Numerical Methods for Solving the Vorticity-Stream function Equations
ME535 Final Project
Frank Bremer
June 5, 2008

Abstract
A commonly used atmospheric model involves the vorticity and stream functions. The vorticity-stream function relations take the form of partial differential equations, with spatial as well as time based derivatives. A finite difference discretization can be used in combination with a Runge-Kutta time stepping scheme to solve such a problem. A basic study was performed to gauge the speed of several methods for solving the stream function equation. Several initial vorticity conditions were also investigated. This paper reports the results of the convergence speed study as well as the effect of varying initial conditions.
Introduction & Overview
Many physical problems in science and engineering take a mathematical form that is not easy to solve analytically. For these types of problems the numerical solution technique is most convenient. The current standard for computing power, coupled with programming languages such as Matlab make solving very complex differential equations relatively fast and easy using numerical techniques. This project involved solving the vorticity-stream function equations of atmospheric sciences via numerical methods. Obtaining a solution for the vorticity-stream function equations using a finite difference discretization requires an Elliptic Solve for the stream function. This elliptic solve was accomplished using several numerical methods, and the time to converge for each method was compared to reveal the most efficient solution technique. The numerical methods investigated were:

- Fast Fourier Transform (FFT)
- Gaussian Elimination with Partial Pivoting
- LU Decomposition
- Biconjugate gradients stabilized method
- Generalized minimum residual method

In addition, several initial conditions were used and the effect on the solution was observed.

Theoretical Background
Vorticity-Stream function Relations
The motion of the atmosphere is often modeled with an advection diffusion equation such as given in the vorticity-stream function relations. The time evolution of the vorticity is given by:

\[ \omega_t + [\psi, \omega] = \nabla^2 \omega \]  

(1)

where \( \omega_t \) is the time derivative of the vorticity, \( \nabla^2 \) is the two dimensional laplacian, and

\[ [\psi, \omega] = \frac{\partial \psi}{\partial x} \frac{\partial \omega}{\partial y} - \frac{\partial \psi}{\partial y} \frac{\partial \omega}{\partial x} \]  

(2)

The vorticity is also coupled to the stream function as governed by the following relation:

\[ \nabla^2 \psi = \omega \]  

(3)

Finite Difference Discretization
Finite difference discretization schemes are often used as a relatively simple means for calculating partial derivatives. These schemes rely on linear combinations of Taylor series expansions of the differentials. By various additions and subtractions of Taylor series expansions a set of direct difference formulas can be derived. The second order
center difference scheme is given in equation 4. The reference to second order accuracy refers to the error accumulated as a result of the approximation. A second order accurate scheme implies that for a discretization size, $\Delta t$, the solution yielded by the scheme will be accurate to $\Delta t^2$. The same logic applies to third, fourth, fifth…order accurate routines.

$$ f'(t) = \frac{[f(t + \Delta t) - f(t - \Delta t)]}{2\Delta t} $$

$$ f''(t) = \frac{[f(t + \Delta t) - 2f(t) + f(t - \Delta t)]}{\Delta t^2} $$ (4)

**Runge Kutta Time Stepping Schemes**

Many numerical solution techniques for ODEs of the form of equation 1 center on the realization that a derivative is nothing more than a slope of a function at a given point and on a very fine scale the slope can be approximated as a simple rise over run of adjacent points. Simple slope formulas can then be derived to approximate derivatives and a Taylor series expansion can be performed on those formulas to improve their accuracy. The most popular numerical schemes for approximating derivatives in this manner are the Runge Kutta methods. Runge Kutta methods are stepping schemes that allow you to predict the solution at a future point based on the solution at the current point and form the basis for numerical differential equation solvers. The most commonly used Runge Kutta method is the 4th order Runge Kutta and is given in equation 5.

$$ k_1 = hf(x_n, y_n) $$ (5a)

$$ k_2 = hf\left(x_n + \frac{h}{2}, y_n + \frac{k_1}{2}\right) $$ (5b)

$$ k_3 = hf\left(x_n + \frac{h}{2}, y_n + \frac{k_2}{2}\right) $$ (5c)

$$ k_4 = hf(x_n + h, y_n + k_3) $$ (5d)

$$ y_{n+1} = y_n + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4) $$ (5e)

