AMATH 581 Homework 2
Shallow Liquid Simulation

Erik Neumann
610 N. 65th St., Seattle, WA 98103
erikn@MyPhysicsLab.com

November 19, 2001

Abstract

A model of shallow fluid behavior is evaluated using a variety of numerical solving techniques. The model is defined by a pair of partial differential equations which have two dimensions in space and one dimension of time. The equations concern the vorticity ω and the stream function ψ which are related to the velocity field of the fluid. The equations are first discretized in time and space. The time behavior is evaluated using a Runge-Kutta ordinary differential equation solver. The spatial behavior is solved using either Fast Fourier Transform, Gaussian Elimination, LU Decomposition, or iterative solvers. The performance of these techniques is compared in regards to execution time and accuracy.

Contents

1 Introduction and Overview

We consider the governing equations associated with shallow fluid modeling. The intended application is the flow of the earth's atmosphere or ocean circulation. The model assumes a 2-dimensional flow, with not much movement up or down. Another assumption is that the fluid is shallow, ie. that the vertical dimension is much smaller than the horizontal dimensions.

The velocity field is given by the set of vectors $\vec{v}$ at each point with components

$$\vec{v} = \begin{pmatrix} u \\ v \\ w \end{pmatrix} \tag{1}$$

where u is the x component of the velocity, v is the y component of the velocity and so on. The height of the fluid is given by $h(x, y, t)$. From conservation of mass we can derive the following

$$h_t + (hu)_x + (hv)_y = 0 \tag{2}$$

Conservation of momentum leads to the following two equations

$$(hu)_t + (hu^2 + \frac{1}{2}gh^2)_x + (huv)_y = fhv \tag{3}$$

$$(hv)_t + (hv^2 + \frac{1}{2}gh^2)_y + (huv)_x = -fhu \tag{4}$$

Next, assume that h is constant (to leading order). Then equation (2) becomes

$$u_x + v_y = 0 \tag{5}$$

which expresses that this is an incompressible flow. We can define the stream function ψ by

$$u = -\psi_y \qquad\qquad v = \psi_x \tag{6}$$

which automatically satisfies the incompressibility of equation (5). The remaining two equations become

$$u_t + 2uu_x + (uv)_y = fv \tag{7}$$

$$v_t + 2vv_y + (uv)_x = -fu \tag{8}$$

Define the vorticity ω by

$$\omega = v_x - u_y. \tag{9}$$

We can simplify these equations as follows. Subtract the y-derivative of (7) from the x-derivative of (8) and use equations (5) and (9) to simplify (see appendix C for details). The result is

$$\omega_t + u\omega_x + v\omega_y = 0 \tag{10}$$

Adding a diffusion term (representing viscosity) to the right hand side and using equation (6) leads to

$$\omega_t - \psi_y\omega_x + \psi_x\omega_y = \nu\ \nabla^2\omega \tag{11}$$

where $\nabla^2\omega = \omega_{xx} + \omega_{yy}$ and ν is a small constant factor.

Another relationship between vorticity ω and the stream function ψ is gleaned from the following

$$\omega = v_x - u_y = (\psi_x)_x - (-\psi_y)_y \tag{12}$$

$$\omega = \nabla^2\psi \tag{13}$$

The system is then given by

$$\boxed{\omega_t + [\psi, \omega] = \nu \, \nabla^2 \omega} \tag{14}$$

$$\boxed{\nabla^2 \psi = \omega} \tag{15}$$

where $[\psi, \omega] = \psi_x \omega_y - \psi_y \omega_x$ represents the advection term. This advection-diffusion equation has characteristic behavior of the three PDE classifications,

parabolic:	$\omega_t = \nu \, \nabla^2 \omega$	(diffusion)
elliptic:	$\nabla^2 \psi = \omega$	
hyperbolic:	$\omega_t + [\psi, \omega] = 0$	(advection)

The numerical solution technique consists of the following steps, starting with an initial value for ω.

1. Given a value of ω, solve equation (15) for ψ. Discretizing the ∇^2 operator turns this into a matrix equation of the form $\mathbf{A}\vec{\psi} = \vec{\omega}$ where $\vec{\psi}$ is a discretized vector rearrangement of ψ and similarly for $\vec{\omega}$.
2. Discretize equation (14) so that we get another matrix equation $\vec{\omega_t} = \mathbf{B}\vec{\omega}$. We regard ψ as fixed for a small period of time. This is now in the form of a set of ordinary differential equations.
3. Step forward in time using a Runge-Kutta ordinary differential equation solver to get ω at a small time in the future.

This new value of ω is then used in step 1 and the process can continue indefinitely. We will examine various techniques for solving the matrix equation in step (1), including Fast Fourier Transform, Gaussian Elimination, LU Decomposition, and iterative solvers.

2 Theoretical Background

We seek to numerically simulate the system given by equations (14) and (15). We assume that there is an initial vorticity $\omega(x, y, t)$ specified at time $t = 0$. The area we are solving over is a box defined by

$$x \in [-L/2, L/2] \qquad y \in [-L/2, L/2] \tag{16}$$

for some constant length L.

We assume periodic boundary conditions, so that

$$\omega(-L/2, y, t) = \omega(L/2, y, t) \tag{17}$$

$$\omega(x, -L/2, t) = \omega(x, -L/2, t) \tag{18}$$

and similarly for ψ.

2.1 Solving for ψ - Matrix Method

The first step is to solve equation (15) for ψ from a given value of ω. We use second order central differencing to approximate the second derivatives, so that equation (15) becomes

$$\frac{\psi(x+\triangle x) - 2\psi(x) + \psi(x-\triangle x)}{\triangle x^2} + \frac{\psi(y+\triangle y) - 2\psi(y) + \psi(y-\triangle y)}{\triangle y^2} = \omega(x,y) \tag{19}$$

Assume that the box is discretized into N segments horizontally and vertically so that $\triangle x = \triangle y = L/N = \delta$. Label the points along the x axis as $x_1, x_2, \ldots, x_n$ and similarly for y. We define the following notation

$$\psi(x_m, y_n) = \psi_{m,n} \tag{20}$$

and then we can write equation (19) as

$$-4\psi_{m,n} + \psi_{m+1,n} + \psi_{m-1,n} + \psi_{m,n+1} + \psi_{m,n-1} = \delta^2 \omega_{m,n} \tag{21}$$

Note that the boundary conditions give us

$$\psi_{m,N+1} = \psi_{m,1} \qquad \psi_{m,0} = \psi_{m,N} \tag{22}$$

$$\psi_{N+1,n} = \psi_{1,n} \qquad \psi_{0,n} = \psi_{N,n} \tag{23}$$

and similarly for ω.

Suppose we form ψ into a vector row-wise as follows:

$$\vec{\psi} = (\psi_{1\,1}, \psi_{1\,2}, \ldots, \psi_{1\,n}, \psi_{2\,1}, \ldots, \psi_{2\,n}, \ldots, \psi_{n\,n}) \tag{24}$$

and similarly for $\vec{\omega}$. These vectors are of length N^2. We can then write equation (21) as a matrix equation,

$$\mathbf{A}\vec{\psi} = \delta^2 \vec{\omega} \tag{25}$$

The matrix $\mathbf{A}$ is a sparse banded matrix with dimensions $N^2 \times N^2$. An example for $N = 4$ is shown below.

$$\mathbf{A} = \begin{pmatrix}
-4 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
1 & -4 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 1 & -4 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
1 & 0 & 1 & -4 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
1 & 0 & 0 & 0 & -4 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 1 & -4 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 1 & -4 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 1 & 0 & 1 & -4 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & -4 & 1 & 0 & 1 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & -4 & 1 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & -4 & 1 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & -4 & 0 & 0 & 0 & 1 \\
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & -4 & 1 & 0 & 1 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & -4 & 1 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & -4 & 1 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & -4
\end{pmatrix}$$

Each row of **A** corresponds to one instance of equation (21). For example, the first row of the example matrix above corresponds to the equation

$$-4\psi_{1\,1} + \psi_{1\,2} + \psi_{1\,4} + \psi_{2\,1} + \psi_{4\,1} = \delta^2 \omega_{1\,1}$$

The construction of the **A** matrix is further explained in section 3.1.

Matrix methods can now be used to solve equation (25) for $\vec{\psi}$ given $\vec{\omega}$.

2.2 Solving for ψ - FFT Method

When using the Fast Fourier Transform (FFT) we find ψ in a different way. The Fourier transform and its inverse are defined as

$$\hat{f}(k) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} e^{ikx} f(x)\, dx \tag{26}$$

$$f(x) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} e^{-ikx} \hat{f}(k)\, dk \tag{27}$$

We will denote the Fourier transform in x of a function f by $\hat{f}$. The key relation is among derivatives of functions. The general result for the n-th derivative of a function f is

$$\hat{f}^{(n)} = (-ik)^n \hat{f} \tag{28}$$

We begin by taking the Fourier transform in x of equation (15)

$$\widehat{\psi_{xx}} + \widehat{\psi_{yy}} = \hat{\omega} \tag{29}$$

$$-k_x^2 \hat{\psi} + \widehat{\psi_{yy}} = \hat{\omega} \tag{30}$$

We denote the Fourier transform in y of f by $\tilde{f}$. Then taking the Fourier transform in y of equation (30) gives

$$-k_x^2 \tilde{\hat{\psi}} - k_y^2 \tilde{\hat{\psi}} = \tilde{\hat{\omega}} \tag{31}$$

$$\tilde{\hat{\psi}} = \frac{-\tilde{\hat{\omega}}}{k_x^2 + k_y^2} \tag{32}$$

To find ψ we inverse transform in x and y the right-hand side of equation (32).

