# Solving an Elliptic Partial Differential Equation by the Method of Finite Differences

In these notes I am going to apply the method of finite differences to solve an elliptic partial differential equation. The equation we are going to work with is the Poisson equation

$$\partial_{x,x} u(x, y) + \partial_{y,y} u(x, y) = f(x, y)$$

on a rectangular region $a \le x \le b$, $c \le y \le d$ with appropriate boundary conditions.

## Running the *Mathematica* code in this notebook

This notebook is divided into three sections. The code in these three sections is designed to be run independently. If you would like to run the commands in this notebook, I recommend that you quit and restart the kernel after each section to reset the definitions of the variables used here. If you don't do this, the definitions set up in one section may clash with those in a subsequent section.

---

# Introducing the Basic Method

In this section of the notes I will be demonstrating how the finite difference method works. I will make absolutely no attempt to do anything efficiently; instead, the emphasis in the first pass will be do things in the clearest and most straight-forward way.

To make the discussion more concrete, I will be working through a specific example from the text, example 2 in section 12.1.

## Setting up the difference equations

The method of finite differences works by subdividing the region in question into a grid of sample points. We typically associate a simple numbering system with the grid of points. In this example, each sample point is identified by a pair of indices (i,j) with i being the index in the x-direction and j the index in the y-direction. The lower left hand corner of the region, (a,c) is numbered sample point (0,0), while the upper right hand corner of the region, (c,d) is numbered (n,m).

The first step in the calculation is to convert the PDE into a difference equation. There will be a different difference equation associated with each of the sample points, so I set this up as a *Mathematica* function that maps the indices of a sample point to a difference equation for that sample point.

```
In[•]:= eqn[i_, j_] = (u[i + 1, j] - 2 u[i, j] + u[i - 1, j]) / h^2 +
          (u[i, j + 1] - 2 u[i, j] + u[i, j - 1]) / k^2 == f[x[i], y[j]]
```

Out[•]=

$$\frac{u[i, -1 + j] - 2\,u[i, j] + u[i, 1 + j]}{k^2} + \frac{u[-1 + i, j] - 2\,u[i, j] + u[1 + i, j]}{h^2} == f[x[i], y[j]]$$

The next set of definitions set up the sample point grid and put in the specifics we need to do this example.

```
In[•]:= f[x_, y_] = x Exp[y];
        a = 0.;
        b = 2.;
        c = 0.;
        d = 1.;
        n = 6;
        m = 5;

In[•]:= h = (b - a) / n;
        k = (d - c) / m;
        x[i_] = a + i h;
        y[j_] = c + j k;
```

Note in particular that the number of subdivisions in both the x and y directions are fairly modest. This will limit the number of sample points and thus the number of equations we have to solve.

In a minute we are going to generate a list of equations, with one equation for each sample point in the interior of our region. Before we do that, I need to identify which variables are associated with the interior points and which are boundary point variables. The following two list definitions do this.

```
In[•]:= innerVars = Flatten[ Table[u[i, j], {j, 1, m - 1}, {i, 1, n - 1}]]
Out[•]=
{u[1, 1], u[2, 1], u[3, 1], u[4, 1], u[5, 1], u[1, 2], u[2, 2], u[3, 2], u[4, 2], u[5, 2],
 u[1, 3], u[2, 3], u[3, 3], u[4, 3], u[5, 3], u[1, 4], u[2, 4], u[3, 4], u[4, 4], u[5, 4]}

In[•]:= leftVars = Table[u[0, j], {j, 0, m}];
        rightVars = Table[u[n, j], {j, 0, m}];
        bottomVars = Table[u[i, 0], {i, 1, n - 1}];
        topVars = Table[u[i, m], {i, 1, n - 1}];
        edgeVars = Flatten[{leftVars, rightVars, bottomVars, topVars}]
Out[•]=
{u[0, 0], u[0, 1], u[0, 2], u[0, 3], u[0, 4], u[0, 5],
 u[6, 0], u[6, 1], u[6, 2], u[6, 3], u[6, 4], u[6, 5], u[1, 0], u[2, 0],
 u[3, 0], u[4, 0], u[5, 0], u[1, 5], u[2, 5], u[3, 5], u[4, 5], u[5, 5]}
```

Now we are ready to generate the list of equations to solve. Note that these equations involve both the inner sample point variables (such as u[1,1]) and the boundary sample point variables (such as u[0,1] and u[1,0]).

In[•]:=
```
allEqns = Flatten[Table[Expand[eqn[i, j]], {j, 1, m - 1}, {i, 1, n - 1}]]
```

