Modeling Numerical Solutions to Partial Differential Equations: A study in Waves

Jake Wood

Department of Physics, Middlebury College, Middlebury, Vermont 05753

Project report for PHYS0704

Abstract

From the Higgs Field to the Schrödinger Equation, one would be hard pressed to find a corner of modern physics that didn't concern itself with partial differential equations (PDEs). Engineering, meteorology, and financing, to name a few, equally depend on our knowledge of PDEs and their solutions to build planes, forecast weather, and predict stock behavior. Unfortunately, PDEs are difficult to solve analytically in all but the simplest of circumstances. Luckily, there exists a variety of numerical schemes that convert impossible calculus into myriad simple calculations that a computer can readily perform. This paper aims to illustrate the process of constructing stable, accurate, numerical solutions for PDEs using Finite-Difference Methods (FDM) and Pseudospectral Techniques. I have implemented a significant portion of this research in computer models which will be referenced and summarized in this paper.

Signatures

Advisor:	N. Graham
Advisor:	C. Andrews
3rd Reader:	S. Ratcliff
Date accepted:	

Contents

Introduction & Motivation

Partial differential equations (PDEs) are equations that involve derivatives of continuous multi-variable functions. Making a previously single-variable function time-dependent is a simple way to turn an ordinary differential equation (ODE) into a PDE. PDEs describe a tremendous assortment of distinct phenomena: heat, light, sound, mechanics, circuits, electrostatics, electrodynamics, fluid dynamics, and quantum mechanics, to name a few. Analytical solutions to PDEs employ traditional calculus to come to an exact solution or a class of exact solutions. For simple problems, pencil-and-paper calculus is sufficient; however, important problems are not always simple. Aerospace engineering, for example, confronts prohibitively difficult problems that demand incredibly precise solutions. Designing jet wings to optimize airflow, for instance, requires the computational power of a computer. Numerical analysis is the field that concerns itself with transforming the calculus description of a problem into algorithms that a computer can perform. Numerical solutions are ultimately estimations that we hope adequately approximate the exact analytical ones.

Solving a PDE numerically can be approached different way; this paper focuses on finite difference methods and spectral methods. Both methods are umbrella terms that describe subfields in numerical analysis. Chapter 1 provides a crash course in finite difference methods, which overall are simple, well-studied, and well understood. Chapter 2 explores the basic concepts behind a subfield of spectral methods: pseudo-spectral methods, which, although more algorithmically complicated, have benefits unrealized by finite differences.

Chapter 1

Finite Difference Methods

1.1 Discretization of equations

Finite Difference Methods (FDMs) are a way of numerically approximating PDEs. Continuous functions such as $u(x,t)$ are referenced via discrete indexing with the following form: $u_j^n \equiv u(j\Delta x, n\Delta t)$, where $\Delta x = L/J$ and $\Delta t = M/N$. L and M are the physical domains of space and time while J and N are the number of discrete points sampled over theses domains. FDMs require the following initial information:

1. PDE, e.g. the 1D heat equation, $\frac{\partial}{\partial t}u(x,t) - \alpha\frac{\partial^2}{\partial x^2}u(x,t) = 0$.

2. Space Domain, e.g. $0 \leq u_j \leq L$, where $u_j \equiv j \cdot \Delta x$, $j = 0, 1, 2, ..., J$, and $L =$ some upper bound, e.g. 2π.

3. Time Domain, e.g. $0 \leq u^n < M$, where $u^n \equiv n \cdot \Delta t$, $n = 0, 1, 2, ..., N$, and $M =$ some time in the future, M can be arbitrarily large.

4. Initial Conditions (ICs), the value of $u(x,t)$ at $t = 0$, i.e. $u^0_{0,1,2,..,J}$, e.g. $u(x,0) = \frac{1}{\sqrt{2\pi\sigma}}\exp(-\frac{(x-\sigma)^2}{2\sigma^2})$.

