Computational Lab 8 : Solving PDEs, part I

Due November 1st, 2019 (@ 17:00)

- Read this document and do its suggested readings to help with the pre-labs and labs.
- This lab's topics revolve around computing solutions to PDEs using basic methods.
- Ask questions if you don't understand something in this background material: maybe we can explain things better, or maybe there are typos.
- Whether or not you are asked to hand in pseudocode, you **need** to strategize and pseudocode **before** you start coding. Writing code should be your last step, not your first step.
- Test your code as you go, **not** when it is finished. The easiest way to test code is with `print('')` statements. Print out values that you set or calculate to make sure they are what you think they are.
- Practice modularity. It is the concept of breaking up your code into pieces that as independent as possible form each other. That way, if anything goes wrong, you can test each piece independently.
- One way to practice modularity is to define external functions for repetitive tasks. An external function is a piece of code that looks like this:

```
def MyFunc(argument):
    '''A header that  explains the function
    INPUT:
    argument [float] is the angle in rad
    OUTPUT:
    res [float] is twice the argument'''
    res = 2.*argument
    return res
```

Physics Background

Temperature distributions in heat conducting materials The distribution of temperature $T(x,y)$ in the interior of heat conducting materials, in steady state and in the absence of internal heat sources, satisfies $\nabla^2 T = 0$. To determine the distribution of heat in a body, the solution of Laplace's equation requires boundary conditions, either on the temperature on the boundary or the temperature gradient on the boundary. Mathematically, it is known that there is a unique solution that satisfies Laplace's equation and the boundary conditions. As mentioned in the computational background, you will use a relaxation type method to solve this elliptic equation numerically.

Time dependent thermal condition : Thermal conduction is modeled with a diffusion equation, where the temperature change along a spatial dimension x is determined by the gradients in thermal fluxes

$$\frac{\partial T}{\partial t} = \frac{\partial F}{\partial x}, \tag{1}$$

where the thermal flux F is itself proportional to a temperature gradient

$$F = D\frac{\partial T}{\partial x}, \tag{2}$$

where D is the thermal diffusivity. If a single material is being considered, then D is a constant, and the rate of temperature change becomes

$$\frac{\partial T}{\partial t} = D\frac{\partial^2 T}{\partial x^2}, \tag{3}$$

which can be integrated using the forward–time–centered–space algorithm.

Burger's equation (for Q3): Burgers' equation is a classical nonlinear wave equation which in one spatial dimension takes the form:

$$\frac{\partial u}{\partial t} + \epsilon u\frac{\partial u}{\partial x} = 0 \tag{4}$$

or, in conservative form,

$$\frac{\partial u}{\partial t} + \epsilon\frac{\partial}{\partial x}\left(\frac{u^2}{2}\right) = 0. \tag{5}$$

Burger's equation is like a wave propagation equation in which the phase speed of the wave v is replaced by the actual wave velocity u. This nonlinearity allows the wave to steepen and so is one way to model water waves approaching a beach. As a model system, Burger's equation is the starting point for modelling solitons with the KdeV equation (which includes an additional dispersion term).

Computational Background

Boundary Value Problems The first class of PDEs that Newman discusses in Chapter 9 are solutions of elliptic partial differential equations, of which the Laplace and Poisson equations are classic examples. He discusses the Jacobi method, which is a relaxational method for solving Laplace's or Poisson's equation, and then speedups to this method using overrelaxation and Gauss-Seidel replacement. For this lab we would like you to focus on Gauss-Seidel with overrelaxation which is implemented according to (9.17). To obtain the solution of a field $T(x,y)$ that satisfies Laplace's equation

$$\nabla^2 T = \frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} = 0, \tag{6}$$

subject to boundary conditions (see Physics Background), this method is written

$$T(x,y) \leftarrow \frac{1+\omega}{4}\left[T(x+a,y) + T(x-a,y) + T(x,y+a) + T(x,y-a)\right] - \omega T(x,y), \tag{7}$$

where the left arrow indicates replacement of the left hand side with the right, and ω is a relaxation parameter. In these methods, you can pick a fixed number of iterations or can choose a level of convergence. In Q1 we will be comparing methods and so will use both approaches.

Animations in matplotlib Animations are possible with `matplotlib`. Here is a sample code that will animate the curve `sin(x-t)`.[1]

```
from numpy import arange, pi, sin
from pylab import clf, plot, xlim, ylim, show, pause
t = arange(0, 4*pi, pi/100)  # t coordinate
x = arange(0,4*pi, pi/100)  # x coordinate
for tval in t:
    clf()  # clear the plot
    plot(x, sin(x-tval))  # plot the current sin curve
    xlim([0, 4*pi])  # set the x boundaries constant
    ylim([-1, 1])  # and the y boundaries
    draw()
    pause(0.01)    #pause to allow a smooth animation
```

A "cleaner" method would be to follow the guidelines of
https://matplotlib.org/api/animation_api.html,
but it does represent a bit more work.