2.3 Discretize the Advection-Diffusion Equation

Next we discretize the advection-diffusion equation which is

$$\omega_t + [\psi, \omega] = \nu\, \nabla^2 \omega \tag{14}$$

This will allow us to step forward in time. Writing out the derivatives and rearranging we have

$$\omega_t = \psi_y \omega_x - \psi_x \omega_y + \nu \left(\omega_{xx} + \omega_{yy}\right) \tag{33}$$

Using central differences in x and y as in section 2.1 we have

$$\begin{aligned}\omega(x,y)_t = &\left(\frac{\psi(x,y+\triangle y) - \psi(x,y-\triangle y)}{2\triangle y}\right)\left(\frac{\omega(x+\triangle x,y) - \omega(x-\triangle x,y)}{2\triangle x}\right)\\ &-\left(\frac{\psi(x+\triangle x,y) - \psi(x-\triangle x,y)}{2\triangle x}\right)\left(\frac{\omega(x,y+\triangle y) - \omega(x,y-\triangle y)}{2\triangle y}\right)\\ &+\nu\left(\frac{\omega(x+\triangle x) - 2\omega(x) + \omega(x-\triangle x)}{\triangle x^2} + \frac{\omega(y+\triangle y) - 2\omega(y) + \omega(y-\triangle y)}{\triangle y^2}\right)\end{aligned} \tag{34}$$

If we use the notation of equation (20) this becomes

$$\begin{aligned}(\omega_{m,n})_t = &\frac{1}{4\delta}\Big((\psi_{m,n+1} - \psi_{m,n-1})(\omega_{m+1,n} - \omega_{m-1,n})\\ &\qquad - (\psi_{m+1,n} - \psi_{m-1,n})(\omega_{m,n+1} - \omega_{m,n-1})\Big)\\ &+\nu\frac{1}{\delta^2}\Big(-4\omega_{m,n} + \omega_{m+1,n} + \omega_{m-1,n} + \omega_{m,n+1} + \omega_{m,n-1}\Big)\end{aligned} \tag{35}$$

Suppose we form ω into a vector row-wise as follows:

$$\vec{\omega} = (\omega_{1\,1}, \omega_{1\,2}, \ldots, \omega_{1\,n}, \omega_{2\,1}, \ldots, \omega_{2\,n}, \ldots, \omega_{n\,n}) \tag{36}$$

Then we can write (35) as an equivalent matrix equation

$$\vec{\omega}_t = \left(\frac{1}{4\delta}\mathbf{B} + \nu\frac{1}{\delta^2}\mathbf{A}\right) * \vec{\omega} \tag{37}$$

where $\mathbf{A}$ is as in section 2.1 and $\mathbf{B}$ corresponds to the advection terms in equation (35). Note that we regard ψ as fixed for each small time step, and so the values of ψ become coefficients in matrix $\mathbf{B}$. We now have a set of ordinary differential equations in equation (37), and we can step forward in time using this equation with a Runge-Kutta solver.

3 Algorithm Implementation and Development

The Matlab file `ev2.m` implements all of the methods discussed. Various parameters can be set near the top of the file such as

- Which solver to use (fft, Gaussian elimination, LU decomposition, etc.)

- Which initial condition to use.
- Diffusion factor ν.
- Number of grid points N.
- Time step Δt.
- How long to run the simulation.
- Time between displayed frames.

We consider the following initial conditions for ω

- One Gaussian shaped vortex, longer than it is wide (elliptic).
- Two same "charged" Gaussian vortices next to each other.
- Two oppositely "charged" Gaussian vortices next to each other.
- Two pairs of oppositely "charged" vortices colliding.
- A random assortment (in position, strength, charge, ellipticity) of vortices.

For the time stepping part of the algorithm, we use `ode23` with the right hand side as defined by equation (37).

3.1 Construction of Matrix A

To understand how matrix **A** of equation (25) is constructed consider the following question:

> Within the $\vec{\psi}$ vector, where is $\psi_{m,n+1}$ relative to $\psi_{m,n}$?

Recall that the $\vec{\psi}$ vector is built row-wise from the matrix ψ.

$$\vec{\psi} = (\psi_{1\,1}, \psi_{1\,2}, \ldots, \psi_{1\,n}, \psi_{2\,1}, \ldots, \psi_{2\,n}, \ldots, \psi_{n\,n}) \tag{24}$$

Suppose that $\psi_{m,n}$ is the j-th element of $\vec{\psi}$. Since we build $\vec{\psi}$ row-wise, the $j+1$st element will correspond to $\psi_{m,n+1}$, it moves us over one column. Except when we are at the right-hand edge of ψ. In that case the boundary conditions wrap around and $\psi_{m,n+1} = \psi_{m,1}$. Within the $\vec{\psi}$ vector, this corresponds to the entry at $j - N + 1$.

Similar considerations lead to the following rules for locating terms in equation (21). Note that we use a special version of mod for which $n \pmod n = n$ instead of the usual $n \pmod n = 0$.

term	location in $\vec{\psi}$
$\psi_{m,n}$	j
$\psi_{m,n+1}$	$j+1 \pmod N + N\lfloor\frac{j-1}{N}\rfloor$
$\psi_{m,n-1}$	$j-1 \pmod N + N\lfloor\frac{j-1}{N}\rfloor$
$\psi_{m+1,n}$	$j+N \pmod{N^2}$
$\psi_{m-1,n}$	$j-N \pmod{N^2}$

3.2 Construction of Matrix B

The matrix **B** in equation (37) corresponds to the following terms from equation (35).

$$(\psi_{m,n+1} - \psi_{m,n-1})(\omega_{m+1,n} - \omega_{m-1,n}) - (\psi_{m+1,n} - \psi_{m-1,n})(\omega_{m,n+1} - \omega_{m,n-1}) \tag{38}$$

Define central differences $\psi^y_{m,n} = \psi_{m,n+1} - \psi_{m,n-1}$ and $\psi^x_{m,n} = \psi_{m+1,n} - \psi_{m-1,n}$. Then equation (38) becomes

$$\psi^y_{m,n}\omega_{m+1,n} - \psi^y_{m,n}\omega_{m-1,n} - \psi^x_{m,n}\omega_{m,n+1} + \psi^x_{m,n}\omega_{m,n-1} \tag{39}$$

After we form the central differences ψ^y, ψ^x we need to place them into the positions in matrix **B** as given in the above equation. Recall that the vector $\vec{\omega}$ is organized row-wise as

$$\vec{\omega} = (\omega_{1\,1}, \omega_{1\,2}, \ldots, \omega_{1\,n}, \omega_{2\,1}, \ldots, \omega_{2\,n}, \ldots, \omega_{n\,n}) \tag{40}$$

and suppose that we have similarly organized the central differences ψ^y, ψ^x.

Consider the first term

$$\psi^y_{m,n}\omega_{m+1,n} \tag{41}$$

Suppose the entry $\psi^y_{m,n}$ is the j-th member of the vector $\vec{\psi^y}$. Recall that equation (35) calculates the time derivative for $\omega_{m,n}$. So the entry we make will be on the j-th row of matrix **B**. The column where we place the entry determines which member of the $\vec{\omega}$ vector the entry multiplies. So for the first term (41) the column will be $j + N \pmod{N^2}$ because increasing the row number moves you N forward in the $\vec{\omega}$ vector. (Note: we are using a special version of mod for which $n \pmod n = n$ instead of $n \pmod n = 0$). An example for $N = 4$ is shown below.

j	1	2	3	4	5	6	7	8
$j+N \pmod{N^2}$	5	6	7	8	9	10	11	12
j	9	10	11	12	13	14	15	16
$j+N \pmod{N^2}$	13	14	15	16	1	2	3	4

So $\psi^y_{1\,1}$ is placed into $\mathbf{B}(1,5)$, $\psi^y_{1\,2}$ is placed into $\mathbf{B}(2,6)$, etc.