5. Boundary Conditions (BCs), which specify the value and/or the derivative of $u(x,t)$ at the ends or edges of the space domain. In one spatial dimension, there

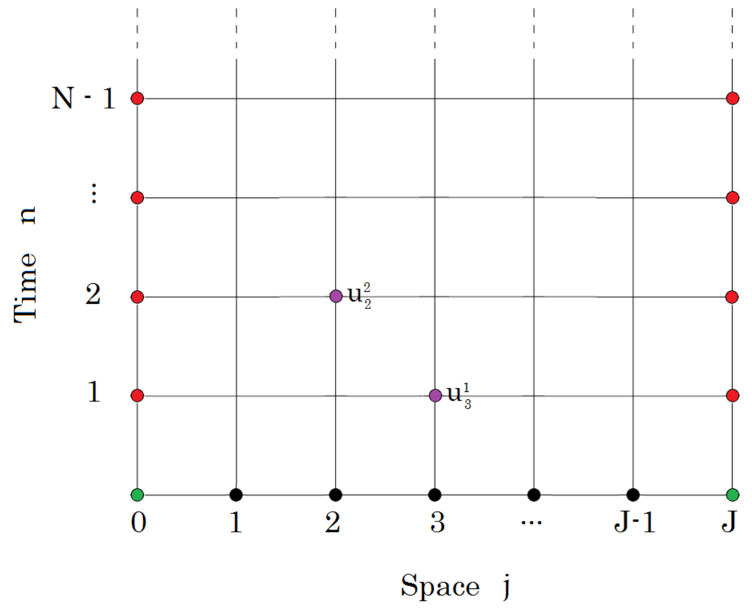

Figure 1.1: ICs and BCs.

are two edge values for any time, t: u_0^n and u_J^n. Their values are predetermined by specific functions of time; $u_0^n = f(n)$ and $u_J^n = g(n)$.

(a) Dirichlet BCs are the simple case where both $f(n)$ and $g(n)$ are constants.

(b) Neumann BCs allow boundary values to fluctuate by setting u_0^n and u_J^n such that they obey the equation $\partial u/\partial x = 0$.

The black, green, and red dots in figure 1.1 represent values initially known for a given PDE. The black dots represent the initial values of function u in the domain from 0 to L, the red dots designate the edge values of function u for all subsequent time steps, and the green dots represent values predetermined both by ICs and BCs. The basic strategy is to use our finite difference equation to fill in the unknown values on the interior of the space/time grid, one row at a time, incrementing the time step, n.

1.2 Explicit Finite Difference Expressions

Finite difference equations use local derivative (slope) information to estimate the solution of the PDE at every spatial location. The derivation is as follows,

$$\begin{aligned}
\dot{u}(t) &= \lim_{h\to 0} \frac{u(t+h)-u(t)}{h} && \text{definition of derivative} \quad (1.1)\\
&\approx \frac{u(t+\Delta t)-u(t)}{\Delta t} && \text{drop limit, replace } h \text{ with } \Delta t \quad (1.2)\\
\dot{u}^n &\approx \frac{u(n\Delta t+\Delta t)-u(n\Delta t)}{\Delta t} && \text{replace continuous } t \text{ with indexed version} \quad (1.3)\\
&\approx \frac{u^{n+1}-u^n}{\Delta t} && \text{write in fully discretized notation} \quad (1.4)
\end{aligned}$$

The same thing can be done (more intuitively) in the spatial domain by using the coordinates on either side of equation 1.6 (ie u_{j+1} and u_{j-1}) to estimate the slope at u_j,

$$\begin{aligned}
u'(x) &= \lim_{h\to 0} \frac{u(x+h)-u(x-h)}{2h} && \text{centered derivative} \quad (1.5)\\
&\approx \frac{u_{j+1}-u_{j-1}}{2\Delta x} && \text{simplified in discrete notation} \quad (1.6)
\end{aligned}$$

To get second order derivatives and higher, we apply the first derivative estimation to the results of previous derivative estimations.

$$\begin{aligned}
u'(x) &= \lim_{h\to 0} \frac{u(x+\frac{h}{2})-u(x-\frac{h}{2})}{h} && \text{alternate centered derivative} \quad (1.7)\\
&\approx \frac{u_{j+1/2}-u_{j-1/2}}{\Delta x} && \text{simplified in discrete notation} \quad (1.8)\\
u''_j &\approx \frac{u'_{j+1/2}-u'_{j-1/2}}{\Delta x} && \text{equation 1.8 with } u'_j \text{ as input} \quad (1.9)\\
&\approx \frac{u_{j+1}-2u_j+u_{j-1}}{(\Delta x)^2} && u'_j \text{ expanded with equation 1.8} \quad (1.10)
\end{aligned}$$