[1] The script simply clears each frame and then replots it, with a little pause to make it run smoothly. This is not an ideal method but gets the job done.

Multiple plots: Since animations don't transfer to PDFs very well (...at all) it's better to make a figure showing multiple frames of the iteration using `matplotlib`. For example

```
import matplotlib.pyplot as plt
frames = []  # a sequence of frames to draw
Ny, Nx = 8, 4  # draw 8 frames down, 4 across
fig, axs = plt.subplots(Ny, Nx,
                        figsize=(8, 10),
                        gridspec_kw={"hspace": 0.1,
                                     "wspace": 0.1},
                        sharex=True,
                        sharey=True)  # raw a full page figure, with shared
                                      # axes, closely packed figures

axs = axs.flatten()  # axs is a list of references to each subplo
for i in range(len(axs)):
    axs[i].pcolormesh(frames[i]) # This is not complete, add axes, maybe transpose, contourf...
    axs[i].text(10, 2,  # Add a frame number at value 10,2 in the plot
                "{}".format(i), # Make it the subplot number, you could scale this to frame number
                horizontalalignment='center') #center it at 10,2
```

The FTCS method : Newman's text in Chapters 9.3.1-9.3.2 discusses the forward–time–centered–space method for solving PDEs, which is easy to program but has some stability issues with particular types of PDEs as discussed in the text. Question 2 asks you to solve a problem based on an exercise in the book, which should produce solutions that are stable using the FTCS method.
In summary, the FTCS method takes the an equation like

$$\frac{\partial \phi}{\partial t} = D\frac{\partial^2 \phi}{\partial x^2}, \tag{8}$$

and calculates the centered–difference equation for the spatial derivative

$$\frac{\partial^2 \phi}{\partial x^2} = \frac{\phi(x+a,t) - 2\phi(x,t) + \phi(x-a,t)}{a^2}, \tag{9}$$

and uses a forward Euler step to calculate the time derivative

$$\frac{\partial \phi}{\partial t} = \frac{D}{a^2}\left(\phi(x+a,t) - 2\phi(x,t) + \phi(x-a,t)\right). \tag{10}$$

Methods for solving Burger's equation This is to remind you of the notation used in Lab08: the space and time directions are discretized into N_x and N_t steps of sizes Δx and Δt, respectively:

$$x_i = i\Delta x, \quad i = 0, ..., N_x - 1 \tag{11}$$
$$t_j = j\Delta t, \quad j = 0, ..., N_t - 1 \tag{12}$$

Then, we write $u_i^j = u(x_i, t_j)$.
As discussed in § 9.3.2, the FTCS method is numerically unstable when integrating the wave equation, which the Burgers equation (5) is an extension of. One simple method that is appropriate, if not particularly efficient, is the leapfrog method. We saw it when solving ODEs, and its extension when solving an equation of the type of (5) is

$$\frac{u_i^{j+1} - u_i^{j-1}}{2\Delta t} = -\frac{\epsilon}{2}\frac{\left(u_{i+1}^j\right)^2 - \left(u_{i-1}^j\right)^2}{2\Delta x}, \tag{13}$$

or

$$u_i^{j+1} = u_i^{j-1} - \frac{\beta}{2}\left[\left(u_{i+1}^j\right)^2 - \left(u_{i-1}^j\right)^2\right], \quad \text{with} \quad \beta = \epsilon\frac{\Delta t}{\Delta x}. \tag{14}$$

That is, the discretization is centred in time and in space. It implies that you always need "starter" values at the edges of your space/time domain, for which eqn. (14) does not apply. In space, this is taken care of by the boundary values, while in time, you need to initiate the leapfrog stepping with one forward Euler step.

Coordinate	x,y (cm)	Temp. (°C)
A	(0,0)	0
B	(9,0)	5
C	(9,4)	25
D	(11,4)	25
E	(11,0)	5
F	(20,0)	0
G	(20,8)	20
H	(0,8)	20

Table 1: Temperature at each corner. The temperature varies linearly between adjoining nodes (i.e. the temperature along AB increases linearly at 1°C/cm.)

Questions

1. **[30%] Temperature distribution in a heat conductor**

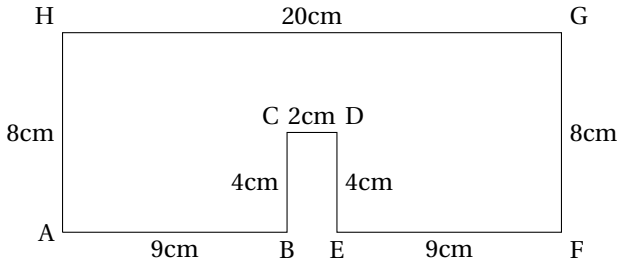

Figure 1: Shape of the heat conductor in question 1.