Out[•]=
```
{9. u[0, 1] + 25. u[1, 0] - 68. u[1, 1] + 25. u[1, 2] + 9. u[2, 1] == 0.407134,
 9. u[1, 1] + 25. u[2, 0] - 68. u[2, 1] + 25. u[2, 2] + 9. u[3, 1] == 0.814269,
 9. u[2, 1] + 25. u[3, 0] - 68. u[3, 1] + 25. u[3, 2] + 9. u[4, 1] == 1.2214,
 9. u[3, 1] + 25. u[4, 0] - 68. u[4, 1] + 25. u[4, 2] + 9. u[5, 1] == 1.62854,
 9. u[4, 1] + 25. u[5, 0] - 68. u[5, 1] + 25. u[5, 2] + 9. u[6, 1] == 2.03567,
 9. u[0, 2] + 25. u[1, 1] - 68. u[1, 2] + 25. u[1, 3] + 9. u[2, 2] == 0.497275,
 9. u[1, 2] + 25. u[2, 1] - 68. u[2, 2] + 25. u[2, 3] + 9. u[3, 2] == 0.99455,
 9. u[2, 2] + 25. u[3, 1] - 68. u[3, 2] + 25. u[3, 3] + 9. u[4, 2] == 1.49182,
 9. u[3, 2] + 25. u[4, 1] - 68. u[4, 2] + 25. u[4, 3] + 9. u[5, 2] == 1.9891,
 9. u[4, 2] + 25. u[5, 1] - 68. u[5, 2] + 25. u[5, 3] + 9. u[6, 2] == 2.48637,
 9. u[0, 3] + 25. u[1, 2] - 68. u[1, 3] + 25. u[1, 4] + 9. u[2, 3] == 0.607373,
 9. u[1, 3] + 25. u[2, 2] - 68. u[2, 3] + 25. u[2, 4] + 9. u[3, 3] == 1.21475,
 9. u[2, 3] + 25. u[3, 2] - 68. u[3, 3] + 25. u[3, 4] + 9. u[4, 3] == 1.82212,
 9. u[3, 3] + 25. u[4, 2] - 68. u[4, 3] + 25. u[4, 4] + 9. u[5, 3] == 2.42949,
 9. u[4, 3] + 25. u[5, 2] - 68. u[5, 3] + 25. u[5, 4] + 9. u[6, 3] == 3.03686,
 9. u[0, 4] + 25. u[1, 3] - 68. u[1, 4] + 25. u[1, 5] + 9. u[2, 4] == 0.741847,
 9. u[1, 4] + 25. u[2, 3] - 68. u[2, 4] + 25. u[2, 5] + 9. u[3, 4] == 1.48369,
 9. u[2, 4] + 25. u[3, 3] - 68. u[3, 4] + 25. u[3, 5] + 9. u[4, 4] == 2.22554,
 9. u[3, 4] + 25. u[4, 3] - 68. u[4, 4] + 25. u[4, 5] + 9. u[5, 4] == 2.96739,
 9. u[4, 4] + 25. u[5, 3] - 68. u[5, 4] + 25. u[5, 5] + 9. u[6, 4] == 3.70923}
```

## Solving the difference equations

Since the PDE we are working with in this example is linear, the difference equations we have to solve are likewise linear.

This suggests that we should try to rewrite our problem as a matrix equation A z = B and solve for z.

The next set of commands construct the matrix A. They do this by going into the equations and asking for the coefficients connected to each of the interior sample point variables.

In[•]:=
```
A = Table[Coefficient[allEqns[[i, 1]], innerVars[[j]]],
    {i, 1, Length[allEqns]}, {j, 1, Length[innerVars]}];
```

In[•]:=
```
MatrixForm[A]
```

Out[•]//MatrixForm=

$$\begin{pmatrix}
-68. & 9. & 0 & 0 & 0 & 25. & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
9. & -68. & 9. & 0 & 0 & 0 & 25. & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 9. & -68. & 9. & 0 & 0 & 0 & 25. & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 9. & -68. & 9. & 0 & 0 & 0 & 25. & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 9. & -68. & 0 & 0 & 0 & 0 & 25. & 0 & 0 & 0 & 0 & 0 \\
25. & 0 & 0 & 0 & 0 & -68. & 9. & 0 & 0 & 0 & 25. & 0 & 0 & 0 & 0 \\
0 & 25. & 0 & 0 & 0 & 9. & -68. & 9. & 0 & 0 & 0 & 25. & 0 & 0 & 0 \\
0 & 0 & 25. & 0 & 0 & 0 & 9. & -68. & 9. & 0 & 0 & 0 & 25. & 0 & 0 \\
0 & 0 & 0 & 25. & 0 & 0 & 0 & 9. & -68. & 9. & 0 & 0 & 0 & 25. & 0 \\
0 & 0 & 0 & 0 & 25. & 0 & 0 & 0 & 9. & -68. & 0 & 0 & 0 & 0 & 25. \\
0 & 0 & 0 & 0 & 0 & 25. & 0 & 0 & 0 & 0 & -68. & 9. & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 25. & 0 & 0 & 0 & 9. & -68. & 9. & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 25. & 0 & 0 & 0 & 9. & -68. & 9. & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 25. & 0 & 0 & 0 & 9. & -68. & 9. \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 25. & 0 & 0 & 0 & 9. & -68. \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 25. & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 25. & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 25. & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 25. & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 25.
\end{pmatrix}$$

The left hand sides of many of our equations also contain factors involving the edge variables. Since the edge variables are determined by the boundary conditions, those terms have to be moved over to the right hand side of the equations and ultimately incorporated into B. The following code isolates the terms that involve the edge variables and moves them into B.

In[•]:=
```
B = Table[allEqns[[i, 2]] - Sum[Coefficient[allEqns[[i, 1]], edgeVars[[j]]] × edgeVars[[j]],
    {j, 1, Length[edgeVars]}], {i, 1, Length[allEqns]}]
```

Out[•]=
```
{0.407134 - 9. u[0, 1] - 25. u[1, 0], 0.814269 - 25. u[2, 0], 1.2214 - 25. u[3, 0],
 1.62854 - 25. u[4, 0], 2.03567 - 25. u[5, 0] - 9. u[6, 1], 0.497275 - 9. u[0, 2], 0.99455,
 1.49182, 1.9891, 2.48637 - 9. u[6, 2], 0.607373 - 9. u[0, 3], 1.21475, 1.82212,
 2.42949, 3.03686 - 9. u[6, 3], 0.741847 - 9. u[0, 4] - 25. u[1, 5], 1.48369 - 25. u[2, 5],
 2.22554 - 25. u[3, 5], 2.96739 - 25. u[4, 5], 3.70923 - 25. u[5, 5] - 9. u[6, 4]}
```

The last step needed to construct our system A z = B is to introduce the boundary conditions. These conditions tell us what values the edge variables should take.

In[•]:=
```
u[0, j_] = 0;
u[n, j_] = 2 Exp[y[j]];
u[i_, 0] = x[i];
u[i_, m] = x[i] E;
```

With the boundary conditions now set, B reduces to a vector of numbers, and our system A z = B is complete and ready to solve.

In[•]:=
```
B
```
Out[•]=
```
{-7.9262, -15.8524, -23.7786, -31.7048, -61.6162, 0.497275,
 0.99455, 1.49182, 1.9891, -24.3665, 0.607373, 1.21475, 1.82212,
 2.42949, -29.7613, -21.9105, -43.821, -65.7315, -87.642, -149.612}
```

Because this system is fairly modest in size, we can solve in a fairly crude way.

In[•]:=
```
z = LinearSolve[A, B]
```
Out[•]=
```
{0.407265, 0.814524, 1.22177, 1.62896, 2.03604, 0.497483,
 0.994958, 1.4924, 1.98978, 2.48696, 0.607596, 1.21518, 1.82274,
 2.43023, 3.03751, 0.742007, 1.48401, 2.22599, 2.96793, 3.70972}
```

## Constructing the solution

The last list we generated is a list of numerical values we want to apply to our inner sample point variables. The next *Mathematica* command goes down the list of inner sample point variables and sets each one equal to the corresponding number in z.

In[•]:=
```
MapThread[Set, {innerVars, z}]
```
Out[•]=
```
{0.407265, 0.814524, 1.22177, 1.62896, 2.03604, 0.497483,
 0.994958, 1.4924, 1.98978, 2.48696, 0.607596, 1.21518, 1.82274,
 2.43023, 3.03751, 0.742007, 1.48401, 2.22599, 2.96793, 3.70972}
```

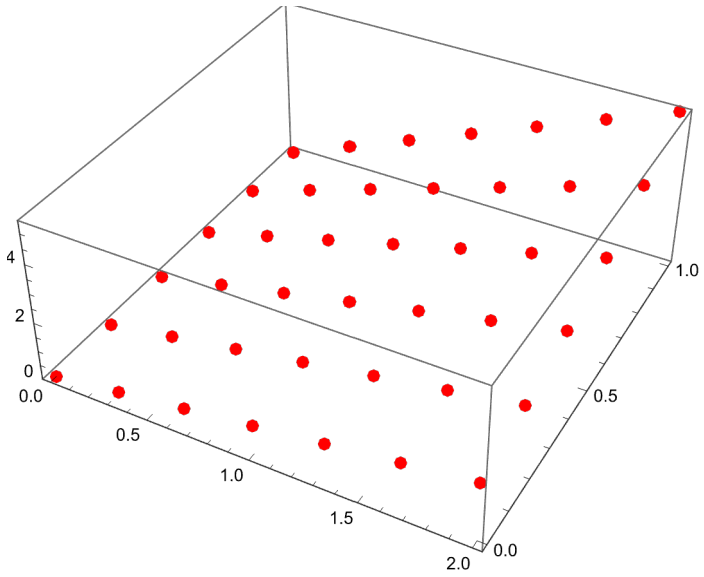
We now have a complete set of values. We have values for both the edge and inner sample points. Here is a table of x, y, and u values for all of our sample points.

In[•]:=
```
pts = Table[{x[i], y[j], u[i, j]}, {i, 0, n}, {j, 0, m}]
```
Out[•]=
```
{{{0., 0., 0}, {0., 0.2, 0}, {0., 0.4, 0}, {0., 0.6, 0}, {0., 0.8, 0}, {0., 1., 0}},
 {{0.333333, 0., 0.333333}, {0.333333, 0.2, 0.407265}, {0.333333, 0.4, 0.497483},
  {0.333333, 0.6, 0.607596}, {0.333333, 0.8, 0.742007}, {0.333333, 1., 0.906094}},
 {{0.666667, 0., 0.666667}, {0.666667, 0.2, 0.814524}, {0.666667, 0.4, 0.994958},
  {0.666667, 0.6, 1.21518}, {0.666667, 0.8, 1.48401}, {0.666667, 1., 1.81219}},
 {{1., 0., 1.}, {1., 0.2, 1.22177}, {1., 0.4, 1.4924},
  {1., 0.6, 1.82274}, {1., 0.8, 2.22599}, {1., 1., 2.71828}},
 {{1.33333, 0., 1.33333}, {1.33333, 0.2, 1.62896}, {1.33333, 0.4, 1.98978},
  {1.33333, 0.6, 2.43023}, {1.33333, 0.8, 2.96793}, {1.33333, 1., 3.62438}},
 {{1.66667, 0., 1.66667}, {1.66667, 0.2, 2.03604}, {1.66667, 0.4, 2.48696},
  {1.66667, 0.6, 3.03751}, {1.66667, 0.8, 3.70972}, {1.66667, 1., 4.53047}},
 {{2., 0., 2.}, {2., 0.2, 2.44281}, {2., 0.4, 2.98365},
  {2., 0.6, 3.64424}, {2., 0.8, 4.45108}, {2., 1., 5.43656}}}
```

Here is a plot of those points in space.