**Fast Fourier Transform**

The Fast Fourier Transform (FFT) is a particular Fast Poisson commonly used for solving numerical problems not based on a Taylor series discretization. The FFT converts time based problems into the frequency domain in order to easily solve the root problem before transforming the solution back to the time domain. The conversions to and from the frequency domain are given by the following equations:

$$ F(k) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} e^{-ikx} f(x) dx $$ (6a)

$$ f(x) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} e^{-ikx} F(k) dk $$ (6b)

A large part of the allure of the FFT method is the relative ease of computing derivatives in the frequency domain. The FFT of the derivative of a function is simply a scalar multiplication of the FFT of the original function, as show in equation 7. This property allows for very fast computation of function derivatives.
\[ F^{(n)} = (-ik)^{(n)} F \]  

(7)

Where \( F \) is the function in the frequency domain and \( n \) is the degree of differentiation. The Fast Fourier Transform is credited with an operation count of \( O(n \log(n)) \), which proves to be much less than most other solution techniques. One caveat for using the FFT method is that due to the nature of the sine and cosine transformations, it requires periodic boundary conditions. The FFT method is invalid if periodic boundary conditions do not exist.

**Gaussian Elimination**

Gaussian Elimination is a method for solving the matrix relation \( Ax=b \) for \( x \). In solving using Gaussian Elimination, the \( A \) matrix is reduced to an upper triangular matrix, allowing back substitution to solve for the elements of vector \( x \). As long as the matrix \( A \) is non-singular, Gaussian Elimination will always yield a result. The operational count required for this method is \( O(n^3) \), making it a very reliable, yet very slow routine.

**LU Decomposition**

In the Gaussian Elimination method, the first step is to reduce the \( A \) matrix into an upper triangular matrix. This reduction requires \( O(n) \) operations each time the solution is found. The back substitution for solving \( x \) once the \( A \) matrix is reduced to triangular matrix requires \( O(n^2) \) operations each time the solution is found. If the same matrix \( A \) is to be used repetitively to solve for \( x \), it would then make sense to decompose the matrix \( A \) such that the repetitive solution would only need to involve triangular matrices. This would eliminate the first \( O(n) \) operations associated with repetitively reducing the matrix to a triangular matrix.

The premise behind LU decomposition is to decompose the \( A \) matrix into lower (L) and upper (U) triangular matrices, such that \( A=LU \). Decomposing the matrix \( A \) into two triangular matrices requires \( O(n^3) \) operations, once. Once the triangular matrices are found, they can then be used in the repetitive solution method in place of \( A \) in order to allow for a solution technique which requires \( O(n^2) \) operations. Thus, the LU decomposition method requires \( O(n^2) \) operations + \( O(n^3) \) operations only once, providing a potentially large time savings over the pure Gaussian Elimination method.

**Iterative Solution Methods**

The solution for \( Ax=b \) can also be found using iterative techniques, often referred to as the Krylov Space Methods. As with most iterative techniques, the Krylov Space Methods employ an initial guess at the solution and run through some iteration scheme until the convergence criteria is met. In general, the structure of a Krylov Space method is as follows:

- Guess the initial values
- Iterate using the particular iteration scheme for the technique
- Check for convergence
Two particular Krylov Space Methods are the biconjugate gradients stabilized method and the generalized minimum residual method.

**Algorithm Implementation**

All solution methods herein require the computational domain to be discretized. The FFT method, in particular, requires the number of discrete steps to be a power of 2. For the purposes of comparing solution time, the number of discrete points in both the x and y direction was chosen to be 64. The computational domain of [-10, 10] was identified as sufficiently large and used for all methods. The time span used for all methods was [0, 50].

Once the computational domain was identified, and the number of discrete points determined, the two-dimensional mesh was created using the MESHGRID command in Matlab. This generated two matrices, X and Y, containing the x and y coordinates for each mesh location.

The initial vorticity was assumed to be an elliptical Gaussian distribution with a ratio of 10:1 between the width in the y and x directions, respectively. The amplitude of the Gaussian was chosen to be unity. Specifically, the initial condition was given as:

\[ \omega_o = e^{(-x^2 \cdot y^2 / 10)} \]

The value for diffusion was assumed to be constant at .001 and the boundary conditions were assumed to be periodic for all solution methods.

