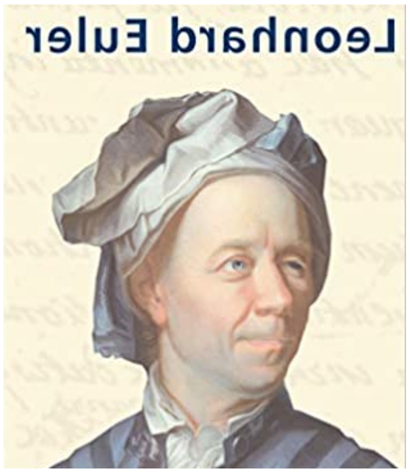
Implicit methods for the heat equation

MATH1091: ODE methods for a reaction diffusion equation

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



Backward Euler can solve our stepsize and stability issues!

Implicit Solvers for the Heat Equation

The CFL condition forces an explicit solver to take very small steps to avoid instability. There are several implicit ODE solvers that can allow us to take generous steps. However, instead of an explicit formula for the next values, we get an implicit linear system that must be solved.

1 Backward Euler is the simplest implicit solver

As a review, let's recall the "stiff ODE" from the previous ODE class:

$$y' = 50(\cos(t) - y)$$
$$y(0.0) = 0.0$$

for which the exact solution is

$$y(t) = 50\,\frac{\sin(t) + 50\cos(t) - 50e^{-50t}}{2501}$$

Being lazy, we decide to apply the forward Euler method, which says that, to discretize the ODE:

$$y' = f(t, y)$$

we write the discrete equation:

$$\frac{y(j+1) - y(j)}{dt} = f(t(j), y(j)) \text{ which we rewrite as}$$
$$y(j+1) = y(j) + dt\, f(t(j), y(j))$$

In other words, evaluate the right hand side of the ODE at the **old time and solution**, $t(j), y(j)$.

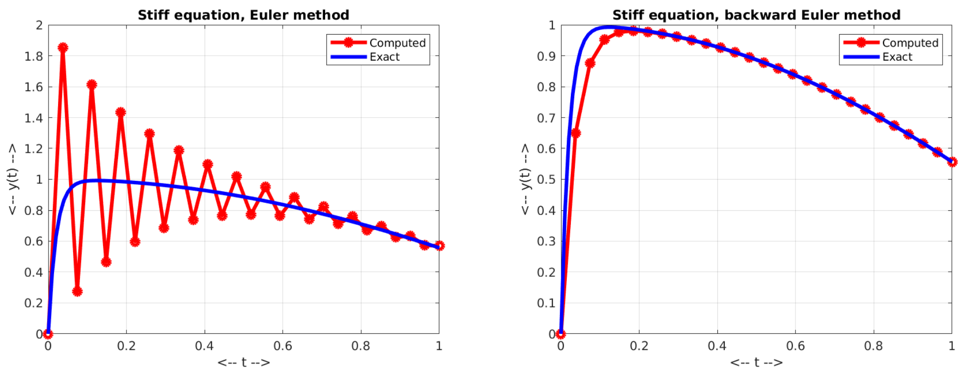
For the stiff equation, this approach gave terrible answers unless we drastically cut down the stepsize. Our next alternative was to try the backward Euler method, which discretizes the ODE as:

$$\frac{y(j+1)-y(j)}{dt} = f(t(j+1), y(j+1))$$

So here we evaluate the right hand side of the ODE at the **new time and solution**, $t(j+1), y(j+1)$.

For our stiff equation, the resulting implicit equation is actually easy to set up and solve. (That is not always so easy, though!)

$$\begin{aligned}
\frac{y(j+1)-y(j)}{dt} &= 50(\cos(t(j+1)) - y(j+1)) \\
y(j+1) &= dt\, y(j) + 50\, dt\, \cos(t(j+1)) - 50\, dt\, y(j+1) \\
(1+50\, dt)\, y(j+1) &= y(j) + 50\, dt\, \cos(t(j+1)) \\
y(j+1) &= \frac{y(j) + 50\, dt\, \cos(t(j+1))}{1+50\, dt}
\end{aligned}$$



Forward and fackward Euler ODE approximations.

2 The linear system for the implicit heat equation

Now let's consider how the backward Euler method would be applied to a heat problem.

We will take our problem to be:

$$\begin{gathered}
\frac{\partial u}{\partial t} = \frac{4}{\pi^2}\frac{\partial^2 u}{\partial x^2} \quad \text{for } 0 \le x \le 1 \\
u(0,t) = 0, u'(1,t) = 0 \text{ for } 0 \le t \le 1 \\
u(x,0) = \sin(\frac{\pi\, x}{2}) \quad \text{for } 0 \le x \le 1
\end{gathered}$$

for which the exact solution is

$$g(x,t) = \sin(\frac{\pi\, x}{2})\, e^{-t}$$

When we seek the solution at time step $j+1$, our linear system looks like this:

$$\begin{aligned}
U(1,j+1) = g(x(1), t(j+1)); &\quad \text{Dirichlet condition at node 1} \\
\frac{k\, dt}{dx^2}U(i-1,j+1) + (1.0 - 2\frac{k\, dt}{dx^2}U(i,j+1) + \frac{k\, dt}{dx^2}U(i-1,j+1) = U(i,j) &\quad \text{Backward Euler at nodes 2, 3, ..., nx-1} \\
\frac{U(nx-1,j+1) - U(nx,j+1)}{dx} = 0 &\quad \text{Neumann condition at node nx}
\end{aligned}$$

In order to compute the solution at timestep $j+1$, we need to reformulate these equations as a linear system $A * U = rhs$ and then solve.

3 Exercise #1: Backward Euler solver

Create a MATLAB program *exercise1.m* which sets up the problem described above, using $nx = 21$ nodes and $nx = 11$ time steps. The outline of your time loop might be something like this:

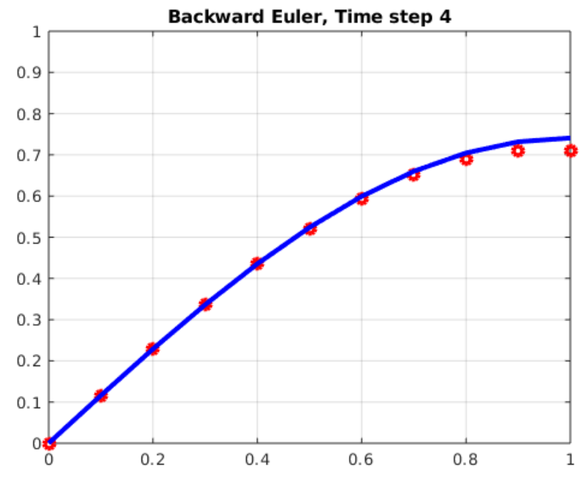
```
for j = 1 : nt - 1

  A(1,1) = ?
  rhs(1) = ?

  for i = 2 : nx - 1
    A(i,i-1) = ?
    A(i,i)   = ?
    A(i,i+1) = ?
    rhs(i)   = ?
  end

  A(nx,nx-1) = ?
  A(nx,nx) = ?
  rhs(nx) = ?

  u = A \ rhs;
%
%  Perhaps display a plot of u versus exact solution g(x,t(j+1)).
%
  U(1:nx,j+1) = u;
end
```

Listing 1: The code for setting up the linear system.

If you compute the CFL condition for this problem, you should see that our time step is much too large for an explicit scheme. However, you should observe that the implicit scheme shows no instability, and provides a fairly accurate approximation.

4 Replace Neumann condition by symmetry condition

In your plots for exercise 1, you may have noticed that the Dirichlet boundary condition was matched perfectly, but that on the right endpoint, something seemed a bit wrong. The solution is a sine curve, which has a zero derivative there, but the computed solution was actually noteiceably flat over the entire last subinterval. This seems to be related to the fact that, near the right endpoint, the computed solution seems to pull away from the exact one.



The approximation deteriorates near the Neumann boundary.

Our approximations to $\frac{\partial^2 u}{\partial x^2}$ involve an error of order dx^2, but our approximation to the Neumann boundary condition involves a much larger error of order dx:

$$\frac{\partial u}{\partial x} \approx \frac{u(x-dx,t)-u(x,t)}{dx} + O(dx)$$

It turns out that we can treat the boundary in another way that results in better accuracy. Suppose that the right endpoint actually defines a line of symmetry for the problem, so that we can imagine a mirror image of the problem over $1 \le x \le 2$. Then the point at $x = 1$ doesn't have to be treated as a boundary point. We can think of it as an interior point, where we would want to write another heat equation. The only trouble is that this equation will reference a point $x(nx+1)$, further to the right of our computational region. But if we can assume symmetry, then the solution value at $x(n+1)$ is the same as that at $x(nx-1)$.

