CS402: Seminar 2
An Introduction to `deqn`

Today we are going to look at `deqn`, a simple simulation code that we will be using for the remaining lab sessions. 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.

1 Diffusion

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:

$$\frac{\partial u}{\partial t} - \alpha\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\right) = 0$$

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.

Let's 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

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

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:

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

and

$$\frac{\partial^2 u}{\partial y^2} = \frac{u^t_{x,y+h} - 2u^t_{x,y} + u^t_{x,y-h}}{h^2}$$

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:

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

and re-arrange to get the formula for u^{t+k}_x:

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

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 let's look at a graphic of what is going on here. Figure **??** 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.

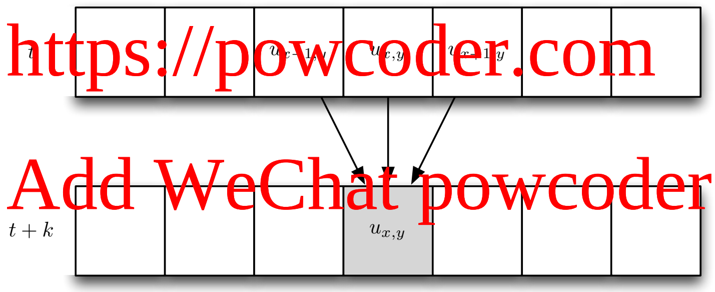

Figure 1: Graphical representation of the discretisation used in `deqn`.

From Figure **??** 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 it's 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, let's take a look.

2 `deqn`

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.

Task 1 Download a copy of `deqn` from the CS402 web page.

Looking at the folder you have just downloaded, you will see it contains a `Makefile`, a `README`, and two subfolders, `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 Let's 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 responible for reading some initial data from the input file, and then creating the `Mesh` and `Diffusion` objects.

Let's now examine the `run` method of the `Driver` class, since this is responsible for running the simulation. The `run` method contains the *timestep* 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 **??** 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 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.

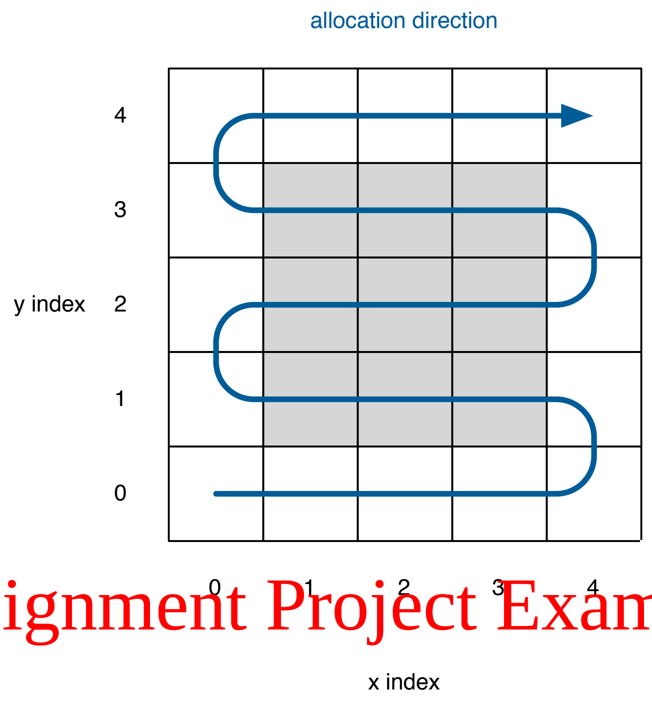

Figure 2: Mesh used by `deqn`. Boundary cells are white, interior cells are grey.

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.

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`.

Task 2 Look at each of the classes used by `deqn` and draw a flow diagram showing the connections between them. Ask a tutor if there are any bits of code that you don't understand.

2.1 Compiling and Running

The previous section gave a very brief overview of the `deqn` code. Now let's compile and run it. The folder you have downloaded contains a Makefile, which we will use to compile the program. All you need to do is type:

```
  make
```

at the command line, and the code will be built and the executable placed in the `build` directory.

Now that we have compiled the program, we can run a simulation. The input file we will use is in the `test` directory. Run the code in the normal way, passing the input file to use as the first argument:

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

as the program runs, you will see some output:

```
+++++++++++++++++++++
  Running deqn v0.1
+++++++++++++++++++++

+ step: 1, dt:   0.04
+ current total temperature: 9000

...

+ step: 20, dt:   0.04
+ current total temperature: 9000

+++++++++++++++++++++
   Run completete.
+++++++++++++++++++++
```

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, and provides a good way to debug the output.

Task 3 Try running a one of the problems in the test directory using the `deqn` code. If you have any problems, speak to a tutor.

3 Extra Resources

For more information on using computers to solve partial differential equations like the diffusion equation, see [?].

References

[1] G. D. Smith, *Numerical Solution of Partial Differential Equations: Finite Difference Methods*, Oxford University Press, 3rd Edition, 1985.