The individual methods discussed in section 2 were used to solve equation 3 for the stream function. Once the stream function was found, all solution methods used a 4th order Runge-Kutta time stepping routine (ode45) to solve equation 1. All solution methods (except for the FFT method) used the second order center difference scheme described by equation 4 to generate the matrices necessary to calculate derivatives in the finite difference discretization. It was necessary to find 3 separate matrices; the A matrix which calculates the two-dimensional Laplacian, the B matrix which calculates one derivative with respect to x, and the matrix C which calculates one derivative with respect to y. The A matrix is required to back solve for the stream function in equation 3 as well as to calculate the last term in equation 1. The B and C matrices are required to calculate the partial differentials in equation 2. The structure of the A, B, and C matrices are given in Figure 1. The diagonals of the sparse A, B, and C matrices are 1 and -1, depending on the particular diagonal. See the functions dim2_grad.m, dx_mat.m, and dy_mat.m in Appendix B for details. Once the matrices were obtained, they were divided by either \( 2\Delta x \) or \( \Delta x^2 \) to account for the denominators in equation 4. The center difference schemes for calculating the A, B, and C matrices require all points in the domain to reference the two neighboring points. This poses a problem for the first and last points in the domain. However, since the boundary conditions were periodic, the first point references the second and the last points, while the last point references the second to last and the first points.
The symmetric, sparse nature of matrix A makes it a singular matrix. This is a result of the stream function not being unique. Due to the partial differentials applied to the stream function within the vorticity-stream function relations, a constant could be added to the stream function and it would still be an equally viable solution (since the constant disappears during differentiation). This poses a problem for solving equation 3, since a singular matrix cannot be inverted. In order to overcome this numerical difficulty, the value for $A(1,1)$ was arbitrarily changed to 1. The A matrix was then non-singular and able to be inverted.

The specific implementation steps required for each solution method are given in the following sections.

**FFT**

The Fast Fourier Transform in Matlab takes a function and applies equation 6a to transform into the frequency domain. In order to calculate derivatives, the k values, or wave numbers, needed to be calculated. The function of the k values is to scale and shift the computational domain to be a periodic $2\pi$ domain. Once the k values are calculated for both the x and y directions, the Meshgrid command is used to assign a k value to each element of the mesh.

Once the initial vorticity is established, it is easily transformed into the frequency domain by the function call $\text{fft}(w)$, where w is the vorticity. Now that the vorticity is in the frequency domain and the k values are calculated, equation 3 can be solved for the stream function as follows

$$\psi = \frac{\partial \phi}{k_x^2 + k_y^2}$$

where $\psi$ denotes the two-dimensional FFT of $\psi$. 
It is easily seen from equation 8, that numerical difficulties can be experienced if both $k_x$ and $k_y$ are zero. This is analogous to the A matrix singularity issue previously mentioned. Much the same as the A matrix singularity solution, the way to avoid numerical difficulties is to arbitrarily reassign the first element of the k vectors to a non-zero value. $10^{-6}$ was selected as a small, but non-zero value to avoid this problem.

Now that both the vorticity and stream function are known for a particular time, they can be plugged into the transform of equation 1, along with the k values to form an ODE. This ODE can then be easily solved using the previously mentioned 4th order Runge Kutta time stepping routine. Since the stream function is dependent on the instantaneous value of the vorticity, as given in equation 3, it too needs to be updated at every time step during the time stepping routine. For this reason the calculation of the stream function takes place within the .m file containing the ode (see wrhs.m in Appendix B).

Since the initial vorticity, and hence the FFT of the vorticity, is given in matrix form, and the ode45 solver requires a vector, the reshape command must be used. Once the transform of the initial vorticity is calculated (and stored as an n x n matrix) it must be reshaped to a vector (of length $n^2$). Within the ODE file (wrhs.m) the vorticity can be reshaped back into a matrix of size n x n and used to calculate the stream function according to equation 8. Once the stream function and the vorticity are known for the given time step, they can be substituted into the Fourier transform of equation 1, with the derivatives calculated as given by equation 7. This results in a value for the time derivative of the vorticity in the frequency domain in matrix form. This matrix then has to be reshaped back into a vector (of length $n^2$) due to the I/O requirements of the ode45 solver. The ode45 solver will then step through the given time span, repeating the calculations at each time step. The result is a matrix of vorticity values in the frequency domain at each time step specified. These values can then be inverse Fourier transformed and plotted to show the time evolution of the vorticity function.