1.3 The Finite Difference Method

The finite difference method is one strategy for solving PDEs, which involves replacing derivatives with finite difference expressions derived in section 1.2. Consider the 1D advection equation, where u_x represents the component of wave's velocity in the x direction,

$$\frac{\partial \phi}{\partial t} + u_x \frac{\partial \phi}{\partial x} = 0 \tag{1.11}$$

Simply replace $\frac{\partial \phi}{\partial t}$ with equation 1.4 and $\frac{\partial \phi}{\partial x}$ with equation 1.6:

$$\frac{\phi_j^{n+1} - \phi_j^n}{\Delta t} = u_x \frac{\phi_{j-1}^n - \phi_{j+1}^n}{2\Delta x} \tag{1.12}$$

Solving for the ϕ one time step in the future yields

$$\phi_j^{n+1} = \sigma(\phi_{j-1}^n - \phi_{j+1}^n) + \phi_j^n \tag{1.13}$$

where $\sigma \equiv \frac{u_x}{2}(\frac{\Delta t}{\Delta x})$. Figure 1.2 illustrates how coordinate information propagates using FDM equation 1.13. Using the finite difference equation 1.13 and row ϕ_n^0, one can calculate each ϕ_n^1 row value. In figure 1.2 the three green ϕ values ($\phi_{1,2,3}^0$) combine to specify ϕ_2^1. After every ϕ_j^1 is found, equation 1.13 is used with the ϕ_n^1 row values to compute the next row, the ϕ_n^2 values.

1.4 FDM example in 2 Spatial Dimensions

The Higgs Field can be described by a version of the Klein-Gordon PDE,

$$\frac{1}{c^2}\ddot{\psi} = \nabla^2 \psi - \frac{\partial u}{\partial \psi} \qquad \text{Klein-Gordon} \tag{1.14}$$

$$u(\psi) = (|\psi|^2 - 1)^2 \qquad \text{potential energy of } \psi \tag{1.15}$$

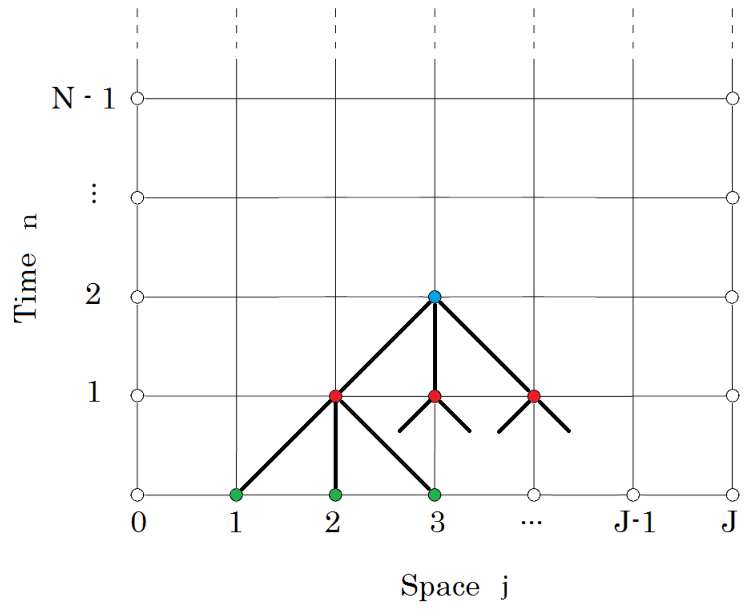

Figure 1.2: Finite difference stencil

Limiting the solution to 2 spatial dimensions, equation 1.14 can be simplified to

$$\frac{1}{c^2}\frac{\partial^2\psi}{\partial t^2} = \frac{\partial^2\psi}{\partial x^2} + \frac{\partial^2\psi}{\partial y^2} - 4\psi(\psi^2 - 1). \tag{1.16}$$

