Advanced Computational Science Assignment 2

Liam O'Sullivan — 10309537

April 28, 2014

The Alternating Direction Implicit Method

The Alternative Direction Implicit (ADI) method is an implicit numerical method similar to the Crank-Nicholson method. It is given by

$$(1 - \frac{\alpha_x}{2}\delta_x^2)(1 - \frac{\alpha_y}{2}\delta_y^2)u_{i,j}^{n+1} = (1 + \frac{\alpha_x}{2}\delta_x^2)(1 + \frac{\alpha_y}{2}\delta_y^2)u_{i,j}^n \tag{1}$$

where $\alpha_\gamma = \frac{\tau}{h_\gamma^2}$. Interestingly, this method can be broken into two steps:

$$(1 - \frac{\alpha_x}{2}\delta_x^2)u_{i,j}^{n+1/2} = (1 + \frac{\alpha_y}{2}\delta_y^2)u_{i,j}^n$$

$$(1 - \frac{\alpha_y}{2}\delta_y^2)u_{i,j}^{n+1} = (1 + \frac{\alpha_x}{2}\delta_x^2)u_{i,j}^{n+1/2}$$

where the x-derivative is taken implicitly in the first step, and the y-derivative taken implicitly in the second.

Expanding the first expression above gives

$$-\frac{\alpha_x}{2}u_{i-1,j}^{n+1/2} + (1 + \alpha_x)u_{i,j}^{n+1/2} - \frac{\alpha_x}{2} = \frac{\alpha_y}{2}u_{i,j-1}^n + (1 - \alpha_y)u_{i,j}^n - \frac{\alpha_y}{2}u_{i,j+1}^n$$

where the right side can be computed explicitly. We can write this in matrix form,

$$\begin{pmatrix} 1+\alpha & -\frac{\alpha}{2} & 0 & \cdot & \cdot & 0 \\ -\frac{\alpha}{2} & 1+\alpha & -\frac{\alpha}{2} & 0 & \cdot & \cdot \\ 0 & -\frac{\alpha}{2} & 1+\alpha & -\frac{\alpha}{2} & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ 0 & \cdot & \cdot & \cdot & -\frac{\alpha}{2} & 1+\alpha \end{pmatrix} \begin{pmatrix} u_{0,j}^{n+1/2} \\ u_{1,j}^{n+1/2} \\ \cdot \\ \cdot \\ u_{n_x,j}^{n+1/2} \end{pmatrix} = \begin{pmatrix} d_{0,j}^n \\ d_{1,j}^n \\ \cdot \\ \cdot \\ d_{n_x,j}^n \end{pmatrix}$$

where $d_{i,j}^n$ is the explicitly computed part of the previous expression. As this matrix is clearly diagonally dominant and tridiagonal, we can use the Thomas Algorithm to solve for the unknown $u^{n+1/2}$. This is done by bringing the matrix into upper triangular form. As an example, the matrix equation

$$\begin{pmatrix} b_1 & -c_1 & 0 & \cdot & \cdot & 0 \\ -a_2 & b_2 & -c_2 & 0 & \cdot & \cdot \\ 0 & -a_3 & b_3 & -c_3 & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ 0 & \cdot & \cdot & \cdot & -a_n & b_n \end{pmatrix} \begin{pmatrix} u_0 \\ u_1 \\ \cdot \\ \cdot \\ u_n \end{pmatrix} = \begin{pmatrix} d_1 \\ d_2 \\ \cdot \\ \cdot \\ d_n \end{pmatrix}$$

can be transformed into

$$\begin{pmatrix} 1 & -e_1 & 0 & \cdot & \cdot & 0 \\ 0 & 1 & -e_2 & 0 & \cdot & \cdot \\ 0 & \cdot & 1 & -e_3 & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ 0 & \cdot & \cdot & \cdot & \cdot & 1 \end{pmatrix} \begin{pmatrix} u_0 \\ u_1 \\ \cdot \\ \cdot \\ u_n \end{pmatrix} = \begin{pmatrix} f_1 \\ f_2 \\ \cdot \\ \cdot \\ f_n \end{pmatrix}$$

with

$$e_k = \frac{c_k}{b_k - a_k e_{k-1}}, \qquad f_k = \frac{d_k + a_k f_{k-1}}{b_k - a_k e_{k-1}}$$

As the final line produces an equation of one unknown, it is possible to solve this system through backwards substitution. This is done for both parts of the ADI method.

