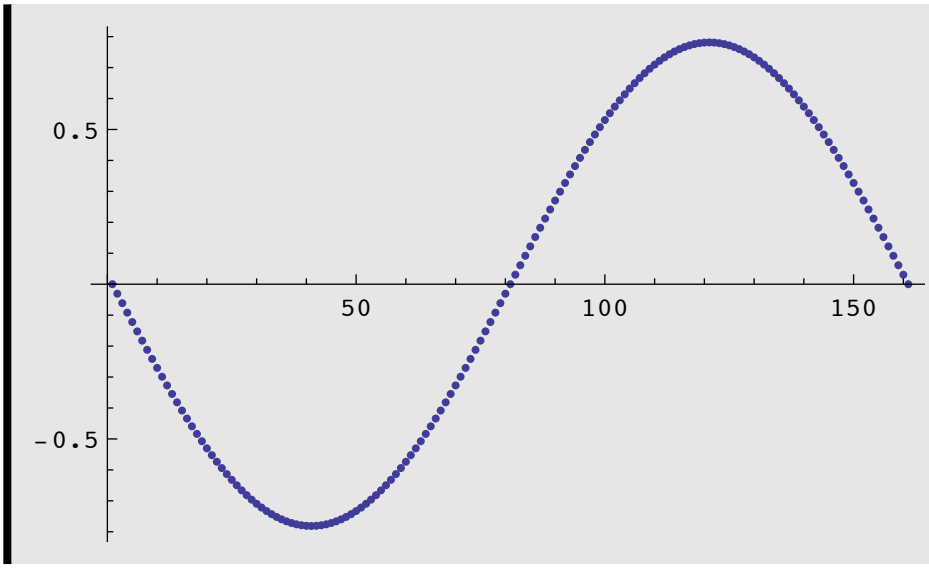
The C++ code again just has a different matrix evolution block:

```
for (j=0; j<=2*N-2;j++)
     {
         a[j] = 1.0-6.0*alpha;
         b[j] = 10.0+12.0*alpha;
         c[j] = 1.0-6.0*alpha;
     }

     for (i= 1; i<= M; i++)
     {
         for (j = 1; j<= 2*N-1; j++)
         {
             r[j-1]=(10.0-12.0*alpha)*solvals[i-1][j]+
```

```
(1+6.0*alpha)*(solvals[i-1][j+1]+solvals[i-1][j-1]);
                }// The boundary conditions are zero so no
end effects
            tridiagsolve(a, b, c, r, x);
```

```
cppdata5 = ReadList["fdoutput5.txt", Number];
```

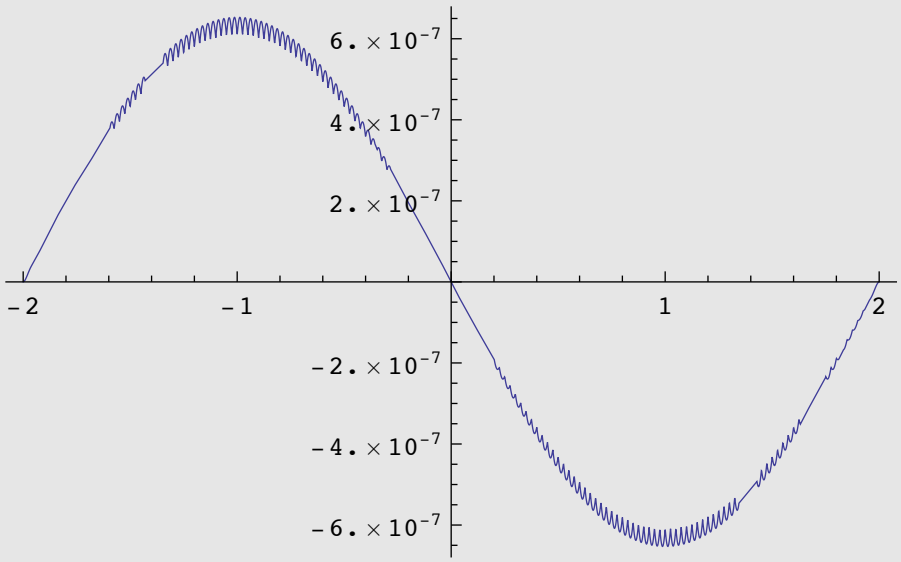
```
ListPlot[cppdata5]
```




```
cppinterpoldata =
Table[{(k-nminus-1)*dx ,cppdata5[[k]]}, {k, 1, nminus+npplus+1}];
cppfunc = Interpolation[cppinterpoldata, InterpolationOrder -> 3];
```