Using finite difference estimations described in the previous section, every partial derivative can be re-expressed in terms of known values:

$$\frac{\psi^{n+1}_{j,k} - 2\psi^{n}_{j,k} + \psi^{n-1}_{j,k}}{(c\Delta t)^2} = \frac{\psi^{n}_{j+1,k} - 2\psi^{n}_{j,k} + \psi^{n}_{j-1,k}}{(\Delta x)^2} + \frac{\psi^{n}_{j,k+1} - 2\psi^{n}_{j,k} + \psi^{n}_{j,k-1}}{(\Delta y)^2} - 4\psi^{n}_{j,k}((\psi^{n}_{j,k})^2 - 1), \tag{1.17}$$

which simplifies to the following, where $\alpha \equiv (\frac{c\Delta t}{\Delta x})^2$ and $\beta \equiv (\frac{c\Delta t}{\Delta y})^2$,

$$\psi^{n+1}_{j,k} = 2\psi^{n}_{j,k}(2 - \alpha - \beta - 2(c\Delta t \cdot \psi^{n}_{j,k})^2) + \alpha(\psi^{n}_{j+1,k} + \psi^{n}_{j-1}, k) + \beta(\psi^{n}_{j,k+1} + \psi^{n}_{j,k-1}) - \psi^{n-1}_{j,k}. \tag{1.18}$$

1.5 Stability

In pure mathematics there exist a number of complicated techniques used to ensure a numerical solution is stable and accurate. In simple cases (usually 1 spatial dimension), one can compare the numerical solutions against the analytical ones; however, in practice, it is easy to guess and check the initial conditions of a numerical solution. The choices for Δx and Δt are the most important components of a stable numerical solution. Running a simulation with unstable parameters causes the calculated solutions to diverge rapidly. When possible, it is a good idea to perform basic stability analysis, the most common measures being awareness of the Courant–Friedrichs–Lewy (CFL) condition and Von Neumann Stability Analysis (VNSA).

1.5.1 CFL Condition

Courant, Friedrichs, and Lewy developed the CFL condition in the 1950's for numerical analysis of PDEs [4]. The CFL condition, σ, is the term that collects the time step, space step, and velocity terms into one variable for a given finite difference scheme. For example, $\sigma = \frac{c\Delta t}{\Delta x}$ for the advection equation in one spatial dimension. For stability it is required, but not sufficient, that the CFL ≤ 1 [4]. Since the sizes of the time and space steps have an impact on stability, the CFL condition provides a guideline for appropriate choices for these values.

1.5.2 Von Neumann Stability Analysis (VNSA)

VNSA only works on finite difference equations describing linear PDEs and relies on Fourier analysis to determine constraints on the CFL condition that preserve stability. In other words, it specifies constraints for the time-step, spatial-step, and velocity that ensure the stability of the solution. FDM formulas describe how the values associated with a collection of spatial components are updated after one time step. The basic idea is to convert these spatial components into Fourier space then examine the behavior of each component after they have been updated by the finite difference equation [2]. Starting from the discrete Fourier Transform,

$$\phi(x_j, t_n) = \sum_k \zeta(k)^n e^{ikj\Delta x}, \tag{1.19}$$

take a single Fourier mode [3],

$$\phi_j^n = \zeta(k)^n e^{ikj\Delta x}, \tag{1.20}$$

and use it to rewrite a finite difference equation of interest[1]. Rearrange the altered difference equation to isolate $\zeta(k)$, then investigate which values of Δx and Δt satisfy the condition $|\zeta(k)| \leq 1$. If no values for Δx and Δt satisfy the condition, then the finite difference scheme is unstable. If a specific ratio of Δx and Δt causes the condition to be true, the scheme is conditionally true. If for all values of Δx and Δt, $|\zeta(k)| \leq 1$, then the scheme is unconditionally stable.

1.5.3 Stability Analysis for Wave Model in Two Spatial Dimensions

The following example illustrates my process for determining appropriate Δx and Δt for the second-order wave equation in one spatial dimension; the foundation of which was used to simulate the same wave equation in two spatial dimensions:

$$\ddot{\phi} - c^2 \frac{\partial^2 \phi}{\partial x^2} = 0. \tag{1.21}$$

[1]Note that in equation 1.20, the superscript n for ϕ_j^n references the n^{th} time step but denotes a power for $\zeta(k)^n$.

