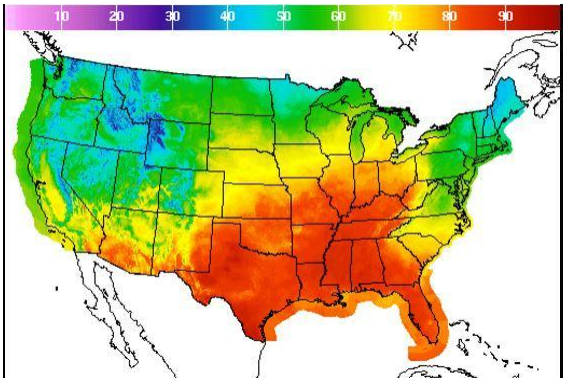
The 2D heat equation

MATH1091: ODE methods for a reaction diffusion equation

http://people.sc.fsu.edu/~jburkardt/classes/math1091_2020/heat_2d/heat_2d.pdf

---



We want to predict and plot heat changes in a 2D region.

The Heat Equation

We learned a lot from the 1D time-dependent heat equation, but we will still have some challenges to deal with when moving to 2D: creating the grid, indexing the variables, dealing with a much larger linear system.

1 Recall the steady 2D Poisson problem

We are interested in solving the time-dependent heat equation over a 2D region. From our previous work on the steady 2D problem, and the 1D heat equation, we have an idea of the techniques we must put together. We will see that the increased complexity of our data means that we will be looking for ways to cut down on the cost of storing data and solving linear systems.

Let's first make sure we recall the idea of approximating the Laplacian of u, that is, in 2D, the quantity known as the Laplacian, symbolized by $\Delta u = u_{xx} + u_{yy}$. Suppose we use grid spacings dx and dy, and that we are looking at a node C with grid index (i, j), with north, south, east and west neighbors N, S, E and W. Then we have the approximation

$$\Delta u(C) \approx \frac{u(W) - 2U(C) + U(E)}{dx^2} + \frac{u(N) - 2U(C) + U(S)}{dy^2}$$

In a steady 2D Poisson problem, we create a linear system by writing an equation for the value of u at each node. If the node is an interior node, we write the approximation to $\Delta u = f(x, y)$ (where $f(x, y)$ is often simply 0), or if the node is on the boundary, we write the appropriate Dirichlet (value), Neumann (derivative) or symmetry ("reflected point") equation.

Our solution $U()$ is an $nx \times ny$ array, and we solve the linear system once, and we are done. To see the solution, a simple contour or surface plot gives us all the information.

2 Build a 2D steady heat code

Our goal is to write some codes for time dependent heat problems. It is natural to think of starting with one of the codes we wrote for the 2D steady Poisson problem. However, I'd like to go over some of the coding ideas and rework them a little.

Defining the grid: We need x and y grid vectors, as we did for the 2D Poisson problem, and, although we might not need it. We naturally use indices i and j to locate any grid point.

Setting up the linear system: Given our $nx \times ny$ grid, we will be computing the solution at $nx * ny$ points, and so we need to set aside space for a matrix $\texttt{A}(\texttt{nx} * \texttt{ny}, \texttt{nx} * \texttt{ny})$ and a right hand side $\texttt{rhs}(\texttt{nx} * \texttt{ny})$.

Counting variables: When we are looking at the node at location (i, j), we need to figure out the nodes variable index, that is, the number between 1 and $nx * ny$. This is because we need to think of our array $U(nx, ny)$ as a vector $u(nx * ny)$ in order to work with the linear solver.

In fact, if we have to approximate the Laplacian at a node with variable index "C" (for "center"), then we will also need the indices of its North, South, East and West neighbors. I have written a simpler function than the one I did before, which returns all five indices at once. You just give it the values of the (i, j) index of the center (C) point, and the value of nx:

```
[C,N,S,E,W] = cnsew ( i, j, nx );
```

Listing 1: Function to return variable indices.

Using this function will make it easier to set the boundary conditions and Laplacian approximations.