In[•]:=
```
plot2 = ListPointPlot3D[ArrayFlatten[pts, 1], PlotStyle → Red]
```

Out[•]=




## Judging the quality of the solution

Since this is a textbook example, we have the exact solution at our disposal.

In[•]:=
```
w[x_, y_] = x Exp[y]
```

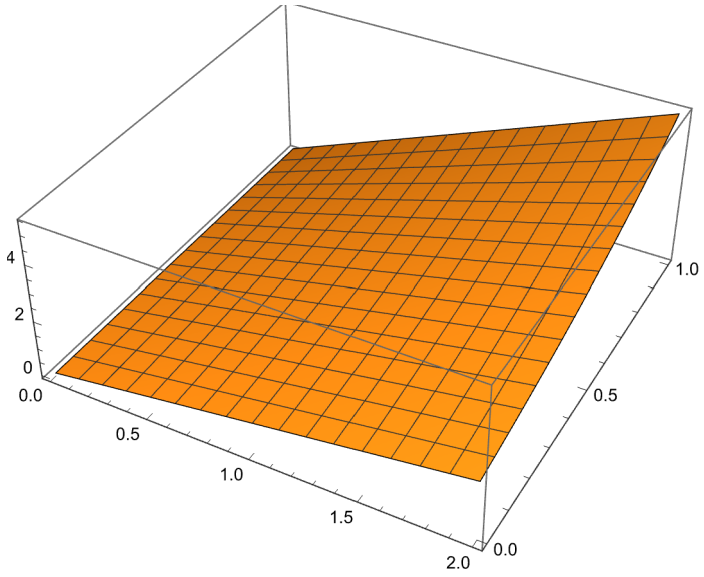
Out[•]=

$e^{y}\, x$

Here is a plot of the exact solution.

In[•]:=
```
plot1 = Plot3D[w[x, y], {x, a, b}, {y, c, d}]
```

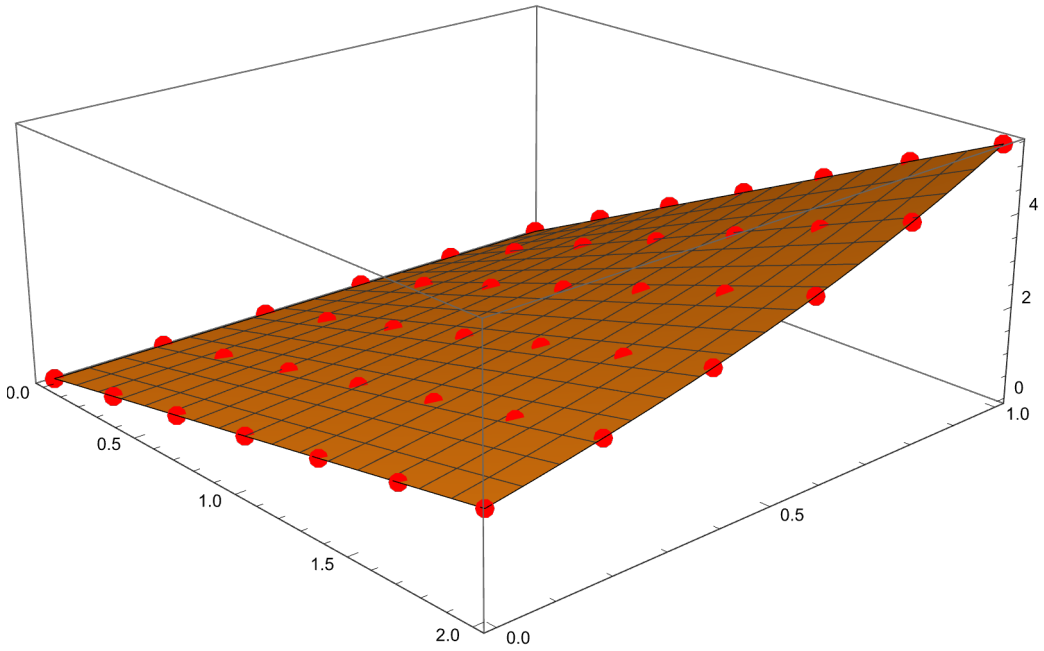
Out[•]=




Finally, here are the estimate and the exact solution plotted side by side. This demonstrates visually

that the estimated values for u[x,y] are not too bad.

In[ ]:=
```
Show[plot1, plot2]
```
Out[ ]=




## Critiquing the method

Although the steps we followed above are all correct and produce reasonable results, the method employed does not scale well to problems with larger values of m and n. There are three specific weaknesses that will cause trouble when we scale up the system:

1) We leaned pretty heavily on *Mathematica*'s symbolic manipulation capabilities to construct the matrix A for us. This takes both time and memory, and may not work too well when the problem scales up.

