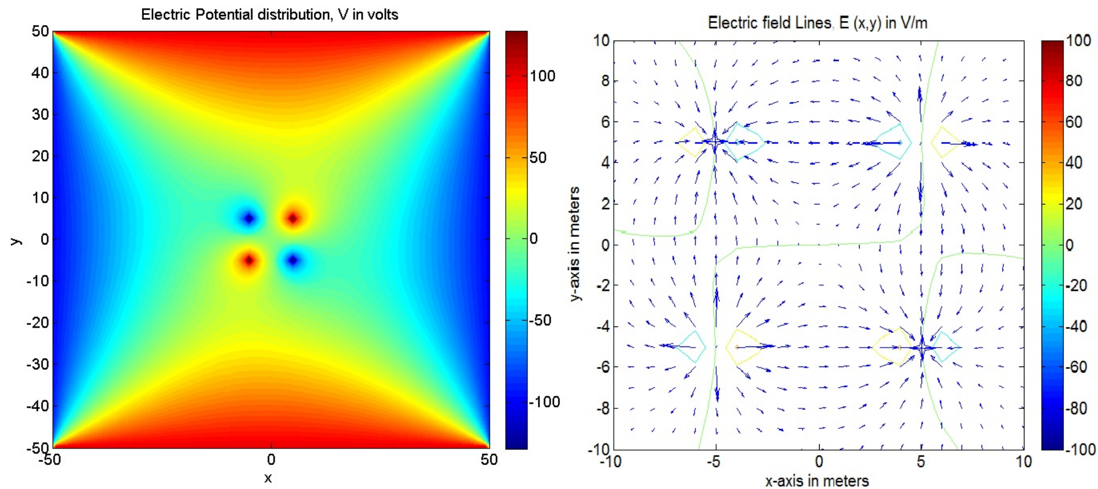
The steady 2D Poisson equation

MATH1091: ODE methods for a reaction diffusion equation

http://people.sc.fsu.edu/∼jburkardt/classes/math1091_2020/poisson_2d_steady/poisson_2d_steady.pdf



Electric field potential is an example of a Poisson problem.

The steady Poisson Problem

Given a Poisson equation on a 2D rectangular region, use finite differences to create a model of the equation, set up the corresponding linear system, plot the approximate solution and compute the norm of the approximation error.

1 Moving to 2D spatial domains

For our initial value problems and boundary value problems, we have been looking at a one dimensional domain:

- a time interval $\Omega = t_{init} \leq t \leq t_{final}$ for IVP's;
- a space interval $\Omega = x_{left} \leq x \leq x_{right}$ for BVP's;

We'd like to look at some problems on more interesting domains. In this discussion, we will move to a 2D spatial domain. Later, we will consider problems involving variation over a domain that extends over both space and time. To handle such problems, we will generalize the methods that we developed for the 1D steady (time independen) Poisson problem.

To keep things simple, our 2D domain Ω will usually be a rectangle. This simplifies settiing up our discretized geometry. Although we don't have time to cover them, there are plenty of techniques for handling more complicated regions, such as a circle, a slice of pizza, or the outline of an arbitrary curve such as the border of France.

2 The 2D Poisson equation

The steady (time-independent) Poisson equation in $\Omega \subset \mathbb{R}^2$ can be written as:

$$-\Delta u = f(x,y) \quad \text{for all } x,y \in \Omega$$

We will assume that Ω is a rectangle, $\Omega = \{(x,y) : a \leq x \leq b, c \leq y \leq d\}$. Here, in 2D, the mysterious symbol Δu represents

$$\Delta u \equiv \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = u_{xx} + u_y y$$

In order to be able to determine how well our approximations are doing, we will always assume that we know a formula $g(x,y)$ for the exact solution. For simplicity, we will generally assume that the boundary conditions will be of Dirichlet type, and that means we can use the $g(x,y)$ function to define these boundary conditions:

$$u(x,y) = g(x,y) \text{ for all } x,y \in \partial\Omega.$$

Here, $\partial\Omega$ is the set of all boundary points of Ω. For a rectangle, $a \leq x \leq b, c \leq y \leq d$, this is simply the points on the perimeter, that is

$$\partial\Omega = \{(x,y) \in \Omega : x = a \text{ or } x = b \text{ or } y = c \text{ or } y = d\}$$

For this lecture, we will stick with this simple version of the Poisson equation. Later we may see variations of the Poisson equation, involving new boundary conditions, the inclusion of spatially dependent coefficients, or extra terms in the equation, involving u, or $\frac{\partial u}{\partial x}$ or $\frac{\partial u}{\partial y}$.