**Gaussian Elimination (A\b)**

The matrix backslash operator (\) in Matlab performs the Gaussian Elimination discussed in the Theoretical Background section. As was discussed in the FFT method, since the stream function is dependent on the instantaneous value of the vorticity, as given in equation 3, it too needs to be updated at every time step during the time stepping routine. As was the case with the FFT method, as well as all other methods, the calculation of the stream function value needs to occur in the ODE file that is passed into the ode45 function call. The ODE function call also needs to include the A, B, and C matrices as well as the value for the diffusion. The ODE file for this solution method was main_rhs.m and can be found in Appendix B. The particular equation used to solve the stream function using Gaussian Elimination is as follows:

$$\psi = A \backslash \omega$$  \hspace{1cm} (9)

Since all of the values required to solve equation 1 (with the exception of the stream function) are passed into the ODE file, once the stream function is calculated using the backslash command the time derivative of the vorticity is easily solved using matrix
multiplication. The ode45 solver will then step through the given time span, repeating the calculations at each time step. The result is a matrix of vorticity values at each time step specified. These values can plotted to show the time evolution of the vorticity function.

**LU Decomposition**

The implementation of the LU decomposition method is very straightforward. The procedure is exactly the same as for the Gaussian Elimination method except that instead of using the A matrix to solve for the stream function, the L and U matrices are used. In particular, equation 9 becomes

\[
y = L \backslash \omega \\
\psi = U \backslash y
\]

(10)

Where y is simply an intermediate placeholder. Equation 10 performs a Gaussian Elimination twice, but the matrices that are being used are triangular so that only 2 O(n²) operations are required rather than 1 O(n³) operation.

**Iterative Methods**

The implementation of the iterative methods is also very straightforward. The two iterative methods investigated were the biconjugate gradients stabilized method and the generalized minimum residual method. Matlab contains implicit functions for both of these methods. For the iterative methods, equations 9 and 10 were replaced with the appropriate function calls; namely bicgstab for the biconjugate gradients stabilized method and gmres for the generalized minimum residual method. The form of the function calls are as follows:

\[
\psi = \text{bicgstab}(A, \omega) \\
\psi = \text{gmres}(A, \omega)
\]

(11)

Some optional inputs were also utilized to increase the speed of the solution method. The two functions used also allow a user specified tolerance, a maximum number of iterations, and an initial guess. These values were added to the function calls given in equation 11.

The tolerance for the iteration schemes was chosen to be the square of the spatial domain step size. The fact that a second order center difference scheme was used to discretize the domain means that a tolerance greater than the square of the spatial step size would be unnecessary.

The maximum number of iterations was selected to be 200 to ensure convergence.

The ODE function call also contained the initial value for the stream function, which was then used as the initial guess for the iteration scheme. The initial value for the stream function was calculated using Gaussian elimination (once). Since the stream function doesn't change very drastically over one time step, the previous time's value of the
stream function was used as an initial guess for the iteration scheme to speed up convergence. See main_rhs_BGS and main_rhs_GMR for the details.

**Alternate Initial Conditions**
In addition to the solution time study, the following initial conditions were investigated:

- Two oppositely ‘charged’ Gaussian vortices next to each other
- Two same ‘charged’ Gaussian vortices next to each other
- Two pairs of oppositely charged Gaussian vortices next to each other such that they attempt to collide
- A random assortment of vortices

The initial conditions were accomplished by modifying equation 7.5 as follows:

Two oppositely charged:

\[ \omega_o = e^{-(X-3)^2 - Y^2/10} - e^{-(X+3)^2 - Y^2/10} \] (12)

Two same charged:

\[ \omega_o = e^{-(X-3)^2 - Y^2/10} + e^{-(X+3)^2 - Y^2/10} \] (13)

Two oppositely charged pairs:

\[ \omega_o = e^{-(X+3)^2 -(Y-4)^2/10} - e^{-(X-3)^2 -(Y-4)^2/10} + e^{-(X-3)^2 -(Y+4)^2/10} - e^{-(X+3)^2 -(Y+4)^2/10} \] (14)

A similar modification was made to equation 7.5 to accommodate the random assortment of vortices, however the equation is much too long to list in this section. Refer to Appendix B for the details. For the random assortment run, the domain was increased to [-20, 20] and the number of modes was increased from 64 to 128. These changes had to be made in order to fit all of the initial lumps in the computational domain while retaining a fine resolution.

**Computational Results**

**Speed Investigation**

The vorticity-stream function relations were solved using the previously mentioned methods and the computational durations were compared to determine the fastest solution technique. The FFT method solved the problem in relatively short order, with a total run time of roughly 10 seconds. The slowest solution technique was the generalized minimum residual method, with a total run time of roughly 256 seconds. A breakdown of the run time for each method is given in Table 1. As was expected, the LU decomposition method provided a slight benefit over the Gaussian Elimination, however it was no where near as fast as the FFT method. It is a bit surprising that the biconjugate gradients stabilized method proved to be faster than the Gaussian Elimination method, and very near the LU decomposition method, since it is an iterative scheme and requires multiple iterations per time step.
The time evolution of the vorticity as solved by the FFT method is shown in Figure 2. The elliptical nature of the initial vorticity yields a spiraling pattern as the solution moves through time.

The secondary benefit of the FFT method over the others considered is the increased accuracy (spectral vs. second order). This is apparent in Figure 3. The figure shows the solution after 25 time units for each solution method. The plot for the FFT method is very clear and high resolution, whereas the others are relatively blurry. As a side note, it is also easy to see that the accuracy of each of the other methods used is very comparable. It is not easy to see any difference in the resolution of the plots for the other methods used.

Table 1. Comparison of solution times, average number of iterations, and average residual of each method (where applicable).

<table>
<thead>
<tr>
<th>Method</th>
<th>Solution Time (s)</th>
<th>Approx # of iterations</th>
<th>Approx Residual</th>
</tr>
</thead>
<tbody>
<tr>
<td>FFT</td>
<td>10.02</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>Gaussian Elimination</td>
<td>136.32</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>LU Decomposition</td>
<td>109.68</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>Bicgstab</td>
<td>111.28</td>
<td>11</td>
<td>0.09</td>
</tr>
<tr>
<td>Gmres</td>
<td>255.76</td>
<td>15</td>
<td>0.09</td>
</tr>
</tbody>
</table>

Figure 2. Time evolution of the vorticity from time unit 4 to 36, as solved by the FFT method.
Figure 3. Vorticity-Stream Function solutions 25 time units into the simulation.

**Varied Initial Conditions**

When a second initial vorticity with the opposite charge as the first was introduced in the initial conditions it created a dipole that translated within the computational domain. The two charged vorticity lumps did not seem to attract or repel, but rather translated together as they continued to spiral. This is shown in Figure 4.

When the second initial vorticity was the same charge as the first initial vorticity lump, a drastically different behavior was observed. The two lumps attracted each other and began to swirl together. This effect is shown in Figure 5.
Figure 4. Two oppositely charged initial vorticities translating in the computational domain.

Figure 5. Two same charged initial vorticities swirling together
An attempt was made to get two oppositely charged pairs to collide into each other, however when they were initialized they immediately began to repel. It can be seen in Figure 6 that by time unit 36 the four lumps have basically moved into each of the four corners of the domain. The lumps do not pass across the domain boundary, as was seen with the two oppositely charged lumps, because the periodic boundary conditions allow the lumps on the opposite side of the domain to repel.

Figure 7 shows the behavior of the oppositely charged lump pairs as the simulation is taken out to 180 time units. It can be seen that once the lumps initially retreat to the corners of the domain, they begin to come in on each other. Once they get sufficiently close to the middle of the domain, they again repel and move to the corners of the domain. If the simulation were taken out longer, it the lumps would continue to oscillate within their respective quadrants of the domain.

Figure 8 shows the time evolution of the vorticity-stream function relations with various initial vorticity lumps. The lumps in this simulation propagate across the domain in a seeming random pattern due to the variously positioned and charged vorticity lumps.

Figure 6. Two oppositely charged pairs of initial vorticities repelling each other. The time evolution is shown from time unit 4 to 36.
Figure 7. Two oppositely charged pairs of initial vorticity taken out to time unit 180.

Figure 8. Random assortment of initial vorticities
Summary & Conclusions
As was expected, the fastest technique for solving the vorticity-stream function relations was the FFT method. In addition to its much faster run time, the spectral accuracy of the FFT method produced much finer resolution results. The LU decomposition method resulted in a slightly faster run time than the Gaussian Elimination method, also to be expected. It was no surprise that the iterative methods produced the overall slowest algorithms for solving the problem. The accuracy and resolution of the finite difference and iteration methods were very nearly identical, much less accurate than the FFT method.

When running the FFT code with the alternate initial conditions it was found that the oppositely charged Gaussian lumps produced a dipole that translated across the domain. The two same charged Gaussian lumps would attract and merge together. When an attempt was made to get two pairs of oppositely charged Gaussian lumps to collide, it was found that the lumps would repel, and because of the periodic boundary conditions they bounced around the computational domain never colliding. A combination of these effects was observed when using a random assortment of Gaussian lumps.

Appendix A. MATLAB Functions
The implicit Matlab functions used as a part of this exercise are as follows:
- ode45 – used to solve ODEs for all methods
- linspace – used to create an evenly spaced vector to define discretized domain
- spdiags – used to populate sparse matrices
- ones – used to generate a vector of 1s for use in defining sparse matrices
- surf – used to generate a 3-d surface plot of the vorticity function
- length – used to find the length of a t vector for plotting purposes
- meshgrid – used to create spatial X and Y computational grid
- reshape – used to reshape matrices into vectors, and vice-versa
- fft,ifft,fft2,ifft2 – used for Fast Fourier Transforms and inverse FFTs in 1 and 2 dimensions
- pcolor – used to visualize vorticity time evolution
- tic,toc – used to find how quick each method found the solution
- num2str – used for easily labeling the time evolution plots

Appendix B. MATLAB Codes
main.m:
```matlab
clear all
close all
cic

L=20;
n=64;
tspan=0:2:50;
nu=.001;

%Discretize the domain
```
x2=linspace(-L/2,L/2,n+1); x=x2(1:n); y=x;
dx=x(2)-x(1);
[X,Y]=meshgrid(x,y);

%Generate derivative matrices and perform LU decomposition of A matrix
A=dim2_grad(n);
A(1,1)=1;
A=A/dx^2;
[l,U]=lu(A);

% initial data
w=exp(-X.^2-Y.^2/10);
w_vect=reshape(w,n^2,1);
w=fft2(w);
w=reshape(wt,n^2,1);
tsi_init=A\w_vect;

%K values for fft method
kx=(2*pi/L)*[0:n/2-1 -n/2:-1]; kx(1)=10^(-6); ky=kx;
[KX,KY]=meshgrid(kx,ky);
K2=1./(KX.^2+KY.^2);
K3=KX.^2+KY.^2;

%Control data for bicgstab and gmres methods
tol=dx^2;
maxit=200;

%Solve using fft
tic
[t,wsol_fft]=ode45('wrhs',tspan,wt2,[],KX,KY,nu,K2,K3,n);
t_fft=toc;

%Solve using A\b method
tic
[t,wsol_backslash]=ode45('main_rhs',tspan,w_vect,[],nu,A,B,C);
t_backslash=toc;

%Solve using LU decomposition
tic
[t,wsol_lu]=ode45('main_rhs_lu',tspan,w_vect,[],nu,A,l,U,B,C);
t_lu=toc;

%Solve using bicgstab method
tic
[t,wsol_BGS]=ode45('main_rhs_BGS',tspan,w_vect,[],nu,A,B,C,tol,maxit,tsi_init);
t_BGS=toc;

%Solve using gmres method
tic
[t,wsol_GMR]=ode45('main_rhs_GMR',tspan,w_vect,[],nu,A,B,C,tol,maxit,tsi_init);
t_GMR=toc

%save all data
save main_data

dim2_grad.m:
function [ A ] = dim2_grad( n )
%2DIM_GRAD takes in the number of finite difference grid points in one
%direction and generates a matrix that computes the second order center
%difference matrix that takes 2 derivatives in x and 2 derivatives in y
%for
%periodic boundary conditions assuming equal grid spacing in the two
%dimensions

e1=ones(n^2,1);
center_diag=-4*e1;
diag1=e1;
diag3=zeros(n^2,1);
for i=1:n
    diag1(i*n)=0;
    diag3(i*n-n+1)=1;
end
for j=1:(n^2)
    diag_plus3(j,1)=diag3(n^2+1-j);
    diag_plus1(j,1)=diag1(n^2+1-j);
end
A=spdiags([e1 e1 diag3 diag1 center_diag diag_plus1 diag_plus3 e1
e1],[-(n^2-n) -n -(n-1) -1 0 1 (n-1) n n^2-n],n^2,n^2);

dx_mat.m:
function [ B ] = dx_mat( n )
%DX_MAT takes in the number of finite difference grid points in the x
%direction and generates a matrix that computes the derivative in x

e1=ones(n^2,1);
d=n^2-n;
B = spdiags([e1 -e1 e1 -e1],[-d -n n d],n^2,n^2);

dy_mat.m:
function [ C ] = dy_mat( n )
%DY_MAT takes in the number of finite difference grid points in the y
%direction and generates a matrix that computes the derivative in y

diag_1=ones(n^2,1);
diag_3=zeros(n^2,1);
for i=1:n
    diag_1(i*n)=0;
    diag_3(i*n+1-n)=1;
end
for j=1:(n^2)
    diag_plus3(j,1)=diag_3(n^2+1-j);
    diag_plus1(j,1)=diag_1(n^2+1-j);
end

C = spdiags([diag_3 -diag_1 diag_plus1 -diag_plus3],[-(n-1) -1 1 n-1],n^2,n^2);

wrhs.m
function wrhs=wrhs(t,wt2,dummy,KX,KY,mu,K2,K3,n)
    wt=reshape(wt2,n,n);
    psit=(-wt.*K2);
    psix=real(ifft2(i*(KX.*psit)));
    psy=real(ifft2(i*(KY.*psit)));
    wx=real(ifft2(i*(KX.*wt)));
    wy=real(ifft2(i*(KY.*wt)));
    wrhs=reshape(-mu*K3.*wt -fft2(psix.*wy-psy.*wx),n^2,1);

main_rhs.m
function [ rhs ] = main_rhs( tspan,w_vect,dummy,nu,A,B,C )
%MAIN_RHS Summary of this function goes here
% Detailed explanation goes here

ts1=A\w_vect;
rhs=(C*tsi).*(B*w_vect)-(B*tsi).*C*w_vect)\nu*(A*w_vect);

main_rhs_lu.m
function [ rhs ] = main_rhs_lu( tspan,w_vect,dummy,nu,A,L,U,B,C )
%MAIN_RHS Summary of this function goes here
% Detailed explanation goes here

y=L\w_vect;
ts1=U\y;
rhs=(C*tsi).*(B*w_vect)-(B*tsi).*C*w_vect)\nu*(A*w_vect);

main_rhs_BGS.m
function [ rhs ] = main_rhs_BGS( tspan,w_vect,dummy,nu,A,B,C,tol,maxit,tsi_init )
%MAIN_RHS Summary of this function goes here
% Detailed explanation goes here

ts1=bicgstab(A,w_vect,tol,maxit,[],[],tsi_init);
ts1_init=tsi;
rhs=(C*tsi).*(B*w_vect)-(B*tsi).*C*w_vect)\nu*(A*w_vect);

main_rhs_GMR.m
function [ rhs ] = main_rhs_GMR( 
    tspan,w_vect,dummy,nu,A,B,C,tol,maxit,tsi_init)

%MAIN_RHS Summary of this function goes here 
% Detailed explanation goes here

tsi=gmres(A,w_vect,[],tol,maxit,[],[],tsi_init);
    tsi_init=tsi;
    rhs=(C*tsi).*(B*w_vect)-(B*tsi).*(C*w_vect)+nu*(A*w_vect);

wfft.m

clear all; close all; clc;

tspan=0:2:50;
L=20;
n=64;
L_d=40;
n_d=128;

mu=0.001;

x2=linspace(-L/2,L/2,n+1); x=x2(1:n); y=x;
[X,Y]=meshgrid(x,y);

kx=(2*pi/L)*[0:n/2-1 -n/2:-1]; kx(1)=10^(-6); ky=kx;
[KX,KY]=meshgrid(kx,ky);
K2=1./(KX.^2+KY.^2);
K3=KX.^2+KY.^2;

x2_d=linspace(-L_d/2,L_d/2,n_d+1); x_d=x2_d(1:n_d); y_d=x_d;
[X_d,Y_d]=meshgrid(x_d,y_d);

kx_d=(2*pi/L_d)*[0:n_d/2-1 -n_d/2:-1]; kx_d(1)=10^(-6); ky_d=kx_d;
[KX_d,KY_d]=meshgrid(kx_d,ky_d);
K2_d=1./(KX_d.^2+KY_d.^2);
K3_d=KX_d.^2+KY_d.^2;

% initial data
% Two oppositely charged lumps
w_a=exp(-(X-3).^2-Y.^2/10)-exp(-(X+3).^2-Y.^2/10);
w_a=fft2(w_a);
wt2_a=reshape(wt_a,n^2,1);