2) When the number of sample points gets large, the size of A goes up as the square of the number of sample points. For example, if we used m = 100 and n = 50, A would end up being a matrix with roughly 5000 rows and 5000 columns. Just storing such a large matrix is going to put strain on *Mathematica*.

3) When A gets very large, we have to use a more efficient method to solve the system A z = B. The method used here is both too slow and takes up too much space.

# The first alternative: SOR with sparse matrices

The first alternative method I am going to pursue is aimed at bypassing issue #1 above completely and moderating the impact of issue #2.

## Setting up the problem

For this iteration I am going to use a different problem. The problem for this example comes from exercise 3c in section 12.1. The following statements set up the problem.

In[ ]:=
```
f[x_, y_] = (x^2 + y^2) Exp[x y];
a = 0.;
b = 2.;
c = 0.;
d = 1.;
n = 70;
m = 40;
```

In[ ]:=
```
h = (b - a) / n;
k = (d - c) / m;
x[i_] = a + i h;
y[j_] = c + j k;
```

Note that this time around we are using larger values of n and m to put greater strain on the system.

Here are the boundary conditions for this example.

In[ ]:=
```
Clear[u]
u[0, j_] = 1.;
u[n, j_] = Exp[2. y[j]];
u[i_, 0] = 1.;
u[i_, m] = Exp[x[i]];
```

## Constructing A

You may have noticed in the first round above that the matrix A we generated has a particularly simple and symmetric form. This suggests that we should try to come up with a simple formula to generate the elements of A and bypass the symbolic manipulation needed to construct A. The starting point for this is the difference equations we have to solve. If we let u(i,j) be the value of the estimated solution at sample point (i,j), the associated difference equation is

$$\frac{2\,u(i,j) - u(i-1,j) - u(i+1,j)}{h^2} + \frac{2\,u(i,j) - u(i,j-1) - u(i,j+1)}{k^2} = -f(x(i), y(j))$$

(Note that here I have multiplied both sides of the original PDE by a factor of -1. I need to do this to guarantee that the matrix A that results is positive-definite. The method in the section after this

requires that A be positive-definite.)

The first step of this method is to introduce a linear numbering system for the interior grid points. We can do this by introducing an index q and using the following formula to translate from (i,j) coordinates to the q index:

q = (j-1)*(n-1) + i

Note that the difference equation above has terms for exactly five sample points, (i,j) and its four immediate neighbors. If we use the formula above to convert indices we get the following:

$(i, j) \to q$
$(i-1, j) \to q - 1$
$(i+1, j) \to q + 1$
$(i, j-1) \to q - n + 1$
$(i, j+1) \to q + n - 1$

with these formulas, we can rewrite the difference equations in terms of variables $u_q$ that use the linear numbering system:

$$-\frac{1}{k^2} u_{q-n+1} - \frac{1}{h^2} u_{q-1} + 2\left(\frac{1}{h^2} + \frac{1}{k^2}\right) u_q - \frac{1}{h^2} u_{q+1} - \frac{1}{k^2} u_{q+n-1} = -f(x(i(q)), y(j(q)))$$

This gives us the pattern we need to construct the rows of A. What we can read off from this is that each row will have at most five non-zero terms which are the coefficients of the five terms on the left. For points close to the boundary, some of the terms will not be unknowns, but will be determined by the boundary conditions and reduce to numbers. Ultimately, those numerical terms will have to be moved over to the right-hand side and incorporated into B.

One useful way to build the matrix A is to use *Mathematica*'s SparseArray function. This function is used to set up matrices where most of the elements are expected to be 0. We provide a list of pattern rules that specify which entries will be non-zero. Here is the statement we would execute to use the pattern we discovered above to build A directly:

In[•]:=
```
A = SparseArray[
    {{q_, q_} → 2 / h^2 + 2 / k^2, {q_, r_} /; And[r == q - 1, Mod[q, n - 1] ≠ 1] → -1 / h^2,
     {q_, r_} /; And[r == q + 1, Mod[q, n - 1] > 0] → -1 / h^2, {q_, r_} /; r == q - n + 1 →
      -1 / k^2, {q_, r_} /; r == q + n - 1 → -1 / k^2}, {(n - 1) * (m - 1), (n - 1) * (m - 1)}];
```

SparseArray offers one further advantage which is very useful here, and that is that it is the most efficient way to store matrics with many 0s. Only the non-zero entries take up space, and no space is used for the remaining entries.

## Constructing B

The next step is construct the B for the right-hand side of our system. The first thing that goes into B is terms derived from the $-f(x, y)$ that appears on the right hand side of the differential equation. For this step, it is most convenient to use the original (i,j) numbering scheme for the interior sample points.

```
In[•]:= B = Table[-f[x[i], y[j]], {j, 1, m - 1}, {i, 1, n - 1}];
```

For interior points near the boundary, some of the terms in the corresponding equations will reduce to numbers determined by the boundary conditions. Those numbers have to be moved over to the right hand side of the system and get incorporated into B. The best way to handle this is to iterate over the four edges of the boundary and add the appropriate terms to B as we go along. The following commands do that.

```
In[•]:= For[j = 1, j < m, j++, B[[j, 1]] += u[0, j] / h^2];
        For[j = 1, j < m, j++, B[[j, n - 1]] += u[n, j] / h^2];
        For[i = 1, i < n, i++, B[[1, i]] += u[i, 0] / k^2];
        For[i = 1, i < n, i++, B[[m - 1, i]] += u[i, m] / k^2];
```

Note that B is still in the form of a two-dimensional grid. The last thing we need to do with it is to turn into a vector. The *Mathematica* Flatten command is useful for this.

```
In[•]:= B = Flatten[B];
```

## Solving the System

Because our matrix A is very large, we have to use a more efficient method to solve the very large system A x = B. The method I am going to use is the SOR method described in section 7.3 of the text.