A similar analysis yields these rules for all the terms of equation fragment (39). The rules specify the row and column to put the j-th entry of $\vec{\psi^y}$ or $\vec{\psi^x}$.

term	row of $\mathbf{B}$	column of $\mathbf{B}$
ψ^y	j	$j + N \pmod{N^2}$
$-\psi^y$	j	$j - N \pmod{N^2}$
$-\psi^x$	j	$j + 1 \pmod{N} + N\lfloor\frac{j-1}{N}\rfloor$
ψ^x	j	$j - 1 \pmod{N} + N\lfloor\frac{j-1}{N}\rfloor$

An example of applying these rules for $N = 4$ is shown below.

	j	1	2	3	4	5	6	7	8
ψ^y	$j + N \pmod{N^2}$	5	6	7	8	9	10	11	12
$-\psi^y$	$j - N \pmod{N^2}$	13	14	15	16	1	2	3	4
$-\psi^x$	$j + 1 \pmod{N} + N\lfloor\frac{j-1}{N}\rfloor$	2	3	4	1	6	7	8	5
ψ^x	$j - 1 \pmod{N} + N\lfloor\frac{j-1}{N}\rfloor$	4	1	2	3	8	5	6	7
	j	9	10	11	12	13	14	15	16
ψ^y	$j + N \pmod{N^2}$	13	14	15	16	1	2	3	4
$-\psi^y$	$j - N \pmod{N^2}$	5	6	7	8	9	10	11	12
$-\psi^x$	$j + 1 \pmod{N} + N\lfloor\frac{j-1}{N}\rfloor$	10	11	12	9	14	15	16	13
ψ^x	$j - 1 \pmod{N} + N\lfloor\frac{j-1}{N}\rfloor$	12	9	10	11	16	13	14	15

3.3 Pinning the Value of $\psi(1, 1)$

The first step of the algorithm is, given ω, to solve equation (25) for ψ. For the matrix solve approach, $\mathbf{A}$ must be non-singular. The condition number $\mathcal{K}(\mathbf{A})$ of a matrix gives us a good idea of how close a matrix is to being singular. The definition is

$$\mathcal{K}(\mathbf{A}) = \|\mathbf{A}\| \, \|\mathbf{A}^{-1}\| \tag{42}$$

where $\|\mathbf{A}\|$ denotes a matrix norm. If $\mathcal{K}(\mathbf{A})$ is close to 1, then we can be confident that $\mathbf{A}$ is non-singular. If $\mathcal{K}(\mathbf{A})$ is very large, then $\mathbf{A}$ is nearly singular. It turns out (using the Matlab command `condest`) that $\mathbf{A}$ in equation (25) has a very large condition number, on the order of $\mathcal{K}(\mathbf{A}) = 10^{17}$. And indeed matrix methods do not give useful results in solving equation (25) as is.

This situation arises because we are essentially performing integration in solving for ψ and so there is an additive constant introduced. This means that there are an infinite number of solutions to equation (25) that differ by an additive constant. When a matrix equation has an infinite number of solutions it is singular.

To fix this, we need to pin down one value of ψ. Then there will be only one solution possible and the matrix will be non-singular. So we add one more equation, namely $\psi(1, 1) = 0$, which pins down the value of $\psi(1, 1)$. This is

represented as an additional row added to $\mathbf{A}$ consisting of $(1, 0, 0, \ldots, 0)$ and a zero added to $\vec{\omega}$.

If we want to keep $\mathbf{A}$ square, we can add the additional row to an existing row instead. For the results that follow, this was the method used unless otherwise indicated. It turns out that the non-square (additional row) method is much slower and also less accurate.

Note that these modifications are only needed when solving equation (25) with matrix methods. We do not need the modification when using the FFT method, or in setting up equation (37) for time stepping.

3.4 Comparing Solvers

We have various methods of solving equation (15), such as FFT, Gaussian Elimination, LU decomposition, and iterative solvers. To check that they work correctly we examine how well the resulting ψ satisfies the equation (15). We use second order central differences on ψ to approximate $\nabla^2\psi$ and compare this to ω.

Another consideration is that in solving equation (15) we expect ψ to be symmetric given a symmetric ω.

3.5 An FFT problem

When using the Fast Fourier Transform to solve equation (15) we make use of equation (32). But when both k_x and k_y are zero, the right hand side goes to infinity. To avoid this problem we add a small amount, such as 10^{-4}, to the zero entry in the matrix of k numbers.

It turns out that if this value is too small, on the order of 10^{-14}, then the result is not usable. This can be established using the technique described in section 3.4. My guess is that the resulting component is so large that it swamps the accuracy of the other components.

4 Computational Results

4.1 Results for various initial conditions

The following figures show selected frames from the simulations with various initial conditions. Movies of these are viewable at www.MyPhysicsLab.com. The spatial dimensions in each case are 10×10.

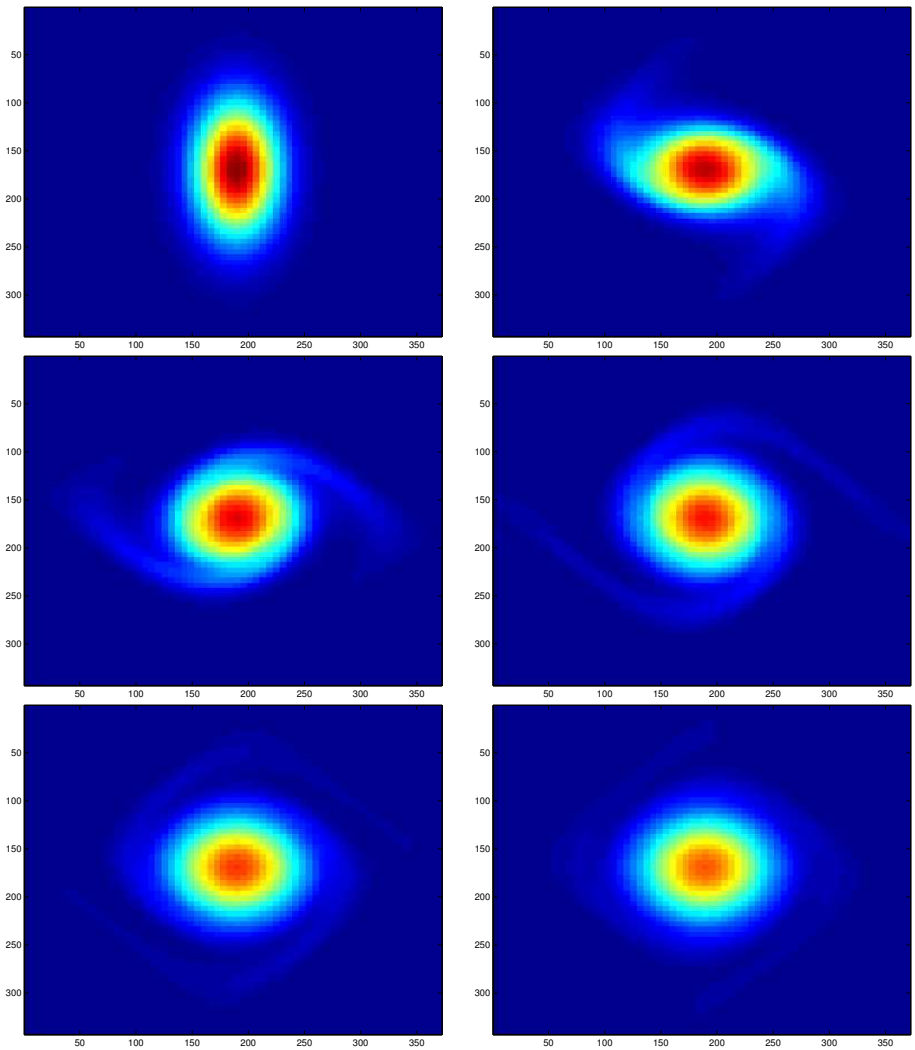

Figure 1: Single vortex at $t = 0, 2.4, 4.8, 7.2, 9.6, 12$ Using LU decomposition and $\nu = 0.01, \Delta t = 0.15, n = 64$. Amplitude ranges from 0 (blue) to 4 (red).