%Two same charged lumps
w_b=exp(-(X-3).^2-Y.^2/10)+exp(-(X+3).^2-Y.^2/10);
w_b=fft2(w_b);
wt2_b=reshape(wt_b,n^2,1);

%Two pairs of oppositely charged lumps
w_c=fft2(w_c);
wt2_c=reshape(wt_c,n^2,1);

%Variety of initial lumps
w_d=exp(-(X_d+15).^2-(Y_d+15).^2/10)-5*exp(-(X_d+12).^2-
(Y_d+12).^2/10)+.5*exp(-(X_d-15).^2-(Y_d-15).^2/10)+...
- exp(-(X_d-10).^2-(Y_d-10).^2/10)+.8*exp(-(X_d-3).^2-(Y_d-3).^2/10)-
- 2*exp(-(X_d-6).^2-(Y_d+6).^2/10)+...
- .2*exp(-(X_d).^2-(Y_d).^2/10);
wt_d=fft2(w_d);
wt2_d=reshape(wt_d,n_d^2,1);

% Solve ODE for all initial conditions
[t_a,wt2sol_a]=ode45('wrhs',tspan,wt2_a,[],KX,KY,mu,K2,K3,n);
[t_b,wt2sol_b]=ode45('wrhs',tspan,wt2_b,[],KX,KY,mu,K2,K3,n);
[t_c,wt2sol_c]=ode45('wrhs',tspan,wt2_c,[],KX,KY,mu,K2,K3,n);
[t_d,wt2sol_d]=ode45('wrhs',tspan,wt2_d,[],KX_d,KY_d,mu,K2_d,K3_d,n_d);

% Make plots for all initial conditions
figure (1)
for j=1:9
    subplot(3,3,j)
    wsol_a=ifft2(reshape(wt2sol_a(2*j,:),n,n));
    pcolor(X,Y,wsol_a), shading interp, colormap(hot)
    title(['t = ' num2str(4*j)])
    axis off
end

figure (2)
for j=1:9
    subplot(3,3,j)
    wsol_b=ifft2(reshape(wt2sol_b(2*j,:),n,n));
    pcolor(X,Y,wsol_b), shading interp, colormap(hot)
    title(['t = ' num2str(4*j)])
    axis off
end

figure (3)
for j=1:9
    subplot(3,3,j)
    wsol_c=ifft2(reshape(wt2sol_c(2*j,:),n,n));
    pcolor(X,Y,wsol_c), shading interp, colormap(hot)
    title(['t = ' num2str(4*j)])
    axis off
end

figure (4)
for j=1:9
    subplot(3,3,j)
    wsol_d=ifft2(reshape(wt2sol_d(2*j,:),n_d,n_d));
    pcolor(X_d,Y_d,wsol_d), shading interp, colormap(hot)
    title(['t = ' num2str(4*j)])
    axis off
end

main_plots.m
clear all
close all

load main_data

figure(1)
title('Vorticity solution at time = 25')

j=length(t)/2;

w_fft=ifft2(reshape(wsol_fft(j,:),n,n));
w_backslash=reshape(wsol_backslash(j,:),n,n);
w_lu=reshape(wsol_lu(j,:),n,n);
w_BGS=reshape(wsol_BGS(j,:),n,n);
w_GMR=reshape(wsol_GMR(j,:),n,n);

subplot(2,3,1)
pcolor(X,Y,w_fft), shading interp, pause(0.5)
title('FFT Method')
axis off

subplot(2,3,2)
pcolor(X,Y,w_backslash), shading interp, pause(0.5)
title('A\b Method')
axis off

subplot(2,3,3)
pcolor(X,Y,w_lu), shading interp, pause(0.5)
title('LU Decomposition Method')
axis off

subplot(2,3,4)
pcolor(X,Y,w_BGS), shading interp, pause(0.5)
title('BICGSTAB Method')
axis off

subplot(2,3,5)
pcolor(X,Y,w_GMR), shading interp, pause(0.5)
title('GMRES Method')
axis off

figure(2)

for j=1:9
    subplot(3,3,j)
    w_fft2=ifft2(reshape(wsol_fft(2*j,:),n,n));
pcolor(X,Y,w_fft2), shading interp, colormap(hot)
title(['t = ' num2str(4*j)])
axis off
end