To get a finite difference equation, both derivatives are replaced with a second-order approximation, equation 1.10 (The LHS is simply the temporal version of the second derivative in equation 1.10).

$$\frac{\phi_j^{n+1} - 2\phi_j^n + \phi_j^{n-1}}{(\Delta t)^2} = c^2\left(\frac{\phi_{j+1}^n - 2\phi_j^n + \phi_{j-1}^n}{(\Delta x)^2}\right) \qquad \text{write as FDM eqn} \tag{1.22}$$

Rewrite equation 1.22 with Fourier terms (1.20),

$$\left(\frac{1}{\Delta t}\right)^2\left(\zeta(k)^{n-1}e^{ikj\Delta x} - 2\zeta(k)^n e^{ikj\Delta x} + \zeta(k)^{n+1}e^{ikj\Delta x}\right) \qquad \text{LHS} \tag{1.23}$$

$$\left(\frac{c}{\Delta x}\right)^2\left(\zeta(k)^n e^{ik(j-1)\Delta x} - 2\zeta(k)^n e^{ikj\Delta x} + \zeta(k)^n e^{ik(j+1)\Delta x}\right) \qquad \text{RHS} \tag{1.24}$$

After moving the term with Δt to the other side, divide both sides of the equation by $\zeta(k)^n e^{ikj\Delta x}$ and let $\alpha \equiv \left(\frac{c\Delta t}{\Delta x}\right)^2$,

$$\zeta^{-1} - 2 + \zeta = \alpha(e^{-ik\Delta x} - 2 + e^{ik\Delta x}) \tag{1.25}$$

$$\zeta^{-1} + \zeta = 2 + 2\alpha(\cos(k\Delta x) - 1) \tag{1.26}$$

$$= 2 - 4\alpha \sin^2(\frac{1}{2}k\Delta x) \tag{1.27}$$

$$= 2(1 - 2\alpha \sin^2(\frac{1}{2}k\Delta x)) \tag{1.28}$$

In keeping with physics tradition, I replaced $\sin^2(\frac{1}{2}k\Delta x)$ with 0 and 1 respectively (as 0 and 1 represent the minima and maxima of the sin term for all k) and inspected both equations.

$$\zeta(k)^{-1} + \zeta(k) = 2 \qquad \text{replacing sin term with 0} \tag{1.29}$$

$$\zeta(k)^{-1} + \zeta(k) = 2(1 - 2\alpha) \qquad \text{replacing sin term with 1} \tag{1.30}$$

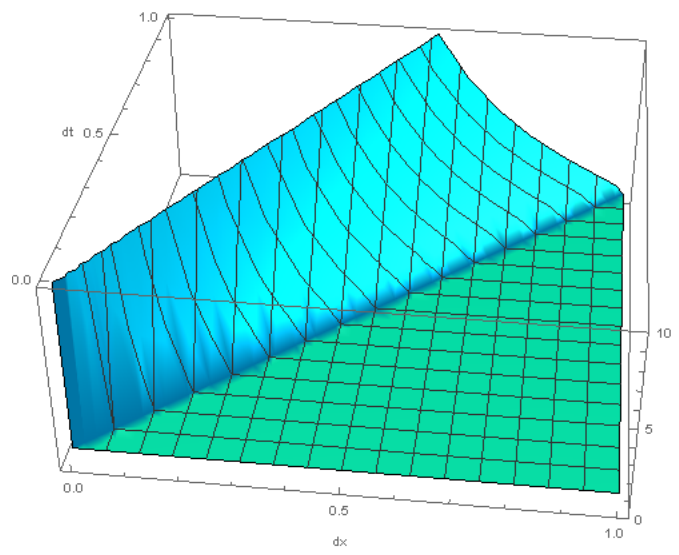

Figure 1.3: Mathematica plot of absolute value of equation 1.31 demonstrating the effect that Δx and Δt have on $\zeta(k)$. The $\Delta x < c\Delta t$ region is visibly greater than 1, and thus unstable.