To apply SOR, we have to decompose A into matrices D, L, and U. Rather than decompose A, I will construct these three matrices directly with SparseArray:

```
In[•]:= mD = SparseArray[{{i_, i_} → 2 / h^2 + 2 / k^2}, {(n - 1) * (m - 1), (n - 1) * (m - 1)}];
        mL = SparseArray[{{i_, j_} /; And[j == i - 1, Mod[i, n - 1] ≠ 1] → 1 / h^2,
             {i_, j_} /; j == i - n + 1 → 1 / k^2}, {(n - 1) * (m - 1), (n - 1) * (m - 1)}];
        mU = SparseArray[{{i_, j_} /; And[j == i + 1, Mod[i, n - 1] > 0] → 1 / h^2,
             {i_, j_} /; j == i + n - 1 → 1 / k^2}, {(n - 1) * (m - 1), (n - 1) * (m - 1)}];
```

The SOR method requires us to pick an over-relaxation parameter $\omega$. In section 12.1, the textbook recommends that we use the following formulas to pick an optimal value for $\omega$:

```
In[•]:= rho = (Cos[Pi / m] + Cos[Pi / n]) / 2.
Out[•]=
        0.997955

In[•]:= omega = 2 / (1. + Sqrt[1 - rho^2])
Out[•]=
        1.87985
```

We are now ready to set up and use the SOR iteration.

```
In[•]:= tOmega = Inverse[mD - omega mL].((1 - omega) mD + omega mU);
```

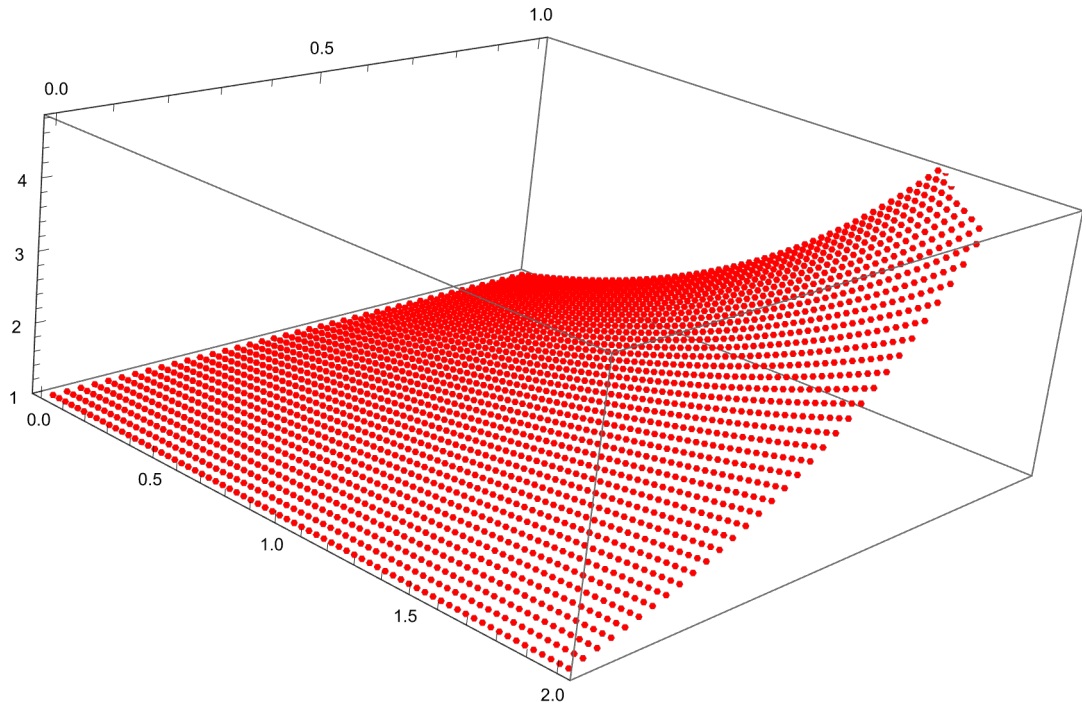
```
In[•]:= cOmega = omega Inverse[mD - omega mL].B;

In[•]:= fOmega[x_] := tOmega.x + cOmega

In[•]:= startGuess = Table[1., {(n - 1) * (m - 1)}];
```

After a moderate number of iterations, this method converges to a reasonable result.

```
In[•]:= z = Nest[fOmega, startGuess, 200];

In[•]:= innerVars = Flatten[ Table[u[i, j], {j, 1, m - 1}, {i, 1, n - 1}]];

In[•]:= MapThread[Set, {innerVars, z}];
```

We can now construct a set of points for this approximate solution and plot them.

```
In[•]:= pts = Table[{x[i], y[j], u[i, j]}, {i, 0, n}, {j, 0, m}];

In[•]:= ListPointPlot3D[ArrayFlatten[pts, 1], PlotStyle → Red]
Out[•]=
```




The problem we are solving has an exact solution, so we can compute and plot errors for each of our sample points.