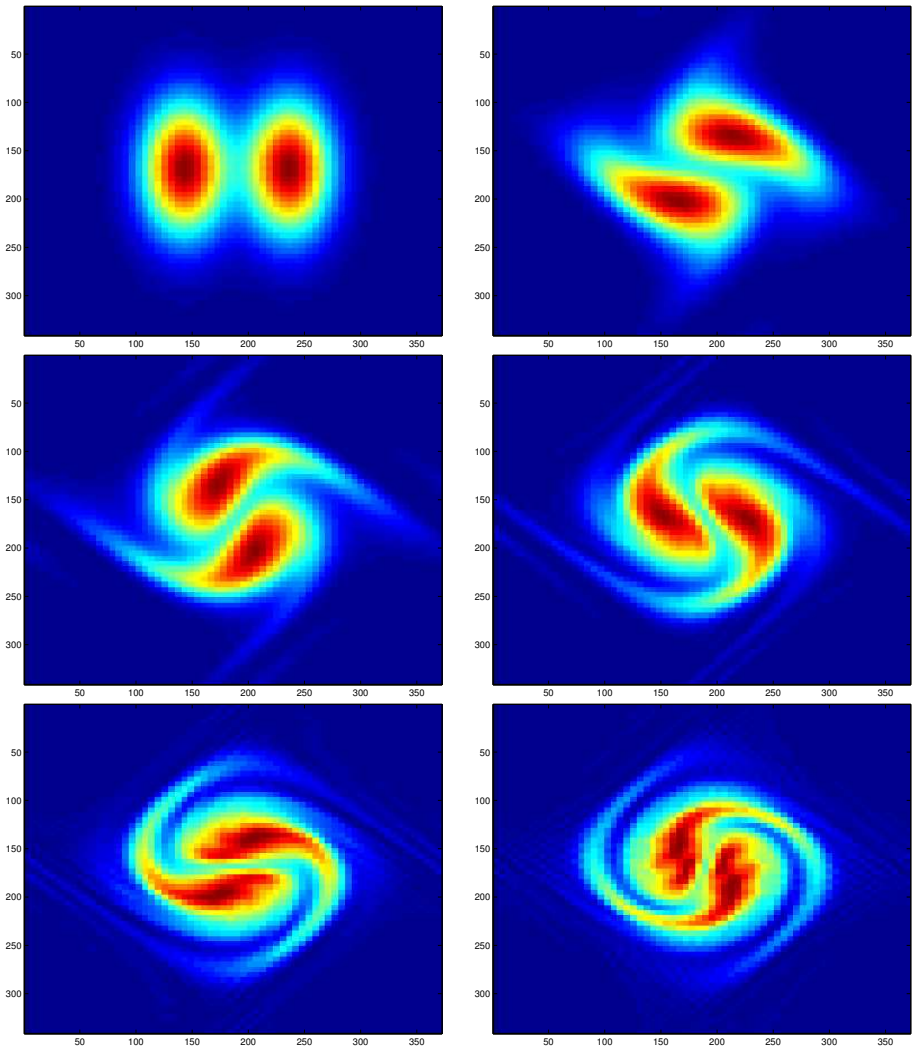

Figure 2: Two positive vortices at $t = 0, 0.9, 1.8, 2.7, 3.6, 4.5$ Using LU decomposition and $\nu = 0.001, \Delta t = 0.10, n = 64$. Amplitude ranges from 0 (blue) to 4 (red).

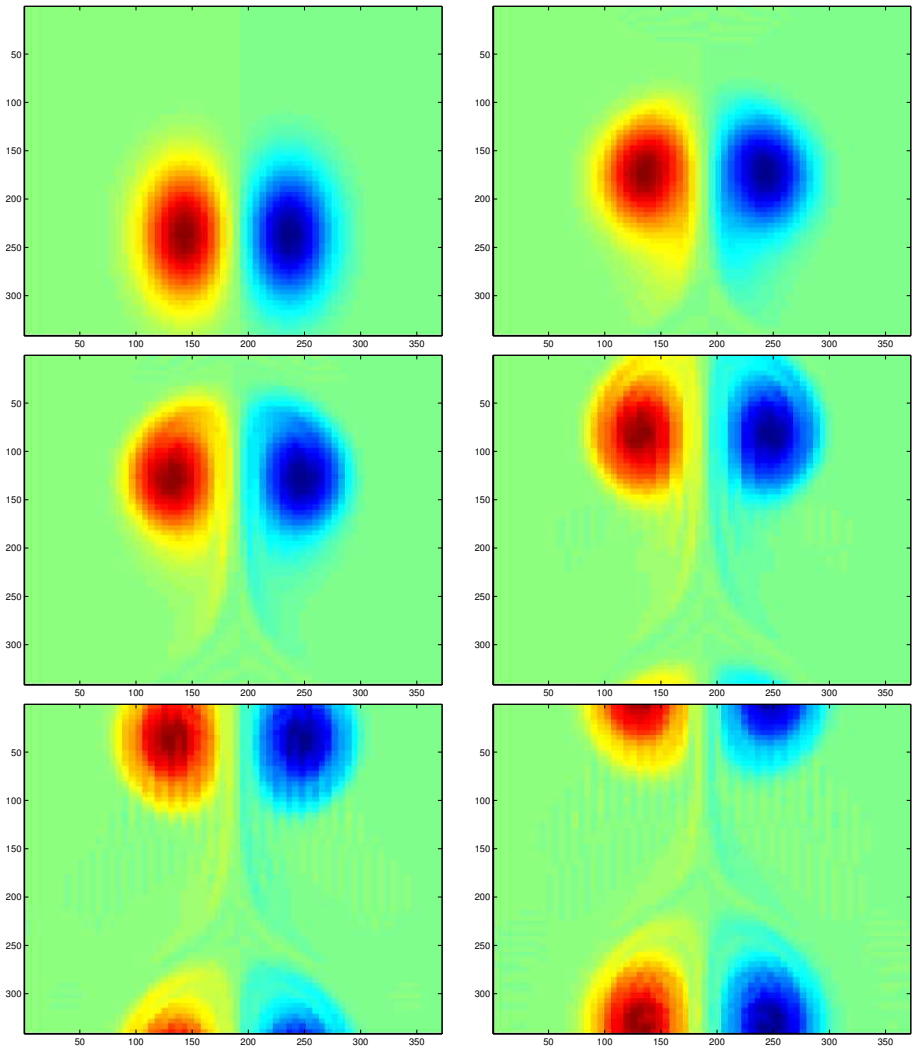

Figure 3: Two opposite vortices at $t = 0, 2.1, 4.2, 6.3, 8.4, 10.5$ Using LU decomposition and $\nu = 0.001, \Delta t = 0.10, n = 64$ Amplitude ranges from -4 (blue) to 4 (red).

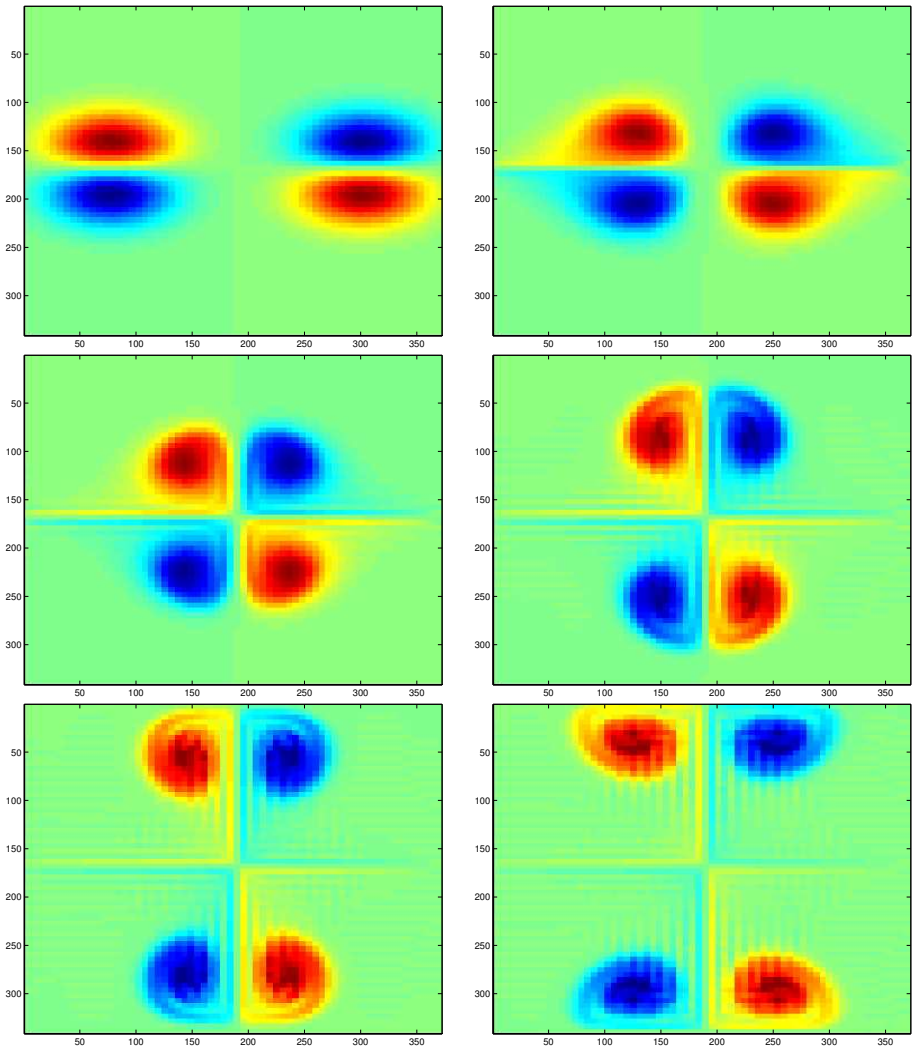

Figure 4: Colliding pairs of vortices at $t = 0, 2.1, 4.2, 6.3, 8.4, 10.5$ Using LU decomposition and $\nu = 0.001, \Delta t = 0.10, n = 64$. Amplitude ranges from -4 (blue) to 4 (red).