(a) Consider the illustration in fig. 1 and properties in table 1 of a two dimensional rectangular metal plate with an indentation whose dimensions and boundary temperatures are given in the table.

(b) Write a program to calculate the distribution of temperature under steady state conditions and in the absence of heat sources, using Gauss-Seidel relaxation with replacement and overrelaxation. Construct a grid of spacing 0.1 cm. You can use the script from the textbook called `laplace.py` for guidance; note that it implements the Jacobi method. The program should produce a contour plot (your choice of contour lines, filled contour lines, or other type of plot) of the temperature distribution as a function of x and y. Have the script plot a sequence of iterations during the relaxation process, so that you can see the temperature of heat conductor relax to the equilibrium. While you're working on the script it can be useful to animate the iterations (see the Computational Background information on animations), *but these won't be useful in grading your work unless you can save a video/gif.*.

As the script runs you should see the solution converge to a solution that stops changing much with subsequent iterations.

After you have done the rest of the question, hand in this code.

NOTHING TO SUBMIT (YET).

(c) To evaluate the impact of overrelaxation, run the program two times for 100 iterations: first with $\omega = 0.0$ and second with $\omega = 0.9$. Produce a plot of the solution for each case. What do you notice about the difference between the two cases?

HAND IN THE PLOTS AND A WRITTEN ANSWER.

(d) Now have your program calculate the value of the temperature at each grid point to a precision of 10^{-6} °C, with $\omega = 0.9$. As you watch the iteration, does the non-converged solution you obtain look symmetric about the vertical bisector $x = 10$? Why or why not? After the solution converges, what is temperature at the point $x = 2.5$ cm, $y = 1$ cm?

HAND IN THE CODE AND PLOT OF THE CONVERGED SOLUTION.

2. **[35%]** Application of FTCS: Thermal diffusion in the Earth's crust (Newman exercise 9.4) A classic example of a diffusion problem with a time-varying boundary condition is the diffusion of heat into the crust of the Earth, as surface temperature varies with the seasons. Suppose the mean daily temperature at a particular point on the surface varies as:

$$T_0(t) = A + b\sin\frac{2\pi\, t}{\tau}, \tag{15}$$

where $\tau = 365$days, $A = 10^\circ$ C, and $B = 12^\circ$ C. At a depth of 20m below the surface almost all annual temperature variation disappears and the temperature is, to a good approximation, a constant 11° C (which is higher than the mean surface temperature of 10° C — temperature increases with depth due to heating from the hot core of the planet). The thermal diffusivity of the Earth's crust varies somewhat from place to place, but for our purposes we will treat it as constant with value $D = 0.1 m^2 \text{day}^{-1}$.

   (a) Write a program to calculate the temperature profile of the crust as a function of depth up to 20m and time up to 10 years. Start with temperature everywhere equal to 10° C, except at the surface and the deepest point, choose values for the number of grid points and the time-step h.

SUBMIT YOUR CODE

   (b) Run the program for the first nine simulated years, to allow it to settle down into whatever pattern it reaches. Then for the tenth and final year make a plot with four temperature profiles taken at 3–month intervals on a single graph to illustrate how the temperature changes as a function of depth and time.

SUBMIT YOUR PLOT

   (c) Does this simulation and plot help explain why Canadian homes often have basements, or why wine cellars are often built several metres underground?

NOTHING TO SUBMIT

3. **[35%]** Solving Burger's equation

   Write a code that solves Burgers' equation using the method as described in the Computational Background for this problem.

   Recall that you will have to apply one forward time step to the initial condition before being able to apply eqn. (14).

   You will need to derive the appropriate equation for this step.

   Also, at the boundary points, i.e. u_0^j and u_{N-1}^j, instead of applying eqn. (14), you should just set the values to be equal to the chosen boundary conditions.

   Your code should have parameters for ϵ, Δx, Δt and the length of the spatial domain L_x and for how long the simulation will be run, T_f. Given Δx and L_x, you can find N_x, and similarly for the time dimension. Run your code with $\epsilon = 1$, $\Delta x = 0.02$, $\Delta t = 0.005$, $L_x = 2\pi$, $T_f = 2$. Set the initial condition to be $u(x, t = 0) = \sin(x)$, and the boundary conditions to be $u(0, t) = 0$ and $u(L_x, t) = 0$.

   Create and save plots of your solution at time $t = 0$, 0.5, 1, 1.5. What is happening to the disturbance as time progresses? Do you notice any artifacts that seem unrealistic in the solution?

HAND IN CODE, PLOTS, AND WRITTEN ANSWERS TO QUESTIONS.

Question:	1	2	3	Total
Points:	30	35	35	100