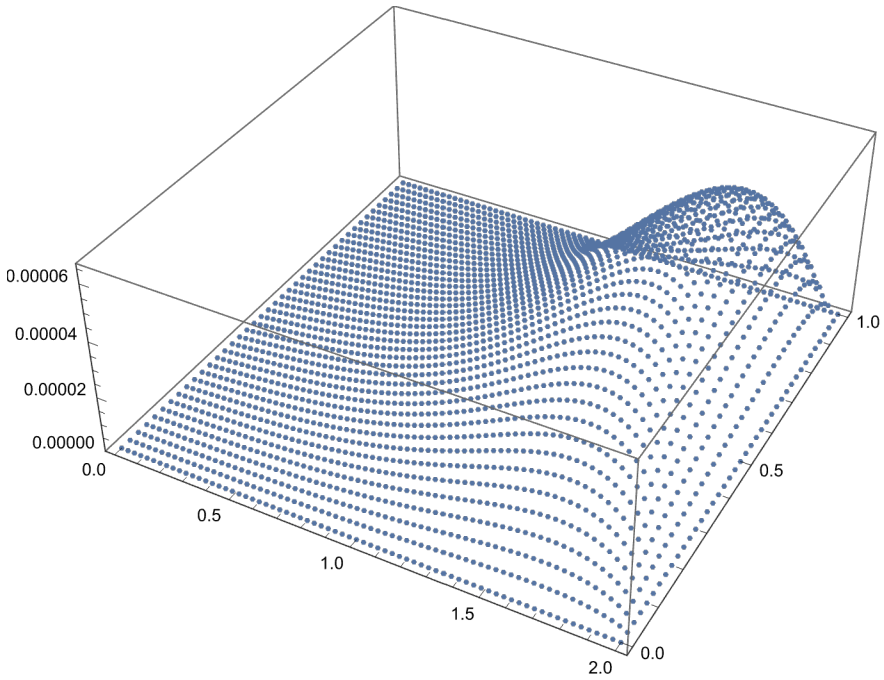
```
In[•]:= soln[x_, y_] = Exp[x y]
Out[•]=
```

$$e^{x\,y}$$

```
In[•]:= errors = Table[{x[i], y[j], u[i, j] - soln[x[i], y[j]]}, {i, 0, n}, {j, 0, m}];
```

In[ ]:= `ListPointPlot3D[ArrayFlatten[errors, 1]]`

Out[ ]=




This shows that the results are quite reasonable.

### Problems with this method

Although this method performs much better than the crude method I used in the first section, it still has problems. The main problem is that the matrix T used in the SOR method is quite large, and takes up a lot of memory. In fact, if we increase the values of m and n a bit more, *Mathematica* actually runs out of memory in the middle of computing tOmega.

The next alternative attempts to aggressively trade space for time, using less storage space while taking slightly more time to do the calculation.

## The second alternative: solving exactly via the Cholesky decomposition

The central problem we have to solve is the linear system A z = B. As we have seen in chapters 6 and 7, there are multiple alternative approaches available for solving linear systems of equations. In this iteration, we will use another method to solve the system.

### Setting up the method

The method we are going to use this time takes advantage of a special property of A. It turns out that when you use the difference equation method to solve an elliptic PDE the matrix A you end up construct-

ing is always a symmetric, positive-definite matrix. Such matrices can be factored into lower triangular and upper triangular matrices via the Cholesky decomposition. This is the approach we are going to use. We will convert the original system

$$A z = B$$

into

$$L L^T z = B$$

via the Cholesky decomposition and hence into a pair of equations

$$L w = B$$
$$L^T z = w$$

Since both of the matrices in the latter two systems are triangular, both systems can be solved relatively quickly by back-substitution.

## Running this method

Here now is the code to run this method.

In[ ]:=
```
Clear[f]
f[x_, y_] := (x^2 + y^2) Exp[x y];
a = 0.;
b = 2.;
c = 0.;
d = 1.;
n = 50;
m = 30;
```

In[ ]:=
```
h = (b - a) / n;
k = (d - c) / m;
Clear[x]
x[i_] := a + i h;
Clear[y]
y[j_] := c + j k;
```

Note that I am using slightly more moderate values for m and n here. This is because the method in this section is slower. If you have the time to waste, you can try running this section with larger values of n and m.

```
In[•]:= A = SparseArray[
          {{i_, i_} → 2 / h^2 + 2 / k^2, {i_, j_} /; And[j == i - 1, Mod[i, n - 1] ≠ 1] → -1 / h^2,
           {i_, j_} /; And[j == i + 1, Mod[i, n - 1] ≠ 0] → -1 / h^2, {i_, j_} /; j == i - n + 1 →
            -1 / k^2, {i_, j_} /; j == i + n - 1 → -1 / k^2}, {(n - 1) * (m - 1), (n - 1) * (m - 1)}];

In[•]:= B = Table[-f[x[i], y[j]], {j, 1, m - 1}, {i, 1, n - 1}];

In[•]:= Clear[u]
        u[0, j_] = 1.;
        u[n, j_] = Exp[2. y[j]];
        u[i_, 0] = 1.;
        u[i_, m] = Exp[x[i]];

In[•]:= For[j = 1, j < m, j++, B[[j, 1]] += u[0, j] / h^2];
        For[j = 1, j < m, j++, B[[j, n - 1]] += u[n, j] / h^2];
        For[i = 1, i < n, i++, B[[1, i]] += u[i, 0] / k^2];
        For[i = 1, i < n, i++, B[[m - 1, i]] += u[i, m] / k^2];

In[•]:= B = Flatten[B];
```