Defining boundary conditions: Our simple problems all take place on a rectangle. And so we can identify a boundary node based on its grid indexes i and j. Let's also suppose that we have a function $g(x, y)$ which stores the exact solution. Setting the boundary data is like this:

```
for i = 1 : nx
  for j = 1 : ny
    [C,N,S,E,W] = cnsew ( i, j );
    if ( i == 1 | i == nx | j = 1 | j = ny )
      A(C,C) = 1
      rhs(C) = g ( x(i), y(j) )
    else
      ...approximate the Laplacian...
    end
  end
end
```

Listing 2: Identifying and setting Dirichlet conditions.

Approximating the Laplacian: Once we have the variable index information, then computing the Laplacian is also not too bad. We assume the function $f(x, y)$ is available to evaluate the right hand side of the steady heat equation, unless this is actually simply 0:

```
for i = 1 : nx
  for j = 1 : ny
    [C,N,S,E,W] = cnsew_from_ij ( i, j );
    if ( i == 1 | i == nx | j = 1 | j = ny )
      ...set boundary conditions
    else
      A(C,C) =  2.0 / dx^2 + 2.0 / dy^2;
      A(C,W) = -1.0 / dx^2;
      A(C,E) = -1.0 / dx^2;
      A(C,S) =              -1.0 / dy^2;
      A(C,N) =              -1.0 / dy^2;
```

```
        rhs(C) = f ( x(i), y(j) );
      end
    end
  end
```

Listing 3: Approximating the Laplacian.

Solving the linear system: As before, when we solve the linear system, we get a vector u:

```
  u = A \ rhs;
```

Listing 4: Solve the linear system.

Report RMS error: Since we know the exact solution $g(x, y)$, we can print the RMS error. We set `V` to be the exact solution at each node, set `E` to the difference, and call `rms()`. Since `rms()` expects a vector, not an array, we have to use `reshape()` as part of the call;

```
  V = g ( X, Y );
  E = U - V;
  err = rms ( reshape ( E, nx*ny, 1 ) );
  fprintf ( 1, '  RMS error = %g\n', err );
\vskip 0.1in
```

Listing 5: Get the RMS error.

Plotting the results: To plot the solution we reshape it into an array, then call `contourf()` or `surf()`:

```
  U = reshape ( u, nx, ny );
  [ X, Y ] = ndgrid ( x, y );
  contourf ( X, Y, U );
  title ( 'Solution of steady heat equation' );
  colorbar ( );
```

Listing 6: Creating a contour plot of the solution.

Helper functions: We would want to have a copy of `csnew()`, the exact solution function `g(x,y)`, and, if the heat equation source term isn't zero, the function `f(x,y)`.

This outlines a way to write our solver for a steady heat equation in 2D.

3 Exercise #1: Solver for the 2D steady heat equation

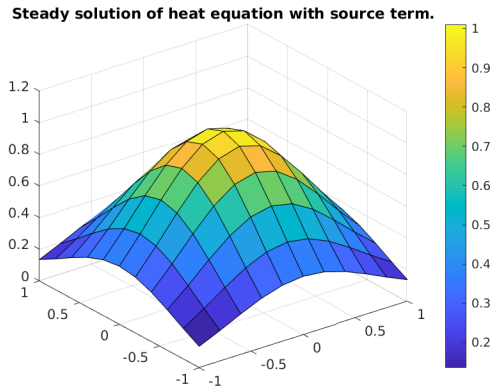
Make a file *exercise1.m* which is a copy of *exercise2.m* from the **poisson_2d_steady**. Modify the file, especially in the loop that sets the linear system, as described above. For the exact solution, use the following Gaussian function

$$g(x, y) = e^{-(x^2+y^2)}$$

and use the following for the right hand side function

$$f(x, y) = 4(1 - x^2 - y^2)\, e^{-(x^2+y^2)}$$

Use `nx=11` and `ny=11`, over the square $-1 \leq x, y \leq +1$.



4 Moving to a 2D time-dependent problem

We want to use our steady code as a base from which we build the time-dependent solver.

We will start by writing a forward Euler code, but we will do this in a way that makes it easy to move on to backward Euler, Crank Nicolson, and the theta version.

In the initial part of the code, we want to include a discretization in time, creating a time grid vector t of `nt` equally spaced values between 0.0 and some upper time limit, which we might assume is simply 1.0.

We also need to set a value for the quantity `kappa`, because the equation we are going to solve is

$$\frac{\partial u}{\partial t} = \kappa(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2})$$

Here `kappa` represents a physical quantity called *diffusivity*. (In the 1D case, I simply called this k, but now we need that name to count time steps!)

Along with setting up the linear system arrays `A` and `rhs`, we need to create a few more arrays. Notice that we are not going to try to save every value of temperature. Instead, the array $U()$ only has room for all the solution values at the current time. We need to start worrying about not using too much memory.

```
  [ X, Y ] = ndgrid ( x, y );
  U = zeros(nx,ny);
```

Listing 7: Create U X Y arrays.

The time loop Obviously, we need to do a time loop. Notice that I have moved the initial condition into the time loop, which seems a cleaner style, so that the loop sets U one way or another, and then plots it. (The previous code missed the first or last plot.) The details of the linear system depend on the solution method we choose.

```
for k = 1 : nt
  if k == 1
    U = initial condition
  else
    for i = 1 : nx
      for j = 1 : ny
        set boundary condition or heat equation
      end
    end
    solve for U
  end
```

```
  compute rms error
  plot U
end
```

Listing 8: An outline of the time loop.

Defining boundary conditions: As in the steady case, we will assume we have an exact solution function, but now $g()$ is a function of three variables, x, y, t. Note that k is keeping track of the time. Rather than setting $U(i,j)$ directly, we will think in terms of a linear system, so if node C is on the boundary, we set up the equation

```
  A(C,C) * U(C) = rhs(C)
```

Listing 9: Think about the boundary condition as a linear equation.

where `A(C,C)=1` and `rhs(C)` is determined from the $g()$ function:

```
  for i = 1 : nx
    for j = 1 : ny
      [C,N,S,E,W] = cnsew ( i, j );
      if ( i == 1 | i == nx | j = 1 | j = ny )
        A(C,C) = 1
        rhs(C) = g ( x(i), y(j), t(k) )
      else
        ...approximate the heat equation...
      end
    end
  end
```

Listing 10: Identifying and setting Dirichlet conditions.

Computing the RMS error: Conputing the error is almost the same as before, except that the function $g()$ now also depends on the time:

```
  V = g ( X, Y, t(k) );
  E = U - V;
  err = rms ( reshape ( E, nx*ny, 1 ) );
  fprintf ( 1, '  RMS error = %g\n', err );
```

Listing 11: Get the RMS error.

Plotting the results: Plotting U will occur inside the time loop, at the bottom, after the linear system is solved. Here is an outline of the time loop which emphasizes the plotting part. Note that we need the `X` and `Y` arrays here, which we presumably set up earlier in the code:

```
for k = 1 : nt
  if k == 1
    U = initial condition
  else
    for i = 1 : nx
      for j = 1 : ny
        set boundary condition or heat equation
      end
    end
    solve for U
  end
  report rms error

  contourf ( X, Y, U );
  label = sprintf ( 'Time step %d', k );
  title ( label );
  colorbar ( );
```

```
end
```

Listing 12: Creating a contour plot of the solution.

Now all of the coding discussed so far will be the same, no matter whether we are using forward or backward Euler, Crank Nicolson, or the theta method. We will implement each of those solvers by sliding the necesary commands inside the time loop, where we approximate the heat equation. It's like replacing an engine in a car.

5 Exercise #2: Create the general heat equation code

Create a file *exercise2.m* from a copy of your *exercise1.m*.

Try to make all the changes discussed above.

Use `nx=11`, `ny=11`, `nt=11`, with $-1 \leq x, y \leq +1$ and $0 \leq t \leq 1$. Set the diffusivity $kappa = \frac{1}{\pi^2}$.

We won't need a function $f(x,y)$. However we will need an exact function $g()$. Use this one:

$$g(x,y,t) = \sin(\frac{3\pi x}{5}) \cos(\frac{4\pi y}{5}) e^{-t}$$

Now, we can't actually drive a car with no engine, so let's put a "fake" engine in. In the spot where normally we would put the heat equation commands, put a copy of the boundary condition equations instead:

```
      if ( i == 1 | i == nx | j = 1 | j = ny )
        A(C,C) = 1
        rhs(C) = g ( x(i), y(j), t(k) )
      else
        A(C,C) = 1
        rhs(C) = g ( x(i), y(j), t(k) )
      end
```

Listing 13: Use boundary conditions everywhere.