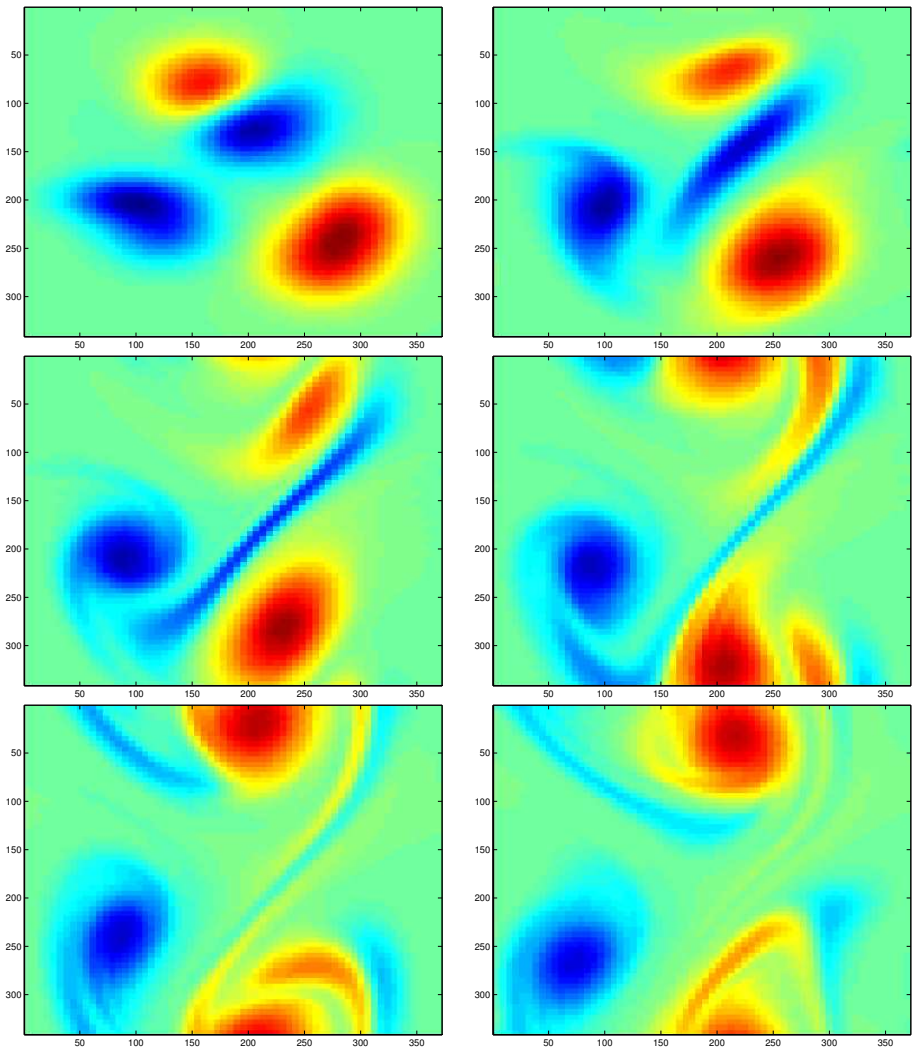

Figure 5: Random vortices at $t = 0, 1.5, 3.0, 4.5, 6.0, 7.5$ Using LU decomposition and $\nu = 0.01, \Delta t = 0.15, n = 64$. Amplitude ranges from -4 (blue) to 4 (red).

4.2 Running times

The table below lists the running time as measured by `cputime` for the various solver methods used. Time step was $\Delta t = 0.1$, diffusion was $\nu = 0.01$, the initial condition was the random vortices. For $N = 64$ we have an average time per frame not counting the first frame. Note that iterative methods are faster once an initial guess has been developed, so the first frame is not as representative. For $N = 128$ we have only the time for the first frame.

Method	Time Per Frame $N = 64$	First Frame Time $N = 128$
Fast Fourier Transform	1.3	7.4
LU Decomposition	2.5	35
BiConjugate Gradients Stabilized	5.1	100
Gaussian Elimination	7.0	98
Generalized Minimum Residual	150	n.a.

The Generalized Minimum Residual method ran out of memory for $N = 128$. It was close to running out of memory for $N = 64$ which caused a lot of hard disk activity and significant slowing.

Providing a starting guess for the iterative solvers `bicgstab` and `gmres` cut the number of iterations and running time by roughly 50%. The table below shows this for $N = 64$. The later frames have a starting guess while the first frame does not.

		Frame 1		Later Frames	
Method	Residual	Iterations	Time	Iterations	Time
bicgstab	10^{-4}	135	12	54	5.1
gmres	10^{-4}	169	212	76	150

4.3 Accuracy of solvers

To check the accuracy of the solvers we use the methods described in section 3.4. The initial condition used was the single vortex. The difference $\nabla^2\psi - \omega$ at the point $\omega_{1,1}$ is shown in the first column of the table below. For the `fft` method the solution $\nabla^2\psi$ seems to differ from ω by a large constant amount. So we also present what the variation would be if we corrected for this constant difference. The second and third columns present the maximum and minimum of the calculation

$$\nabla^2\psi - \omega - (\nabla^2\psi - \omega)_{1,1} \tag{43}$$

Method	Difference at $\omega_{1,1}$	Variation Positive	Variation Negative
Fast Fourier Transform	-0.25	0.0055	-0.17
LU Decomposition	-1.3e-10	3.9e-10	-1.5e-10
BiConjugate Gradients Stabilized	-2.9e-5	2.2e-4	-1.8e-4
Gaussian Elimination	1.3e-11	6.4e-10	-4.8e-10
Generalized Minimum Residual	1.1e-5	1.9e-4	-2.3e-4

We can see that the `fft` method is far less accurate than the other methods. This inaccuracy affects the simulation results as well. With the `fft` method the double vortex spins at a slower rate than with the other methods. You can see this by comparing figure 6 to figure 2.

The accuracy of the iterative methods `bicgstab` and `gmres` are determined by the tolerance setting, which was set to $1e-4$. So better accuracy can be achieved with these methods by increasing the tolerance.

4.4 Symmetry of Solution

Since the elliptic equation(15) is symmetrical in x and y we expect that if ω is symmetrical about the origin, then so should be the resulting ψ. However, this turns out to not be the case when using the square matrix method of making $\mathbf{A}$ non-singular. In that method we add an arbitrary constant to $\mathbf{A}_{1\,1}$. The result is that one of the spikes at the corner of ψ is larger than the others, see figure 7.

If instead we use the non-square method of making $\mathbf{A}$ non-singular, we get a symmetric ψ. But in that case, ψ is generally inaccurate, and also it is much slower for solving.

4.5 Time Resolution Needed

To determine how small the time step needs to be, we can decrease the time step until we see that the solutions are converging. As an example, figure 8 shows the two opposite "charge" vortices at time $t = 20.1$ computed with $\Delta t = 0.1$ and $\Delta t = 0.3$. The vortices are moving upward in the figure, and are moving slower and drifting farther apart with time step $\Delta t = 0.3$.

4.6 Mesh Drift Instability

With low diffusion $\nu < 0.01$ we can see linear artifacts in some of the simulations. For example, figure 3 and figure 4 both show some parallel stripes that look artificial. These are the result of what is called "mesh-drift instability" as described on p. 844 of *Numerical Recipes in C, Second Edition*. It occurs

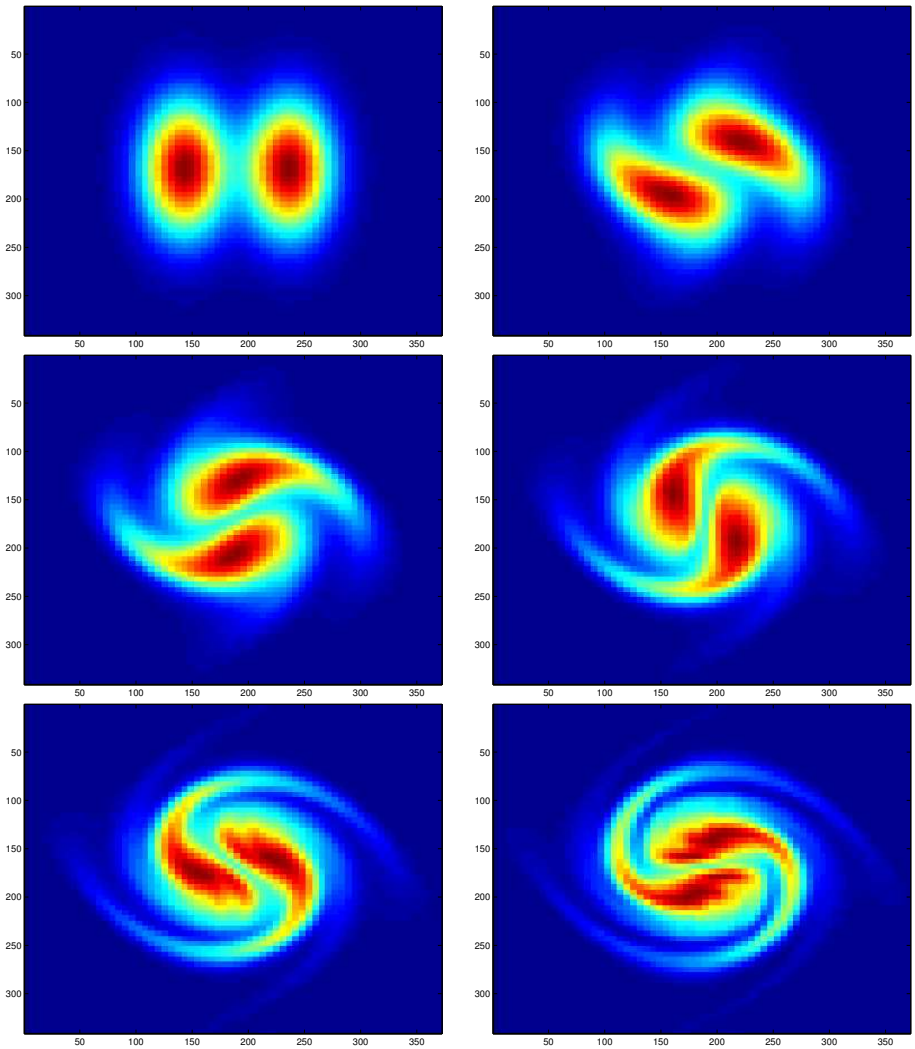

Figure 6: Two positive vortices at $t = 0, 0.9, 1.8, 2.7, 3.6, 4.5$ Using Fast Fourier Transform and $\nu = 0.001, \Delta t = 0.10, n = 64$. Amplitude ranges from 0 (blue) to 4 (red).

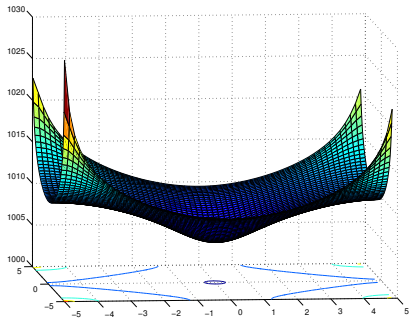

Figure 7: ψ resulting from single symmetric vortex.

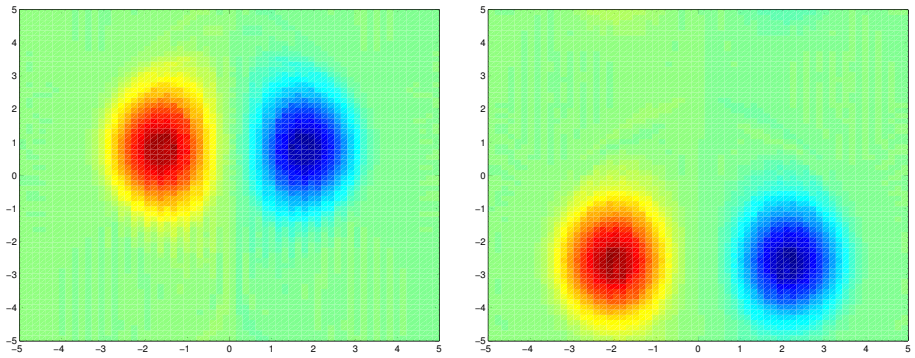

Figure 8: Two positive vortices at $t = 20.1$ calculated with time steps $\Delta t = 0.1$ (left) and $\Delta t = 0.3$ (right).

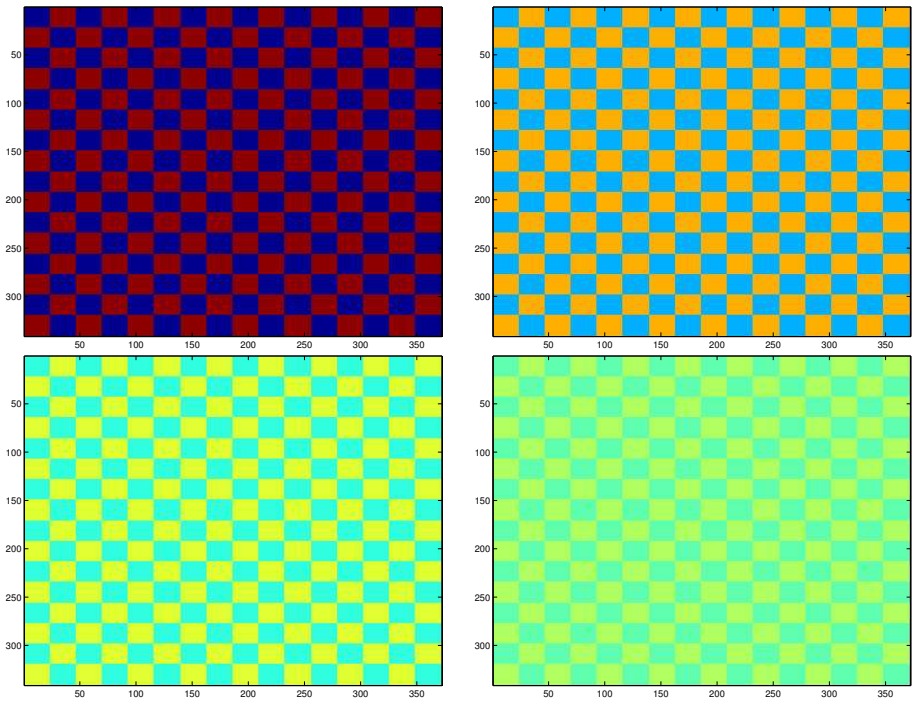

Figure 9: Grid mesh pattern at $t = 0, 4.2, 8.4, 12.6$ Using LU Decomposition and $\nu = 0.01, \Delta t = 0.30, n = 16$. Amplitude ranges from -4 (blue) thru 0 (green) to 4 (red).

because of the central first differences used in estimating the advection term $[\psi, \omega]$ in equation (14). The central first difference in y is

$$\left(\frac{\psi(x, y+ \triangle y) - \psi(x, y- \triangle y)}{2 \triangle y} \right) \tag{44}$$

which depends only on neighboring points. The result is that the grid is that there are effectively two independent simulations. If the grid is colored like a chess board, then the white squares are independent of the black squares.

Figure 9 shows that the grid mesh is stable and only decays due to diffusion. If the diffusion is set low then the decay takes an extremely long time. The solution recommended in *Numerical Recipes* is to add some numerical diffusion. And with enough diffusion, roughly $\nu \geq 0.01$, the artifacts do not appear.

5 Summary and Conclusions

We have used a variety of numerical solving techniques to model shallow fluid behavior. Two-dimensional partial differential equations with time behavior

are computationally challenging, and the efficiency of the technique chosen is crucial. On the computer used for these simulations, a mesh size of $N = 128$ was about the maximum for most methods considered. The exception is the Fast Fourier Transform which performed very quickly and could be used for a much larger mesh than $N = 128$.

Accuracy is another consideration, and here the Fast Fourier Transform did not hold up as well. The accuracy of the FFT in solving the elliptic equation $\nabla^2\psi = \omega$ was significantly worse than the matrix methods, and this showed up in the simulation results.

As a result, LU Decomposition was the preferred technique having the best speed with very good accuracy. The iterative matrix solvers did not prove to be better than LU Decomposition.

There was a compromise made by altering the **A** matrix in order to make it non-singular. This was necessary to be able to use matrix methods in solving the elliptic equation $\nabla^2\psi = \omega$. The resulting ψ was not symmetrical as it should have been, so the results are somewhat suspect. It was not clear if this affected the results.

Numerical artifacts in the form of stable lines appeared with low diffusion ν. This was found to be due to "grid-mesh instability" and necessitates our using a larger value for the diffusion.

To determine an adequate time resolution, we decreased the time step until the solutions converged. This should also be done spatially if one has adequate computer power.

A MATLAB functions used

evhump.m For generating ω initial conditions. Returns a matrix with a Gaussian mound at a given x,y location, amplitude, etc.

evrhs.m The right hand side used in time stepping with the `ode23` solver.

wh.m A small helper function to specify location of initial vortices.

fr.m A tool used for selecting and printing frames from movies.

ev2.m Calculates the time evolution of the vorticity and streamfunction.