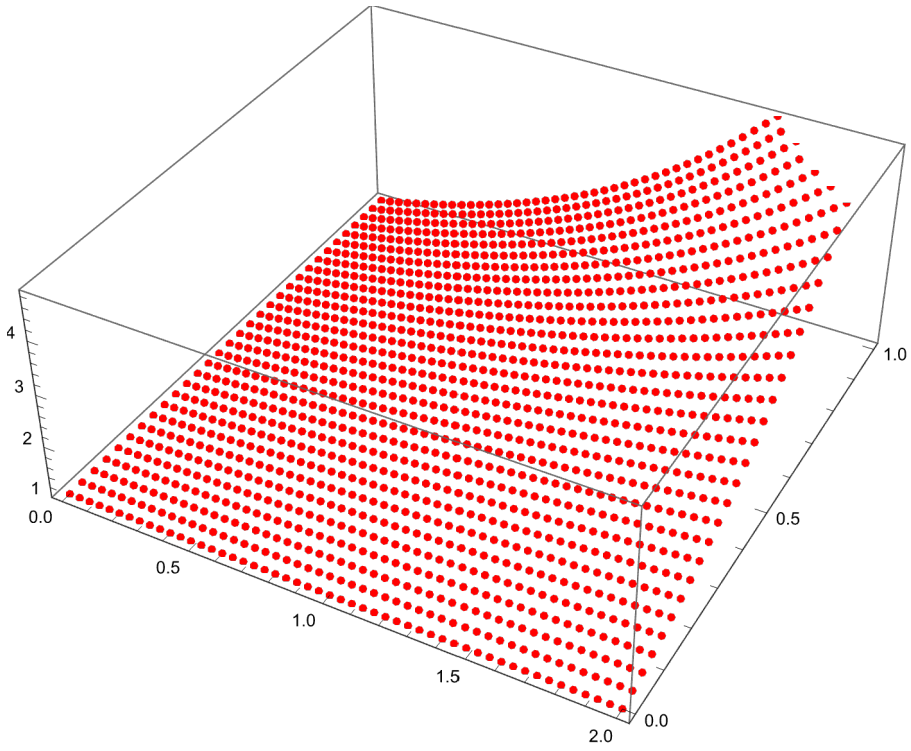
*Mathematica*'s built-in method for computing the Cholesky decomposition is fairly efficient.

```
In[•]:= mL = CholeskyDecomposition[A];
```

The next step is to do the back-substitutions. These functions carry out those steps. USolve is designed to solve a system U x = c where U is an upper triangular matrix, while LSolve solves L x = b for a lower triangular L. Note that USolve assumes that the parameter u is an upper triangular matrix. The matrix mL returned by *Mathematica*'s CholeskyDecomposition function is upper triangular, so I will pass mL directly to USolve. Since I don't want to have to actually compute the transpose of mL to pass to LSolve, LSolve also assumes that its parameter l is an upper triangular matrix. It compensates for that by switching indices when it actually uses that parameter.

```
In[•]:= USolve[u_, c_] := Module[{y, n},
          n = Length[c];
          y = Table[0., {n}];
          For[i = n, i > 0, i--, y[[i]] = (c[[i]] - Sum[u[[i, j]] × y[[j]], {j, i + 1, n}]) / u[[i, i]];
          y]

In[•]:= LSolve[l_, b_] := Module[{x, n},
          n = Length[b];
          x = Table[0., {n}];
          For[i = 1, i ≤ n, i++, x[[i]] = (b[[i]] - Sum[l[[j, i]] × x[[j]], {j, 1, i - 1}]) / l[[i, i]];
          x]

In[•]:= Timing[w = LSolve[mL, B]];

In[•]:= %[[1]]
Out[•]=
        309.417
```

```
In[•]:= Timing[z = USolve[mL, w]];

In[•]:= %[[1]]
Out[•]=
        308.097

In[•]:= innerVars = Flatten[ Table[u[i, j], {j, 1, m - 1}, {i, 1, n - 1}]];

In[•]:= MapThread[Set, {innerVars, z}];

In[•]:= pts = Table[{x[i], y[j], u[i, j]}, {i, 0, n}, {j, 0, m}];

In[•]:= ListPointPlot3D[ArrayFlatten[pts, 1], PlotStyle → Red]
Out[•]=
```




Here are the errors for this method.

```
In[•]:= soln[x_, y_] = Exp[x y]
Out[•]=
        e^(x y)

In[•]:= errors = Table[{x[i], y[j], u[i, j] - soln[x[i], y[j]]}, {i, 0, n}, {j, 0, m}];
```

In[ ]:= `ListPointPlot3D[ArrayFlatten[errors, 1]]`

Out[ ]=