

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

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

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,

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

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

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

can be transformed into

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

with

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

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,

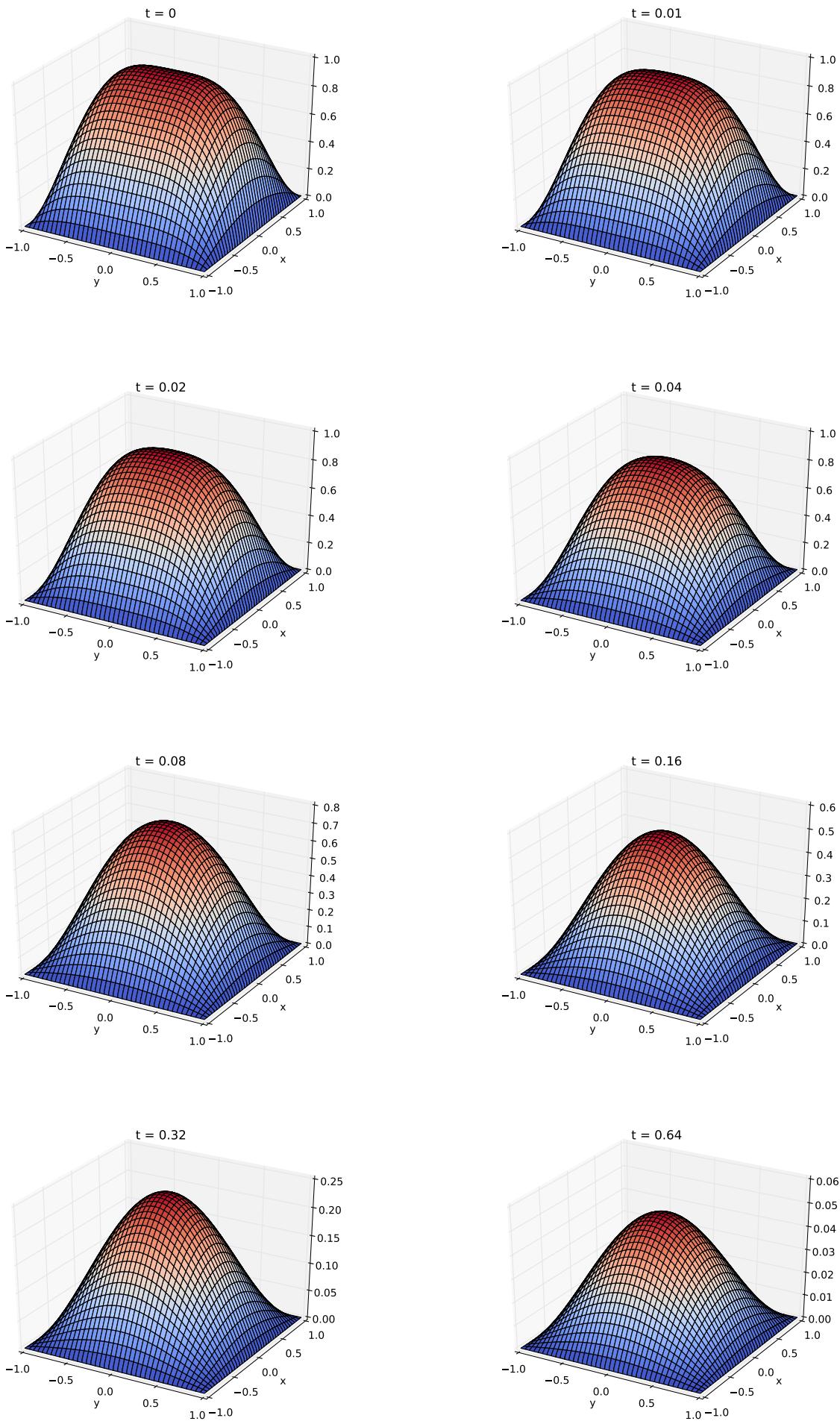
$$\frac{\partial u}{\partial t} = \nabla \cdot \nabla u \quad (2)$$

in two dimensions, making our equation

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

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.



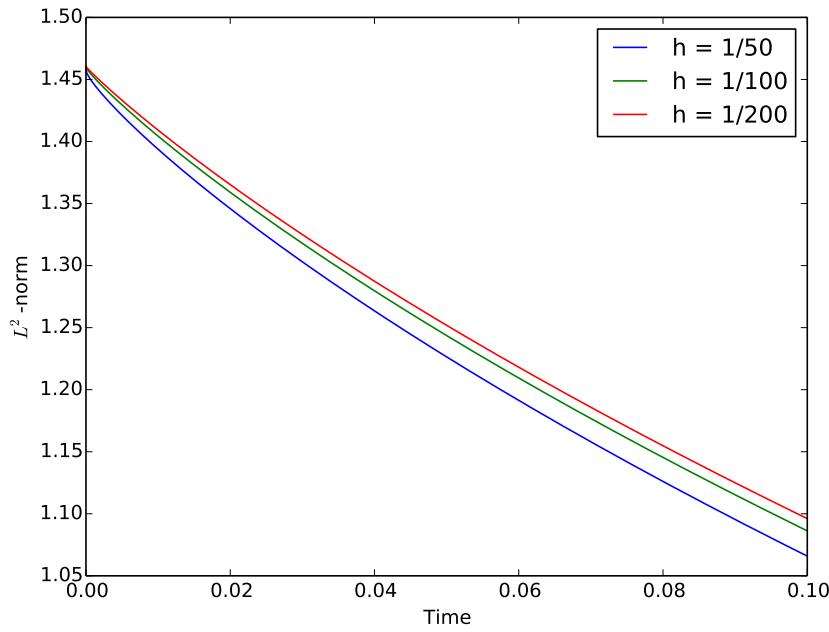


Figure 1: Evolution of L^2 -norms in time for three resolutions.