So instead of the last linear equation being a Neumann boundary condition, we have a symmetry condition, which allows us to determine the solution value there by writing another heat equation. So now we generate a coefficient that would go with the non-existent value $U(nx+1, j+1)$, and we use the symmetry condition so that this coefficient is simply "reflected", that is, added to the coefficient of $U(nx-1, j)$.

5 Exercise #2: Replace Neumann condition by symmetry

Make a MATLAB file *exercise2.m* by copying *exercise1.m*.

Again, we will use $nx = 21$ nodes and $nx = 11$ time steps. The only thing you need to change is the last equation. You can start by copying the lines that define the coefficients and right hand side for interior equation i. In this copy, if you replace i by nx, you will have set a coefficient $A(nx, nx+1)$, which is not legal. Instead, add this amount to $A(nx, nx-1)$ and drop the reference to $A(nx, nx+1)$.

If you run your code, the computed solution so look noticeably closer to the exact result. sectionHomework: A simple diffusivity problem

6 The Crank Nicolson method

In terms of solving a differential equation, $\frac{dy}{dt} = f(t, y)$, the evaluation of $f()$ is done at the old time for the forward Euler method, and at the new time for the backward Euler method. The Crank Nicolson method tries to average these two results. As an ODE solver, it would be written

$$\frac{dy}{dt} = f(t,y) \approx \frac{1}{2}(f(t_{old}, y_{old}) + f(t_{new}, y_{new}))$$

For our discretized heat equation, this means we write

$$\frac{U_{i,j+1} - U_{i,j}}{dt} = \frac{1}{2}\,k\,(\frac{U_{i-1,j} - 2U_{i,j} + U_{i+1,j}}{dx^2} + \frac{U_{i-1,j+1} - 2U_{i,j+1} + U_{i+1,j+1}}{dx^2})$$

Rewriting this as a linear system, with the unknown variables on the left, we have:

$$U_{i,j+1} + \frac{k\,dt}{2\,dx^2}(U_{i-1,j+1} - 2U_{i,j+1} + U_{i+1,j+1}) = U_{i,j} + \frac{k\,dt}{2\,dx^2}\,(U_{i-1,j} - 2U_{i,j} + U_{i+1,j})$$

which we can rearrange, collecting coefficients on the left, and evaluating the right hand side, to get the linear system $A * u = rhs$.

7 Exercise #3: A Crank Nicolson Solver

Create *exercise3.m* by copying *exercise1.m*, the code that still uses the Neumann condition on the right, and $nx = 21$ nodes and $nx = 11$ time steps.

In order to create a Crank Nicolson code, the only thing you have to change is the calculations, inside the time loop:

```
  for i = 2 : nx - 1
    A(i,i-1) = ?
    A(i,i)   = ?
    A(i,i+1) = ?
    rhs(i)   = ?
  end
```

Listing 2: Statements that define the linear system.

Look carefully at the formula above, where I worked out the linear system. Set the coefficients of the unknown quantities as the coefficients in A. The value of rhs is simply the sum of the known terms on the right of that equation.

Run your code on the same problem we have been using before. You should expect to see a good solution estimate.

8 Exercise #4: Crank Nicolson + symmetry condition

Create *exercise4.m* by copying *exercise3.m*. Now we want to modify the code so that it uses a symmetry condition instead of a Neumann boundary condition. That means that we want to write one more heat equation, just like the ones at nodes $i = 2 : nx - 1$, but now associated with node nx. We do this just after the i loop, setting $i = nx$ and then copying the assignment statements from inside the loop.

```
  for i = 2 : nx - 1
    A(i,i-1) = formula from last exercise
    A(i,i)   = formula from last exercise
    A(i,i+1) = formula from last exercise
    rhs(i)   = formula from last exercise
  end
%
%  Code to apply same formula at node nx, but using symmetry:
```

```
%
  i = nx;
  A(i,i-1) = ?
  A(i,i)   = ?
  A(i,i+1) = ?  (This matrix entry doesn't exist!}
  rhs(i)   = ?  (This formula will refer to a value of U that doesn't exist.)
```

Listing 3: Replacing the Neumann boundary by symmetry.