1

In this section, we are asked simply to write a program which will solve the linear diffusion equation,

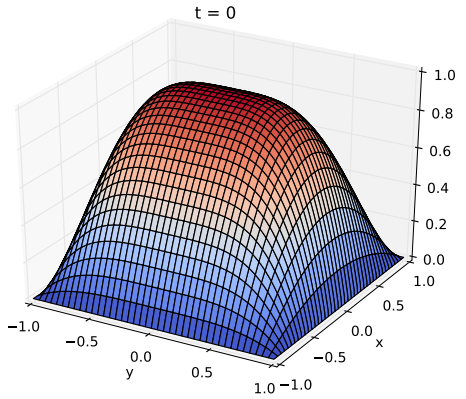
$$\frac{\partial u}{\partial t} = \nabla \cdot \nabla u \tag{2}$$

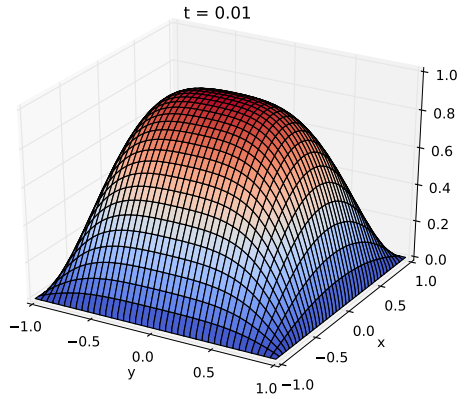
in two dimensions, making our equation

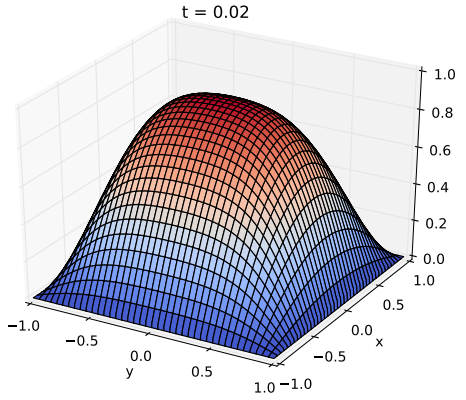
$$\frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}. \tag{3}$$

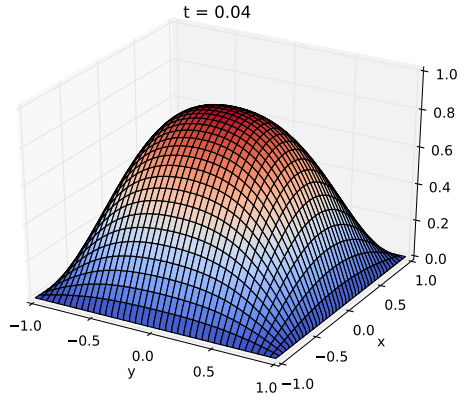
We are to use the ADI method, described previously. We are told to solve the equation on a rectangular grid, $x \in [-1, 1],\ y \in [-1, 1]$, with initial condition $u(x, y, t = 0) = (1 - x^2)(1 - y^4)$, and Dirichlet boundary conditions, $u(-1, y, t) = u(1, y, t) = u(x, -1, t) = u(x, 1, t) = 0$.

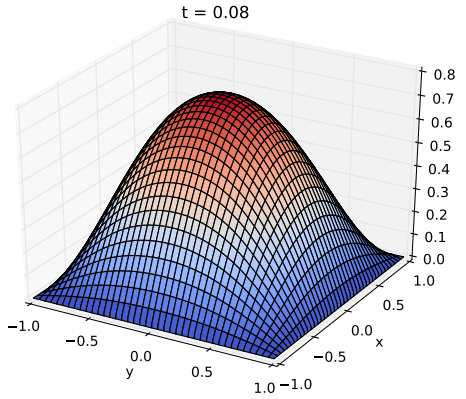
The program was written in Python 3, and is attached. On the following page are various snapshots of the function at indicated times. This case was solved on a grid with spacing $h_x = h_y = 1/200$, and 1000 time steps.