3 A Poisson equation on a 2D rectangle

As we did for the 1D equation, we will replace the mathematical problem by a discrete computational problem. We start by making a discrete model of the the geometry. We use a regular mesh of points; those along the boundary will be associated with the boundary conditions, while points in the interior will be associated with the Poisson equation. At each point (i,j), we will write an equation involving the approximation $U_{i,j}$. These equations will form a linear system to be solved. Our solution, then, will be a table of these values.

Because our problem is 2D, we have to be more careful about how we generate the mesh points, how we form the linear system, and how we try to report the results.

In particular, we will now be working with a mix of vectors (lists) and matrices (tables). A useful MATLAB convention suggests using capital letters for matrices.

To discretize the geometry, we might use a grid of `nx` points in the x direction and `ny` points in y. In MATLAB, this can be done by:

```
1  x = linspace ( a, b, nx );
2  y = linspace ( c, d, ny );
```

so that the node in column i and row j of the mesh has coordinates `(x(i),y(j))`

We will seek an approximate solution only at this discrete set of points, and we may use the notation $U_{i,j}$ to indicate the value of our discrete solution at (x_i, y_j).

Points corresponding to $i = 1$, $i = nx$, $j = 1$ or $j = ny$ are *boundary points* where the Dirichlet boundary conditions will be imposed, using the function `g(x,y)`:

$$U_{i,j} = g(x_i, y_j)$$

At the remaining *interior points* we impose a discretized version of the Poisson equation. Recall that for the 1D equation, we had a way to approximate the second derivative u''. Now that u is a function of x and y, we simply use the same approximate technique for $\frac{\partial^2 u}{\partial x^2}$ and $\frac{\partial^2 u}{\partial y^2}$, so that our discretized version of the Poisson equation at the point (x_i, y_j) is:

$$\frac{-U_{i-1,j} + 2U_{i,j} - U_{i+1,j}}{hx^2} + \frac{-U_{i,j-1} + 2U_{i,j} - U_{i,j+1}}{hy^2} = f(x_i, y_j)$$

This equation involves the solution values at 5 nodes, which we can think of as `N=north`, `S=south`, `E=east`, `W=west` and `C=center`. We think of this as equation number k in our linear system, with coefficient matrix A. Then we can identify each coefficient $A(*,*)$ as occuring in matrix row k, the column corresponding to each of the five variables.

```
A(k,W)*U(W) + A(k,E)*U(E) + A(k,N)*U(N) + A(k,S)*U(S) + A(k,C)*U(C) = rks(k)
```

The values of the coefficients and right hand side will be:

```
A(k,W) = -1/hx^2
A(k,E) = -1/hx^2
A(k,N) = -1/hy^2
A(k,S) = -1/hy^2
A(k,C) = 2/hx^2 + 2/hy^2
rhs(k) = f(x(i),y(j) )
```

Now we have a system of $nx \times ny$ equations, which we symbolize by

$$A * u = rhs$$

We expect our solution `U` to be a matrix, but the MATLAB linear equation solver is going to return `u` as a vector, so that's another issue we are going to have to deal with. If we handle things correctly, then the command that turns that vector into a table will be something like this:

```
1  u = A \ rhs;
2  U = reshape ( u, nx, ny );
```

but we will explain what is going on a little later!

4 Exercise #1: Create a 4x3 discretized system

Consider a Poisson problem defined on Ω, the rectangle $[0,3] \times [0,2]$, with $f(x,y) = -2 * y$, exact solution $g(x,y) = x^2 y$, and Dirichlet boundary conditions.

If we use a 4×3 mesh of points, we get 12 nodes, and 12 equations, 10 of which are boundary equations, and 2 are discretized Poisson equations. Suppose we number the nodes (and the corresponding equations) as follows:

```
  2 | 9 10 11 12
Y=1 | 5  6  7  8
  0 | 1  2  3  4
    +-----------
      0  1  2  3
           X
```

Write the 12 linear equations of the discretized system.

5 Assembling the linear system

We have $nx \times ny$ equations to construct, and we need a way to order them, just as I did in the diagram above. We will use i and j to count x and y coordinates, and we want k to count nodes and equations. A procedure to visit each node in order might look something like this:

```
k = 0;
for j = 1 : ny
  for i = 1 : nx
    k = k + 1;
    if ( boundary)
      equation k is boundary equation at xi yj
    else
      equation k is Poisson equation at xi yj
    end
  end
end
```

Listing 1: Logic needed to visit each node.

