COURSEWORK 1

1. Introduction

Today we are going to look at deqn, a simple simulation code that we will be using for Coursework 1. The main goal of this session is for you to be able to download, compile and run a simple version of the code. We will start with a brief overview of the physics of the application, before covering the design of the program and how to use it.

The deqn code solves the diffusion equation in two-dimensions. The diffusion equation is a partial differential equation that describes the transfer of heat through a material:

$$\frac{\partial u}{\partial t} - \alpha(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial x^2}) = 0 \tag{1}$$

where u is the temperature of the material. x and y are the spatial coordinates, and t is time. The diffusion coefficient, α, will be taken as 1.

In order to solve this equation computationally, it must be discretised. This means we need to change the continuous function specified by the partial differential equation into something that can be approximated by a finite number of elements. We do this using a finite difference scheme, with a second-order central difference in space, and a forward difference in time. The discretisation lets us represent the value of this function at a number of points in space. These points will be stored in an array, making it easy for us to evaluate the function at each point. Lets step through the discretisation of the equation. This section contains some maths, but most of it should be easy to follow. If you don't understand everything, don't worry, we are only trying to give you an overview of how computers are used to solve mathematical problems that often occur in the physical sciences. The first thing we need to do is discretise the the equation in time. If we let

$$\frac{\partial u}{\partial t} = \frac{u^{t+k}_{x,y} - u^{t}_{x,y}}{k} \tag{2}$$

where k $= \Delta t$, then this is our forward difference in time. We can now discretise the remaining terms of the equation in space, letting:

$$\frac{\partial^2 u}{\partial x^2} = \frac{u^{t}_{x+h,y} - 2u^{t}_{x,y} + u^{t}_{x-h,y}}{h^2} \tag{3}$$

and

$$\frac{\partial^2 u}{\partial y^2} = \frac{u^{t}_{x,y+h} - 2u^{t}_{x,y} + u^{t}_{x,y-h}}{h^2} \tag{4}$$

where h is the difference in space (Δx or Δy), then this is our central difference in space. We now substitute these discretised equations into the original formula:

$$\frac{u^{t+k}_{x.y} - u^t_{x,y}}{k} = \frac{u^t_{x+h_x,y} - 2u^t_{x,y} + u^t_{x-h_x,y}}{h_x^2} + \frac{u^t_{x,y+h_y} - 2u^t_{x,y} + u^t_{x,y-h_y}}{h_y^2} \tag{5}$$

and re-arrange to get the formula for $u^{t+k}_{x,y}$:

$$u^{t+k}_{x,y} = ru^t_{x+h_x,y} + ru^t_{x-h_x,y} + r'u^t_{x,y+h_y} + r'u^t_{x,y-h_y} + (1-2r-2r')u^t_{x,y} \tag{6}$$

where $r = \frac{k}{h_x^2}$ and $r' = \frac{k}{h_y^2}$.

This can now be solved explicitly, provided you specify initial values and boundary conditions. This mathematical treatment can seem quite abstract, so lets look at a graphic of what is going on here. Figure 1 is a 1D example of the scheme we are using. Time flows downwards, and using the diagram, we can see how the formulae we have derived let us calculate the value of $u^{t+k}_{x,y}$ using the values of u^t.

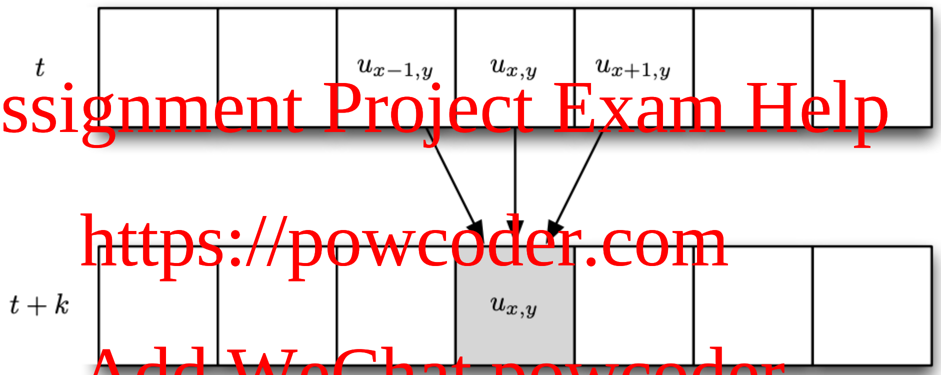

FIGURE 1. Graphical representation of the discretisation used in deqn

From Figure 1 we can see how this kind of calculation might be accomplished by a computer program. As we mentioned, the data can be stored in an array, and its easy to see how we could write a loop to iterate over the array, and update the value of each of the cells using the correct formula. The deqn program is designed to do just this, lets take a look.

The deqn program is designed to solve the diffusion equation in two dimensions. It's written in C++, and the program is encapsulated into multiple objects. In this section we will walk through the source code of deqn, so that you can see how it all fits together.

Looking at the folder you have just downloaded, you will see it contains a Makefile, a README, and two sub-folders, *src* and *test*. Start off by looking in the *src* folder. Throughout this discussion, when we refer to C++ classes we remind you that they will be defined and implemented in two files: a .C file, and a .h file. For a brief overview of the methods and data the class uses, look at the .h file, but for more information, look at the .C file.

Main Lets start by looking at the file main.C - this contains the main method of the program. The file is very simple. It parses the input file name, then creates a new Driver object and calls its run method.