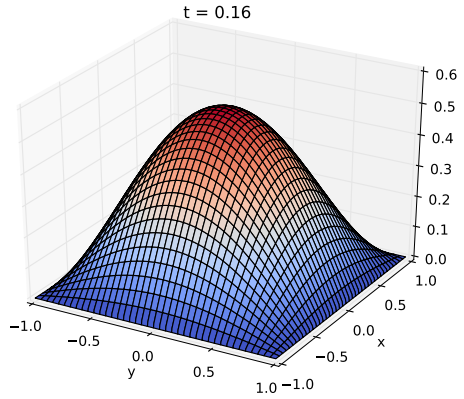

t = 0
1.0
0.8
0.6
0.4
0.2
0.0
−1.0
−0.5
0.0
0.5
1.0
y
x

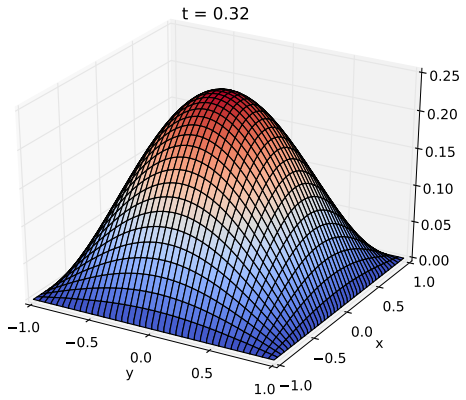

t = 0.01
1.0
0.8
0.6
0.4
0.2
0.0
−1.0
−0.5
0.0
0.5
1.0
y
x

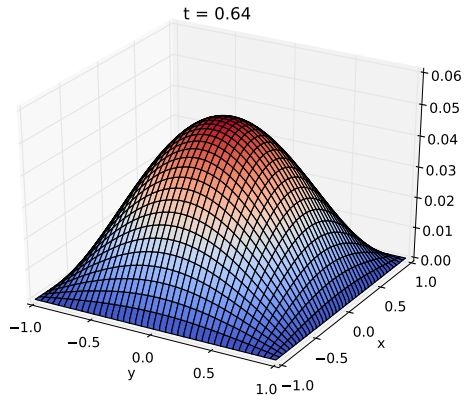

t = 0.02
1.0
0.8
0.6
0.4
0.2
0.0
−1.0
−0.5
0.0
0.5
1.0
y
x


t = 0.04
1.0
0.8
0.6
0.4
0.2
0.0
−1.0
−0.5
0.0
0.5
1.0
y
x


t = 0.08
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0.0
−1.0
−0.5
0.0
0.5
1.0
y
x


t = 0.16
0.6
0.5
0.4
0.3
0.2
0.1
0.0
−1.0
−0.5
0.0
0.5
1.0
y
x


t = 0.32
0.25
0.20
0.15
0.10
0.05
0.00
−1.0
−0.5
0.0
0.5
1.0
y
x


t = 0.64
0.06
0.05
0.04
0.03
0.02
0.01
0.00
−1.0
−0.5
0.0
0.5
1.0
y
x

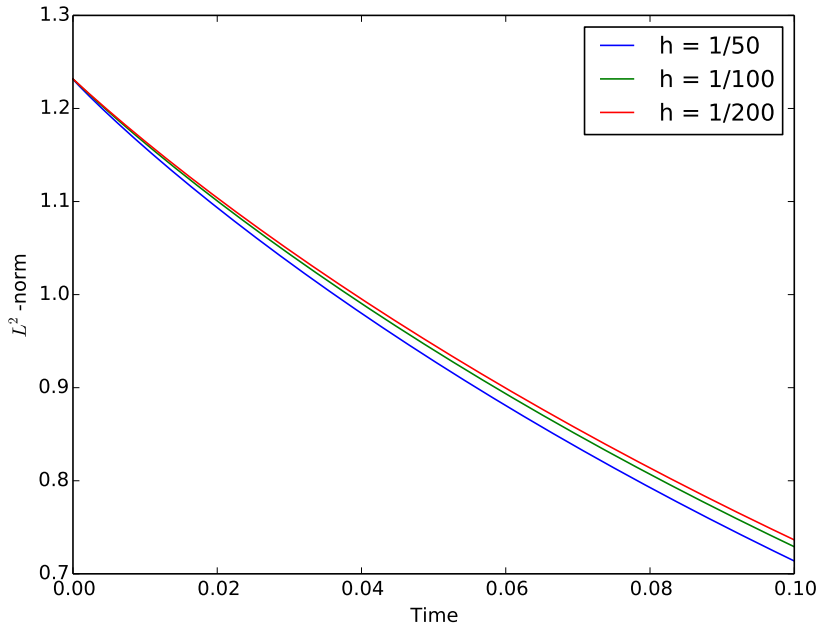


Figure 1: Evolution of L^2-norms in time for three resolutions.