This will allow us to test-drive the code, even though we won't really be calculating anything.

Now run your `exercise2`. After you clean up any little bugs, you should see a surface plot that gradually flattens as time moves on.

6 Designing the forward Euler engine

For the 1D heat equation, to compute U at node i and time k, we wrote

$$Unew(i) \leftarrow U(i) + \kappa \frac{dt}{dx^2} * (U(i-1) - 2U(i) + U(i+1))$$

In 2D, let's use the CNSEW symbols for node indexing. Now the equation becomes

$$Unew(C) \leftarrow U(C) + \kappa \frac{dt}{dx^2} * (U(W) - 2U(C) + U(E)) + \kappa \frac{dt}{dy^2} * (U(N) - 2U(C) + U(S))$$

Now we need to think of this as a linear system involving $Unew$ on the left hand side with coefficients stored in the matrix A, and the right hand side summing up everything involving the known, old values of U. In other words, we will have

```
A(C,C) = 1.0
rhs(C) = U(C) + kappa * (dt/dx^2) * (U(W) - 2 U(C) + U(E))
              + kappa * (dt/dy^2) * (U(N) - 2 U(C) + U(S));
```

Listing 14: The forward Euler "engine".

7 Exercise #3: A forward Euler solver for the 2D heat equation

Create a MATLAB file *exercise3.m* by copying *exercise2.m* and "installing" your forward Euler engine. Run your code.

8 Designing the backward Euler engine

In 2D, the backward Euler step looks like this:

$$Unew(C) - \kappa * \frac{dt}{dx^2} * (Unew(W) - 2Unew(C) + Unew(E)) - \kappa \frac{dt}{dy^2} * (Unew(N) - 2Unew(C) + Unew(S)) = U(C)$$

This means that, for the backward Euler, the equation for node `C` will involve defining `A(C,C)`, `A(W,C)`, `A(E,C)`, `A(N,C)`, `A(S,C)`, and `rhs(C)`.

9 Exercise #4: A backward Euler solver for the 2D heat equation

Create a MATLAB file *exercise4.m* by copying *exercise2.m* and "installing" your backward Euler engine. Run your code.

10 Designing the Crank Nicolson engine

Remember that the Crank Nicolson method can be thought of as the "average" of the forward and backward Euler methods. Simply, this means that if the matrix entry `A(*,*)` has the value `fw` in the forward method, and `bw` in the backward method, it has the value `0.5*fw+0.5*bw` in the Crank Nicolson method. The right hand side entry `rhs()` is treated the same way, by averaging.

For our Laplacian estimate, this means we have to average the quantities `A(C,C)`, `A(W,C)`, `A(E,C)`, `A(N,C)`, `A(S,C)`, and `rhs(C)`.

Here is how I set the main matrix entry `A(C,C)`:

```
fw = 1.0;
bw = 1.0 + 2.0 * kappa * dt / dx^2 + 2.0 * kappa * dt / dy^2;
A(C,C) = 0.5 * fw + 0.5 * bw;
```

Listing 15: Setting a Crank Nicolson matrix entry.

11 Exercise #5: A Crank Nicolson solver

Create a MATLAB file *exercise5.m* by starting with a copy of *exercise2.m*. Then copy both the forward and backward Euler "engines", and try to figure out how to average them to set up the Crank-Nicolson engine. Run your code.

12 Homework: A theta method solver

To construct the theta method, we replace the equal averaging of the Crank-Nicolson method by the sum of `(1-theta)` times the forward Euler values, and `theta` times the backward Euler values.

Make the file *hw5.m* by copying your *exercise5.m* program, and then replacing the averaging coefficients. Some 0.5 values become `(1-theta)` and others become `theta`.

You will also need to set `theta` inside the program, or, if you write it as a function, allow it to be set as an input argument:

```
function hw6 ( theta )
```

Listing 16: Adding theta as in input parameter.

Try your program with `theta` = 0, 0.5, 1.0 and see if you get the results for the forward Euler, Crank-Nicolson, and backward Euler methods.

Send your program *hw5.m* to me at **jvb25@pitt.edu**. I would like to see your work by Friday, 05 June 2020.