By inspection, the solution to equation 1.29 is $\zeta = 1$. This implies that $|\zeta| \leq 1$ for any choice of α, and thus replacing $\sin^2(\frac{1}{2}k\Delta x)$ with 0 indicates unconditional stability. At this point, I have only verified that the FDM is stable when $\frac{k}{2}\Delta x \in Z$. Using the NSolve function in Mathematica on equation 1.30 yields,

$$\zeta(\Delta x, \Delta t) = \frac{-2\Delta t^2 \pm 2\sqrt{\Delta t^4 - \Delta t^2 \Delta x^2} + \Delta x^2}{\Delta x^2}. \tag{1.31}$$

Figure 1.3 shows the 3D plot of ζ with domains of Δx and Δt from 0 to 1. From figure 1.3 it can be ascertained that the FDM for the second order wave equation is conditionally stable iff

$$\Delta x \geq c\Delta t \tag{1.32}$$

because it is only in this portion of the graph that $|\zeta| \leq 1$. In this case, VNSA strongly validates the CFL condition, although it just as easily could have put tighter bounds on Δx and Δt. In the case of the heat equation in one spatial dimension, for instance, VNSA determines that $(\Delta x)^2 \geq 2c\Delta t$ [2].

Chapter 2

Pseudospectral Methods

2.1 Pseudospectral Method

Pseudospectral methods are a subset of spectral methods that use global information to determine how a PDE evolves with time. Concretely, to determine the derivative at a point in space, instead of using local (neighboring points') values, the derivative is estimated via knowledge of every value in the domain, which is commonly (although not necessarily) done with a discrete Fourier transform (DFT) [5]. My research and implementations of the pseudospectral method exclusively used Fourier Polynomials[1], which only applies when the modeled PDE exhibits periodicity. This prerequisite exists because Fourier transforms, and thus DFTs, require periodic signals. As a result, spectral methods are sound choices for modeling systems with periodic behavior. Pure spectral methods treat time periodically as well, which, in addition to being algorithmically more complex, is not always something that can be assumed. The prefix *pseudo* in pseudo-spectral means spatial operations are handled spectrally, while FDMs are used to update time components.

[1]In other words, instead of representing the solution as a sum of Legendre polynomials, for instance, I used the exponentials associated with Fourier Transforms.

2.1.1 Implementation

The following steps outline the basic procedure for solving a PDE with the pseudo-spectral method [6].

1. Write PDE in discrete form.
2. Convert spatial terms into Fourier space.
3. Apply operators to Fourier terms[2].
4. Return Fourier terms back to real space.
5. Use FDM to time step function.

This procedure works because differentiating in real space is equivalent to multiplication by coefficients in Fourier space. Consider the heat equation in 1 spatial dimension where the $\mathscr{F}$ operator performs a Fourier Transform,

$$\dot{u}(x,t) - \alpha\frac{\partial^2 u}{\partial x^2} = 0 \qquad \text{start with PDE} \tag{2.1}$$

$$\frac{\partial u_j^n}{\partial t} - \alpha\frac{\partial^2 u_j^n}{\partial x^2} = 0 \qquad \text{step 1} \tag{2.2}$$

$$\frac{\partial u_j^n}{\partial t} - \alpha\mathscr{F}^{-1}\{\frac{\partial^2}{\partial x^2}\mathscr{F}\{u_j^n\}\} = 0 \qquad \text{step 2-4} \tag{2.3}$$

$$\frac{u_j^{n+1} - u_j^n}{\Delta t} - \alpha\mathscr{F}^{-1}\{\frac{\partial^2}{\partial x^2}\mathscr{F}\{u_j^n\}\} = 0 \qquad \text{step 5} \tag{2.4}$$

Simplify to get,

$$u_j^{n+1} = u_j^n + \alpha(\Delta t)\mathscr{F}^{-1}\{\frac{\partial^2}{\partial x^2}\mathscr{F}\{u_j^n\}\} \tag{2.5}$$

Formally, the $\mathscr{F}$ operator applies a continuous transform; however, our equations act on discrete values so a discrete Fourier Transform operator is needed which, for this paper, will be represented by the letters DFT. This indicates for our calculation that equation 2.5 should replace $\mathscr{F}$ with *DFT*. In practice we want to optimize this

[2]Usually, but not necessarily, a derivative.