2

i

For this exercise, we are told to take $h_x = h_y = \frac{1}{200}$, and to maximise the number of time steps in $t \in [0, 0.1]$ based on our available memory. In this section, the data for each time step is saved in one contiguous memory array. The datatype used is a 64-bit float, making each entry 8 bytes in size. With this, our total memory usage in GB will be

$$M = \frac{8 \times 400^2}{1024^3} n_t$$

where n_t is the number of time-steps. Taking $n_t = 1000$ gives just under 1.2GB used by U, which is reasonable on my machine with 2GB of memory, as this should at no point cause processes to use swap. During the runs, I was running other programs(Firefox, etc.), making using the entire 2GB undesirable.

ii

This exercise concerns the L^2-norms of the function. The L^2-norm is defined as

$$||u(\cdot, t)||_2 = \sqrt{\int u(x, y, t)^2 dy dx}.$$

In the code, I have approximated the integral as a sum:

$$||u(\cdot, t)||_2 \approx \sqrt{\sum_{i,j} u_{i,j}^2 h_y h_x}$$

Figure 1 shows the evolution of the L^2 norms in time for resolutions $h_0 = 1/200$, $h_1 = 1/100$, $h_2 = 1/50$, with $h_x = h_y = h$.

iii

We are told that comparing the quantity

$$\epsilon_k(t) = 1 - \frac{||u_k(\cdot, t)||_2}{||u_0(\cdot, t)||_2}$$

where $||u_k(\cdot, t)||_2$ is the L^2-norm of the solution corresponding to h_k. We are told to take $h_0 = 1/200$ and $h_1 = 1/100$, $h_2 = 1/50$. The results are shown in figure 2.

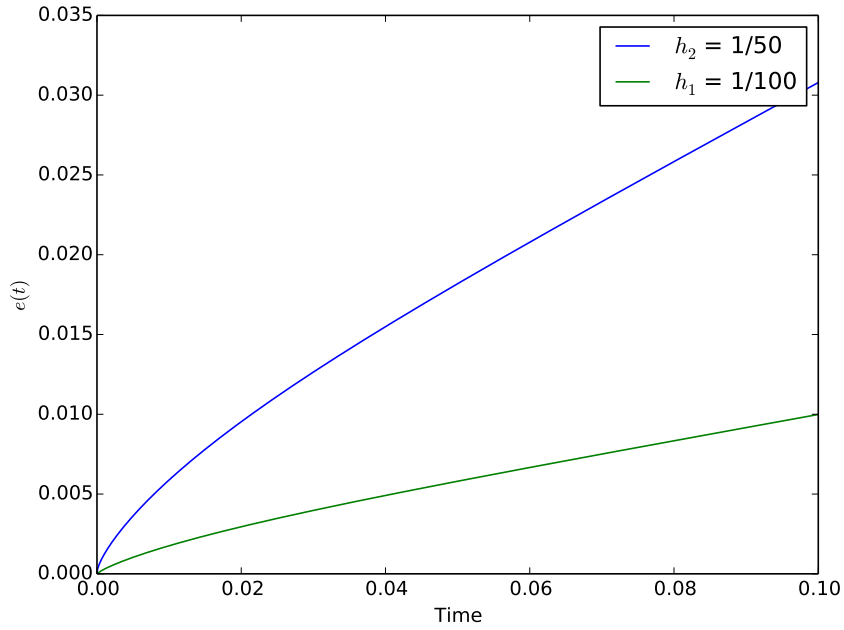


Figure 2: $\epsilon_{1,2}(t)$

iv

This section asked to extend the previous resolution study to higher resolutions. To do this, it is recommended that we do not store the entire array for u, but only the amount necessary for computation. I found that only one array for u and one for $u^{n+1/2}$ are necessary, and the computation can effectively be done in place. After each run, the L^2-norm is computed and assigned into an array as before. Unfortunately, the runtime for 10 time steps is 300 seconds at the resolution of $h = 1/800$, making timing and comparing a variety of resolutions arduous. Due to time constraints, $nt = 100$ was used for this section.

Figure 3 shows the relatively small difference in $\epsilon(t)$ for $h < 1/200$ compared to the previous cases. It can be seen that higher resolutions converge better.

Figure 4 shows the runtime dependence of the algorithm on resolution. When the `run()` function is called, it noted the process' CPU time on entry, and subtracts the CPU time on exit, returning the difference as the runtime. It can be seen to scale somewhat inversely with resolution.