Now we plot the error in the answer:

```
Plot[cppfunc[x] - Sin[π x / 2] e^(1/4 (-π^2) 0.1`), {x, -2, 2}, PlotPoints → 50]
```




```
mmadata = Transpose[interpoldata][[2]];
mmacppdiff = mmadata - cppdata5;
Max[Abs[mmacppdiff]]
```

2.10942×10^{-15}

Theta Method - C++ code

```
#include <cmath>
#include <iostream>
#include <fstream>
#include "nrutil_nr.h" //allow access to public domain Numerical Recipes matrix routines
using namespace std;


void tridiagsolve(NRVec<double> &a, NRVec<double> &b, NRVec<double> &c, NRVec<double> &r, NRVec<double> &x)
{
    int j;
    int P=r.size();
    NRVec<double> u(P);
    NRVec<double> d(P);
    NRVec<double> l(P);
    NRVec<double> z(P);
    u[0]=c[0];
    d[0]=b[0];
    z[0]=r[0];
    // Do LU decomposition and forward subsititution as you go
    for (j=1;j<P;j++) {
        u[j]=c[j];
        l[j]=a[j]/d[j-1];
```

```
        d[j]=b[j]-l[j]*u[j-1];
        z[j] = r[j]-l[j]*z[j-1];
        }
    //follow with back substitution
    x[P-1] = z[P-1]/d[P-1];
    for (j=(P-2);j>=0;j--){
        x[j] = (z[j] - u[j]*x[j+1])/d[j];
        }
}

void ThetaImplicitDiffusion(int N, int M, double theta)
{
       int i, j;
       double dt = 0.1/M ;
       double dx = 2.0/N;;
       const double PI = 3.141592653589793;
       NRMat<double> solvals(0.0,M+1,2*N+1); //NR dynamic matrix
       double alpha = 0.0; // key diffusion parameter
       NRVec<double> r(0.0,2*N-1); // create vectors for implicit
solver
       NRVec<double> x(0.0,2*N-1);
       NRVec<double> a(0.0,2*N-1);
       NRVec<double> b(0.0,2*N-1);
       NRVec<double> c(0.0,2*N-1);

       alpha = dt/(dx*dx);
```

```
//output dt
cout << "dt = " << dt << "\n";
    //output dt
cout << "dx = " << dx << "\n";
// output alpha
cout << "alpha = " << alpha << "\n";

// initialize initital data

for (j = 0; j <= 2*N; j++)
{
    solvals[0][j] = sin(PI*(j-N)*dx/2);
}

// initialize boundary conditions

for (i = 1; i< M; i++)
{
    solvals[i][0] = 0;
    solvals[i][2*N] = 0;
    }
// Run the theta implicit algorithm
// First fill the arrays to feed to implicit solver
// Note form for theta implicit scheme
```

```
        for (j=0; j<=2*N-2;j++)
        {
            a[j] = -alpha*theta;
            b[j] = 1.0+2.0*alpha*theta;
            c[j] = -alpha*theta;
        }

        for (i= 1; i<= M; i++)
        {
            for (j = 1; j<= 2*N-1; j++)
            {
                r[j-1]=(1.0-2.0*alpha*(1-theta))*solvals[i-1][j]+al-
pha*(1-theta)*(solvals[i-1][j+1]+solvals[i-1][j-1]);
                }// The boundary conditions are zero so no end effects
            tridiagsolve(a, b, c, r, x);
            // Note that for a grid with constant parameters as here
            // It would be more efficient to make one call to get L and
U
            // before entering the evolution and then
            // only call the fwd and back substitution at each step
            // The version here would be more appropriate for the case
when
            // the matrix varied across the entire grid.
            for (j = 1; j<= 2*N-1; j++)
            {
```

```
            solvals[i][j] = x[j-1];
            }
        }

    ofstream out("fdoutput6.txt");
    for (j=0; j<= 2*N; j++)
    {   out.precision(15);
        out << solvals[M][j] << "\n";
        }
       return ;
}

#include <iostream>
#include <cmath>
using namespace std;
int main()
{
    double result;
    char name[20];
    int N;
    int M;
    double theta;
    cout << "Theta Method Implicit Diffusion Example \n ";
    cout << "Enter number of time steps (M) \n ";
    cin >> M;
```

```
    cout << M  << " time steps \n" ;
    cout << "Enter space step parameter (N) \n ";
    cin >> N;
    cout << 2*N  << " space steps \n" ;
    cout << "Enter evolution parameter (theta) \n ";
    cin >> theta;
    cout << 2*N  << " space steps \n" ;
    cout << " Result being written to fdoutput6.txt \n " << endl;
    ThetaImplicitDiffusion(N,M,theta);
    cout << "Hit any key+<RET> to finish \n ";
    cin >> name;
    return(0);
}
```