This is almost right, except that now we have a statement assigning a value to $A(i, i+1)$, referring to node $i+1 = nx+1$, and a term in the right hand side that refers to $U(i+1, j)$. We fix the bad coefficient reference by adding that value to $A(i, i-1)$, since this references node $nx - 1$ which does exist, and which should have the same value as the symmetric, but fictitious, node $nx + 1$. Similarly, in the right hand side, where we wrote $U(i + 1, j)$, we want to write $U(i - 1, j)$.

If you make these changes and run the code, you should see your Crank Nicolson program giving a good approximation to the true solution, as we got in exercise #2.

Since it is noticeably more work to program the Crank Nicolson method, this raises the question *What's so great about Crank Nicolson compared to Backward Euler?*. We will look into this question in the homework.

9 The Theta Method

When we have any two quantities, A and B, we can form a blend C by using using $1 - \theta$ of A and θ of B. If we can assume $0 \leq \theta \leq 1$, then we are forming what is called a **convex combination** of A and B, which has some special properties:

- $\theta = 0 \rightarrow C = A$;
- $\theta = 1 \rightarrow C = B$;
- $0 \leq \theta \leq 1 \rightarrow \min(A, B) \leq C \leq \max(A, B)$;

Now the Crank Nicolson method can be written as

$$\frac{y_{new} - y_{old}}{dt} = \frac{1}{2} f(t_{old}, y_{old}) + \frac{1}{2} f(t_{new}, y_{new})$$

and so we can think of it as a special case, for $\theta = \frac{1}{2}$, of the following **theta method**:

$$\frac{y_{new} - y_{old}}{dt} = (1 - \theta)\, f(t_{old}, y_{old}) + \theta\, f(t_{new}, y_{new})$$

so that

- $\theta = 0 \rightarrow$ forward Euler method;
- $\theta = 0.5 \rightarrow$ Crank Nicolson method;
- $\theta = 1 \rightarrow$ backward Euler method;

If we think about this carefully, we can write a single program, which accepts a value for the parameter θ, and carries out the corresponding method. We could even use a value of $\theta = 0.75$, as well.

It turns out that if you have written a Crank Nicolson method, it's not too hard to build a general theta method code from it.

10 Homework: Implement the Theta Method

Make a file *hw4.m* by copying *exercise4.m*.

If you wrote `hw4()` as a function, change the function statement so that it accepts an input value for `theta`:

```
  function hw4 ( theta )
```

Listing 4: Your function should accept θ as a parameter.

If your program is just a script, then you will just have to remember to define `theta` before calling your script.

Our only change is in the definition of the linear system. Setting up the linear system inside your loop can be thought of like this, where many terms were divided by 2:

```
    for i = 2 : nx - 1
      A(i,i-1) =         - Astuff / 2
      A(i,i)   = 1.0 + 2 * Astuff / 2
      A(i,i+1) =         - Astuff / 2
      rhs(i) = U(i,j) +    rhsstuff / 2
    end
```

Listing 5: Where to change from 1/2 to theta or (1-theta).

In order to make a theta code, replace every term `Astuff/2` by `theta * Astuff`, and the `rhsstuff/2` term by `(1-theta)*rhsstuff`.

We have one more chunk of linear system definitions, for the symmetry condition at the last node. Do the same thing here.

Note that we do not change the 1.0 term in the definition of `A(i,i)`, and we do not change the `U(i,j)` term in the definition of `rhs(i)`!

I realize this can be confusing, but take your time and think it through. If you cannot understand my suggestions, or your code doesn't work as predicted, let me know!

If you get your code written properly, then you do the following:

- `hw4(0.0)`, forward Euler, should blow up around step 10;
- `hw4(0.5)`, Crank Nicolson, should seem to match the exact solution;
- `hw4(1.0)`, backward Euler, should stay very close to the exact solution;

Save the plot of time step 10 of the forward Euler method as *hw4.png*

Send your plot *hw4.png* to me at **jvb25@pitt.edu**. I would like to see your work by Friday, May 29.