Discussion

Generally, Python seemed significantly slower than the Matlab programs of my classmates, up to an order of magnitude in some cases. This was a severe constraint in the final section, limiting the number of time steps. Various efforts were made to improve the performance, but most failed. Vectorising expressions instead of `for` loops was tested

```
for i in ...: # Normal.
    for j in ...:
        U[i][j] = TMP[j]
```

```
for i in ...: # Vectorised.
    U[i][:] = TMP[:]
```

but actually worsened performance by an order of magnitude in the section of interest.

Obvious bottlenecks include repeatedly calling the function to solve the Thomas algorithm, which is not particularly efficient, and the use of many `for` loops.

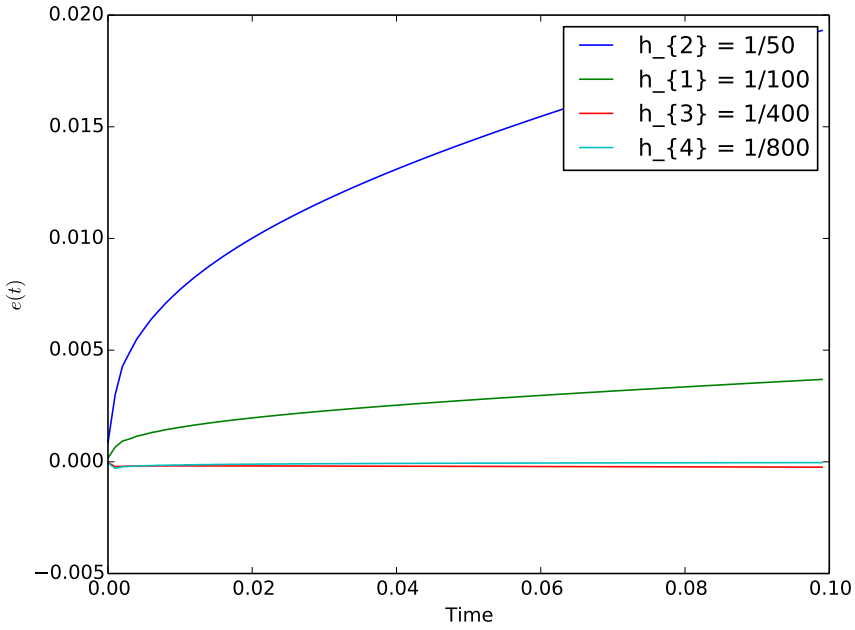


Figure 3: $\epsilon(t)$ for various resolutions.

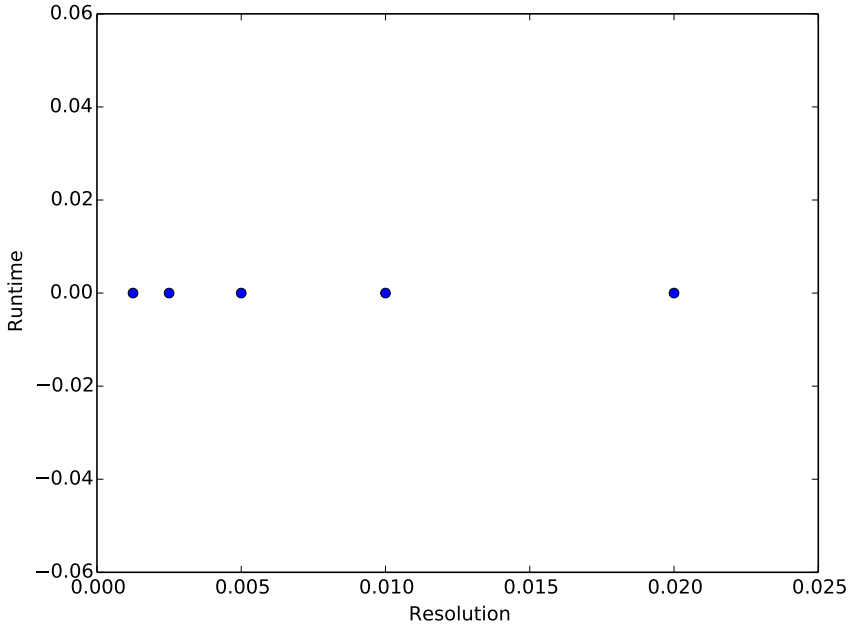


Figure 4: Runtime dependence on resolution.