Example: Coolant flow in channel (2^{nd}-order, PDE Toolbox) • Mech. Eng. • 20min

Now let's illustrate how you might set up a more complex geometry in MATLAB. The PDE Toolbox is a graphical interface for designing and calculating equations on a mesh geometry.

- Invoke the PDE Toolbox with `pdetool`.
- Create a rectangle and a circle overlapping a quarter of it. Change the set formula to R1-E1 to make the circle subtractive.
- First initialize a mesh, then refine the mesh. Now jiggle it.
- Change to Boundary Mode. Mixed boundary conditions can only be done in systems, so we'll cheat in this case and just set the outer and inner boundaries directly (perfect heat transfer at the surface).
    - Set the inner circle (coolant channel surface) to Dirichlet BC with h=1, r=400.
    - Set the outer (top) boundary (combustion surface) to Dirichlet BC with h=1, r=1700.
    - Set the three inner boundaries to Neumann BC with g=0, q=1.
- Set up the PDE. The types of PDE for which built-in solvers are available are shown here. Create an elliptic PDE with c = 1.0; a = 0.0; f = 0.0; and d = 1/3.6.

  (If, like me, you don't recall the mathematical terms for each class of PDE, here's a list:
    - elliptic = Laplace equation
    - parabolic = heat equation
    - hyperbolic = wave equation

  Other types (including nonlinear PDEs) require direct implementation of a numerical method to solve. Next time we will look at a nonlinear PDE, the Burgers' equation, which introduces a shock wave in its solution.)
- Solve the PDE.
- Plot it. Add contours (10 lines), flat shading, and change the colormap to 'hot'. Examine the abs(grad(u)) plot as well; if the sides are not horizontal, try playing with parameters such as the mesh resolution to solve that problem.

Example: Shock waves (nonlinear PDE) • Aerospace Eng. • 30min

The inviscid Burgers' equation

$$\frac{\partial u}{\partial t} = -u\frac{\partial u}{\partial x} = -\frac{1}{2}\frac{\partial (u^2)}{\partial x}$$

describes advecting waves which can produce shocks, or discontinuities. It is used to describe acoustics or hydrodynamics (particularly in the more complicated viscous form). As this is a partial differential equation in both x and t, we will need a numerical method to solve the system in time and another to solve the system in space. Accordingly, we will adopt the first-order upwind scheme in time and space. (This method isn't terribly stable, but will do for our purposes: consult a numerical methods text for more information on stability.)

This builds on some of the ideas of the finite difference example from last time, but extends them to more dimensions. We also will not construct an explicit matrix, but will use nested for loops to achieve the same effect.

The first-order upwind scheme uses the quantities at time step n to calculate the values at time step n, making its implementation straightforward.

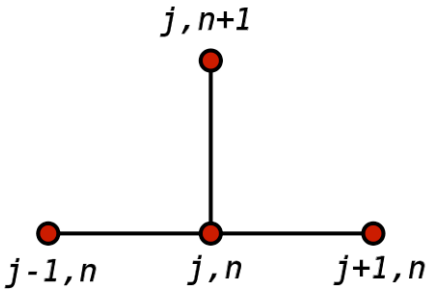
We can approximate the Burgers' equation above with this scheme as

$$\frac{u_j^{n+1} - u_j^n}{\Delta t} = -u_j^n \frac{u_{j+1}^n - u_{j-1}^n}{2\Delta x}$$

yielding

$$u_j^{n+1} = u_j^n \left(1 - \Delta t \frac{u_{j+1}^n - u_{j-1}^n}{2\Delta x}\right)$$

as the fundamental expression for each calculation of u.