process, so we'll use the Fast Fourier Transform Algorithm (FFT) to compute the DFT and the Inverse Fast Fourier Transform (IFFT) to compute DFT^{-1}:

$$u_j^{n+1} = u_j^n + \alpha(\Delta t) IFFT\big(\frac{\partial^2}{\partial x^2} FFT(u_j^n)\big) \tag{2.6}$$

It may seem mathematically illegitimate to perform Fourier transforms on the spatial component of our finite difference equation. While the motivation is yet to be explained fully, adding transforms to equation 2.3, for instance, is valid because

$$u = \mathscr{F}^{-1}\{\mathscr{F}\{u\}\} \tag{2.7}$$

which has the discrete analog [7],

$$u = DFT^{-1}(DFT(u)) \tag{2.8}$$

Since the Fourier Transform operator and Differentiation are commutative, it is true that

$$\frac{\partial^2 u}{\partial x^2} = \mathscr{F}^{-1}\{\frac{\partial^2}{\partial x^2}\mathscr{F}\{u\}\}, \tag{2.9}$$

which is why we can replace the derivative in equation 2.2 with the expression in equation (2.9).

2.1.2 The Discrete Fourier Transform

The most natural question at this point is how does changing $u(x,t)$ into Fourier space make the derivative easier to perform numerically. Consider the definition of the discrete Fourier transform,

$$\tilde{u}_k = \frac{1}{J}\sum_{j=0}^{J-1} u_j e^{-2\pi i \frac{jk}{J}} \qquad \tilde{u} \equiv \mathrm{DFT}(u) \tag{2.10}$$

$$u_j = \sum_{k=0}^{J-1} \tilde{u}_k e^{+2\pi i \frac{jk}{J}} \qquad u \equiv \mathrm{DFT}^{-1}(\tilde{u}) \tag{2.11}$$

where $\tilde{u_k}$ holds the values of u_j in Fourier space [1]. The DFT occurs before the derivative so it does not operate on each term in the sum. The naïve approach looks like this,

$$\frac{\partial^2}{\partial x^2}\tilde{u}_k = \frac{\partial^2}{\partial x^2}\left(\frac{1}{J}\sum_{j=0}^{J-1} u_j e^{-2\pi i \frac{jk}{J}}\right) \tag{2.12}$$

$$= -\left(\frac{2\pi k}{J}\right)^2 \tilde{u}_k \tag{2.13}$$

Fourier transforms decompose functions of time into the frequencies that comprise them. Since we are limited by the density of sampled points in our grid, high frequencies cannot be accurately represented and thus introduce error. To account for this, we will employ a frequency shifted weight described by S. Johnson [1].

$$\frac{\partial^2}{\partial x^2}\tilde{u}_k = \begin{cases} -(\frac{2\pi}{J}k)^2\tilde{u}_k & k \leq J/2 \\ -(\frac{2\pi}{J}(k-J))^2\tilde{u}_k & k > J/2 \end{cases} \tag{2.14}$$

Applying the DFT^{-1} (2.11) on the results of equation 2.14 finally yields the spatial derivative of u_j.

$$\frac{\partial^2}{\partial x^2}u_j = DFT^{-1}\left(\frac{\partial^2}{\partial x^2}\tilde{u}_k\right) \tag{2.15}$$

Putting all the pieces together, one can treat equation 2.5 in the same way as a finite difference formula, which procedurally estimates a given PDE with incremental time-steps.

Chapter 3

My Simulations

To put the math foundations, discussed in the previous two chapters, into practice, I wrote two simulations using Processing[1]. Both programs visualize the time evolution of a linear, second-order, wave equation in 2 spatial dimensions.

$$\frac{\partial^2 \phi}{\partial t^2} = c^2\left(\frac{\partial^2 \phi}{\partial x^2} + \frac{\partial^2 \phi}{\partial y^2}\right) \tag{3.1}$$

I didn't use a more complicated PDE for a few reasons. The first reason is accessibility; the purpose is to provide a tangible link between a person's intuition about waves and the mathematics behind them. Telling people they are looking at the solution to the sine-Gordon equation[2] will almost never evoke an emotional response (let alone a positive one). On the other hand, telling them that they are looking at the simulation of the surface of a swimming pool immediately unites the math with the observers' own experiences. Does that look like the right behavior? Suddenly the layperson has tools to critically examine and interact with the simulation. Furthermore, the principles required to produce a numerical solution for equation 3.1 are the exact same as those for analytically more challenging problems. Modeling more complicated

[1]https://processing.org/

[2] $\psi_{tt} - \psi_{xx} + sin(\psi) = 0$

equations, such as the Korteweg–de Vries equation[3], I saw that my methods replicated behavior described in the literature without any real nuance introduced beyond that which I employed in my wave equation simulation.