Assuming we have defined `function value = g(x,y)`, a boundary equation can be thought of as

```
U_{i,j} = g(x(i),y(j))
```

while, using `hx` and `hy` as the x and y spacings, and `function value = f(x,y)` for the right hand side, we have seen that the Poisson equation at node in column i and row j can be thought of as

```
 (-U(i-1,j) +2*U(i,j)-U(i+1,j))/hx^2
+(-U(i,j-1) +2*U(i,j)-U(i,j+1))/hy^2 = f(x(i),y(j))
```

6 Relating column and row indices to nodes or (x,y) coordinates

If we think of our nodes as a table or matrix, it is natural to index them using i and j. However, in order to work with the linear solver, we also need to index the nodes by a single index variable k, that counts from 1 to $nx * ny$.

The illustration below suggests an $nx = 6 \times ny = 5$ rectangle, for which we can identify a node by specifying `(i,j)` coordinates, or a node index which we might call `k`. In particular, when writing the discretized Poisson equation at a node with coordinates `(i,j)` and index `k`, it is natural to describe the 4 neighbors as N=north at `(i,j+1)`, S=south at `(i,j-1)`, E=east at `(i+1,j)`, W=west at `(i-1,j)`. However, our data is stored by node index, so while we think of the North value in terms of its i and j indices, we need to be able to produce the corresponding node index k.

```
  ^
5 |   25 26 27 28 29 30
4 |   19 20 21 22 23 24
3 J   13 14 15 16 17 18
2 |    7  8  9 10 11 12
1 |    1  2  3  4  5  6
  |
  +-----------I-------->
       1  2  3  4  5  6
```

Looking at the illustration, we can see that if we let the center variable be in column i=5, row j=3, and node index k=17, then its North node is in column i=5, row j=4, and we need a way to automatically compute that the North node index is k=23.

The following one line function will take any pair (i, j) in a grid and return the corresponding k index:

```
1  function k = k_from_ij ( i, j, nx )
2    k = i + ( j - 1 ) * nx;
3    return
4  end
```

Listing 2: k_from_ij.m computes the node index.

In other words, since nx=6, the node with i=5, j=4 has k=23 because $5 + (4 - 1) * 6 = 23$.

Although we won't need it today, a similar function can be written to return the (i, j) coordinates given the k index:

```
1  function [ i, j ] = ij_from_k ( k, nx )
2    i = mod ( k, nx );
3    j = floor ( k / nx ) + 1;
4    return
5  end
```

Listing 3: ij_from_k.m computes the (ij) indices.

7 Exercise #2: Set up the linear system for the 4x3 problem

Here is a sketch of a MATLAB function which sets up and solves the linear system for the 4x3 Poisson problem that you looked at in Exercise #1.

In order to create an equation for every node, we consider every combination of i and j using a pair of nested loops. When we are ready to create a new equation, we increase the k index by 1. The value k identifies both the node we are considering, and the corresponding equation.

We set the equation by defining the coefficients of node k, and its four neighbors, which are the nodes to the east, west, north and south of it. Since our center node k has mesh indices (i, j), it's easy to generate the corresponding i and j indices of the four neighbors. However, in order to put the coefficients into the matrix, we need to identify those nodes by their corresponding node index. That's where we can use the `k_from_ij()`function. Here is a skeleton code that you need to fill in:

```
1    a = ?
2    b = ?
3    c = ?
4    d = ?
5    nx = ?
6    ny = ?
7    hx = ?
8    x = ?
9    hy = ?
10   y = ?
11
12   A = zeros(?,?);
13   rhs = zeros(?,1);
14
15   k = 0;
16   for j = 1 : ny
17     for i = 1 : nx
18       k = k + 1;
19       if ( boundary )  % what values of i and j mean we are on the boundary?
20         A(k,k) = ?
21         rhs(k) = ?
22       else
23         CENTER = k;
24         A(k,CENTER) = ?
```

```
25        WEST = ?       % Work out the (i,j) coordinates of WEST, and use k_from_ij().
26        A(k,WEST) = ?
27        EAST = ?
28        A(k,EAST) = ?
29        SOUTH = ?
30        A(k,SOUTH) = ?
31        NORTH = ?
32        A(k,NORTH) = ?
33        rhs(k) = ?
34      end
35    end
36  end
37
38  u = A \ rhs
39
40 define function g(x,y) here
41 define function f(x,y) here
```

Listing 4: skeleton2.m

Starting from *skeleteon2.m***, create a MATLAB function** *exercise2.m* **that sets up and solves the tiny Poisson linear system.**

Your `u` solution should be `[0,0,0,0,0,1,4,9,0,2,8,18]`.

8 Exercise #3: Postprocessing by tabulating

For small problems, we can report the results by simply printing the solution vector. However, it's helpful to include extra information, such as the mesh indices and the spatial coordinates.