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

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.

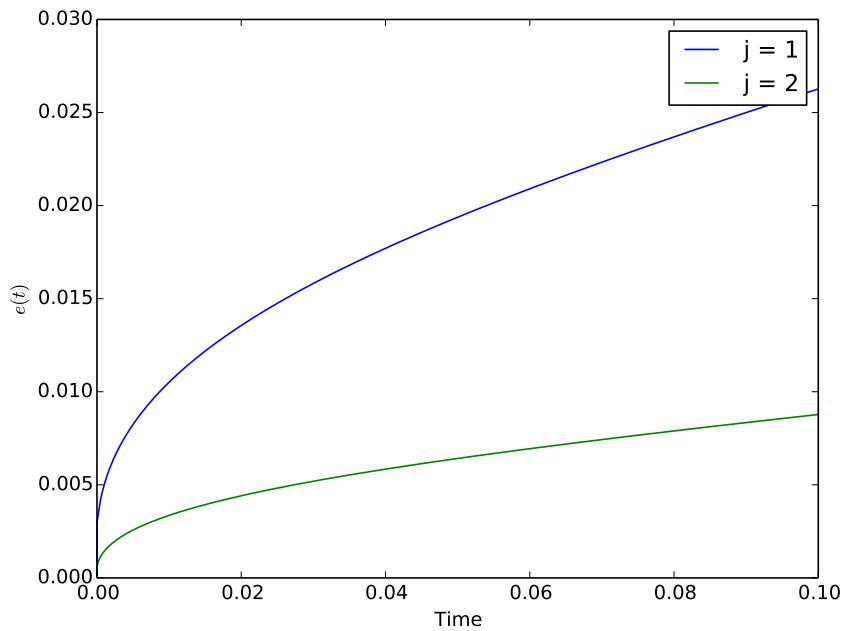


Figure 2:

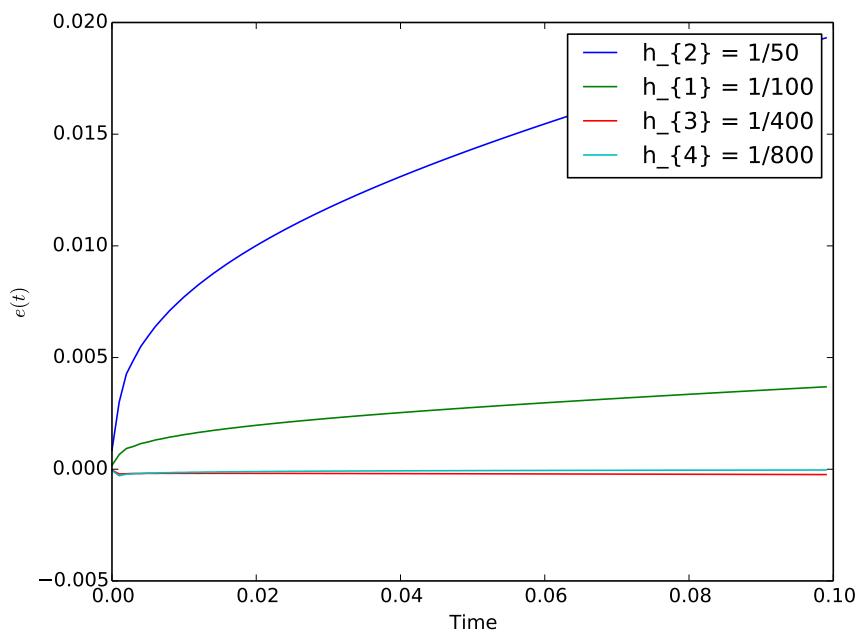


Figure 3:

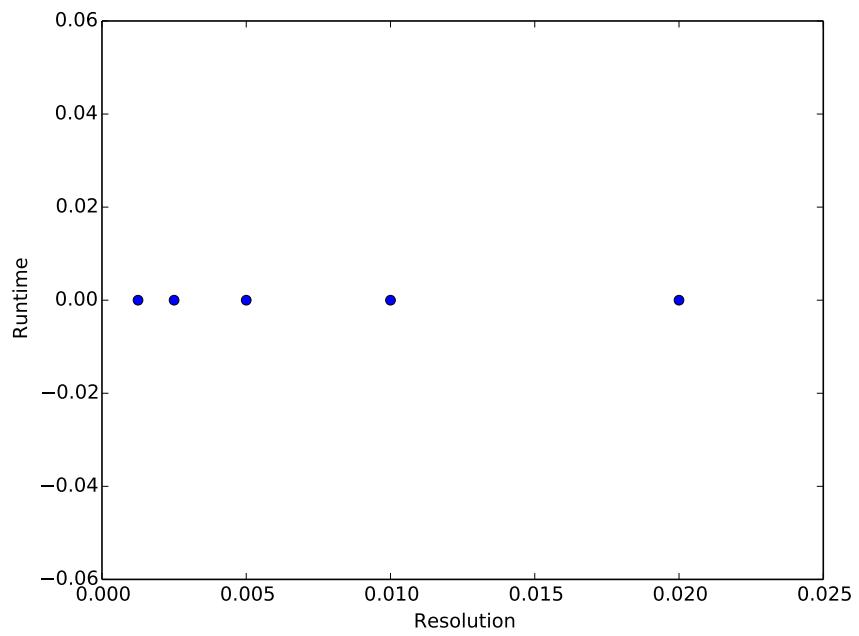


Figure 4: