An Experimental and Numerical Analysis of 2-Dimensional Kolmogorov Fluid Flows

---

A Thesis
Presented to
The Division of Mathematics and Natural Sciences
Reed College

In Partial Fulfillment
of the Requirements for the Degree
Bachelor of Arts

Devon Kesseli

May 2015

Approved for the Division
(Physics)

Daniel Borrero

Acknowledgments

Thanks to all of the people who guided me though this process, and helped me get here in the first place. First to Daniel Borrero, for leading me through the most turbulent sections, and Reed's extremely helpful and inspiring faculty and staff. Also to my parents and Rory, who backed me up every step of the way. Finally to my friends on this coast and the other one, who make everything worth battling for.

Table of Contents

List of Tables

List of Figures

Abstract

We studied the effect of shear forces in a two-dimensional fluid flow experimentally and computationally by forcing fluid in a sinusoidal pattern known as Kolmogorov flow. Both the experimental and numerical data show the expected qualitative change in behavior, with the flow changing from the laminar solution to a lattice of interlocking vortices as the Reynolds number is increased. Experimentally, we modeled the flow by electromagnetically forcing a thin layer of fluid. We extracted information about the flow using particle image velocimetry, and found the bifurcation to occur at a critical Reynolds number of 85. This result agrees reasonably well with similar experiments, which have yielded critical Reynolds numbers of 61 [1] and 70 [2]. Significant error is caused by the friction between the working fluid and the bottom of the tray. Numerically, we developed two codes, one time-independent and the other time-dependent. Both successfully demonstrated the initial bifurcation. Using the time-dependent code, we simulated the Kolmogorov flow in both a rigid box with no slip boundary conditions, and in a periodic cell. From these simulations, we observed the qualitative motion of the fluid, and created bifurcation diagrams, both of which showed a strong correlation to the experimental data.

Introduction

Fluid dynamics is a vast and well-storied branch of physics that has progressed a huge amount in the last century. In addition to physics, it is a field of interest for both engineering and mathematics. Fluid dynamics encompasses both liquid and gas flows, making it useful in many engineering problems, from aerodynamics to fluid cooling mechanisms. The governing equations of fluid dynamical systems are nonlinear, making them interesting and incredibly difficult mathematical problems, with chaotic and turbulent solutions.

In this project I experimentally and computationally model 2-dimensional fluid flows. While a 2-dimensional fluid cannot actually exist in our 3-dimensional universe, solving 2-dimensional fluid dynamics problems can be surprisingly useful. Systems such as Earth's oceans and atmosphere, or just a thin layer of fluid in the lab can often be accurately approximated as 2-dimensional systems owing to the fact that their area is many times greater than their depth. I chose to work with a 2-dimensional thin fluid layer because it is much easier to work with than a 3-dimensional system. Experimentally, a 2-dimensional system enables me to place and view tracer particles on the surface instead of mixed throughout the fluid. Computationally, 2-dimensional systems are also far easier to model, requiring a simpler code, and significantly less computational power. This allowed me to write a program that I could run on a regular personal computer.

The system I worked with is called a 2-dimensional Kolmogorov flow. This system, first studied by Soviet physicist and mathematician A. N. Kolmogorov, is defined by a sinusoidal forcing pattern along one axis, and no forcing along the other. In an x-y coordinate system, this is described by the equation

$$\vec{f}(x) = A \sin(bx)\hat{y}, \tag{1}$$

as shown by the vectors in Figure 1.1, where A is the amplitude of the forcing, b is its spatial frequency.

This creates anti-parallel bands of flow, which exert a shear force on one another. The magnitude of this shear is determined by the forcing strength and viscosity of the fluid, which are both captured in the Reynolds number. The Reynolds number is a constant that arises in the governing equations of fluid dynamics derived in Chapter 2. When the Navier-Stokes momentum equation is nondimensionalized, constants are combined into a single value. Convention defines this as the inverse of the Reynolds number, the independent variable in my experiment. Physically, the Reynolds Number represents the relative importance of fluid inertia and viscous dampening [1]. Low

Reynolds numbers would be assigned to slow moving, viscous fluids like a pipe full of maple syrup, while high Reynolds numbers would correspond to fast, turbulent flows. As the Reynolds number increases, so does the shear force, resulting in a drastic, qualitative change in the overall movement of the fluid, called a bifurcation. The Reynolds number is a universal construct, and is essential to describing the behavior of any fluid dynamical system.

In this experiment I am examining the bifurcations of the Kolmogorov system. As mentioned above, a bifurcation point is a point where, while varying some parameter, the system experiences some qualitative change of behavior [3]. An example of this would be a beam snapping under too much stress, or a population exploding once the number of predators is lowered below a critical value. These changes represent a shift in the stable state of the system. Generally, when we observe a system, it is either moving towards, or has arrived at its most stable form. Because of this, we can describe the long term behavior of a system by studying its fixed points, points at which the system's behavior will stay fixed over time. Stable fixed points are points where, after a small perturbation, the system will return to its initial state, like the lowest point of a pendulum's swing. Unstable fixed points are fixed points at which a small perturbation will result in movement away from the fixed point, like a pencil balanced on its point.

A bifurcation diagram allows us to visualize the behavior of a system as some parameter is varied. These diagrams are simply plots of fixed points as a function of the independent variable. Stable fixed points are represented by solid lines, while unstable fixed points are represented as dotted lines.

An extremely simple example would be the mass on a rod example from [3]. Here, a mass can slide horizontally on a rigid rod. The mass is attached to a spring, connecting it to a solid platform directly below it, as pictured in Figure 1. The independent variable here is platform height, which can be raised or lowered. The spring can be compressed or extended about some natural length, x_c. With the platform well below x_c, the only fixed point for the mass is directly above the platform. Any perturbation from this point will cause it to oscillate and eventually return to the center. When the platform is raised to within a distance x_c from the rod, the system changes. Now, with the spring compressed, the point directly above the platform is unstable. While the mass can be balanced at this point, any perturbation will cause it to move outwards and eventually settle at one of the two new stable fixed points on either side of the center. This is shown in Figure 1, along with a bifurcation diagram for the system. This is called a supercritical pitchfork bifurcation similar to the one we are looking for in the Kolmogorov system [1, 3, 2]. This bifurcation is defined by the fact that at some critical value, a stable fixed point becomes unstable, spawning two new stable fixed points on either side.

As stated above, the bifurcation diagram for Kolmogorov flow contains a supercritical pitchfork, where the laminar solution become unstable. If Reynolds number continues to increase, we get a Hopf bifurcation [4], turning from a steady flow to a limit cycle. Here, the flow moves through a periodic pattern of fixed points, resulting in an oscillating lattice of vortices. These are fairly common types bifurcation presented in [3]. The expected results I am using demonstrate these bifurcations

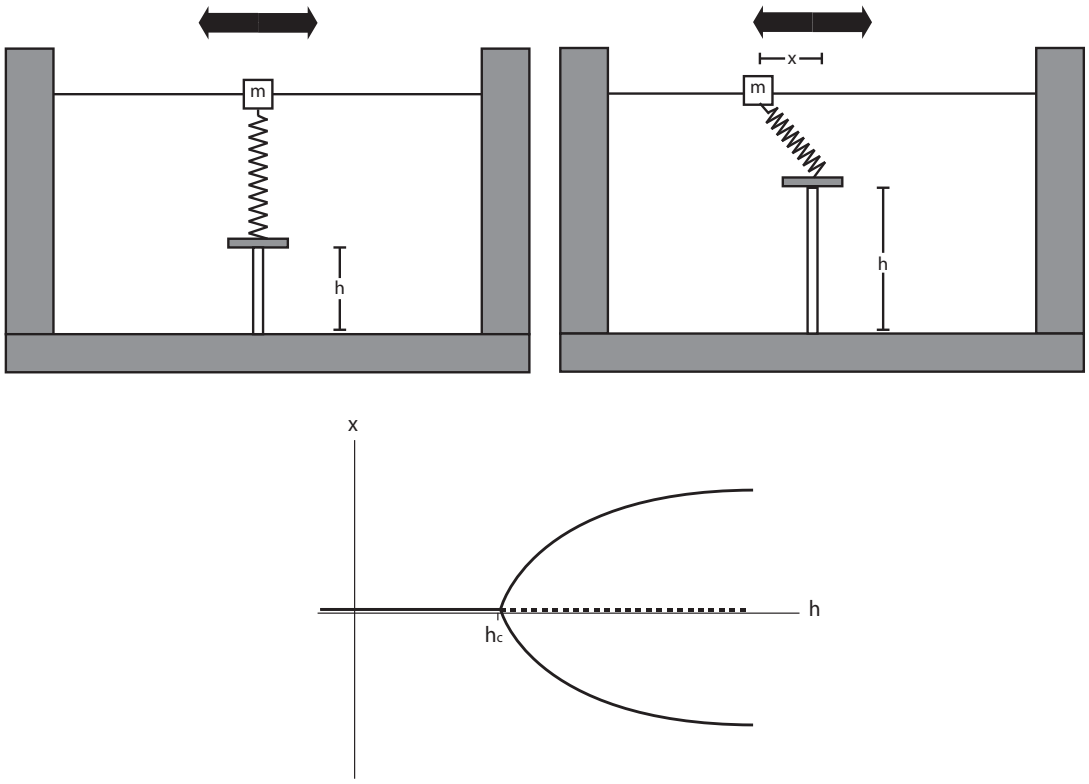


Figure 1: An example bifurcation diagram for an over-damped pendulum with a rigid rod. The x-axis is some arbitrary parameter that does not affect the fixed points, like size or color of the mass, and the y-axis is angle θ. The pendulum is over-damped, so it will not swing past its lowest point.

using several different experimental setups that simulate Kolmogorov flow. Each experiment has frictional damping, a source of error that is difficult to account for, which causes the points at which these bifurcations occur to vary from experiment to experiment.

0.1 Some Numerical Background

New techniques for analyzing nonlinear systems paired with a few important technological developments have revolutionized fluid dynamics over the last century. Fluid dynamical systems are usually difficult to solve analytically, so were effectively unsolvable until computers became powerful enough to provide numerical solutions [5], and experimental measuring tools were precise enough to provide an accurate model of a fluid flow [6]. Below I provide some historical background and basic information about both of these important developments.

The Navier-Stokes equations are the governing equations of fluid dynamics. These equations are based off of conservation of mass, energy, and momentum and were derived in the first half of the nineteenth century separately by M. Navier and G. Stokes [5]. These equations were derived by applying these classical conservation

laws to an infinitesimal volume of fluid. These equations are derived in Chapter 2, Theory.

Computational Fluid Dynamics became a viable way to analyze fluid systems in the mid 1960s [5]. At this time the Cold war was in full swing and a huge amount of research was going into making better aerodynamics for both Intercontinental Ballistic Missiles and space travel technology. Before this time, the only options were analytic solutions, which were extremely rare due to the nonlinearity of the Navier-Stokes equations, and physical tests, either outdoors or in giant wind tunnels, which were difficult and impractical, especially to model things moving at supersonic speeds. Numerical solutions on the other hand are limited only by the amount of information we can input into the computational model and the amount of information a computer can process, both of which have increased enormously over the last fifty years.

As part of my study, I developed two different programs in *Mathematica* to solve an idealized 2-dimensional Kolmogorov flow system. The code is in Appendices A and B, and an explanation of the techniques used can be found in Chapter 2.

0.2 Some Experimental Background

Experimental fluid dynamics faced similar issues. The principles of flow visualization through particle tracking have been around since the early twentieth century, when Ludwig Prandtl observed mica particles on the surface of a flowing fluid [6]. Although this allowed him to visualize the patterns in a moving fluid, techniques for extracting quantitative data did not emerge until much later. Advanced optics such as lasers and fluorescent particles created bright and easily imaged particles. Better resolution and high-speed cameras allowed these images to be accurately captured. Finally, better computer software enabled huge amounts of this data to be analyzed simultaneously. The Von Karman Institute in Brussels first developed these techniques in the early 1980s [6].

0.3 Format

This thesis has two main portions. The first is the experimental section, where I applied forcing to a thin layer of fluid and used Particle Image Velocimetry to analyze the resulting flow fields. The second portion is entirely computational. I worked on codes in *Mathematica* to model the Kolmogorov system using several different techniques. These two portions represent very different ways to deal with a nonlinear system. Each method has strengths and weaknesses, and together should give a more complete picture of Kolmogorov flow.

The main results I am looking for in both the experimental and numerical sections is a qualitative picture of the bifurcation, with behavior matching those described in [1], [7], and [4]. Once the qualitative behavior of the fluid is confirmed in the numerical and experimental systems, I will look at my data more quantitatively. For the experimental portion, I will calculate the Reynolds numbers that correspond to different forcing amplitudes, and estimate the Reynolds number at which the system

bifurcates. I can compare this to the results found in [1]. I will analyze the bifurcation both experimentally and numerically, creating a bifurcation diagram for each method. I will compare these diagrams, and classify the bifurcation as well as possible.

In Chapter 1, Experimental Methods, I describe my experimental setup and the techniques I used in detail. I explain the techniques used in Particle Image Velocimetry to quantitatively measure fluid flow, and describe the specific setup I used to create these flows. Chapter 2 describes the governing equations and fluid dynamics and computational methods that I used to simulate the experimental flow. I derive the Navier-Stokes equations that govern the moving fluid, and explain how these are integrated into a computer program and solved numerically. Chapter 3 describes the results of both the experiment and simulations, and how the data generated in each piece was analyzed. Finally, Chapter 4 summarizes the important results and sources of error, and discusses possible extensions to expand and improve on this project.

Chapter 1

Experimental Methods

1.1 Experimental Concepts

Although 2-dimensional flows are impossible in nature, they can be modeled by thin fluid layers, like our atmosphere which is much thinner in its vertical extent than in its other two dimensions. For practical reasons, I am using a thin layer of fluid in a box. This section describes a few basic concepts necessary for this experiment. First is the forcing method. Some common ways to apply a force to a fluid are with pressure, friction, and electromagnetism. Pressure is used mainly in experiments involving 3-dimensional flows, such as the flow through a pipe. In the driven cavity problem, a cavity full of fluid is driven by a moving boundary, which uses friction to drag along the nearby fluid. The problem I am working with requires a more complicated, sinusoidal forcing, described by Equation 1, so using an electromagnetic force is ideal.

The method used here is based on the Lorentz force. This is the basic electromagnetic principle that describes how a charged particle moving through a magnetic field experiences a force equal to the cross product of its velocity and the magnetic field. This is illustrated by the Lorentz equation. The current form of this equation gives the force, $\vec{F}$, acting on a moving charge distribution, and is written

$$\vec{F} = \vec{J} \times \vec{B}. \tag{1.1}$$

Here, $\vec{J}$ is the current density, a vector equal to the charge density times its velocity, and $\vec{B}$ is the magnetic field vector. The direction of the force is easily found by calculating the cross product or by using the right hand rule. From this equation, it is clear that particles in a sinusoidal magnetic field pointing in the z-direction, shown by the red and blue stripes in Figure 1.1, will yield a sinusoidal forcing pattern, shown by the vector arrows, when a uniform sheet of current travels from left to right. The force will in be in sinusoidal, anti-parallel stripes, as depicted in Figure 1.1, and described by Equation 1.

In my computational model, the fluid layer can be made infinitely thin, and its motion across the bottom of the tray frictionless. In reality, there is significant friction between the fluid and the bottom of the tray, making it difficult to match the experiment to numerical data. One way to get around this is to put a layer of denser,

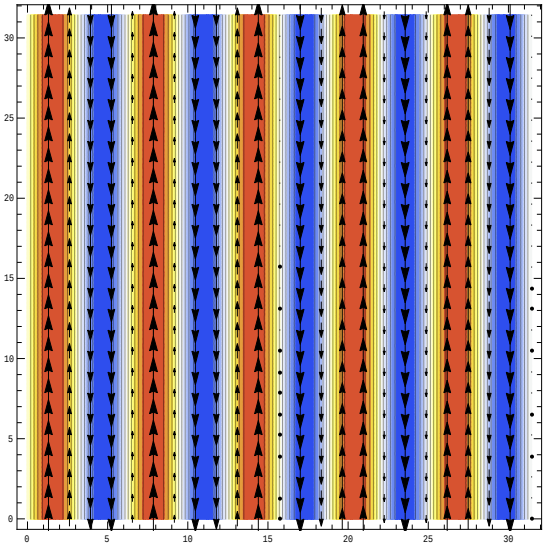


Figure 1.1: A picture of the forcing pattern over the magnetic field. Blue represents a magnetic field into the page, while red is field out of the page. The forcing pattern, indicated by the arrows, is sinusoidal and represented by Equation 1.

hydrophobic liquid between the fluid and the tray, as done in [8]. This is important if you are trying to match your experiment very closely to a numerical model.

1.2 Setup

The setup I used is based on the one described by Kelley and Ouellette [1]. It consists of a tray containing a thin layer of electrolyte resting on a configuration of magnets. Rows of magnets alternating in polarity will create a roughly sinusoidal magnetic field. An electrical current is driven through the fluid perpendicular to the field, yielding the desired forcing pattern. Figure 1.2 shows a diagram of this setup, the main elements of which are described below.

To create the magnetic field, I used circular neodymium-iron-boron grade N52 magnets from K & J Magnetics. These magnets have a diameter of 12.7 mm and a thickness of 3.18 mm. The manufacturer's specifications list the magnetic field at the surface to be 0.3309 T. The magnets are held in a grid made of two sheets of 3.15 mm thick acrylic. With our mechanist, Jay Ewing's help, I laser cut circles into the topmost sheet and epoxied the two sheets together, creating a magnet holding grid. The center to center spacing between grid points is 19 mm. I then arranged the magnets into the grid in stripes of alternating polarity, covering finished rows with thin strips of painter's tape to keep the magnets in place. The fluid tray was also made using the laser cutter and sealant (entirely by Jay). The tray is 29.5 mm, much larger than the magnet grid. While the size of the tray will affect the flow towards the edges of the grid, I made sure to only observe the flow near the center of the tray, where the boundary conditions will not be as important. The bottom was painted black for better contrast with the particles I will use to image the flow. Two pieces of

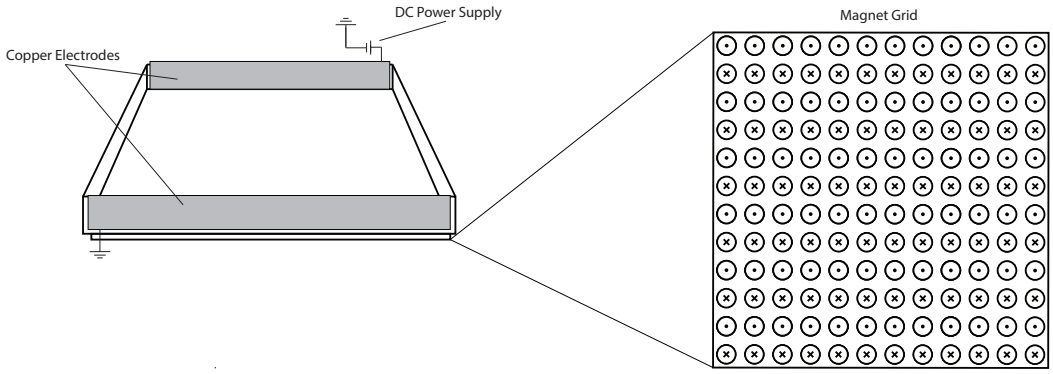


Figure 1.2: The basic experiment plan. The magnets are arranged in rows of alternating polarity under the tray, and the copper electrodes are connected to a DC power supply.

copper bent over opposite sides of the tray were used as electrodes, and connected to a Tektronix PS280 DC Power Supply using alligator clips to drive a current through the fluid layer. The experimental setup is shown in Figure 1.3.

The working fluid is a mixture of water, copper sulfate (to make it more conductive), and glycerol (to make it more viscous). Our solution is 10% $CuSO_4$ by mass and 20% glycerol by volume as described in [1]. To get quantitative measurements out of this setup, I needed to put tracer particles into the fluid, record their movements, and analyze these recordings in Matlab. The particles that I used are S60 microscopic glass bubbles from 3M. I recorded the motion of these particles using a Canon T5i DSLR camera supported by a ring stand over the tray of fluid. I wrapped all reflective surfaces above the fluid in black cloth to reduce any glare on the bottom of the tray. The camera was switched to video mode, with all settings left at automatic, except the standby function, which was turned off to stop it from interrupting the focusing process, and the focusing, which was switched to manual. I recorded at a resolution of 1920×1080 pixels, at 30 frames per second.

For my lighting setup, I used three 38cm × 38cm square white-light emitting diode (LED) panels made by iOS Light, Venice, CA. These were arranged around the tray as shown in Figure 1.3, connected in series and powered by another 12V DC power supply, which can be seen to the left of Figure 1.3. This worked very well for the particle setup I used. For recording at higher currents, and therefore higher particle velocities, it is common to use strobed LEDs. I attempted to use a strobe set up made of a powerful blue LED connected to a waveform generator. By flashing at the same speed as the frame rate of the camera, the effective shutter speed of the camera is reduced to the duration of the strobe flash. With a bright enough strobe, this is a good way to get high resolution images, with less blur than a longer shutter speed would give. I tried this method first, but had a lot of trouble getting the lighting bright enough, and evenly distributed across the tray. I also ran into some problems with the camera's indexing. If the period of the strobe light is not perfectly lined up with the frame rate of the camera, you get bands of doubly-exposed or unexposed sections that travel up and down the screen. This is a result of how the camera

records records each pixel, and makes for unusable data.

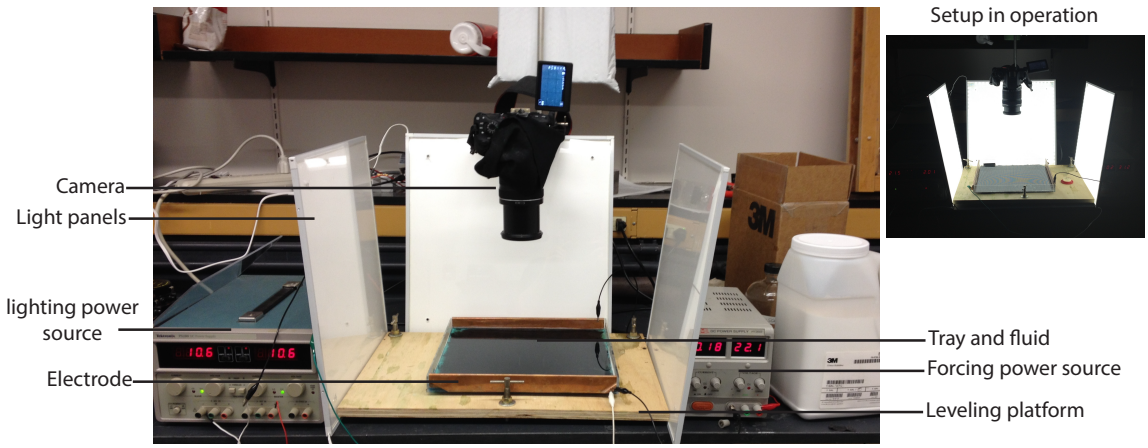


Figure 1.3: The experimental setup with key components labeled.

1.3 Data Collection

To collect data, I first made sure the grid of magnets was level using an adjustable leveling platform. I then aligned the camera with the magnet grid to ensure that the lines of magnets were parallel to the edges of the frame on the camera viewer. I centered the camera on a (roughly) 8.5 cm by 13 cm section at the center of the magnet grid, and used a level to make sure the camera was also level, pointing directly downwards. Next, I positioned the tray over the magnets, attached the copper electrodes and connected the alligator clips from the electrodes to the power source. After pouring in 450 mL of fluid, I measured the fluid depth at the center of the tray using Pittsburgh sliding electronic precision calipers. I then seeded the fluid with tracer particles by sprinkling them on top little by little, and then mixing until average particle spacing looked to be approximately 0.3 mm. Next, I used the camera's digital zoom to focus, zooming in and focusing for maximum particle resolution, and zooming back out to the original sized window. The exact size of the window was not important, as long as the final video held at least a few periods of the forcing pattern, to get a full sample of the motion. Usually this turned out to be about 8 rows of magnets.

Next, I recorded the particles' motion at different Reynolds numbers. I varied the Reynolds numbers by changing the electrical current (since this is much easier to adjust than the fluid's viscosity). I turned on the current to 20 mA and began recording. I then slowly turned to current up in intervals of 10 or 20 mA, giving the flow one minute to settle into its new pattern. After the fluid had settled, I recorded for an additional 5 seconds, noting the time on the camera recording and the current coming out of the power source, and then raised the current again. I repeated this as many times as possible, until the current source could not go any higher. This ended up being at about 220 mA.

This setup could be improved by using better cameras, light sources and fluorescent particles, but all of these things are expensive and not entirely necessary for the type of results I am looking for.

1.4 PIV Methods

While creating the fluid flow that effectively mirrors what you want to measure is difficult, extracting quantitative data from the flow also poses a challenge.

There are a few different adjustable parameters we can change in a particle imaging setup to get the best possible results. Different flow systems require different conditions for particle-based data collection to be effective. Some important parameters include particle size, particle material, density of particles within the fluid, the particles' effectiveness at scattering light, as well as the lighting itself and the quality of the recording device.

Luckily, in a 2-dimensional fluid flow, a particles' material is much less of a concern, making it much easier to find good particles. This is because we are only tracking the surface of the flow, so as long as the particles are made of a material that is less dense than the fluid, they will stay on the surface.

In all particle tracking systems, smaller particles, better image quality, and better lighting with more luminescent particles result in more accurate tracking. While small particles more faithfully follow a flow without interfering with the motion of the fluid [6], particles also must be big enough to show up on camera. With regards to lighting and image quality, as explained above, these things can definitely always be improved through more expensive cameras and brighter lighting to achieve more precise results.

The final criterion for particle tracking, density of particles within the fluid, depends on the tracking technique being used. In traditional particle tracking, the trajectories of individual particles are followed. The drawback of this method is that it requires a very high image frame rate, low velocity, or low particle density. This is because individual particles are indistinguishable. If a particle moves too far from its initial position between frames, the tracking algorithm has no way of distinguishing it from another particle that moved towards its previous location. This means particles must be sparse and slow, resulting in a less well-resolved velocity field.

Particle Image Velocimetry, or PIV, is a newer method that allows us to use a higher particle density than traditional particle tracking methods. PIV software divides each image into tiny "interrogation windows," each consisting of a tiny group of particles. The unique distribution of particles in each window is distinguishable, and moves fairly homogeneously between frames. PIV software tracks each of these configurations using statistical methods to determine the overall motion of the fluid.

PIV Statistical Methods

The statistical methods themselves can be pretty complex. A complete description can be found in the third chapter of [6]. Here I explain a very simplified version, covering the basic concepts and techniques a PIV system uses. This technique involves

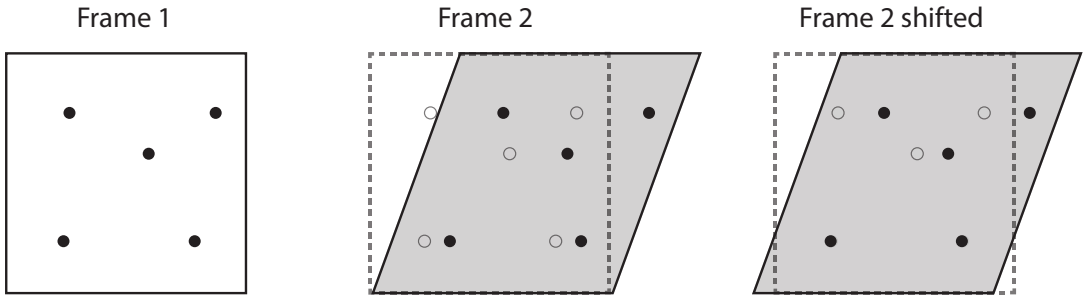


Figure 1.4: A comparison of interrogation windows in two adjacent frames. Black dots represent particles (in reality particles would be white spots on a dark background). Initially the PIV software shifts the interrogation window to track the overall motion of the window. The leftmost image shows frame two shifted left, leading to some correlation (the bottom two points).

first assigning a value to each pixel according to its brightness. In the gray-scale input image, lighter pixels are given higher numerical values, to create an image intensity field. The program then breaks the entire area being analyzed down into "interrogation windows", smaller boxes which it will look for in adjacent frames. The program then attempts to pick out the same pattern in the next frame, shifting and warping the original interrogation window to try to find the best correlation with the same window in the new frame. This process is described in greater detail below.

The first thing a PIV program does is create an intensity field. In theory, this should be fairly simple. The computer would just need to pick out two different gradient values; the infinitely bright, white particles on a black background. In reality, there is a lot more to this. To mathematically represent an intensity distribution, a program must take into account both the input image, and the point spread function. The point spread function is the function that describes how a single point spreads out when imaged. For an ideal, well focused lens, the point spread function is the Airy function [6], which looks similar to (and is commonly approximated as) a Gaussian function. PIV software can also take into account defects in the lens and in the image, like scratches and reflections in the tray. I tried to minimize these defects, but did not worry about tailoring the software specifically to my experiment, since the error from these small defects was not significant compared to errors from other aspects of my setup.

After creating an image intensity field, the PIV program needs to create the cross-correlation function. In an extremely steady flow, there would be no warping of interrogation windows. Each window would move along, with the internal particle configuration staying the same from frame to frame. This is obviously not the case, but some correlation can be found simply by shifting the window around its previous location. This is depicted in Figure 1.4.

The numerical method for actually determining the correlation between two windows involves multiplying their intensity fields at each point, and summing over the entire window. One window is then shifted and the process repeated, yielding a correlation value for each shift. This is pretty effective, since configurations in which more

high intensity values (particles) from two windows overlap will have larger numbers being multiplied together, and therefore will yield higher correlations. The end result of this shifting around is a 2-dimensional grid, with each discrete point representing a single pixel shift. The value associated with each point represents the cross-correlation of the window in the new frame shifted by the given amount, with the interrogation window from the original frame. This can be visualized as a 3-dimensional graph, with spikes protruding from a plane. Taller spikes represent better correlation for a given shift. A possible set of peaks is shown in Figure 1.5.

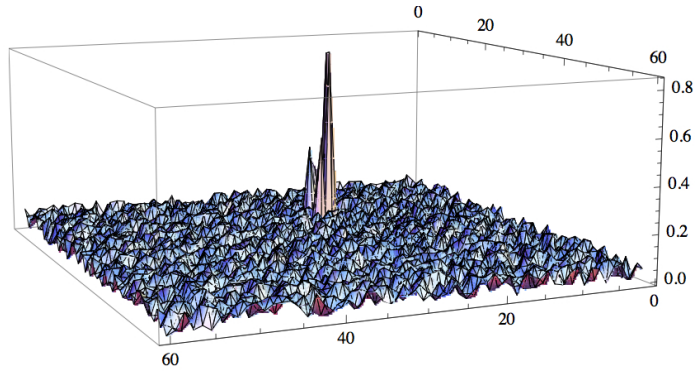


Figure 1.5: A hypothetical plot of cross correlations. The peak corresponds to a good correlation, revealing the shift of an interrogation window.

From here, the program is able to guess the general motion of the window, ignoring any shear or distortion. By comparing the general motion of nearby windows, the software can estimate how each window is distorted. This allows it to compute a distortion function. A rough sketch of this is shown in Figure 1.6.

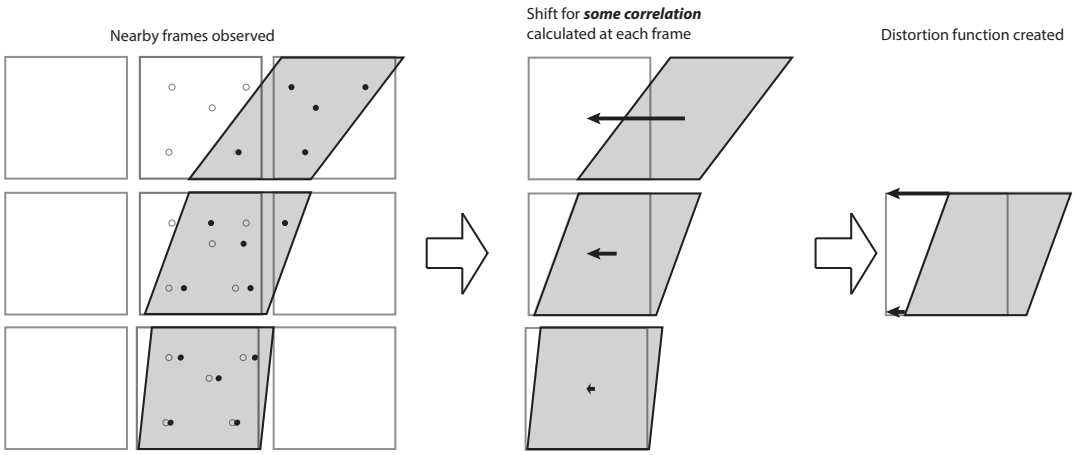


Figure 1.6: The black squares represent interrogation windows, three of which contain configurations of particles. The left image shows the movement of three adjacent windows from frame 1 (gray) to frame 2 (black). The center image show general shift of each window observed, as explained above. The rightmost image shows the resulting distortion function for the center window.

With general shift and distortion roughly figured out, the PIV program can put these together, applying both distortion and shift to the new interrogation window in the second frame, and checking for correlation. This process can be repeated for multiple iterations until the correlation peak reaches a certain threshold. A good correlation, after shift and distortion, is depicted in Figure 1.7. The plot of correlation peaks for this idealized example (or any good correlation) would look like a single tall peak, with little noise around it. Because the frame can only move in discrete shifts (of one pixel), a plot of correlation peaks may have a high peak made up of a set of adjacent, different sized peaks. To find the actual movement, the software fits the correlation peaks to a Gaussian, whose maximum provides a more exact result, with sub pixel resolution.

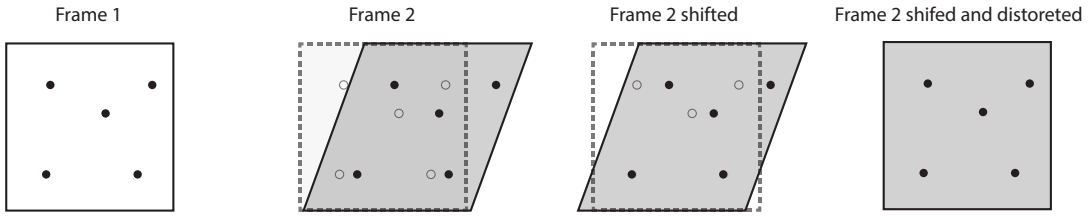


Figure 1.7: An idealized depiction of two consecutive frames. The shift is applied to frame 2 in the third figure and the distortion in the fourth, resulting in a perfect correlation.

The specific program we used is called Prana, a free, open-source PIV package for Matlab. This can be downloaded at http://sourceforge.net/projects/qi-tools/files/. This program was originally developed at the AEThER Laboratory at Virginia Tech. It accepts consecutive gray-scale frames of a video, and returns matrices of x and y velocity values. To get our raw videos into a form that the software would accept, I cropped each video to a 2-3 second section for each driving current, and wrote a simple *Mathematica* code that extracts, gray-scales, and exports the first few frames. We then ran sets of frames through the code, each of which took a few hours total. We did two passes, the first with larger interrogation windows, 32 pixels by 32 pixels, the second with smaller 24 by 24 windows. The two pass method allows the program to narrow down the motion even further. It was important not to go below 24 by 24 pixels, since it is important to have at least 10 particles per window, to give each a unique signature. Prana also eliminates spurious vectors that sometimes occur by comparing them to the median of neighboring vectors and looking for outliers, using the technique presented in [9]. Finally, the resulting vector field is smoothed out with a Gaussian filter.

Chapter 2

Theory

2.1 The Navier-Stokes Equations

The Navier-Stokes equations are the governing equations of fluid dynamics. They were developed independently by M. Navier and G. Stokes near the beginning of the 19th century [5]. These equations are essential to understanding fluid dynamics and are at the very heart of the code. The derivation presented here loosely follows the derivation found in the second chapter of [5].

The governing equations of fluid dynamics are based on three fundamental principles. These are the conservation of mass, the conservation of momentum, and the conservation of energy. I am not including the derivation for the energy equation, since it is not necessary for my analysis. The Navier-Stokes equations take the place of Newton's laws for fluid dynamics, and are based on applying Newton's second law to an infinitesimal fluid element.

2.1.1 Tools and concepts

The infinitesimal fluid element is the first concept we will need to derive and understand these equations. For rigid bodies, a force on one side moves the entire body as a single unit. This is obviously not the case with fluids. The way we can deal with nonrigid materials is to divide them into lots of minuscule boxes. There are two different ways to do this.

The first method looks at fluid using a finite control volume. For this method, we are looking at a tiny control volume fixed in space, with fluid flowing freely through it. The second method uses the infinitesimal fluid element. Here, we are looking at a tiny box of fluid as it flows around in time and space. I will be primarily discussing the infinitesimal fluid element method, but the distinction between the two will be important later.

The next tool we will need is the substantial derivative. This is really just the full derivative (as opposed to the partial derivative). This is the derivative we would use for the infinitesimal fluid element. For one of these chunks of fluid, as it moves through our stationary reference frame, its position (x,y), and time are all changing. This means that any of its properties, such as pressure, temperature, or x-velocity

(each functions of position and time) moving over the time interval $[t_1, t_2]$ can be written

$$T(x_1, y_1, t_1) \longrightarrow T(x_2, y_2, t_2), \tag{2.1}$$

where the T is temperature, x_1 and y_1 are the x and y coordinates at t_1, and x_2 and y_2 are the x and y coordinates at t_2. From here, we can write a first order Taylor approximation for point 2 using point 1.

$$T_2 = T_1 + (t_2 - t_1)\left(\frac{\partial T}{\partial t}\right)\Big|_{x_1,y_1,t_1} + (x_2 - x_1)\left(\frac{\partial T}{\partial x}\right)\Big|_{x_1,y_1,t_1} + (y_2 - y_1)\left(\frac{\partial T}{\partial y}\right)\Big|_{x_1,y_1,t_1}. \tag{2.2}$$

Rearranging this yields,

$$\frac{T_2 - T_1}{t_2 - t_1} = \left(\frac{\partial T}{\partial t}\right)\Big|_{x_1,y_1,t_1} + \frac{x_2 - x_1}{t_2 - t_1}\left(\frac{\partial T}{\partial x}\right)\Big|_{x_1,y_1,t_1} + \frac{y_2 - y_1}{t_2 - t_1}\left(\frac{\partial T}{\partial y}\right)\Big|_{x_1,y_1,t_1}. \tag{2.3}$$

Now, taking the limit as $\Delta t = t_2 - t_1$ goes to zero yields

$$\frac{DT}{Dt} = \frac{\partial T}{\partial t} + v_x\frac{\partial T}{\partial x} + v_y\frac{\partial T}{\partial y}, \tag{2.4}$$

where $\frac{D}{Dt}$ defines the substantial derivative. This could be extended into as many dimensions as we need. My thesis only deals with 2, but it is easy to write the substantial derivative more generally as

$$\frac{D}{Dt} \equiv \frac{\partial}{\partial t} + \left(\vec{v} \cdot \vec{\nabla}\right). \tag{2.5}$$

This is enormously useful for looking at properties of the infinitesimal fluid element. Besides being used to find the element's pressure and temperature as it moves through time, the substantial derivative is what we need to define the acceleration of the infinitesimal fluid element. For a moving fluid element, the acceleration in the x-direction can be written

$$a_x = \frac{Dv_x}{Dt}. \tag{2.6}$$

This follows from the definition of the substantial derivative, since the acceleration of a box of fluid is the rate of change of velocity for our fluid element moving through time and space.

2.1.2 Conservation of mass

With the concepts and tools above we can start applying some basic physics principles to fluids. First, we will deal with conservation of mass. To express this principle, I am using the first method for looking at fluid elements; the finite control volume. Both methods could yield useful equations, but the tiny box fixed in space with fluid

passing through it will give the conservative form of the mass continuity equation, which is the form that we will need later.

So, picturing a tiny box fixed within the fluid, assuming mass is conserved allows me to state that the rate of change of the mass within the box will be equal to the amount of mass exiting or entering the box through the surface. This must be the case, since there are no magical sources or sinks of mass inside our tiny cube.

When written mathematically, the change in mass of the cube is

$$\delta M_{\text{inside}} = -\frac{\partial}{\partial t}\int_V \rho \, dV \tag{2.7}$$

and the flow of mass through its surface is

$$\delta M_{\text{through surface}} = \oint_V \rho \vec{v} \cdot \vec{dS}. \tag{2.8}$$

Here, $\vec{v}$ is the velocity vector, with a component in each direction at each point in space, and ρ is the density of the fluid, with a scalar value at each point in space. In Equation 2.7, we are integrating over the volume of our box of fluid, with $dV = dxdydz$. In 2.8, we are integrating over the closed surface of the control volume, with the directional area element $\vec{dS}$ pointing perpendicular out of each face. Equation 2.8 can be rewritten as a volume integral using the vector divergence theorem;

$$\oint_V \rho \vec{v} \cdot \vec{dS} = \int_V \vec{\nabla} \cdot (\rho \vec{v}) \; dV. \tag{2.9}$$

We can also rewrite Equation 2.7 as

$$-\frac{\partial}{\partial t}\int_V \rho \, dV = -\int_V \frac{\partial \rho}{\partial t} \, dV, \tag{2.10}$$

since the volume of the box is fixed in time. Setting equations 2.10 and 2.9 equal to each other by the logic explained above, we can rearrange, and combine the volume integrals to yield;

$$\int_V \left[\frac{\partial \rho}{\partial t} + \vec{\nabla} \cdot (\rho \vec{v}) \right] dV = 0. \tag{2.11}$$

This is essentially saying that the change of mass within some volume is equal to the total divergence of the mass flux inside that volume. Since Equation 2.11 must hold for any choice of volume, it is clear that

$$\frac{\partial \rho}{\partial t} + \vec{\nabla} \cdot (\rho \vec{v}) = 0. \tag{2.12}$$

This is the general form of the mass continuity equation but it is not entirely necessary for our system. Because our fluid moves at relatively slow speeds at relatively low pressures, we can assume it is incompressible, meaning that its density, ρ is constant in both time and space. This simplifies Equation 2.12 into the incompressibility condition:

$$\boxed{\vec{\nabla} \cdot \vec{v} = 0.} \tag{2.13}$$

2.1.3 Conservation of momentum

While mass conservation is considered a Navier-Stokes equation, the name usually refers to the Navier-Stokes momentum equation. This is the equation at the heart of my computer code, and is derived by applying Newton's Second Law to the infinitesimal moving fluid element. Applying Newton's law in just the x-direction for now yields

$$F_x = ma_x. \tag{2.14}$$

Here, F_x is the total force in the x-direction, m is the mass of the fluid element, and a_x is the acceleration of the fluid element in the x-direction.

Dealing with only the left hand side for now, the total force accelerating a fluid element in the x-direction will be equal to the body force on the element plus the surface forces from the flow around it. The body force is the force pulling on each molecule of the fluid. This could be something like gravity, pulling down on each particle, or an electromagnetic force like the one in my experiment. This force affects each particle regardless of the motion of other particles around it and can be represented (in 2 dimensions) as force per unit area (f_x), a vector with the x-component

$$F_{x,\,body} = \int_V f_x \, dxdy. \tag{2.15}$$

The surface force can be further divided into shear, normal, and pressure surface forces. A normal surface force in the x-direction would be a force acting on the $x = 0$ and $x = dx$ faces of the element as depicted in Figure 2.1 (the y-z plane for a 3-dimensional fluid element). This force would be perpendicular to the face, pushing or pulling the box of fluid. In contrast, the x-component of shear will be a force in the x-direction acting on the $y = 0$ and $y = dy$ faces of the element (the x-y and x-z planes of a 3-dimensional fluid element) through viscous friction. This force is shown in Figure 2.1, with the $y = 0$ arrow pointing left and the $y = dx$ arrow pointing right. The directions are just to follow the convention that a positive velocity gradient increases in the positive direction. This creates a shear force in which the top is moving to the right and the bottom is moving left relative to overall motion of the fluid element. Finally, the pressure surface force pushes inwards on the fluid element from both sides. All of these forces are shown for a 2-dimensional fluid element in Figure 2.1.

Notice that each of the surface forces above acts on opposite sides of the fluid element, so naturally the net surface force will depend on the difference between the two sides, and therefore the gradient of pressure and velocity in our fluid.

Combining all of these forces, the net surface force per volume in the x-direction can be written

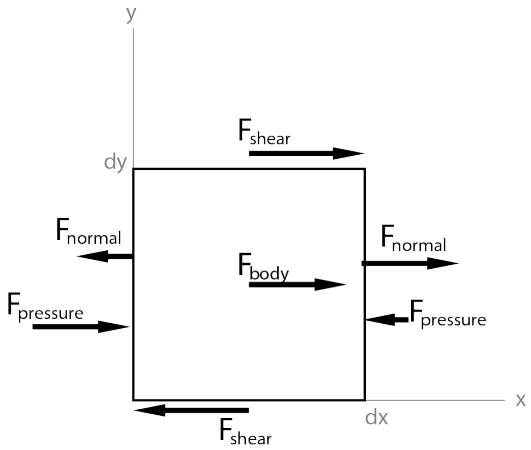


Figure 2.1: A diagram of the forces on an infinitesimal 2-dimensional fluid element.

$$\text{Net surface force in the x direction} \;=$$

$$\underbrace{\left[p - \left(p + \frac{\partial p}{\partial x}dx\right)\right]dy}_{\text{pressure difference}} + \underbrace{\left[\left(\tau_{xx} + \frac{\partial \tau_{xx}}{\partial x}dx\right) - \tau_{xx}\right]dy}_{\text{normal stress difference}} + \underbrace{\left[\left(\tau_{yx} + \frac{\partial \tau_{yx}}{\partial y}dy\right) - \tau_{yx}\right]dx}_{\text{shear stress difference}}.$$

Where τ_{ij} is the force in the j direction on the surface perpendicular to the i axis. After canceling and combining with the body force, this becomes

$$F_x = \left[-\frac{\partial p}{\partial x} + \frac{\partial \tau_{xx}}{\partial x} + \frac{\partial \tau_{yx}}{\partial y} + f_x\right] dxdy. \tag{2.16}$$

Combining this, Equation 2.6 and the mass per volume of a 2-dimensional infinitesimal fluid element, $m = \rho\, dxdy$, Newton's second law (in the x-direction) becomes

$$\underbrace{-\frac{\partial p}{\partial x} + \frac{\partial \tau_{xx}}{\partial x} + \frac{\partial \tau_{yx}}{\partial y} + f_x}_{\vec{F} \text{ per volume}} = \underbrace{\rho \frac{Dv_x}{Dt}}_{m\vec{a} \text{ per volume}}. \tag{2.17}$$

This is the x-component of Navier-Stokes momentum equation in conservative form, which will govern the motion of the fluid in my setup. All that is left now is to define the τ terms and then apply some assumptions we can make about our flow. This will allow us to rearrange the equation into an easier to use form.

2.1.4 Assumptions and rearranging

The first assumption we can make is that the fluid that I am using is Newtonian. This means the shear stress is proportional to the velocity gradient. For these fluids, Stokes found the τ elements are

$$\tau_{xx} = \lambda\vec{\nabla}\cdot\vec{v} + 2\mu\frac{\partial v_x}{\partial x} \tag{2.18}$$

and

$$\tau_{yx} = \mu\frac{\partial v_y}{\partial x} + \mu\frac{\partial v_x}{\partial y} \tag{2.19}$$

[5], where μ is a constant known as the viscosity of the fluid and λ is the bulk viscosity coefficient, which often stated to be $-\frac{2}{3}\mu$, and will drop out of our solution when we enforce incompressibility.

Because we can assume the fluid to be incompressible, Equation 2.13 states that $\vec{\nabla}\cdot\vec{v} = 0$. This eliminates the first part of the τ_{xx} term, reducing it to $\tau_{xx} = 2\mu\frac{\partial v_x}{\partial x}$. Taking the derivatives of these terms necessary for Equation 2.17 yields

$$\frac{\partial\tau_{xx}}{\partial x} + \frac{\partial\tau_{yx}}{\partial y} = 2\mu\frac{\partial^2 v_x}{\partial x^2} + \mu\frac{\partial^2 v_y}{\partial x\partial y} + \mu\frac{\partial^2 v_x}{\partial y^2}, \tag{2.20}$$

which can be rearranged and rewritten as

$$\mu\left(\frac{\partial^2 v_x}{\partial x^2} + \frac{\partial^2 v_x}{\partial y^2}\right) + \mu\frac{\partial}{\partial x}\left(\frac{\partial v_x}{\partial x} + \frac{\partial v_y}{\partial y}\right) = \mu\nabla^2 v_x + \mu\frac{\partial}{\partial x}\left(\vec{\nabla}\cdot\vec{v}\right). \tag{2.21}$$

Again, the divergence term can be eliminated by enforcing incompressibility, leaving only the Laplacian term.

Plugging equations 2.5 and 2.21 into Equation 2.17, and dividing by density, ρ, gives

$$\frac{\partial v_x}{\partial t} + (\vec{v}\cdot\vec{\nabla})v_x = -\frac{1}{\rho}\frac{\partial p}{\partial x} + \nu\nabla^2 v_x + \frac{1}{\rho}f_x, \tag{2.22}$$

where ν is the kinematic viscosity, equal to $\frac{\mu}{\rho}$. Finally, this can be applied to each dimension of vector $\vec{v}$ simultaneously by writing it as

$$\boxed{\frac{\partial\vec{v}}{\partial t} + (\vec{v}\cdot\vec{\nabla})\vec{v} = -\frac{1}{\rho}\nabla p + \nu\nabla^2\vec{v} + \frac{1}{\rho}\vec{f},} \tag{2.23}$$

the final form of the Navier-Stokes momentum equation. The last thing we have to do to make this equation usable in our code is to make it unitless through nondimensionalization. To do this, each variable will be rewritten according to scales that are intrinsically part of the Kolmogorov system. The length scale will be the wavelength of the sinusoidal forcing. Time scales will be determined by the velocities of the fluid, (U) using $U = \frac{\text{length}}{\text{time}}$, which in turn will allow us to determine mass using the viscosity (μ) using $\mu = \frac{\text{mass}}{\text{length time}}$. These scales can be combined to create the pressure and density scales from $p = \frac{\text{mass}}{\text{length time}^2}$ and $\rho = \frac{\text{mass}}{\text{length}^3}$ (here I am using 3-dimensional units for familiarity). Defining the nondimensional variables $v = v'\frac{L}{T}$, $x = x'L$, $t = t'T$, $p = p'\frac{M}{LT^2}$, $\rho = \frac{M}{L^3}$, $\nabla = \nabla'\frac{1}{L}$ and $\nu = \nu'\frac{L^2}{T}$, where L is the length scale, M the mass

scale, and T the time scale. Also note that $\vec{f}$ is force per volume, so $\vec{f} = \vec{f'}\frac{M}{T^2L^2}$. With these new nondimensionalized terms, 2.23 becomes

$$\left(\frac{L}{T^2}\right)\frac{\partial\vec{v'}}{\partial t'} + \left(\frac{L}{T^2}\right)(\vec{v'}\cdot\nabla')\vec{v'} = -\left(\frac{L}{T^2}\right)\nabla' p' + \left(\frac{L}{T^2}\right)\nu'\nabla'^2\vec{v'} + \left(\frac{L}{T^2}\right)\vec{f'}. \quad (2.24)$$

Dividing through by $\frac{L}{T^2}$, cancels all dimensions, leaving the flow parameter ν', which we will denote as one over the Reynolds number, giving the nondimensionalized Navier-Stokes equation:

$$\boxed{\frac{\partial\vec{v'}}{\partial t'} + (\vec{v'}\cdot\nabla')\vec{v'} = -\nabla' p' + \left(\frac{1}{Re}\right)\nabla'^2\vec{v'} + \vec{f'},} \quad (2.25)$$

The Reynolds number is unitless, and can be rewritten in terms of our scale parameters as $\frac{\nu}{UL}$, a unitless quantity which compares inertial and viscous effects.

2.2 Computational Methods

I am simulating 2-dimensional Kolmogorov flow using two different programs in *Mathematica*. The first is a time-independent code, meaning that it assumes the system is not changing in time. For this I started from a code posted on the Wolfram Blog, [10]. The second method does not assume time independence. It begins with no flow at all, and calculates the flow at each time-step for an arbitrary number of time-steps. For this code, I followed the methods from the University of Michigan's Computational Fluid Dynamics I course from 2001 [11], in which they wrote a time-dependent driven cavity flow solver in Matlab. Driven cavity flow is a common fluid dynamics problem in which an enclosed cavity of fluid is being driven by a moving boundary. It was extremely helpful to follow a recipe I knew would work, since these codes (the time-dependent ones especially) can be very temperamental and can easily become unstable, yielding impossibly huge results. Below, I explain the methods required to write each code.

2.2.1 Discretization and finite difference approximations

In order to get a computer to simulate a fluid dynamics problem, there are a few concepts that we need. These methods will allow the computer to approximate and solve complex functions to an arbitrary degree of accuracy. Firstly, discretization is a method around which all of my computation is based. The Navier-Stokes equation is extremely hard to solve analytically, except in some special cases, so in order to solve it on our 2-dimensional surface, we break the surface up into a grid of discrete points and solve it numerically. More points will give an answer of higher resolution, but will make the code more computationally demanding. The positions on the grid are represented as an $n \times n$ matrix, at each of which the equation will be solved.

The next tool that I am using is the finite difference approximation for derivatives. In the most basic sense, a derivative is the slope of a function at a single point. To

find this, we can use a Taylor expansion to approximate the value of a function f at a point just past x_0,

$$f(x_0 + dx) = f(x_0) + \left.\frac{df}{dx}\right|_{x_0} dx + O(dx^2). \tag{2.26}$$

From this we can approximate the derivative at x_0 as

$$\left.\frac{df}{dx}\right|_{x_0} \approx \frac{f(x_0 + dx) - f(x_0)}{dx}. \tag{2.27}$$

Since any continuous function can be represented by a Taylor series to arbitrary accuracy, we can extend these finite difference approximations out to be as accurate as needed. This process for approximating derivatives is called the finite difference method. Generally, the first order approximation is fine. We can use this to calculate first and second partial derivatives in two dimensions based solely off of nearby points. On our 2-dimensional grid, dx will be our grid spacing. Ignoring higher order terms, the finite difference approximations can be written as

$$\frac{\partial f_{i,j}}{\partial x} \approx \frac{f_{i+1,j} - f_{i,j}}{dx} \tag{2.28}$$

and

$$\frac{\partial^2 f_{i,j}}{\partial x^2} \approx \left(\frac{\partial f_{i+1,j}}{\partial x} - \frac{\partial f_{i,j}}{\partial x}\right)\frac{1}{dx} = \frac{f_{i+1,j} - 2f_{i,j} + f_{i-1,j}}{dx^2} \tag{2.29}$$

where $f_{i,j}$ represents f for the grid point at the in the i^{th} column (x-coordinate) and the j^{th} row (y-coordinate). Note that the second derivative is essentially the slope of the slopes on either side of x_0. The first derivative can easily be extended out to second order accuracy by averaging the derivatives on either side of x_0, making it second order accurate;

$$\frac{\partial f_{i,j}}{\partial x} \approx \frac{f_{i+1,j} - f_{i-1,j}}{2dx}. \tag{2.30}$$

These can be changed to y partials, or any other partial, by modifying the index associated with that variable (j for the y partial, instead of i). Equation 2.30 is called the midpoint formulation of $\frac{\partial f_{i,j}}{\partial x}$, since it finds this derivative using the values on either side of it. Equation 2.28 would be considered an endpoint approximation, since it approximates the derivative at a point using only points to one side of it. Approximations can be made to arbitrary accuracy for both midpoint and endpoint approximations.

For the time-independent code, I extended the accuracy out to fourth order terms in space using [12], which contains an algorithm to find the fourth order approximations, and a table of fourth order weights. For the time-dependent code, which is much more computationally intensive, I left it at first and second order, as is the standard procedure [13, 11, 14].

One final thing to mention is how points at the boundaries of the grid are dealt with. To create solid boundary walls like those at the edges of our experimental tray,

we would enforce a no-slip boundary condition. This means the points at the boundary are held fixed in space, with velocities set to zero. This accurately approximates a realistic solid boundary, as the adhesive forces holding molecules touching a boundary will force them to move at the same speed.

2.2.2 The time-independent solver

The first method I looked at to solve my system was a time-independent solver. This method relies on the assumption that the flow has reached a steady solution that is not changing in time. For this code, I modified the code from [10]. The original code was designed to solve the 2-dimensional driven cavity problem. To make it representative of Kolmogorov flow, I fixed the moving (driving) boundary so that it was stationary, and added a forcing term at each point on the grid. The forcing term I used is the one from Equation 1, which defines Kolmogorov flow, but it would be simple change this to any desired forcing. I also generated my own fourth order differential matrices, extending the ones from [13] out from second order and using the appropriate endpoint approximations on the boundaries, instead of using a pre-programmed *Mathematica* function. This code can be found in Appendix A.

The code works by discretizing the area into an $n \times n$ grid of arbitrary resolution. Each grid-point holds a unique x and y velocity value, and a pressure value in a stationary reference frame. These variables are written as a single n^2 long vector of variables, which can be re-fitted onto the grid by applying an indexing function [13];

$$g(i, j) = (j - 1)n + i. \tag{2.31}$$

The finite difference derivatives expressed above are written at each point, making it a linear combination of grid points around it. This means that for each point, the derivative operator can be written as an n^2 long vector, with finite difference coefficients attached to the grid points (put into vector form using Equation 2.31) around the point of interest. Taking the dot product of this n^2 long vector and an n^2 vector of variables on the grid will yield the derivative at that point. For example, the vector for a finite difference derivative in x at point p dotted with the x-velocity vector would yield $\frac{\partial v_x}{\partial x}|_p$.

To find the derivative at every point, we create an n^2 long derivative vector at for each of the n^2 points, and combine them into an $n^2 \times n^2$ derivative operator matrix. These operators act on a vector of variables through matrix multiplication, resulting in an n^2 vector of derivatives, which can then be indexed to an $n \times n$ grid of derivative values. The code I wrote (Appendix A) generates fourth-order accurate first and second derivative operator matrices, using fourth order accurate midpoint formulas [12] for the points in the bulk of the fluid, and the appropriate endpoint formulas near the edges. This makes finding derivatives on a discrete grid into the matrix problem below;

$$\mathbb{A}_x \vec{v_x} = \vec{a_x}, \tag{2.32}$$

where $\vec{v_x}$ is the vector of x-velocities at each point, length n^2, $\mathbb{A}_x$ is the $n^2 \times n^2$ x-derivative operator matrix, and $\vec{a_x}$ is a vector of $\frac{\partial v_x}{\partial x}$ values, also length n^2.

Since even at fourth order accuracy each derivative vector only has 5 terms [12], the entire matrix $\mathbb{D}_x$ will only have $5 \times n^2$ of the n^4 terms be non-zero. This means it will be a "sparse" matrix, and allows us to use the *SparseArray* command to create them, and some simplified techniques to deal with them.

Using these derivative approximations and assuming the flow does not change in time, the Navier-Stokes equation becomes an enormous set of solvable linear algebra equations.

Expanding the nondimensionalized Navier-Stokes equation, (Equation 2.25), the change in x-velocity with respect to time can be written

$$-\frac{\partial v_x}{\partial t} = v_x \left(\frac{\partial}{\partial x} v_x\right) + v_y \left(\frac{\partial}{\partial y} v_x\right) + \left(\frac{\partial}{\partial x} p\right) - \frac{1}{\mathrm{Re}} \left(\frac{\partial^2}{\partial x^2} v_x + \frac{\partial^2}{\partial y^2} v_x\right), \qquad (2.33)$$

with an analogous equation for $\frac{\partial v_x}{\partial t}$. Here Re is the Reynolds number, defined earlier, v_x, v_y and p are n^2 vectors representing x and y velocities and pressure at each point on the $n \times n$ grid, and the partial derivatives are given by the $n^2 \times n^2$ matrices described above.

For a steady solution, $-\frac{\partial v_x}{\partial t} = 0$. This means we are essentially finding the root (zeros) of a huge system of linear algebraic equations represented by the right hand side of the equation above.

After defining a unique variable at each grid point and creating the differential matrices, Equation 2.33 is defined. The boundary variables are then set to 0, to simulate the no-slip walls of the container. Finally, the roots are found using a built in *Mathematica* `FindRoot` function. The resulting values, v_x, v_y, and p can be plotted either as scalars on a grid in a contour plot, combined to create a vector field, or plotted in a line integral convolution, as explained in the results section.

2.2.3 The time-dependent solver

As the driving is increased, the laminar solution to the Kolmogorov flow loses stability and gives way to a vortex pattern [1, 2]. This pattern is initially steady in time, so there a time-independent code works just fine. As explained in the Introduction, Kolmogorov flow's initial bifurcation from its steady, laminar state resembles a supercritical pitchfork bifurcation [2], where the laminar flow becomes unsteady, and a new flow pattern replaces it as the stable state of the system. At higher fluid velocities, this steady pattern gives way to periodic and potentially chaotic flows [4]. At this point the time-independent code becomes ineffective, and will give incorrect results. The technique above becomes vastly more complex and computationally expensive when we add in a temporal dimension. I wrote a time-dependent code using methods described in [5], [1], and [11].

First, I need to the simplify the Navier-Stokes momentum equation into a form that is easier to deal with. By taking the curl of the entire equation, I can eliminate the pressure term, since the curl of a gradient is always zero. To simplify the result I

can define vorticity as

$$\vec{\omega} = \vec{\nabla} \times \vec{v} = \left(\frac{\partial v_y}{\partial x} - \frac{\partial v_x}{\partial y} \right) \hat{z}, \tag{2.34}$$

which is a vector, but points only in the $\hat{z}$ direction for flow confined to the x,y plane. After taking the curl, the nondimensionalized Navier-Stokes equation becomes

$$\frac{\partial \vec{\omega}}{\partial t} + (\vec{v} \cdot \vec{\nabla})\vec{\omega} - (\vec{\omega} \cdot \vec{\nabla})\vec{v} = \mathrm{Re}^{-1}\vec{\nabla}^2\vec{\omega} + (\vec{\nabla} \times \vec{f}). \tag{2.35}$$

To simplify further, we can define the stream function ψ as

$$v_x = \frac{\partial \psi}{\partial y} \tag{2.36}$$

and

$$v_y = -\frac{\partial \psi}{\partial x}. \tag{2.37}$$

The physical meaning of the stream function is like a path that a moving infinitesimal piece of fluid will follow. Isocontours of ψ act like flow "rails", where a particle riding on top of the fluid follows a path of constant ψ.

Now, we can rewrite the momentum Navier-Stokes equation in terms of only vorticity and stream functions. Plugging in the stream function, the third term on the left hand side of Equation 2.35 becomes

$$(\vec{\omega} \cdot \vec{\nabla})\vec{v} = \left(\frac{\partial v_y}{\partial x} - \frac{\partial v_x}{\partial y} \right) \frac{\partial}{\partial z}\vec{v} = 0, \tag{2.38}$$

since $\vec{v}$ does not depend on z. Inserting the stream function into the rest of the Navier-Stokes equation yields

$$\frac{\partial \vec{\omega}}{\partial t} + \frac{\partial \psi}{\partial y}\frac{\partial \vec{\omega}}{\partial x} - \frac{\partial \psi}{\partial x}\frac{\partial \vec{\omega}}{\partial y} = \mathrm{Re}^{-1}\vec{\nabla}^2\vec{\omega} + \vec{f_\nabla}, \tag{2.39}$$

where I have expanded out the second term on the left hand side of Equation 2.35 in terms of the stream function, and defined a new forcing term $\vec{f_\nabla}$ as $(\vec{\nabla} \times \vec{f})$. Changing the Navier-Stokes equation into this form is advantageous because we eliminated the pressure term. Now instead of having v_x, v_y, and p as variables, we only have $\vec{\omega}$ and ψ. While $\vec{\omega}$ is technically a vector, for 2-dimensional flow, it points exclusively in the $\hat{z}$ direction, and so can be treated like a scalar.

From here the problem breaks down into a few different steps. For each time-step, first we find the stream function from the vorticity calculated in the previous time-step, then we find the updated vorticity function.

Updating the stream function using SOR

Note that from equations 2.34, 2.36, and 2.37, we can write the vorticity as

$$\vec{\omega} = \left(-\frac{\partial^2 \psi}{\partial x^2} - \frac{\partial^2 \psi}{\partial y^2} \right) \hat{z} = \mathbb{A}\psi, \tag{2.40}$$

so we can get ψ from a known $\vec{\omega}$. This is an example of an elliptic equation, and is the governing equation of subsonic, incompressible flows [5, pp 84–85].

Rewriting this using finite difference second derivatives as in Equation 2.29 yields

$$\omega_{i,j} = -\frac{\psi_{i+1,j} - 2\psi_{i,j} + \psi_{i-1,j}}{dx^2} - \frac{\psi_{i,j-1} - 2\psi_{i,j} + \psi_{i,j-1}}{dy^2}. \tag{2.41}$$

Assuming an evenly spaced grid ($dx = dy$) we can combine terms to get

$$\omega_{i,j}dx^2 = -\psi_{i+1,j} - \psi_{i-1,j} - \psi_{i,j-1} - \psi_{i,j-1} + 4\psi_{i,j}. \tag{2.42}$$

As in the time-independent section, we are solving a large system of linear equations. Like before, the finite difference derivative matrices are sparse. Unlike last time, we need to solve this problem many times (once per time step), so it is much faster to use an iterative method called successive over-relaxation, or SOR.

Successive over-relaxation is an iterative method for solving linear systems of equations. *Mathematica*'s `LinearSolve` or `FindRoot` functions solve systems of equations by exact methods, such as Gaussian elimination. While these would still work, they are much slower than iterative methods, which can be incredibly fast, but do not find exact solutions. Iterative methods make an initial guess and iterate, quickly converging on the solution until it is within some specified accuracy.

As explained in [13, pp 279–281], the basic method here is to decompose $\mathbb{A}$ from Equation 2.40 into three more easily invertible matrices; a matrix $\mathbb{D}$ containing only elements along the diagonal of $\mathbb{A}$, the strictly upper triangular matrix $\mathbb{U}$ containing only elements above the diagonal, and a lower triangular matrix $\mathbb{L}$, containing elements below the diagonal.

In this specific iterative method, the matrix is updated using

$$(\mathbb{D} - \beta\mathbb{L})\psi^{m+1} = ((1-\beta)\mathbb{D} + \beta\mathbb{U})\psi^m + \beta(\omega dx^2) \tag{2.43}$$

[13], where ψ and ω are n^2 long vectors, representing each grid point indexed using Equation 2.31, and $\mathbb{D}$, $\mathbb{U}$, and $\mathbb{L}$ are all $n^2 \times n^2$ square matrices. In Equation 2.43, m represents the iteration number. Finally, β is the weighting term, which ensures the solutions actually converge, and is set between 1 and 2 for SOR. For a single one of the linear equations shown in Equation 2.44, it is clear that point $\psi_{i,j}$ will fall along the diagonal of $\mathbb{A}$. From the indexing function, Equation 2.31, it is also clear that $\psi_{i+1,j}$ and $\psi_{i,j+1}$ will be located to the right of the diagonal, and therefore fall in the upper triangular matrix, while $\psi_{i-1,j}$ and $\psi_{i,j-1}$ will be to the left, and therefore fall in the lower.

$$\begin{bmatrix} & & & \\ [\ldots & A_{(i,j),(i,j)} & \cdots & \ldots] \\ & & & \end{bmatrix} \begin{bmatrix} \vdots \\ \psi_{i,j} \\ \vdots \\ \vdots \end{bmatrix} = \omega_{i,j}dx^2 \tag{2.44}$$

This means we can rewrite the SOR iteration much more simply as

$$4\psi_{i,j}^{m+1} - \beta\psi_{i-1,j}^{m+1} - \beta\psi_{i,j-1}^{m+1} = (1-\beta)4\psi_{i,j}^m + \beta\psi_{i+1,j}^m + \beta\psi_{i,j+1}^m + \beta dx^2\omega_{i,j}^m. \tag{2.45}$$

which, solving for $\psi_{i,j}^{m+1}$ can be rearranged into

$$\psi_{i,j}^{m+1} = \beta\frac{1}{4}\left(\psi_{i-1,j}^{m+1} + \psi_{i,j-1}^{m+1} + \psi_{i+1,j}^{m} + \psi_{i,j+1}^{m} + \omega_{i,j}^{m}dx^2\right) + (1-\beta)\psi_{i,j}^{m}. \tag{2.46}$$

Since the stream functions are being solved in order, for the $\psi_{i,j}^{m+1}$ step, $\psi_{i-1,j}^{m+1}$ and $\psi_{i,j-1}^{m+1}$ will have already been solved, allowing us to use these values in our derivation. This is the final form of SOR used in my code. The program iterates this to get an updated stream function until Equation 2.46 is satisfied to within a specified degree of accuracy, set by a parameter called MaxError.

Boundary Conditions

I applied and analyzed two different sets of boundary conditions in my code. First is solid boundary walls, which could easily be implemented by following the same procedure from [11], since solid boundaries are a necessary part of the driven cavity problem. For this, all we have to do is pin down the vorticity at the boundaries, and we can update it everywhere else. As mentioned earlier, to create solid boundary walls, we can assume that at points touching the wall (or infinitely close), the velocity of the fluid will be the same as the velocity of the wall. For our fixed walls, this means the stream function, the x and y derivatives of which correspond to y and x velocities, is constant all along the boundary. Using Equation 2.40, it is clear that if vorticity is fixed all along the left wall, $\frac{\partial^2\psi}{\partial y^2}$ will be 0 all along that wall, leaving

$$\vec{\omega}_{\text{left}} = -\frac{\partial^2\psi}{\partial x^2}. \tag{2.47}$$

To get $\vec{\omega}_{\text{left}}$, I can follow the method explained in [11]. Here, they follow a routine almost identical to the finite difference derivation, writing a Taylor approximation for ψ one column away from the boundary as

$$\psi_{2,j} = \psi_{1,j} + \frac{\partial\psi_{1,j}}{\partial x}dx + \frac{\partial^2\psi_{1,j}}{\partial x^2}\frac{dx^2}{2} + \text{higher order terms}. \tag{2.48}$$

Noting that $\frac{\partial\psi_{1,j}}{\partial x} = -v_y$ and $\psi_{1,j}$ are both zero at the boundary, combining with Equation 2.47 above, I can rearrange this into

$$\vec{\omega}_{\text{left}} = -\frac{2}{dx^2}\psi_{2,j}. \tag{2.49}$$

This is a function we can solve, since $\psi_{2,j}$ is not a boundary stream function, and has been calculated by SOR. This approximation is repeated for all of the boundaries, allowing for usable $\vec{\omega}_{\text{boundaries}}$ in the update. This is important because it gives the code an updated starting point to work from.

Although the tray does have solid boundary walls, I am focusing on a section in the middle, mostly unaffected by boundary conditions. Here, the fluid can be approximated as part of an infinite plane, which can be more accurately modeled

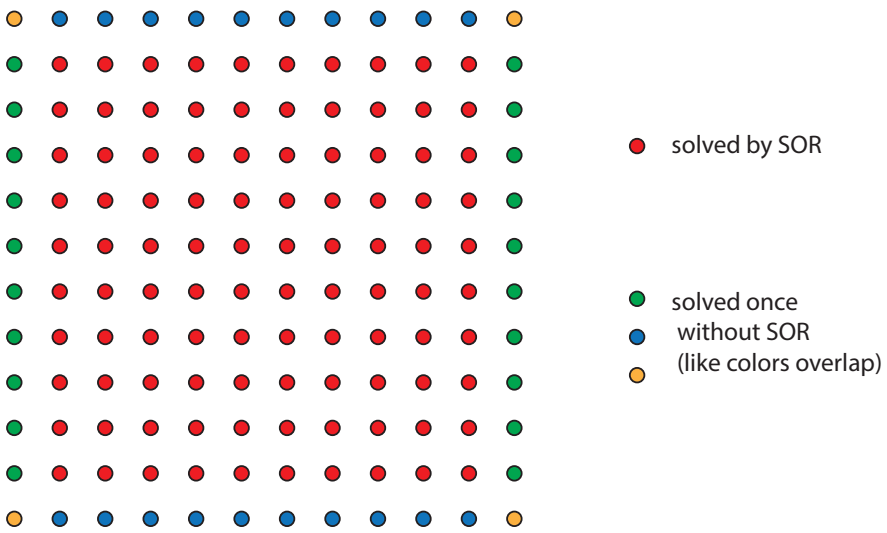


Figure 2.2: A figure describing how the stream function was calculated for periodic boundary conditions.

by periodic boundary conditions. This means a particle of fluid can exit out the right and come back in the left side, maintaining its same speed and trajectory. I eventually implemented these boundary conditions on the sides, top and corners of the system. The resulting code can be found in Appendix B. Basically I followed the same procedure as at points in the bulk of the flow, but wrote three separate procedures for each group of boundary points; one for the top and bottom, one for the left and right, and one for the corners. The code has one grid point of overlap on all of the boundaries, meaning points on the leftmost boundary are equal to those directly across on the rightmost boundary. This is shown in Figure 2.2. Because of the overlapping opposite sides, the boundary stream function only needed to be calculated on two of the boundaries, once for top and bottom, and once for left and right, and one more time for all four corners. Since this only needs to be done $2n+1$ times, I do not use an iterative method, and instead simply solve Equation 2.42 for $\psi_{i,j}$ at the boundary points.

Update

Lastly, the vorticity update is calculated by solving 2.39 for $\frac{\partial \vec{\omega}}{\partial t}$ and using standard finite difference approximations, equations 2.28 and 2.29. This creates the update

$$\frac{\partial \vec{\omega}}{\partial t} = -\frac{\partial \psi}{\partial y}\frac{\partial \vec{\omega}}{\partial x} + \frac{\partial \psi}{\partial x}\frac{\partial \vec{\omega}}{\partial y} + \mathrm{Re}^{-1}\vec{\nabla}^2\vec{\omega} + \vec{f_\nabla}, \tag{2.50}$$

which, after applying finite difference derivative approximations and removing the forcing term that is added to the vorticity later with the update, becomes

$$\omega_{i,j} \text{ update} =$$
$$\frac{\psi^m_{i,j-1} - \psi^m_{i,j+1}}{2dy}\frac{\omega^m_{i+1,j} - \omega^m_{i-1,j}}{2dx} + \frac{\psi^m_{i+1,j} - \psi^m_{i-1,j}}{2dx}\frac{\omega^m_{i,j+1} - \omega^m_{i,j-1}}{2dy}$$
$$+\mathrm{Re}^{-1}\frac{\omega^m_{i+1,j} - 2\omega^m_{i,j} + \omega^m_{i-1,j}}{dx^2}\frac{\omega^m_{i,j+1} - 2\omega^m_{i,j} + \omega^m_{i,j+1}}{dy^2}.$$

I create a separate updated variable to ensure that the newly updated points are not used yet. I also created separate update terms for the sides and corners. Each update is then multiplied by dt (since the updates are actually $\frac{\partial\vec{\omega}}{\partial t}$) and added to the old vorticity along with the curl of the forcing term. The sides, corners and center points are updated separately, and the resulting grid of vorticity and stream functions are stored. Finally, the time step is updated by adding dt, and the vorticity and stream functions are set to the new, updated values for the next time step. The last feature I added was a progress-meter, which prints the percentage of the code that has been completed at certain intervals.

The actual code could be cleaned up, especially the periodic boundary conditions. I did not investigate the possibility of using iterative methods in more places, but this should be considered if the code is being run at higher resolutions where it will take longer. I feel like there is still a huge amount of work I could do to improve this simulation, and hope to continue to upgrade and refine it in the future. Both codes and experiment yielded promising results, as described in Chapter 3.

Chapter 3

Results

3.0.4 Experimental

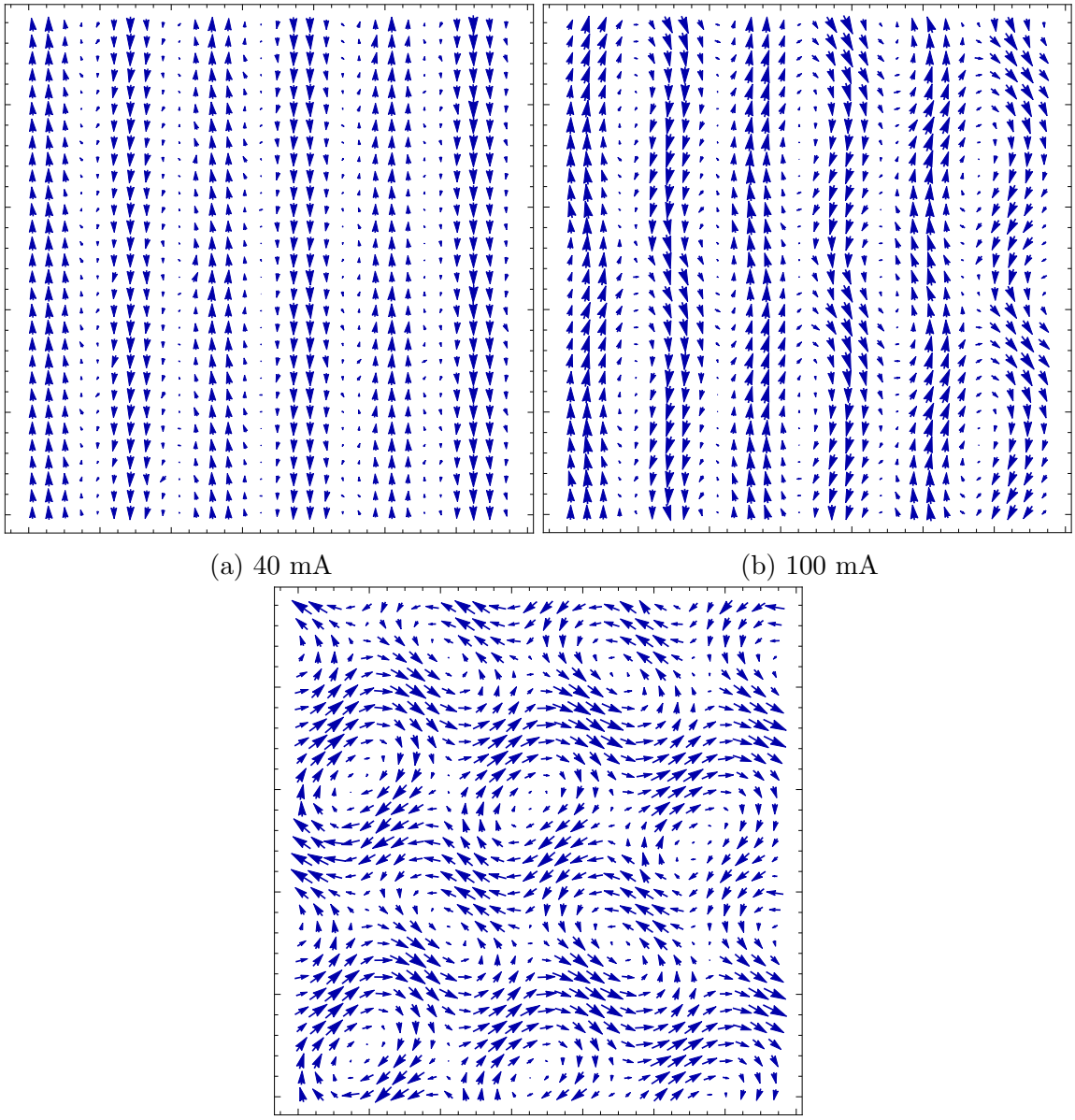
From each set of experimental data, the PIV program generated an enormous flattened matrix, giving the x and y velocities at each point. Vector fields depicting these results are shown in Figure 3.1. These fields clearly show the motion of the fluid before and after the bifurcation, with the initial laminar solution mirroring the arrangement of magnets, and later being replaced by stable interlocking vortices as the forcing is increased.

To get more quantitative results, I calculated the root mean square (rms) of x-velocities in *Mathematica* to gauge the deviation from the laminar flow, which mimics the forcing and is only in the $\hat{y}$ direction. I repeated this process at each electrical current, with the end result being a single rms x-velocity value for each electrical current measurement. To make this into a more universally understandable and useful set of data, I can approximate the Reynolds number (Re) associated with each current.

Here I get Reynolds numbers using a handful of parameters from my experimental setup and series of approximations. I follow the steps laid out by Kelley and Ouellette [1] to arrive at the correct expression for Re. As mentioned in the Introduction, Reynolds number is the relative importance of fluid inertia and viscous damping. "Relative importance" can be translated to the ratio of these two terms. Viscous damping is represented by the damping time scale $\frac{L^2}{\nu}$, where L is our length scale (here the period of the forcing, or experimentally, the distance between two rows of magnets), and ν is the kinematic viscosity measured with a viscometer and shown in Table 3.1. Fluid inertia is captured by the advective timescale, equal to $\frac{L}{U}$, where the velocity scale, U can be set to the total rms speed of the system, $\langle \vec{v} \cdot \vec{v} \rangle^{\frac{1}{2}}$, with the triangular brackets representing a mean [1]. Therefore, the Reynolds number can be expressed as

$$Re = \frac{\frac{L^2}{\nu}}{\frac{L}{U}} = \frac{UL}{\nu}. \tag{3.1}$$

Note that the units of kinematic viscosity (ν) are $\frac{\text{length}^2}{\text{time}}$, making the Reynolds number unitless, as expected. To use this method, I needed to make sure the measurements



(a) 40 mA

(b) 100 mA

(c) 180 mA

Figure 3.1: Experimental vector fields at different driving current values. These were created in *Mathematica* using the PIV data. The laminar solution (at 40 mA) deviates from straight channels (100 mA) and eventually forms a pattern of interlocking vortices (180 mA).

that I used have units that cancel correctly. Firstly, the velocities the PIV software returned were not in conventional units. These values were in pixels per frame, since the program measured shifts in pixels, over the time elapsed between two frames. To correct this I measured the number of pixels across one period of the forcing pattern in a frame of the laminar solution. Because it is difficult to see where exactly the center of a channel is, I measured over two periods to get an average width. The

period of forcing is a known distance, and allows me to find the number of pixels per meter by measuring the magnet spacing. Using this, along with the fact that we were recording at 30 frames per second, the velocity vectors returned by the PIV software can be appropriately scaled by

$$\frac{\text{meters}}{\text{second}} = \frac{\text{pixels}}{\text{frame}} \times \underbrace{\frac{\text{meters}}{\text{pixel}} \times \frac{\text{frames}}{\text{second}}}_{\text{velocity correction term}}. \tag{3.2}$$

I measured the kinematic viscosity of the working fluid using a Cannon-Fenske Routine Viscometer. This involved measuring the time for a fluid to travel through a small capillary, and comparing the result with the same measurement for water. From this, I found the working fluid to have a kinematic viscosity 2.54 times that of water. The accepted kinematic viscosity (ν) of water is given by $\nu = \frac{\eta}{\rho}$ where η is the dynamic viscosity, (or coefficient of viscosity), and ρ is the density. Obtaining these values from [15], I found the ν_{water} to be $1.00 \times 10^{-6} \frac{\text{m}}{\text{s}}$, giving a value of $2.54 \times 10^{-6} \frac{\text{m}}{\text{s}}$ for the kinematic viscosity of the working fluid. This is shown in Table 3.1.

Table 3.1: A table of measurements taken from the experimental setup necessary to determine Re.

Kinematic viscosity	$2.54 \times 10^{-6}\ \frac{\text{m}^2}{\text{s}}$
Pixels per meter	$18,632.6$
Frames per second	30
Length scale (L)	0.038 m

Using Equation 3.1, the correctly scaled velocity and length measurements, and the measured viscosity, I calculated the Reynolds numbers. The parameters used to calculate these are shown in Table 3.1. The calculated Reynolds numbers, along with the corresponding currents are shown in Table 3.2. I plotted rms x-velocity as a function of both the electrical currents and the estimated Reynolds numbers. This yielded the plots shown in Figure 3.2.

These agree with the predictions presented by Kelley and Ouellette [1]. The critical Reynolds number, the one at which the bifurcation occurs, has been found to be around 61 [1], or 70 [2] in a similar experiment using driven soap films. The damping effects of friction on the bottom of the tray are most likely the cause of these discrepancies [1, 16, 2], as these can have a huge effect. A very simple model of linear stability analysis predicts a critical Reynolds number of $\sqrt{2}$, or ≈ 1 [4, 2], far off from any experimental results. Our experimentally measured value, around 85, seems to be fairly consistent with the previous experimental results. This result could definitely be improved by using a smoother tray, or a thin layer of oil as done in [17].

The current data seems to scale like a square root function, as expected [1]. Assuming the bifurcated data scaled in the form $y = x^{\alpha}$, I found the exponent (α) by

Table 3.2: A table of estimated Reynolds numbers with corresponding current values.

Current (mA)	Reynolds number
20	23.7
40	37.6
60	64.9
80	85.0
100	100.5
110	106.4
120	112.4
130	116.4
140	126.1
150	130.8
160	135.5
180	141.8
200	150.9

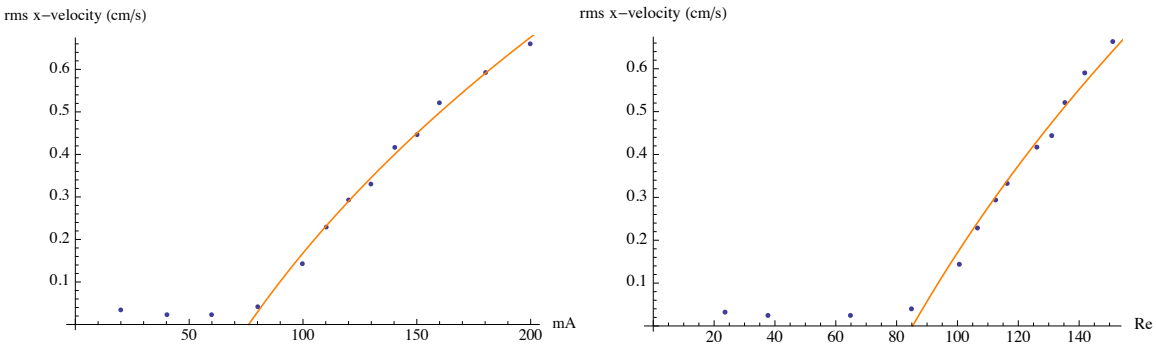


Figure 3.2: A bifurcation diagram made from experimental data. The left plot shows rms x-velocities as a function of forcing current. The right plot shows rms x-velocities as a function of Reynolds number. Both plots show best fit curves given by equations 3.3 and 3.4.

plotting the natural logarithms of rms velocities, current, and Reynolds numbers, and calculated the slope of the resulting linear functions. This gave a best fit power of $\alpha = 0.353$ for the current data, and $\alpha = 0.2175$ for the Reynolds number data. These were linearly fitted to create the functions

$$-1.662 + 0.3595x^{0.353} \tag{3.3}$$

and

$$-4.844 + 1.8421x^{0.2175} \tag{3.4}$$

for rms x-velocities as functions of current and Reynolds numbers respectively. These

yield R-squared values of 0.9946 and 0.9856. The functions are plotted with the experimental data in Figure 3.2.

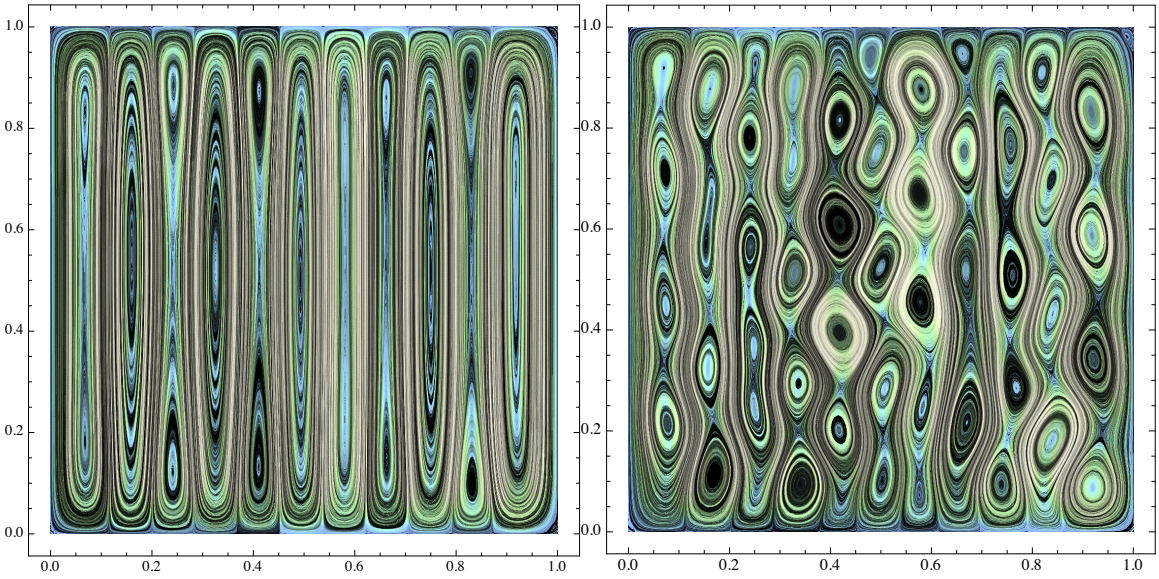
This implies that the Kolmogorov bifurcation is indeed a supercritical pitchfork. While a root is still a good fit, the Reynolds number plot seems to scale almost linearly. The results from [1] also seem to follow this trend. This error could maybe be attributed to the friction of the fluid on the bottom of the tray. This friction increases with velocity [2], so because we found Reynolds numbers experimentally using observed velocity, we might expect higher forcing to yield disproportionately small Reynolds numbers. This could cause the curve to appear more linear, and is something worth investigating further.

3.0.5 Numerical

I ran both the time-independent and time-dependent codes describe in Chapter 2, the latter with both no-slip and periodic boundary conditions.

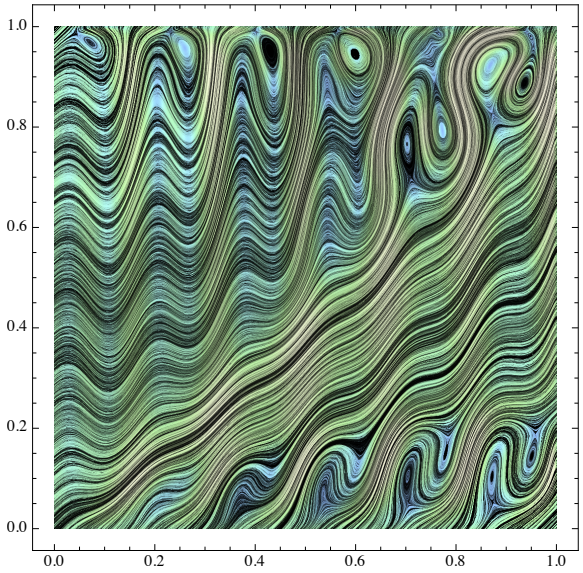
The time-independent code captures the initial bifurcation very well, but could not capture turbulent solutions, or instabilities and periodic solutions as they began to form. Nonetheless, the time-independent code does show the qualitative motion of the fluid fairly well. Figure 3.3 shows line integral convolution plots of the time-independent data. These were done using *Mathematica*'s `LineIntegralConvolutionPlot` function, as done by Mokhasi [10]. This introduces static of a specified color scheme into the grid, and integrates each point along the path of the fluid, dragging the color with it and creating easy to visualize fluid flows. Note that the time-independent code does not allow us to determine if the image furthest to the right depicts a steady, stable solution, a periodic or chaotic regime. This is something we can deal with using the time-dependent code.

For the time-dependent code, I determined the optimal set of parameters, all of which can be found in Table 3.3. I chose each of these after testing a wide range of options. My goal was to make the code run as quickly as possible without losing too much accuracy. I determined the ideal grid size to be 30 by 30. This seemed like enough resolution to accurately capture the motion of the fluid over four full periods of the sinusoidal forcing pattern, while usually not taking more than 30 minutes for the code to run in its entirety. I confirmed that this was enough resolution by trying a smaller grid, and making sure the macroscopic motion of the fluid progressed the same way. This is the same strategy I used to determine most of the other parameters. For time-step size (dt), I ran several tests over a variety of values for dt. I did this for the most turbulent case, since faster moving fluid requires smaller time-steps to capture its motion. I found the threshold at which a larger dt either causes an overflow, or changes the behavior of the simulation to be above 0.0001 but below 0.001, so I used the former value as my time-step size. By the same process, I varied the number of time-steps until I reached a point where the code had settled down. I then changed the Kv value, which is proportional to the inverse of the Reynolds number (Re^{-1}) and forcing amplitudes to find the point at which the bifurcation occurred. While Kv does take the place of $\frac{1}{Re}$ in the code, we can only say that the Re value here is proportional to the actual Reynolds number, since forcing amplitude affects velocity



(a) Low Re

(b) Higher Re



(c) Highest Re

Figure 3.3: Line integral convolution plots of the time-independent code. (a) The laminar solution, visible at low Re. (b) The fluid after its initial bifurcation, and (c) The flow at an even higher Reynolds number. Notice that by being time-independent, (c) is unable to capture periodic or chaotic motion, and therefore generates an impossible result.

timescales, and is not captured in Kv.

Each code generated a stream function and vorticity value at every grid point. Plots of vorticity at different values of Kv (and at later time-steps) can be found in Figure 3.4 for the solid boundary case and 3.5 for the periodic boundary case. Note that the the laminar case looks like the initial forcing pattern, since the vorticity of

Table 3.3: A table of parameters used for the time-dependent code.

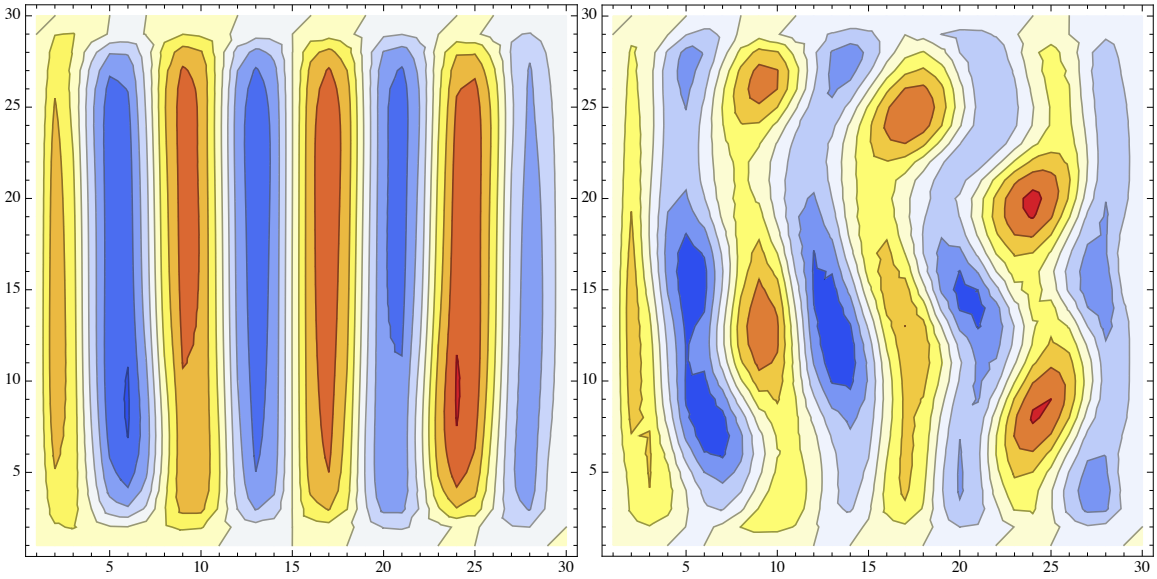
Parameter	Value for solid boundaries	Value for periodic boundaries
Number of gridpoints (nxy)	30	30
Length of a time-step (dt)	0.0001	0.0001
Number of time-steps	3000	5000
Maximum SOR iterations	100	100
Kv ($\propto$ to Re^{-1})	variable: from 0.1 to 0.4	variable: from 0.02 to 0.2

Equation 1 will also give stripes. In each of these plots, red (positive vorticity) represents counter-clockwise rotation, and blue (negative vorticity) represents clockwise rotation. This means the dark bands in the laminar vorticity plots actually represent the areas between channels. The differences in band thickness and magnitude in the laminar cases is a result of the sine wave pattern not being perfectly described by 30 grid points. Even though the forcing is exactly four periods, the grid points can fall directly on the peaks, or off to the side, resulting in slightly different forcing across the grid. This certainly introduces some error into the simulations, and could be remedied by increasing the resolution.

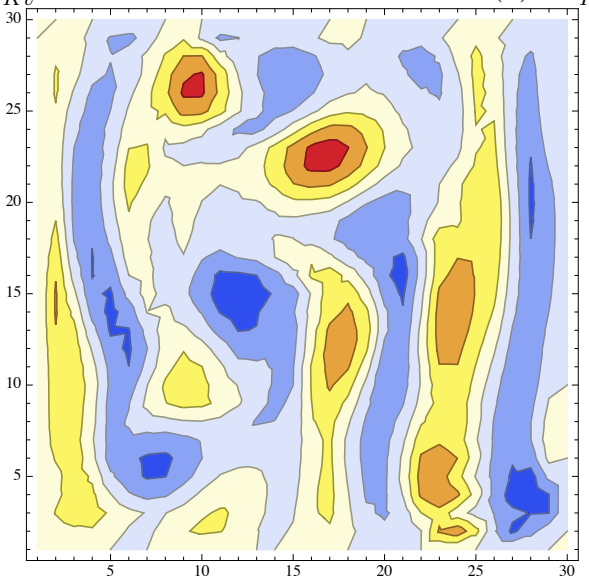
From the stream functions, which are also generated by the simulation, I could extract x and y velocities using equations 2.36 and 2.37. To take the derivative, I used the fourth order differential matrices from the time-independent code, described in Chapter 2. One check to make sure the code is performing realistically is to check the incompressibility condition. As stated in Chapter 2, for an incompressible fluid, $\nabla \cdot \vec{v} = \frac{\partial v_x}{\partial x} + \frac{\partial v_y}{\partial y} = 0$. Figure 3.6 is a plot of this value at each point, all of which are very close to zero. Looking at the formula, this result should be obvious, since we are essentially taking $\frac{\partial}{\partial y}\frac{\partial}{\partial x}\psi$ twice, which must always be zero.

With both velocities found, I can calculate the rms x-velocities, and plot them as a function of $\frac{1}{Kv}$. Figures 3.7 and 3.8 show bifurcation diagrams of this data. I used the points immediately after the bifurcation to calculate exponents for a function of the form $y = x^{\alpha}$ by the method described earlier. I got exponents of 0.42 for the no-slip bounded code, and 0.40 for the periodic bounded code. These functions are plotted for the steady bifurcation region in Figure 3.7 and 3.8. The plots exhibit the same behavior as the experimental data with exponents between 0 and 1, confirming that the time-dependent codes are performing correctly. The scaling factor here is pretty close to the experimental data, with error most likely due to friction in the experiment as described earlier.

The behavior of these plots at higher Reynolds numbers ($Re \propto \frac{1}{Kv}$) is also very revealing. In the no-slip boundary plot, Figure 3.7, points seem to fluctuate upwards and downward in time without trending towards any particular value, implying that the behavior here is either periodic or turbulent. The periodic bounded code seems to generally display points in temporal order, implying that these values are trending towards some value.



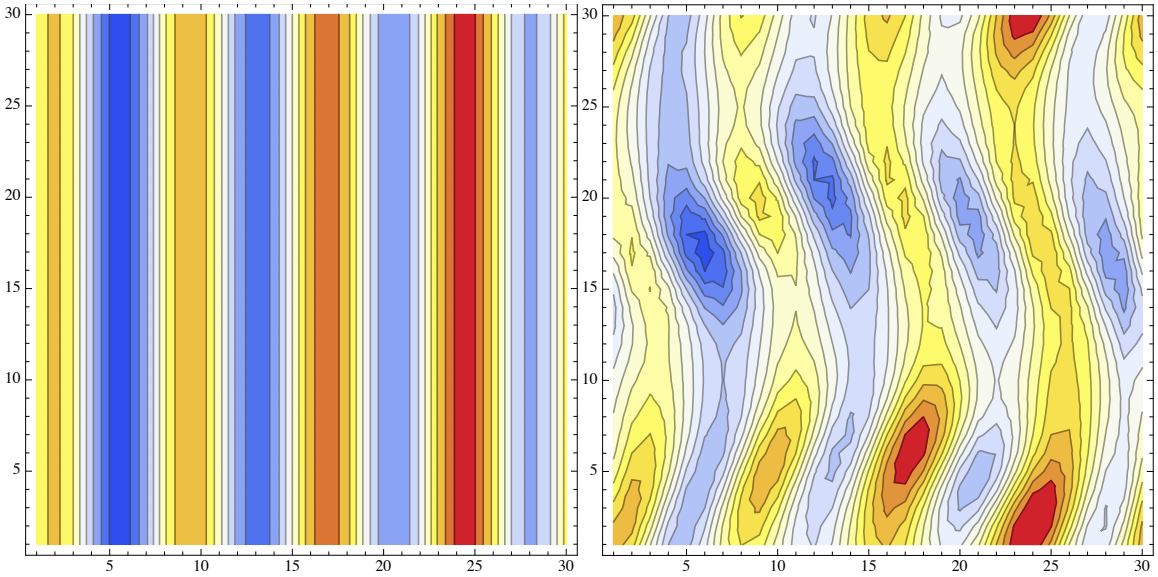
(a) at $\frac{1}{Kv} = 2$ (b) at $\frac{1}{Kv} = 5$



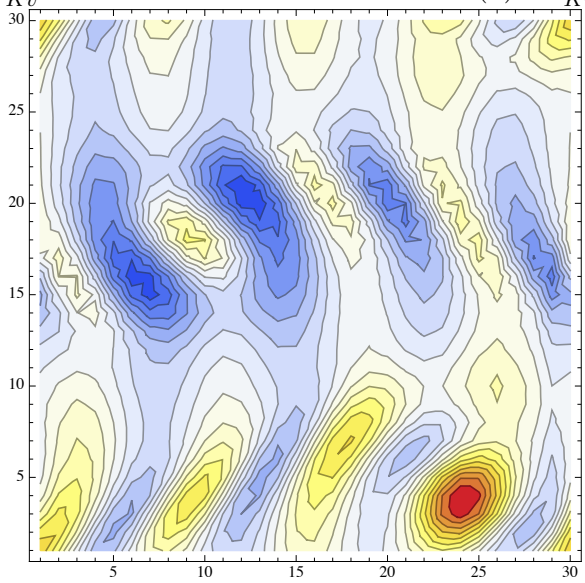
(c) at $\frac{1}{Kv} = 10$

Figure 3.4: Plots of vorticity generated from the time-dependent code with no-slip boundary conditions. Each image is the result after 2,000 time-steps. Here, red represents a positive vorticity, and blue a negative vorticity. Each plot shows the results at a different value of $\frac{1}{Kv}$, a number proportional to the Reynolds number.

These observations seem to be a direct result of having solid boundaries. The bounded fluid code settled into turbulent or steady motion much more quickly than its periodic counterpart, and did so at lower Reynolds numbers. I suspect this is caused by the fact that the bounded code is subjected to many more perturbations, as the flow runs into, and is changed by the boundaries. It is also possible that given enough time, even the extremely chaotic looking regions in the no-slip bounded code



(a) at $\frac{1}{Kv} = 5$ (b) at $\frac{1}{Kv} = 11.1$



(c) at $\frac{1}{Kv} = 20$

Figure 3.5: Plots of vorticity generated from the time-dependent code using periodic boundary conditions. Each image is the result after 10,000 time-steps. Again, each plot shows the results at a different value of $\frac{1}{Kv}$, a number proportional to the Reynolds number.

would have eventually settled down into some other form of solution. Behavior after extremely long times is definitely something worth investigating.

Because the experiment only looks at an area at the center of the forcing pattern, the periodic bounded, time-dependent code with periodic boundary conditions mirrors it well. The bifurcation diagrams generated from the experimental data, Figure 3.2, and from the periodic bounded code, Figure 3.8, closely resemble each other, with

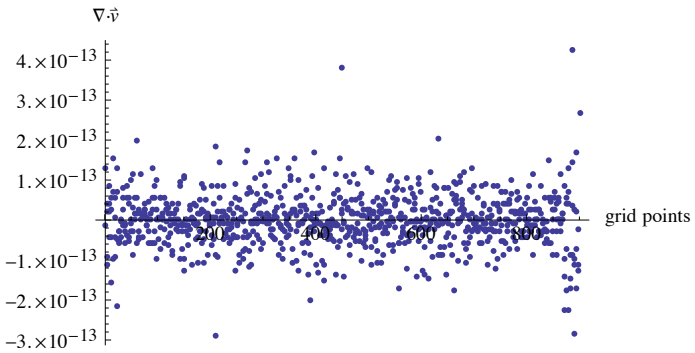


Figure 3.6: A plot of the divergence of velocity at each point. These all being close to zero means the code enforces the incompressibility condition.

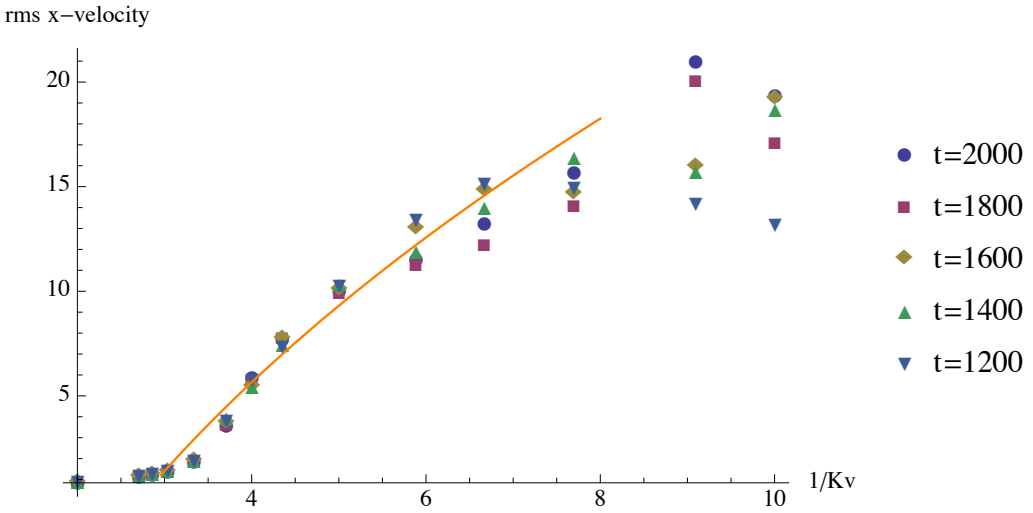


Figure 3.7: A bifurcation diagram for the no-slip boundary condition's code, showing rms x-velocities as function of $\frac{1}{Kv}$, after different numbers of timesteps.

a sharp bifurcation point. These two curves both scale as square root functions after the bifurcation point, as described in Kelley and Ouellette's experiment [1]. This confirms that the flow undergoes a supercritical bifurcation as stated in [2]. While we did not seem to reach the Hopf bifurcation that is expected to appear after the steady regime [4], both the experiment and code should be able to demonstrate these phenomena at higher Reynolds numbers.

The periodic bounded data could be improved by running out to further timesteps, to see if it was in any sort of limit cycle. The data in Figure 3.8 shows rms x-velocities trending in one direction in time, implying that they are going to stabilize, or are moving in a slow limit cycle. Running the code out further in time could clarify this. More timesteps could also be an improvement to the time-dependent, no-slip bounded code, to confirm if the motion at later time steps is chaotic or some sort of periodic limit cycle.

The next step in comparing these results would be matching a Reynolds number

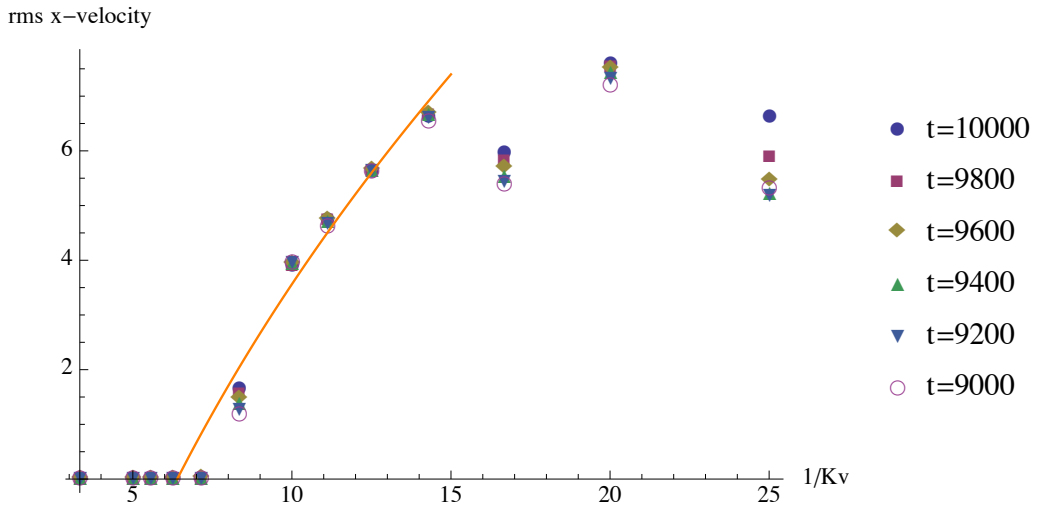


Figure 3.8: A bifurcation diagram for the periodic bounded code, showing rms x-velocities as function $\frac{1}{Kv}$, after different numbers of timesteps.

to the numerical data, and comparing this to the experiment. We would expect a significant discrepancy in the scaling of the bifurcation diagrams, which would mostly be attributed to friction on the bottom of the tray, allowing us to estimate and add a friction parameter to the code. This parameter would be linearly dependent on velocity [4, 2], a constant term we could estimate.

Chapter 4

Conclusions

I successfully demonstrated a 2-dimensional Kolmogorov flow bifurcation using an experimental and numerical approach. The experiment yielded results consistent with [1, 4, 2] qualitatively matching up with the patterns generated by their setups. The experiment showed a bifurcation, shifting from the laminar solution to a set of stable vortices.

For the experimental portion, I used Particle Image Velocimetry to analyze the flow. I was able to show the motion of the fluid qualitatively in vector fields shown in Figure 3.1 and quantitatively in a bifurcation diagram of root mean square x-velocities, shown in Figure 3.2. The Reynolds numbers were approximated, leading to a critical Reynolds number of around 85. This is reasonably close to the results found in [1] and [2] of 61 and 70 respectively. The discrepancies in these measurements are most likely due to damping from friction on the bottom of the tray, a source of error that is difficult to avoid.

Numerically the problem was studied using two different numerical schemes. Every variation of the code produced good results, each having its own strengths and weaknesses. The time-independent code, modified from [10], was by far the fastest, and generated very good results for the initial bifurcation of the system, as shown in Figure 3.3. However, due to its time-independent nature, it could not show any periodic or turbulent solutions, making its results of limited use without a time-dependent code to confirm that the solutions are actually stable.

The time-dependent code, followed the technique from [11] and was implemented using methods from [13]. This was able to show more interesting behavior, but was much slower and therefore had to be run at lower resolution. Two different boundary conditions were studied, both of which demonstrated the initial bifurcation. The bounded code seemed to fall into periodic solutions and turbulence much more quickly, while the periodic bounded code slowly settled to more stable flows.

There is still a lot of improvements that can be made to this experiment, and additional extensions worth investigating. The code works well, but could definitely be cleaned up to run more quickly.

The experiment could be made more accurate by implementing a more complicated tray setup [8, 17]. New forcing patterns could also be investigated. For example, concentric squares of magnet polarity with one electrode at the middle, and the other

around the edge of the tray would yield anti-parallel concentric circles (or squares) of forcing. This could create some very interesting new patterns.

Armed with the time-dependent code, any future extensions could be tested fairly quickly using the simulation, before reconstructing the whole experiment. The two pieces of this thesis together provide an excellent set of tools for analyzing any sort of two-dimensional fluid flow.

While no piece of this project is perfect, it served as an excellent introduction to fluid dynamics. I was able to study a subject I had never really seen before, and make sense of it using the physics and math I had learned over the last few years. I honestly had a really great time moving through this turbulent thesis process, and hope to keep working on wild and exciting fluid systems far into the future.

Appendix A

Differential Matrices and Time-independent Code

This appendix shows a time-independent Kolmogorov solver code, modified from [10], and differential matrices, modified from [13] using finite difference approximations from [12]. The code here generates $\frac{\partial}{\partial x}$, $\frac{\partial}{\partial y}$, $\frac{\partial^2}{\partial x^2}$, and $\frac{\partial^y}{\partial y^2}$ differential operator matrices.

Time-independent code

Below, grid and Reynolds number (Rey) defined (again, forcing amplitude and other parameters are not taken into account, so the actual value of Rey here is not too meaningful).

```
In[54]:= pts = 25;
         Nx = N[pts];
         Ny = N[pts];
         dx = 1.0/(pts - 1);
         dy = 1.0/(pts - 1);
         nx = Table[a * dx, {a, 0, (Nx - 1)}];
         grid = Flatten[Table[{nx[[a]], nx[[b]]}, {a, 1, Ny}, {b, 1, Nx}], 1];
         Rey = 200;
```

Differential matrices created.

```
In[2]:= top[x_] := 0;
        bottom[x_] := 0;
        left[y_] := 0;
        right[y_] := 0;
In[23]:= {DX, bx} = D1Xmat[Nx, Ny, dx, dy, left, right];
         {DY, by} = D1Ymat[Nx, Ny, dx, dy, top, bottom];
         {D2X, b2x} = D2Xmat[Nx, Ny, dx, dy, left, right];
         {D2Y, b2y} = D2Ymat[Nx, Ny, dx, dy, top, bottom];
```

Variables defined (for x-velocity, y-velocity, pressure, and forcing).

```
In[27]:= nL = Length[grid];
         varsU = Table[Unique[u$], {nL}];
         varsV = Table[Unique[v$], {nL}];
         varsP = Table[Unique[p$], {nL}];
         varsF = Flatten[Table[i, {i, 1, pts}, {j, 1, pts}]];
```

Navier-Stokes momentum equations defined (for $\frac{\partial v}{\partial t}$), along with the incompressibility condition.

```
In[42]:= eqnsU = varsU * (DY.varsU) + varsV * (DX.varsU) + DY.varsP -
             (1 / Rey) * ((D2Y + D2X).varsU);
         eqnsV = varsU * (DY.varsV) + varsV * (DX.varsV) + DX.varsP - (1 / Rey) * ((D2Y + D2X).varsV) +
             100 * Sin[(6 Pi / 20) varsF];
         eqnsCont = DY.varsU + DX.varsV;
```

No-slip boundaries defined.

```
In[39]:= boundaries =
           DeleteDuplicates@
            Flatten[Position[grid, #] & /@ {{0., y_}, {1., y_}, {x_, 0.}, {x_, 1.}}];
         topBndry = Flatten@Position[grid, {x_, 1.}];
         bndryComp = Complement[boundaries, topBndry];
```

Boundaries applied.

```
In[42]:= eqnsU[[boundaries]] = varsU[[boundaries]];
         eqnsU[[topBndry]] = varsU[[topBndry]];
         eqnsV[[boundaries]] = varsV[[boundaries]];
         eqnsCont[[topBndry[[1]]]] = varsP[[topBndry[[1]]]];
```

Equations joined into a single expression.

```
In[46]:= govExpr = Join[eqnsU, eqnsV, eqnsCont];
         allVars = Join[varsU, varsV, varsP];
```

Equations solved vie root finding (to find where $\frac{\partial v}{\partial t}$, and divergence are zero).

```
sol = allVars /. FindRoot[govExpr, Thread[{allVars, 1}]];
```

Solutions put into plot-able form

```
{usoltemp, vsoltemp, psoltemp} = Partition[sol, nL];
usol = Interpolation@Join[grid, Transpose@List@usoltemp, 2];
vsol = Interpolation@Join[grid, Transpose@List@vsoltemp, 2];
psol = Interpolation@Join[grid, Transpose@List@psoltemp, 2];
```

Solutions plotted in a line integral convolution plot.

```
In[136]:= LineIntegralConvolutionPlot[
            {{usol[x, y], vsol[x, y]}, {"noise", 1000, 1000}},
            {x, 0, 1}, {y, 0, 1}, LineIntegralConvolutionScale → 3,
            ColorFunction → "AlpineColors"]
```

Differential matrices

In[10]:=
```
D1Xmat[Nx_, Ny_, dx_, dy_, left_, right_] :=
 Module[{g, retmat, bside, index, indey, i, i2, i3, i4, j, k, k2,
   k3, k4},
  g[n_, m_] := (m - 1) * Nx + n;

  retmat = SparseArray[{}, {Nx Ny, Nx Ny}, 0.0];
  bside = Table[0.0, {i, 1, Nx Ny}];

  For[indey = 1, indey ≤ Ny, indey = indey + 1,
   j = g[1, indey];
   k = g[2, indey];
   k2 = g[3, indey];
   k3 = g[4, indey];
   k4 = g[5, indey];

   retmat[[j, j ]] += (-25.0 / 12) / dx;
   retmat[[j, k ]] += 4.0 / dx;
   retmat[[j, k2 ]] += -3.0 / dx;
   retmat[[j, k3 ]] += (4.0 / 3) / dx;
   retmat[[j, k4 ]] += (-1.0 / 4) / dx;
   bside[[ j ]] += left[indey dy] / dx;

   i = g[1, indey];
   j = g[2, indey];
   k = g[3, indey];
   k2 = g[4, indey];
   k3 = g[5, indey];

   retmat[[j, i ]] += (-1.0 / 4) / dx;
   retmat[[j, j ]] += (-5.0 / 6) / dx;
   retmat[[j, k ]] += (1.5) / dx;
   retmat[[j, k2 ]] += (-1.0 / 2) / dx;
   retmat[[j, k3 ]] += (1.0 / 12) / dx;

   bside[[ j ]] += left[indey dy] / dx;

   For[index = 3, index ≤ Nx - 2, index = index + 1,
    i2 = g[index - 2, indey];
    i = g[index - 1, indey];
    j = g[index, indey];
    k = g[index + 1, indey];
    k2 = g[index + 2, indey];
    retmat[[j, i2]] += (1.0 / 12) / dx;
    retmat[[j, i]] += (-2.0 / 3) / dx;
    retmat[[j, j]] += 0.0;
    retmat[[j, k]] += (2.0 / 3) / dx;
    retmat[[j, k2]] += (-1.0 / 12) / dx;
   ];

   i3 = g[Nx - 4, indey];
   i2 = g[Nx - 3, indey];
   i = g[Nx - 2, indey];
   j = g[Nx - 1, indey];
   k = g[Nx, indey];

   retmat[[j, k ]] += (1.0 / 4) / dx;
   retmat[[j, j ]] += (5.0 / 6) / dx;
   retmat[[j, i ]] += (-1.5) / dx;
   retmat[[j, i2 ]] += (1.0 / 2) / dx;
   retmat[[j, i3 ]] += (-1.0 / 12) / dx;


   i4 = g[Nx - 4, indey];
   i3 = g[Nx - 3, indey];
   i2 = g[Nx - 2, indey];
   i = g[Nx - 1, indey];
   j = g[Nx, indey];
   retmat[[j, i4 ]] += (1.0 / 4) / dx;
   retmat[[j, i3 ]] += (-4.0 / 3) / dx;
   retmat[[j, i2 ]] += 3.0 / dx;
   retmat[[j, i ]] += -4.0 / dx;
   retmat[[ j, j]] += (25.0 / 12) / dx;
   bside[[ j]] += -right[indey dy] / dx;

  ];
  Return[{retmat, bside}];
 ]
```

```
D1Ymat[Nx_, Ny_, dx_, dy_, top_, bottom_] :=
 Module[{g, retmat, bside, index, indey, i, i2, i3, i4, j, k, k1,
   k2, k3, k4},
  g[n_, m_] := (m - 1) * Nx + n;

  retmat = SparseArray[{}, {Nx Ny, Nx Ny}, 0.0];
  bside = Table[0.0, {i, 1, Nx Ny}];

  For[index = 1, index ≤ Nx, index = index + 1,
   j = g[index, 1];
   k = g[index, 2];
   k2 = g[index, 3];
   k3 = g[index, 4];
   k4 = g[index, 5];

   retmat[[j, j]] += (-25.0 / 12) / dy;
   retmat[[j, k]] += 4.0 / dy;
   retmat[[j, k2]] += -3.0 / dy;
   retmat[[j, k3]] += (4.0 / 3) / dy;
   retmat[[j, k4]] += (-1.0 / 4) / dy;
   bside[[j]] += bottom[index dx] / dy;

   i = g[index, 1];
   j = g[index, 2];
   k = g[index, 3];
   k2 = g[index, 4];
   k3 = g[index, 5];

   retmat[[j, i]] += (-1.0 / 4) / dy;
   retmat[[j, j]] += (-5.0 / 6) / dy;
   retmat[[j, k]] += (1.5) / dy;
   retmat[[j, k2]] += (-1.0 / 2) / dy;
   retmat[[j, k3]] += (1.0 / 12) / dy;

   bside[[j]] += bottom[index dx] / dy;



   For[indey = 3, indey ≤ Ny - 2, indey = indey + 1,

    i2 = g[index, indey - 2];
    i = g[index, indey - 1];
    j = g[index, indey];
    k = g[index , indey + 1];
    k2 = g[index , indey + 2];
    retmat[[j, i2]] += (1.0 / 12) / dy;
    retmat[[j, i]] += (-2.0 / 3) / dy;
    retmat[[j, j]] += 0.0;
    retmat[[j, k]] += (2.0 / 3) / dy;
    retmat[[j, k2]] += (-1.0 / 12) / dy;

   ];

   i3 = g[index, Ny - 4];
   i2 = g[index, Ny - 3];
   i = g[index, Ny - 2];
   j = g[index, Ny - 1];
   k = g[index, Ny];
   retmat[[j, i3]] += (-1.0 / 12) / dy;
   retmat[[j, i2]] += (1.0 / 2) / dy;
   retmat[[j, i]] += (-1.5) / dy;
   retmat[[j, j]] += (5.0 / 6) / dy;
   retmat[[j, k]] += (1.0 / 4) / dy;
   bside[[j]] += -top[index dx] / dy;

   i4 = g[index, Ny - 4];
   i3 = g[index, Ny - 3];
   i2 = g[index, Ny - 2];
   i = g[index, Ny - 1];
   j = g[index, Ny];
   retmat[[j, i4]] += (1.0 / 4) / dy;
   retmat[[j, i3]] += (-4.0 / 3) / dy;
   retmat[[j, i2]] += 3.0 / dy;
   retmat[[j, i]] += -4.0 / dy;
   retmat[[j, j]] += (25.0 / 12) / dy;
   bside[[j]] += -top[index dx] / dy;



  ];
  Return[{retmat, bside}];
 ]
```

```
D2Xmat[Nx_, Ny_, dx_, dy_, left_, right_] :=
 Module[{g, retmat, bside, index, indey, i, i2, i3, i4, i5, j, k, k2, k3, k4, k5},
  g[n_, m_] := (m - 1) * Nx + n;

  retmat = SparseArray[{}, {Nx Ny, Nx Ny}, 0.0];
  bside = Table[0.0, {i, 1, Nx Ny}];

  For[indey = 1, indey ≤ Ny, indey = indey + 1,
   j = g[1, indey];
   k = g[2, indey];
   k2 = g[3, indey];
   k3 = g[4, indey];
   k4 = g[5, indey];
   k5 = g[6, indey];

   retmat[[j, j ]] += (15.0 / 4) / dx^2;
   retmat[[j, k ]] += (-77.0 / 6) / dx^2;
   retmat[[j, k2 ]] += (107.0 / 6) / dx^2;
   retmat[[j, k3 ]] += (-13.0) / dx^2;
   retmat[[j, k4 ]] += (61.0 / 12) / dx^2;
   retmat[[j, k5 ]] += (-5.0 / 6) / dx^2;
   bside[[ j ]] += -left[indey dy] / dx^2;

   i = g[1, indey];
   j = g[2, indey];
   k = g[3, indey];
   k2 = g[4, indey];
   k3 = g[5, indey];
   k4 = g[6, indey];

   retmat[[j, i ]] += (5.0 / 6) / dx^2;
   retmat[[j, j ]] += (-5.0 / 4) / dx^2;
   retmat[[j, k ]] += (-1.0 / 3) / dx^2;
   retmat[[j, k2 ]] += (7.0 / 6) / dx^2;
   retmat[[j, k3 ]] += (-1.0 / 2) / dx^2;
   retmat[[j, k4 ]] += (1.0 / 12) / dx^2;
   bside[[ j ]] += -left[indey dy] / dx^2;

   For[index = 3, index ≤ Nx - 2, index = index + 1,
    i2 = g[index - 2, indey];
    i = g[index - 1, indey];
    j = g[index, indey];
    k = g[index + 1, indey];
    k2 = g[index + 2, indey];
    retmat[[j, i2]] += (-1.0 / 12) / dx^2;
    retmat[[j, i]] += (4.0 / 3) / dx^2;
    retmat[[j, j]] += (-5.0 / 2) / dx^2;
    retmat[[j, k]] += (4.0 / 3) / dx^2;
    retmat[[j, k2]] += (-1.0 / 12) / dx^2;
   ];

   i4 = g[Nx - 5, indey];
   i3 = g[Nx - 4, indey];
   i2 = g[Nx - 3, indey];
   i = g[Nx - 2, indey];
   j = g[Nx - 1, indey];
   k = g[Nx, indey];

   retmat[[j, i4]] += (1.0 / 12) / dx^2;
   retmat[[j, i3 ]] += (-1.0 / 2) / dx^2;
   retmat[[j, i2 ]] += (7.0 / 6) / dx^2;
   retmat[[j, i ]] += (-1.0 / 3) / dx^2;
   retmat[[ j, j]] += (-5.0 / 4) / dx^2;
   retmat[[ j, k]] += (5.0 / 6) / dx^2;
   bside[[ j]] += -right[indey dy] / dx^2;

   i5 = g[Nx - 5, indey];
   i4 = g[Nx - 4, indey];
   i3 = g[Nx - 3, indey];
   i2 = g[Nx - 2, indey];
   i = g[Nx - 1, indey];
   j = g[Nx, indey];
   retmat[[j, i5]] += (-5.0 / 6) / dx^2;
   retmat[[j, i4]] += (61.0 / 12) / dx^2;
   retmat[[j, i3 ]] += (-13.0) / dx^2;
   retmat[[j, i2 ]] += (107.0 / 6) / dx^2;
   retmat[[j, i ]] += (-77.0 / 6) / dx^2;
   retmat[[ j, j]] += (15.0 / 4) / dx^2;
   bside[[ j]] += -right[indey dy] / dx^2;
  ];
  Return[{retmat, bside}];
 ]
```

```
D2Ymat[Nx_, Ny_, dx_, dy_, top_, bottom_] :=
 Module[{g, retmat, bside, index, indey, i, i2, i3, i4, i5, j, k, k2, k3, k4, k5},
  g[n_, m_] := (m - 1) * Nx + n;

  retmat = SparseArray[{}, {Nx Ny, Nx Ny}, 0.0];
  bside = Table[0.0, {i, 1, Nx Ny}];

  For[index = 1, index ≤ Nx, index = index + 1,

   j = g[index, 1];
   k = g[index, 2];
   k2 = g[index, 3];
   k3 = g[index, 4];
   k4 = g[index, 5];
   k5 = g[index, 6];
   retmat[[j, j ]] += (15.0 / 4) / dy^2;
   retmat[[j, k ]] += (-77.0 / 6) / dy^2;
   retmat[[j, k2 ]] += (107.0 / 6) / dy^2;
   retmat[[j, k3 ]] += (-13.0) / dy^2;
   retmat[[j, k4 ]] += (61.0 / 12) / dy^2;
   retmat[[j, k5 ]] += (-5.0 / 6) / dy^2;
   bside[[ j ]] += -bottom[index dx] / dy^2;

   i = g[index, 1];
   j = g[index, 2];
   k = g[index, 3];
   k2 = g[index, 4];
   k3 = g[index, 5];
   k4 = g[index, 6];
   retmat[[j, i ]] += (5.0 / 6) / dy^2;
   retmat[[j, j ]] += (-5.0 / 4) / dy^2;
   retmat[[j, k ]] += (-1.0 / 3) / dy^2;
   retmat[[j, k2 ]] += (7.0 / 6) / dy^2;
   retmat[[j, k3 ]] += (-1.0 / 2) / dy^2;
   retmat[[j, k4 ]] += (1.0 / 12) / dy^2;
   bside[[ j ]] += -bottom[index dx] / dy^2;

   For[indey = 3, indey ≤ Ny - 2, indey = indey + 1,
    i2 = g[index, indey - 2];
    i = g[index, indey - 1];
    j = g[index, indey];
    k = g[index , indey + 1];
    k2 = g[index , indey + 2];
    retmat[[j, i2]] += (-1.0 / 12) / dy^2;
    retmat[[j, i]] += (4.0 / 3) / dy^2;
    retmat[[j, j]] += (-5.0 / 2) / dy^2;
    retmat[[j, k]] += (4.0 / 3) / dy^2;
    retmat[[j, k2]] += (-1.0 / 12) / dy^2;
   ];

   i4 = g[index, Ny - 5];
   i3 = g[index, Ny - 4];
   i2 = g[index, Ny - 3];
   i = g[index, Ny - 2];
   j = g[index, Nx - 1];
   k = g[index, Nx];
   retmat[[j, i4]] += (1.0 / 12) / dy^2;
   retmat[[j, i3 ]] += (-1.0 / 2) / dy^2;
   retmat[[j, i2 ]] += (7.0 / 6) / dy^2;
   retmat[[j, i ]] += (-1.0 / 3) / dy^2;
   retmat[[ j, j]] += (-5.0 / 4) / dy^2;
   retmat[[ j, k]] += (5.0 / 6) / dy^2;
   bside[[ j]] += -top[index dx] / dy^2;

   i5 = g[index, Ny - 5];
   i4 = g[index, Ny - 4];
   i3 = g[index, Ny - 3];
   i2 = g[index, Ny - 2];
   i = g[index, Ny - 1];
   j = g[index, Nx];
   retmat[[j, i5]] += (-5.0 / 6) / dy^2;
   retmat[[j, i4]] += (61.0 / 12) / dy^2;
   retmat[[j, i3 ]] += (-13.0) / dy^2;
   retmat[[j, i2 ]] += (107.0 / 6) / dy^2;
   retmat[[j, i ]] += (-77.0 / 6) / dy^2;
   retmat[[ j, j]] += (15.0 / 4) / dy^2;
   bside[[ j]] += -top[index dx] / dy^2;
  ];
  Return[{retmat, bside}];
 ]
```

Appendix B

Time-dependent Code

The time-dependent code, created by following techniques from [11] using methods from [13]. The code shown here is the version in which I use periodic boundaries.

```
In[1]:= Vorticity[st0_, vort0_, dt_, Nt_, Kv_, Maxit_] :=
 Module[{it, t, Nxy, dxy, retvort, retst, i, j, k, m, j0, k0, jj, kk, j3, k3, weq, vort,
   upvort, st, upst, B, Errr, Maxerror, Fx, sweq, cweq, j4, j5},
  (*First, everything is defined*)
  Maxerror = 0.001;
  B = 1.5;
  t = 0.0;
  Nxy = Length[st0];
  dxy = 1.0 / (Nxy - 1);
  vort = vort0;
  st = st0;
  upvort = vort0;
  upst = st0;
  weq = Table[0, {a, 1, Nxy}, {b, 1, Nxy}];
  sweq = Table[0, {a, 1, Nxy}, {b, 1, 2}];
  retvort = Table[0, {a, 1, Nt}];
  retst = Table[0, {a, 1, Nt}];
  Fx = Table[Sin[(8 Pi / Nxy) b], {a, 1, Nxy}, {b, 1, Nxy}];
  Errr = 0.0;
  (*Begin timestep iteration*)
  For[i = 1, i ≤ Nt, i ++,
   For[it = 1, it ≤ Maxit, it ++,
    weq = st;
    (*Find center stream function by SOR*)
    For[j = 2, j ≤ (Nxy - 1), j ++,
     For[k = 2, k ≤ (Nxy - 1), k ++,
      st[[j, k]] =
       (0.25) * B * (st[[j + 1, k]] + st[[j - 1, k]] + st[[j, k + 1]] + st[[j, k - 1]] +
          (dxy ^ 2) * vort[[j, k]]) + (1.0 - B) * st[[j, k]];
     ];
    ]

    Errr = 0.0;
    For[j0 = 1, j0 ≤ (Nxy), j0 ++,
     For[k0 = 1, k0 ≤ (Nxy), k0 ++,
      Errr = Errr + Abs[weq[[j0, k0]] - st[[j0, k0]]]
     ];];
    If[Errr ≤ Maxerror, Break[]];

   ];
   (*Find boundary and corner stream function*)
   For[m = 2, m ≤ Nxy - 1, m ++,
    (*left*)
    st[[1, m]] =
     (0.25) * (st[[2, m]] + st[[Nxy - 1, m]] + st[[1, m + 1]] + st[[1, m - 1]] + dxy ^ 2 * vort[[1, m]]);
```

```
  (*right*)
  st[[Nxy, m]] = st[[1, m]];
  (*top*)
  st[[m, 1]] =
   (0.25) * (st[[m + 1, 1]] + st[[m - 1, 1]] + st[[m, 2]] + st[[m, Nxy - 1]] + dxy^2 * vort[[m, 1]]);
  (*bottom*)
  st[[m, Nxy]] = st[[m, 1]];
 ];
 (*corners*)
 st[[1, 1]] =
  (0.25) * (st[[2, 1]] + st[[Nxy - 1, 1]] + st[[1, 2]] + st[[1, Nxy - 1]] + dxy^2 * vort[[1, 1]]);
 st[[Nxy, 1]] = st[[1, 1]];
 st[[1, Nxy]] = st[[1, 1]];
 st[[Nxy, Nxy]] = st[[1, 1]];
 (*Calculate vorticity updates*)
 For[j4 = 2, j4 ≤ (Nxy - 1), j4++,
  (*left and right*)
  sweq[[j4, 1]] =
   -0.25 * ((st[[1, j4 + 1]] - st[[1, j4 - 1]]) * (vort[[2, j4]] - vort[[Nxy - 1, j4]]) -
        (st[[2, j4]] - st[[Nxy - 1, j4]]) * (vort[[1, j4 + 1]] - vort[[1, j4 - 1]])) / (dxy^2) +
    Kv * (vort[[2, j4]] + vort[[Nxy - 1, j4]] + vort[[1, j4 + 1]] + vort[[1, j4 - 1]] -
        4.0 * vort[[1, j4]]) / (dxy^2);
  (*top and bottom*)
  sweq[[j4, 2]] =
   -0.25 * ((st[[j4, 2]] - st[[j4, Nxy - 1]]) * (vort[[j4 + 1, Nxy]] - vort[[j4 - 1, Nxy]]) -
        (st[[j4 + 1, Nxy]] - st[[j4 - 1, Nxy]]) * (vort[[j4, 2]] - vort[[j4, Nxy - 1]])) / (dxy^2) +
    Kv * (vort[[j4 + 1, Nxy]] + vort[[j4 - 1, Nxy]] + vort[[j4, 2]] + vort[[j4, Nxy - 1]] -
        4.0 * vort[[j4, Nxy]]) / (dxy^2);
 ];
 (*corners*)
 cweq =
  -0.25 * ((st[[1, 2]] - st[[1, Nxy - 1]]) * (vort[[1, Nxy]] - vort[[Nxy - 1, Nxy]]) -
       (st[[2, Nxy]] - st[[Nxy - 1, Nxy]]) * (vort[[1, 2]] - vort[[1, Nxy - 1]])) / (dxy^2) +
   Kv * (vort[[2, Nxy]] + vort[[Nxy - 1, Nxy]] + vort[[1, 2]] + vort[[1, Nxy - 1]] - 4.0 * vort[[1, Nxy]]) /
     (dxy^2);
 (*center points*)
 For[jj = 2, jj ≤ (Nxy - 1), jj++,
  For[kk = 2, kk ≤ (Nxy - 1), kk++,
   weq[[jj, kk]] =
    -0.25 * ((st[[jj, kk + 1]] - st[[jj, kk - 1]]) * (vort[[jj + 1, kk]] - vort[[jj - 1, kk]]) -
         (st[[jj + 1, kk]] - st[[jj - 1, kk]]) * (vort[[jj, kk + 1]] - vort[[jj, kk - 1]])) / (dxy^2) +
     Kv * (vort[[jj + 1, kk]] + vort[[jj - 1, kk]] + vort[[jj, kk + 1]] + vort[[jj, kk - 1]] -
         4.0 * vort[[jj, kk]]) / (dxy^2);
  ];
 ];
 (*Apply updates*)
 (*sides*)
 For[m = 2, m ≤ Nxy - 1, m++,
  upvort[[m, 1]] = vort[[m, 1]] + dt * sweq[[m, 2]] + Fx[[m, 1]];
  upvort[[m, Nxy]] = upvort[[m, 1]];
  upvort[[1, m]] = vort[[1, m]] + dt * sweq[[m, 1]] + Fx[[1, m]];
  upvort[[Nxy, m]] = upvort[[1, m]];
 ];
 (*corners*)
 upvort[[1, 1]] = vort[[1, 1]] + dt * cweq + Fx[[1, 1]];
 upvort[[Nxy, 1]] = upvort[[1, 1]];
 upvort[[1, Nxy]] = upvort[[1, 1]];
 upvort[[Nxy, Nxy]] = upvort[[1, 1]];
 (*center*)
 For[j3 = 2, j3 ≤ (Nxy - 1), j3++,
  For[k3 = 2, k3 ≤ (Nxy - 1), k3++,

   upvort[[j3, k3]] = vort[[j3, k3]] + dt * weq[[j3, k3]] + Fx[[j3, k3]];
  ];
 ];
 (*Update timesteps, retain vorticities and stream functions, maybe print percent done*)
 t = t + dt;
 vort = upvort;
 retvort[[i]] = vort;
 retst[[i]] = st;
 If[Mod[i, 500] == 0, Print[i / Nt * 100.0]];
];
Return[{retvort, retst}];
]
```

References

[1] D. H. Kelley and N. T. Ouellette, "Using particle tracking to measure flow instabilities in an undergraduate laboratory experiment," *American Journal of Physics*, vol. 79, pp. 267–273, March 2011.

[2] J. M. Burgess, C. Bizon, W. D. McCormick, J. B. Swift, and H. L. Swinney, "Instability of the Kolmogorov flow in a soap film," *Physical Review E: Statistical, Nonlinear, and Soft Matter Physics*, vol. 60, pp. 715 – 721, July 1999.

[3] S. H. Strogatz, *Nonlinear Dynamics and Chaos: With Applications to Physics, Biology, Chemistry, and Engineering*. Westview Press, 1994.

[4] A. Thess, "Instabilities in two-dimensional spatially periodic flow. Part I: Kolmogorov flow," *Physics of Fluids A: Fluid Dynamics*, vol. 4, p. 1385, July 1992.

[5] J. A. J. D. G. Degrez, E. Dick, R. Grundmann, and J. Vierendeels, *Computational Fluid Dynamics: An Introduction*. Springer-Verlag Berlin Heidelberg, third edition ed., 2009.

[6] M. Raffel, C. E. Willert, S. T. Wereley, and J. Kompenhans, *Particle Image Velocimetry: A Practicle Guide*. Springer Science+Business Media, 2nd edition ed., 2007.

[7] A. M. Obukhov, "Kolmogorov flow and laboratory simulation of it," *Russian Math. Surveys*, vol. 38, no. 4, pp. 113–126, 1983.

[8] B. Suri, J. Tithof, R. M. Jr., R. O. Grigoriev, and M. F. Schatz, "Velocity profile in a two-layer Kolmogorov-like flow," *Physics of Fluids*, vol. 26, no. 053601, 2014.

[9] J. Westerweel and F. Scarano, "Universal outlier detection for PIV data," *Experiments in Fluids*, vol. 39, pp. 1096 – 1100, August 2005.

[10] P. Mokhasi, "Using Mathematica to simulate and visualize fluid flow in a box," *Wolfram Blog*, July 9 2013.

[11] H. G. Im, "A finite difference code for the Navier-Stokes equations in vorticity/stream function formulation." University of Michigan, Computational Fluid Dynamics I, University of Michigan, Computational Fluid Dynamics I 2001.

[12] B. Fornberg, "Generation of finite difference formulas on arbitrarily spaced grids," *Mathematics of Computation*, vol. 51, pp. 699–706, October 1988.

[13] J. Franklin, *Computational Methods for Physics*. Cambridge University Press, 2013.

[14] B. Seibold, "A compact and fast Matlab code solving the incompressible Navier-Stokes equations on rectangular domains." MIT, March 2008.

[15] D. C. Giancoli, *Physics for Scientists and Engineers*. Upper Saddle River, NJ 07458: Pearson Prentice Hall, 4th ed., 2009.

[16] G. Boffetta and R. E. Ecke, "Two-dimensional turbulence," *Annual Review of Fluid Mechanics*, vol. 44, pp. 427–451, 2012.

[17] J. Tithof, B. Suri, R. Pallantla, R. O. Grigoriev, and M. F. Schatz, "Transition to weak turbulence in a Kolmogorov-like flow." Center for Nonlinear Science and School of Physics, Georgia Institute of Technology, Atlanta, Georgia, March 2015.

[18] M. Gould, *An Analysis of Shear Forces in a Thin Fluid Layer and the Effects of Certain Obstructuions to Fluid Flow*. Undergraduate thesis, Reed College, May 2012.

[19] D. Lucas and R. R. Kerswell, "Recurrent flow analysis in spatiotemporally chaotic 2-dimensional Kolmogorov flow," *arXiv*, June 2014. arXiv.