Here is one way to do this:

```
1   u = A \ rhs;
2 %
3 %  Tabulate the solution.
4 %
5   fprintf ( 1, ' k   i   j    x       y       u       exact\n' );
6   k = 0;
7   for j = 1 : ny
8     fprintf ( 1, '\n' );
9     for i = 1 : nx
10      k = k + 1;
11      fprintf ( 1, '%2d  %2d  %2d  %6.2f  %6.2f  %6.2f  %6.2f\n', ...
12                k,   i,   j, x(i),  y(j),  u(k),  g(x(i),y(j)) );
13    end
14  end
```

Listing 5: Tabulate solution

Starting from *exercise2.m***, create a MATLAB function** *exercise3.m* **which tabulates your solution.**

9 Exercise #4: Use a general solver

Suppose we want to solve a different Poisson problem? Since the program we wrote is fairly complicated, is it possible to identify a part that doesn't have to change? If so, we can put that as a separate function, *poisson_solver.m*, and just feed it the new information.

Here is how it might work, allowing the user to specify the size and discretization of the rectangle, and the two special functions $f(x, y)$ and $g(x, y)$. For this example, we repeat the previous calculation.

```
a = 0.0;
b = 3.0;
c = 0.0;
d = 2.0;
nx = 4;
ny = 3;
%
%  Notice that data is now returned as TABLES (matrices).
%
[ X, Y, U ] = poisson_solver ( a, b, c, d, nx, ny, @f, @g );
%
%  You could do some post processing here using tables
%  k, i, j, X(i,j), Y(i,j), U(i,j), g(X(i,j),Y(i,j)).
%

function value = f ( x, y )
  value = - 2.0 * y;
  return
end

function value = g ( x, y )
  value = x.^2 .* y;
  return
end
```

Listing 6: exercise4 calls poisson_solver.m

Create this program *exercise4.m*, using a downloaded copy of *poisson_solver.m*, and report the results in a table.

10 Postprocessing: The RMS error norm

Printing the data in a table is OK for small problems, but for large problems we want something easier to understand quickly. If we know the exact solution $g(x, y)$, which we usually will for these sample problems, then we may be interested in reporting the error in our approximation.

We assume that `poisson_solver()` has returned the results `X, Y, U` as matrices. We can create a corresponding table of the exact solution at all the nodes by $V = g(X, Y)$. (This assumes that the $g(x, y)$ function has been written using MATLAB's "dot" operators for multiplication, division, and exponentiation!). Then the error is the table defined by `E=U-V`. To use MATLAB's `rms()` function, we need to temporarily make `E` look like a vector, and so the error norm is calculated this way:

```
  enorm = rms ( reshape ( E, nx*ny, 1 ) );
```

Once `enorm` has been computed, it should be reported as the RMS norm of the error:

```
  fprintf ( 1, '  nx = %d, ny = %d, rms error norm = %g\n', nx, ny, enorm );
```

11 Exercise #5: Compute RMS error norm for a new problem

Our first example problem was too easy to solve. We actually get zero error. In order to study convergence, we want a problem whose solution is harder to approximate.

Consider the Poisson problem on the rectangle $0 \le x \le 1$, $1 \le y \le 3$, with right hand side $f(x, y) = \pi^2 * (x^2 + y^2) * \sin(\pi * x * y)$, for which the exact solution is $g(x, y) = \sin(\pi * x * y)$.

Create a MATLAB file *exercise5.m* which sets up this new problem, solves it with `poisson_solver()`, and reports the error norm. Use $nx = 11$ and $ny = 21$.

If you run your program again, with $nx = 21$ and $ny = 41$, the mesh spacing h has decreased by a factor of $\frac{1}{2}$. By what factor does the error norm decrease?

12 Homework: Plot the solution of the new problem

The `poisson_solver()` code returns the solution data as tables `X, Y, U`.

MATLAB uses exactly this type of data to make contour line, contour color, and surface plots. Here are the commands to make this happen:

```
figure ( )
contour ( X, Y, U )
title ( 'Contour lines' );
print ( '-dpng', 'hw2_contour_lines.png' );

figure ( )
contourf ( X, Y, U )
title ( 'Contour colors' );
print ( '-dpng', 'hw2_contour_colors.png' );

figure ( )
surf ( X, Y, U );
title ( 'Surface' );
print ( '-dpng', 'hw2.png' );
```

Listing 7: Contour and surface plots

Starting from *exercise5.m*, make a MATLAB file *hw2.m* that creates these three plots of the solution.

For the color contour and surface plots, you could call `colorbar()` to display the relationship between color and value.

By default, the surface plot includes black grid lines on the surface. Especially when a large number of mesh points are used, this can make the surface hard to see. An alternative command then is `surf ( X, Y, U, 'edgecolor', 'none' );`

Notice that you can click in the surface plot window and "move" the surface so you can study it.

Send your surface plot *hw2.png* to me at `jvb25@pitt.edu`. I would like to see your work by Friday, 15 May 2020.