- Open a new script file and save it as burgers.m.
- Specify the fundamental parameters as variables to make them easy to adjust. (We will use the conventions of centimeters and seconds.)

```
nx = 50;       % space discretization
xmin = -1;     % minimum spatial extent, cm
xmax =  1;     % maximum spatial extent, cm
jmid = floor(nx/2)+1; % index of central point of grid
dx = (xmax-xmin)/nx;  % grid size, cm
x = linspace(xmin,xmax,nx+1); % x-coordinates of calculation

nt = 30000;    % time discretization
dt = 0.00001;  % time step size, s
tmax = nt*dt;  % maximum time achieved, s
t = linspace(0,tmax,nt+1);    % t-coordinates of calculation
```

- For anything interesting to happen, the initial condition must be nontrivial. Let's have a cosine-shaped wave at $t = 0$: $u(x,0) = u_0(x) = 1\text{-}\cos(x)$.

```
u0 = zeros(1,nx+1);
% Initial condition:  u(x,0) = u0(x) = 1-cos(x)
for j=1:nx+1
    u0(j) = 1 - cos(x(j)*pi);
```

```
end
u = zeros(nt,nx+1);
u(1,:)=u0;
```

- This completely specifies the parameters for our problem, so let's put some thought into the structure of the main calculation routine. We require the solution at each step to meet the equation above, so

```
u(n+1,j) = u(n,j) * (1 - dt * (u(n,j+1)-u(n,j-1))/(2*dx));
```

should just about fit the bill.

- Note that the above equation requires the introduction of three new variables: `u`, `n`, `j`. `n` and `j` refer to a specific case of a repeated calculation, and so are good candidates for using a for loop. `u` can be initialized directly from `u0` in the first time through the loop. Write this code and try to execute the script.

```
for n=1:nt
    for j=1:nx+1
        if (x(j) == xmin)
            u(n+1,j) = u(n,j) * (1 - dt * (u(n,j+1)-u(n,nx))/(2*dx));
        elseif (x(j) == xmax)
            u(n+1,j) = u(n,j) * (1 - dt * (u(n,1)-u(n,j-1))/(2*dx));
        else
            u(n+1,j) = u(n,j) * (1 - dt * (u(n,j+1)-u(n,j-1))/(2*dx));
        end
    end
end
```

Try to run the code now. It fails because we still have edge cases: j+1 and j-1 aren't always valid statements since you can't use a negative index in MATLAB. Careful thought reveals that we need to define the endpoint boundary conditions: do we want them to be at a fixed temperature or merely insulating (nonconductive)? (Incidentally, this is where matrices can be very convenient, especially in higher dimensions than 1 space plus 1 time.)

We choose to have periodic boundary conditions (which *de facto* means that $u_0 = u_{max}$). (If instead we had chosen insulating boundary conditions, we would simply have set the outermost two points equal to each other; fixed value boundary conditions would have required us to set the endpoints to a definite value.)

- Set up a condition to test if we are at an endpoint and to wrap the equation around if we are (this is inside both loops):

```
for n=1:nt
    for j=1:nx+1
        if (x(j) == xmin)
            u(n+1,j) = u(n,j) * (1 - dt * (u(n,j+1)-u(n,nx))/(2*dx));
        elseif (x(j) == xmax)
            u(n+1,j) = u(n,j) * (1 - dt * (u(n,1)-u(n,j-1))/(2*dx));
        else
            u(n+1,j) = u(n,j) * (1 - dt * (u(n,j+1)-u(n,j-1))/(2*dx));
        end
    end
end
```

Notice that past about $t = 0.25$s the solution becomes unstable. This is actually the numerical expression of a physical result: the inviscid Burgers' equation develops a shock wave at this point. To accurately model it numerically involves introducing a viscosity term for the full Burgers' equation, but we will forgo that at the present time. (Feel free to implement this yourself; see

Wikipedia for details on the full equation form.) The numerical integrator can also be improved.

For nonlinear equations, you often have to worry about an additional component to the mesh spacing and time step size: the equation itself may introduce unexpected behavior that cannot be analytically expressed!

MATLAB also provides the `pdenonlin` function which may be a useful tool if your nonlinear PDE is of the correct form.