B MATLAB code

B.1 evhump.m

```
function m=ev2_hump(n,v,x,y,scalex,scaley,amp)
m = zeros(n,n);
for j=1:n
  m(:,j)=amp.*exp(-((v-y).^2/scaley + (v(j)-x)^2/scalex))';
end
if 0
    figure(1);
    pcolor(v,v,m);
    axis([v(1) v(end) v(1) v(end)]);
end
```

B.2 evrhs.m

```
function rhs=ev2_rhs(xspan,x,dummy,B)
rhs = B*x;
```

B.3 wh.m

```
function r=wh(x)
global L
r = -L/2 + L*x;
```

B.4 fr.m

```
function fr(F,fn,fnam)
%fr(F,1+5*(0:5))  shows every 5th frame
for j=fn
    figure(1)
    [X,Map] = frame2im(F(j));
    image(X)
    drawnow

    if nargin>=3
        framest = num2str(j);
        while(size(framest,2))<3,
          framest = ['0',framest];
          end;
        fname1 = [fnam framest '.pdf'];
```

```
        print('-depsc',fname1);
        fprintf('print %s \n',fname1);
    else
        pause(0.5)
    end
end
```

B.5 ev2.m

```
% ev2.m
% Time evolution of the vorticity and streamfunction
% in a shallow fluid simulation.

clear all; close all; format compact;
use_fft = 1;   % fast fourier transform
use_backslash = 2;  % gaussian elimination
use_lu = 3;    % lu decomposition
use_bicg = 4;   % bi-conjugate stabilized
use_gmres = 5;  % Generalized Minimum Residual method
one_hump = 'one hump';
two_hump = 'two hump same charge';
two_hump_opp = 'two hump opposite charge';
two_pair = 'two pair';
four_square = 'four squares';
stripes = 'stripes';
grid_dots = 'grid dots';
rand_humps = 'random humps';
rand_humps_2 = 'random humps 2';

init_cond = two_hump;  % ******** initial condition
solver = use_lu;  % *******  solver to use
extra_row = 0; % ****** add extra row to matA to pin down psi(1,1)
nu=.01;  % **********  diffusion factor
n=2^6;    % **********  number of points in x & y
tstep=.10; % ************** time step delta
end_time = 12;  % ********** compute up to this time
frame_delta_time = .3;  % ******* timespan between displayed frames
fixed_scale = 1;  % ******** fix the z-scale in display
omega_display = 1; % display omega
psi_display = 0;  % display psi only
psi_2nd_diff = 0; % display the 2nd diff of psi compared to omega
show_cbar = 0;  % whether to display colorbar

clear global L
global L
```

```
if extra_row
    if solver == use_lu
        fprintf('extra row method enabled\n');
    else
        fprintf('extra row only works with lu solver\n');
        break
    end
end
if psi_2nd_diff
    fprintf('displaying 2nd diff of psi\n');
end
N2=n*n;
nx=n;ny=n;
L=10;
Lx=L;Ly=L;
x2=linspace(-Lx/2,Lx/2,nx+1);
y2=linspace(-Ly/2,Ly/2,ny+1);
x=x2(1:nx);
y=x2(1:ny);
dx = x(2) - x(1);  % delta x = width between grid points
dy = y(2) - y(1);

time=0;
frame=1;
last_frame_time = -10;  % when we last displayed a frame
frames=60;  % total number of frames to compute

% store solver names in a cell array
solvers = {'fft','backslash','lu','bicgstab','gmres'};
s = solvers{solver};  % extract string from cell array
fprintf('using solver %s, n=%i\n',s,n);
fprintf('time step = %f,  nu = %f,  spatial width = %f\n',tstep,nu,L);
fprintf('end time = %f,  frame time step=%f\n',end_time,frame_delta_time);
fprintf('initial condition is %s \n',init_cond);

% -----------------------------------------------------
% Generate initial vorticity omega

if strcmp(init_cond, stripes)
    omega = zeros(n,n);
    for j=1:2:n
        omega(:,j)=ones(1,n)';
    end
end

if strcmp(init_cond, grid_dots)
```

```
    omega = [];
    for j=1:n
        omega= [omega; (1:n)+j-1];
    end
    omega = mod(omega,2);
end

if strcmp(init_cond, one_hump)
    omega = ev2_hump(n,x,0,0,1,4,4);
end

if strcmp(init_cond, two_hump_opp)
    omega = ev2_hump(n,x,-L/8,-8*L/40,1,4,4);
    omega = omega - ev2_hump(n,x,+L/8,-8*L/40,1,4,4);
end

if strcmp(init_cond, two_hump)
    omega = ev2_hump(n,x,-L/8,0,1,4,4);
    omega = omega + ev2_hump(n,x,+L/8,0,1,4,4);
end

if strcmp(init_cond, two_pair)
    omega =         ev2_hump(n,x,-30*L/100,+L/16,2,1,4);
    omega = omega - ev2_hump(n,x,-30*L/100,-L/16,2,1,4);
    omega = omega - ev2_hump(n,x,+30*L/100,+L/16,2,1,4);
    omega = omega + ev2_hump(n,x,+30*L/100,-L/16,2,1,4);
end

if strcmp(init_cond, rand_humps)
    omega = zeros(n,n);
    rand('state',0);  % set to initial state
    for j=1:10
        pos_x = L*(-30+rand*60)/100;
        pos_y = L*(-30+rand*60)/100;
        scl_x = (4*rand+1)/2;
        scl_y = (4*rand+1)/2;
        strength = (-0.5 + rand)*8;
        omega = omega + ev2_hump(n,x,pos_x,pos_y,scl_x,scl_y,strength);
    end
end

if strcmp(init_cond, rand_humps_2)
    omega = zeros(n,n);
    omega = omega + ev2_hump(n,x,wh(.2),+wh(.8),2,1,-4);
    omega = omega + ev2_hump(n,x,wh(.2),+wh(.6),2,2,2);
    omega = omega + ev2_hump(n,x,wh(.5),+wh(.6),1,1,4);
```

```
    omega = omega + ev2_hump(n,x,wh(.7),+wh(.7),1.5,2,-5);
    omega = omega + ev2_hump(n,x,wh(.3),+wh(.3),1,3,-3);
    omega = omega + ev2_hump(n,x,wh(.6),+wh(.5),2,.5,3);
    omega = omega + ev2_hump(n,x,wh(.3),+wh(.2),.7,.7,6);
    omega = omega + ev2_hump(n,x,wh(.6),+wh(.2),2,.7,-3);
    omega = omega + ev2_hump(n,x,wh(.8),+wh(.3),1,3,3);
end

if strcmp(init_cond, four_square)
    omega = zeros(n,n);
    for ii=1:nx
        for j=1:ny
            if max(abs([x(ii) y(j)]))<2
                omega(ii,j)=sign(x(ii)*y(j))*2;
            end
        end
    end
end

z_range = [min(min(omega)) max(max(omega))];

% ------------------------------------------------------
% set up matrix A
% corresponds to the viscosity nu*del^2(omega)
% This matrix does not change, so compute only once.

e0=zeros(N2,1);
e1=ones(N2,1);

e2=e1;
e4=e0;
for j=1:n
    e2(n*j)=0;  % e2 is ones, but zero every n-th
    e4(n*j)=1;  % e4 is zeros, but 1 every n-th
end

% Note on spdiags(B,d,m,n)  If a column of B is longer than
% the diagonal it's replacing, spdiags takes elements of
% super-diagonals from the lower part of the column of B,
% and elements of sub-diagonals from the upper part
% of the column of B.

% e3 is a super-diagonal (at +1 above main diagonal)
% so need to shift
e3(2:N2,1)=e2(1:N2-1,1);  % e3 is e2 shifted by one
e3(1,1)=e2(N2,1);
```

```
%e5 is a sub-diagonal (at -4 below main diagonal)
e5(2:N2,1)=e4(1:N2-1,1);  % e5 is e4 shifted by one
e5(1,1)=e4(N2,1);

matA=spdiags(...
   [e1      e1  e5  e2  -4*e1 e3  e4   e1   e1],...
   [-(N2-n) -n  -n+1 -1    0   1  n-1  n   (N2-n)],N2,N2);
clear e1 e2 e3 e4 e5

% Modify matA so that it is non-singular.
% Without this, the condition number of matA is huge
% and matrix solvers fail.
% The cludge is only used for matrix solvers... so add
% or subtract depending on when using it.
cludge = 1;

% divide by dx^2
matA = matA/(dx*dx);

if solver==use_lu
    if extra_row
        matA(N2+1,1) = 1;   % add a row to pin value of psi(1,1)
    else
        matA(1,1) = matA(1,1) + cludge;
    end
    [matL,matU] = lu(matA);
    if extra_row
        matA(N2+1,:) = [];  % delete row we just added
    else
        matA(1,1) = matA(1,1) - cludge;
    end
end

% initial guess needed for iterative solvers (bicgstab, gmres)
psi = zeros(n,n);

%spy(matA)

% -----------------------------------------------------
% set up wave number matrix for fft
if solver==use_fft
    % Why rearrange the K values this way?
    % answer has to do with how fft algorithm works (see fftshift)
    Kx=[0:(nx/2-1) (-nx/2):-1]';   % K values
    kx=2*pi*Kx/Lx;                 % rescale to a 2*pi domain
    Ky=[0:(ny/2-1) (-ny/2):-1];   % K values
```

```
    ky=2*pi*Ky/Ly;                % rescale to a 2*pi domain
    k1 = zeros(nx,ny);
    kx = kx.^2;
    ky = ky.^2;
    for i=1:ny
        k1(:,i) = k1(:,i) + kx;
        k1(i,:) = k1(i,:) + ky;
    end
    %surfc(x,y,k1)
    % k1(1,1)=1e-14 is interesting value... seems that this component
    % gets so large that we lose accuracy of the calculation!
    k1(1,1) = 1e-6;  % cludge to prevent divide by zero
    k1 = -k1;
end


% ------------------------------------------------------
% Set up row & column info for matrix B.
% py & px refer to x & y first differences in psi
% which are calculated during each loop (later).

% Place py into the B matrix (see derivation).
% py(k,j) goes into row k,j, column k+1,j
% for n=16 the column pattern becomes
% 5 6 7 8   9 10 11 12   13 14 15 16   1 2 3 4
rows = [1:N2];
cols = [n+1:N2 1:n];

% -py(k,j) goes into row k,j, column k-1,j
% for n=16 the column pattern becomes
% 13 14 15 16   1 2 3 4   5 6 7 8   9 10 11 12
rows = [rows 1:N2];
cols = [cols N2-n+1:N2 1:N2-n];

% -px(k,j) goes into row k,j, column k,j+1
% for n=16 the column pattern becomes
% 2 3 4 1   6 7 8 5   10 11 12 9   14 15 16 13
c = [1:N2];
c = 1 + c - n*(0==mod(c,n));
rows = [rows 1:N2];
cols = [cols c];

% px(k,j) goes into row k,j, column k,j-1
% for n=16 the column pattern becomes
% 4 1 2 3   8 5 6 7   12 9 10 11   16 13 14 15
c = [1:N2];
```

```
c = -1 + c + n*(1==mod(c,n));
rows = [rows 1:N2];
cols = [cols c];
clear c

% To calculate position: size & position a figure window, then:
%     h = figure(1)
%     get(h,'Position')
figure('Position',[622   534   478   417]);

start_time = cputime;

% ======================================================
while 1
    % ------------------------------------------------------
    % plot the results from this step of psi and omega
    if omega_display & ((time + 0.01 > end_time) | ...
            (time - last_frame_time + 0.00001)>= frame_delta_time)
        %figure(1)
        % pcolor doesn't show last row or column, so add dummies.
        o2 = [omega; zeros(1,n)];
        o2 = [o2, zeros(n+1,1)];
        pcolor(x2,y2,o2)
        if fixed_scale
            caxis(z_range)
        end
        if show_cbar
            colorbar('horiz')
        end
        shading flat
        %axis equal
        %shading interp,axis off
        %surfc(x,y,omega)
        %axis([x(1) x(end) y(1) y(end) z_range])
        drawnow
        fprintf('frame %i time %f\n',frame, time);
        F(frame) = getframe;
        frame=frame+1;
        last_frame_time = time;
    end

    if time>=end_time
        fprintf('end time reached\n');
        break
    end
```

```
% ------------------------------------------------------
% Given vorticity omega, Solve for streamfunction psi.

if solver~=use_fft
    if ~extra_row
        matA(1,1) = matA(1,1) + cludge;
    end
end

if solver==use_fft
    % Take FFT of omega in x, then in y.
    % note: fft works along columns, so transpose afterwards to
    % work on other dimension, and transpose back at end.
    ft = fft(fft(omega).').';
    % Then divide by (kx^2 + ky^2)  where kx, ky are x & y values
    % on the grid (in frequency space).
    ft = ft./k1;
    % Then take iFFT in x, then in y, to get psi.
    psi = ifft(ifft(ft).').';
    psi = real(psi);
elseif solver==use_backslash
    % solve using backslash
    psi = matA\reshape(omega,N2,1);
elseif solver==use_lu
    if extra_row
        y1 = matL\([reshape(omega,N2,1); 0]);
    else
        y1 = matL\(reshape(omega,N2,1));
    end
    psi = matU\y1;
elseif solver==use_bicg
    % pass current value of psi as estimate for next iteration
    psi = bicgstab(matA,reshape(omega,N2,1),1e-4,1000,[],[],...
        reshape(psi,N2,1));
elseif solver==use_gmres
    % pass current value of psi as estimate for next iteration
    psi = gmres(matA,reshape(omega,N2,1),[],1e-4,1000,[],[],...
        reshape(psi,N2,1));
end

if solver~=use_fft
    if ~extra_row
        matA(1,1) = matA(1,1) - cludge;
    end
    psi = reshape(psi,n,n);
end
```

```
% ------------------------------------------------------
% Check whether del^2(psi) = omega
% Use central difference formula for 2nd derivative.
if psi_2nd_diff
    % psi_xx + psi_yy = omega
    % So compute numerical 2nd deriv in x & y directions and add.
    % Result should match omega.
    save_n = n;
    n = size(psi,1);
    % use central difference for 2nd derivatives
    df = psi(1:n-2,2:n-1)-2.*psi(2:n-1,2:n-1)+psi(3:n,2:n-1);
    df = df + psi(2:n-1,1:n-2)-2.*psi(2:n-1,2:n-1)+psi(2:n-1,3:n);
    df = df/(dx*dx);
    n = save_n;
    df = df - omega(2:n-1,2:n-1);
    fprintf('diff at omega(1,1)=%e\n',df(1,1));
    df = df - df(1,1);
    %figure(1);
    % pcolor seems to not show the last row & column!
    df2 = [df; df(1,:)];
    df2 = [df2, df2(:,1)];
    pcolor(df2)
    colorbar
    shading flat
    drawnow
    max(max(df))
    min(min(df))
    %clear save_n df psi2
    fprintf('break\n');
    break
end


% ------------------------------------------------------
% set up matrix B
% corresponds to the curl product of psi <cross> omega
% strategy:
%  1. create a B matrix from loops... big & slow but correct
%  2. vectorize, compare to (1) to ensure correctness

% Find first y-derivative of psi
% For psi(x,y) it is (psi(x,y+dy)-psi(x,y-dy))/(2*dy)
py = [psi(2:n,:); psi(1,:)] - [psi(n,:); psi(1:n-1,:)];
py = reshape(py,1,N2);
% Find first x-derivative of psi
```

```
    % For psi(x,y) it is (psi(x+dx,y)-psi(x-dx,y))/(2*dx)
    px = [psi(:,2:n) psi(:,1)] - [psi(:,n) psi(:,1:n-1)];
    px = reshape(px,1,N2);


    % set up data for matrix B and divide by 4*dx*dy
    data = [py -py -px px]/(4*dx*dy);

    % create a n^2 by n^2 sparse matrix
    % with room for 8*n^2 non-zero elements
    matB = sparse(rows,cols,data,N2,N2,8*N2);

    % add in matrix A (note: don't need the cludge here!)
    matB = nu*matA + matB;

    % ------------------------------------------------------
    % run the ode solver forward one time step
    omega = reshape(omega,N2,1);
    tspan = [time,time+tstep];
    [t,omega] = ode23('ev2_rhs',tspan,omega,[],matB);
    time=t(end);
    omega=reshape(omega(end,:),n,n);

    % ------------------------------------------------------
    % plot psi
    if psi_display
        surfc(x,y,psi)
        %axis([x(1) x(end) y(1) y(end) -0.1 0.9])
        drawnow
    end

    % ------------------------------------------------------
    % big loop back here

    %fprintf('cputime = %f\n',cputime - start_time);
    %start_time = cputime;
end

fprintf('cputime = %f\n',cputime - start_time);
if 0 & omega_display & (frames > 1)
    movie(F,5)
end
```