Driver The Driver class only contains one method, which is responsible for running the specified problem. The constructor takes two arguments, an InputFile and a name for the problem. If you take a look at the InputFile class, you will see that it reads in a file, then provides methods for retrieving the data that has been read in. The Driver constructor is responsible for reading some initial data from the input file, and then creating the Mesh and Diffusion objects.

Lets now examine the run method of the Driver class, since this is responsible for running the simulation. The run method contains the time-step loop of our simulation. This loop is responsible for advancing our simulation through the desired amount of time. At each iteration, the doCycle method of the Diffusion class is called.

Diffusion If we take a look at the Diffusion class, we can see that there is at least some maths going on here! The class constructor reads some data from the InputFile and creates a Scheme object. It then calls the init method, which sets the initial values in the Mesh. The Scheme class encapsulates all the mathematical functions necessary to solve the diffusion equation.

Mesh Before we take a look at the ExplicitScheme class (which implements the Scheme interface), let's take a look at the Mesh. This class encapsulates the idea of our discretised grid, storing data in a two 1D arrays. The u0 array holds the current solution value, and the u1 array holds the updated solution. For a given mesh, the data is allocated with one extra cell at each edge to store the boundary values. The data is allocated as one contiguous array for better performance. We treat this memory as a row-order 2D array. Figure 1 shows how the data maps to the discretised grid. The Mesh also holds other information about the grid such as the number of cells, and the minimum and maximum indices. The getTotalTemperature function is used to verify the results of the simulation.

ExplicitScheme Finally we can take a look at the ExplicitScheme class, which provides the necessary maths to advance our simulation of the diffusion equations. The doAdvance method is used to advance the solution by a specific Δt. This

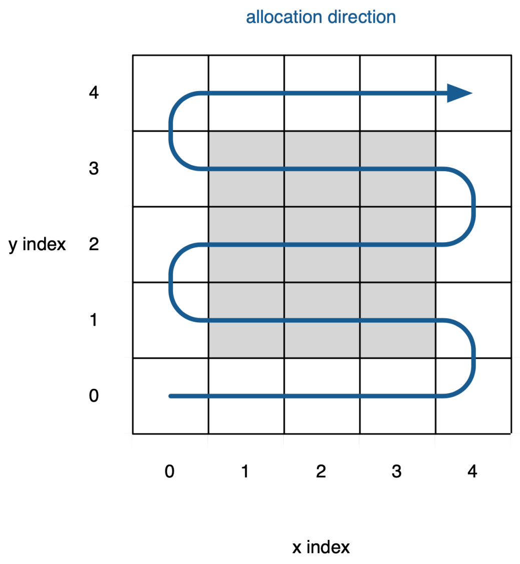

FIGURE 2. Mesh used by deqn. Boundary cells are white, interior cells are grey.

method uses three other methods: diffuse, which calculates the diffusion; reset, which copies data from u1 back to the u0 array; and updateBoundaries, which sets the data in the boundaries of the array. The diffuse method iterates over the entire u0 array and updates the solution value in the u1 array using the equation we derived earlier. Note that we do not update the solution in the boundary cells. The updateBoundaries method uses the reflectBoundaries method to provide a reflective boundary condition, where the solution value next to the boundary is simply reflected (copied) into the boundary cell. The reset method iterates over the entire array, and copies the solution in the u1 array back to u0.

Variable	**Purpose**
dt_max	Stores the maximum Δt value.
dt	Stores the current Δt value.
u0	Stores the current temperature values.
u1	Stores the updated temperature values.

TABLE 1. Variables in the deqn simulation

2. TOOLS

For this coursework you will need the following tools installed on the computer:

- g++
- Make

The tools needed to run the code are already installed and given to you on the lab machines provided. The compilation tool will be Make, which is a common tool to simplify building programs consisting of multiple source code files. The Make tool will call the g++ compiler on the system and will compile the individual source code files, link them together and create the executable in a single command.
It is fine to run the coursework on your own computer when doing this coursework outside of lab hours. If you do want to run this coursework on your own machines then you will need to install these tools yourself.

3. Compilation and Running

To compile the coursework on Lab systems, you can compile using the makefile provided using the following command in the top level directory:

```
$ make
```

This will create the executable in the build directory. Change to this directory using:

```
$ cd build
```

To run the coursework use:

```
$ ./deqn ../test/x.in
```

A common problem encountered while running is that calling make doesn't work. This is likely because you are not in the correct directory. You can verify that the directory is correct using pwd to bet the current working directory or using ls to check to see if the makefile is there.

If you have compiled the coursework and then you want to safely delete it via the terminal then use:

```
$ make clean
```

This will delete the compiled deqn files and object files for you. This will allow you to avoid typing in the 'rm' command yourself if you are nervous about accidentally deleting the wrong files.

If the program run successfully then you should see output similar to the one found in Figure 3

The important thing to note is that the total temperature does not change. This is a consequence of the reflective boundary conditions that we use. If the total temperature does differ then you have a good indication that something has gone wrong in the code, which you will need to fix.

FIGURE 3. Example output of a successful deqn run

```
+++++++++++++++++++++ Running deqn v0.1 +++++++++++++++++++++
+ step: 1, dt: 0.04
+ current total temperature: 9000
...
+ step: 20, dt: 0.04
+ current total temperature: 9000
+++++++++++++++++++++ Run completete. +++++++++++++++++++++
```