C Calculations

Here we detail how adding the two equations (7) and (8) leads to equation (10).

$$\omega_t + u\omega_x + v\omega_y = 0 \tag{10}$$

The y-derivative of equation (7) and the x-derivative of equation (8) are the following.

$$u_{ty} + 2(u_y u_x + u u_{xy}) + (uv)_{yy} = f v_y \tag{45}$$
$$v_{tx} + 2(v_x v_y + v v_{xy}) + (uv)_{xx} = -f u_x \tag{46}$$

Expand some derivatives

$$u_{yt} + 2u_y u_x + 2u u_{xy} + u v_{yy} + 2u_y v_y + u_{yy} v = f v_y \tag{47}$$
$$v_{xt} + 2v_x v_y + 2v v_{xy} + u v_{xx} + 2u_x v_x + u_{xx} v = -f u_x \tag{48}$$

Subtracting (47) from (48) and rearranging gives us

$$\begin{aligned}(v_{xt} - u_{yt}) + (u v_{xx} - u u_{xy}) + (v v_{xy} - v u_{yy}) \\ + 2(u_x v_x + v_x v_y - u_x u_y - u_y v_y) - (u u_{xy} + u v_{yy}) + (v v_{xy} + v u_{xx})\end{aligned} \tag{49}$$

Some more rearranging gives

$$\begin{aligned}(v_x - u_y)_t + u(v_x - u_y)_x + v(v_x - u_y)_y \\ + 2v_x(u_x + v_y) - 2u_y(u_x + v_y) - u(u_x + v_y)_y + v(u_x + v_y)_x\end{aligned} \tag{50}$$

Recall equations (5) and (9),

$$u_x + v_y = 0 \tag{5}$$
$$\omega = v_x - u_y. \tag{9}$$

which applied to equation (50) leads to the final form of equation (10).