3.1 Program One

Program One acts more as a proof of concept than anything else; its purpose is to visualize the finite difference method in action while simultaneously verifying its accuracy. On the left, two mesh surfaces, one superimposed above the other, undulate up and down. One of these mesh surfaces is my numerical approximation and the other is the analytical solution. On the right, a height field of warm colors bobs up and down in rounded symmetric patterns. This field represents the error between the numerical solution and the analytical one. More specifically, it shows the absolute difference between the numerical and analytical solution (described below) divided by $|max\ amplitude - min\ amplitude|$. This gives a reference for the error height field–absolute error is meaningless without context about the range of expected values. An error of plus or minus one is more impressive when the expected values span $(0, 100)$ than when the expected range is $(0, 4)$. By dividing by the range, the error is normalized to represent relative error on the range $(0, 1)$. The reason I chose not to use fractional error is that both the expected and experimental values sweep from positive to negative, which means that very close to zero the error blows up. This occurs not because the model is inaccurate near zero, but because the expected value (denominator) periodically equals 0. I also tested using the first spatial derivative in the denominator: $|\phi + \Delta t \cdot \phi'|$; however, this term still equals 0 periodically, which causes the same problems I experienced before and ultimately convinced me to use the method I did.

[3]The KDV equation is a non-linear soliton with the following form: $\partial_t \phi + \partial_x^3 \phi + 6\phi \partial_x \phi = 0$

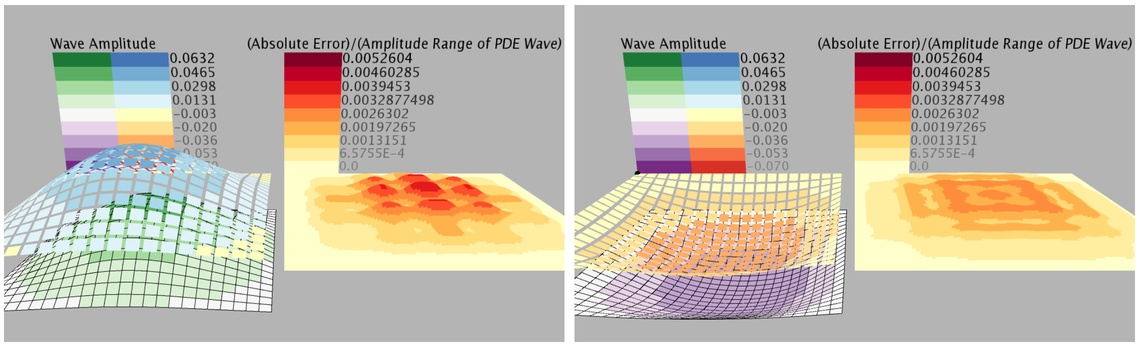

Figure 3.1: Screenshots from Program One.

3.1.1 Analytical Solution

Another important reason I used the 2D wave equation was because it is feasible to implement the 2D solution analytically. I did not perform the derivation myself but relied on the helpful lecture notes of R. Daileda [9], that describe a derivation that relies on Fourier Analysis. The solution is complicated enough that I was restricted to Dirichlet boundary conditions (border height held constant at 0), and used a Bessel function for the initial conditions. The reason I couldn't use a Gaussian for the ICs is that it is impossible to make the ICs match Dirichlet BCs (see figure 3.2). The Gaussian plot on the right shows how the cylindrical symmetry prevents a square border from having a common height unlike the dark orange region of the Bessel function on the left, whose height is exactly zero on the square.

3.1.2 Numerical Solution

The numerical solution in my model uses the following finite difference formula with the same ICs and BCs as the exact solution,

$$\phi_{j,k}^{n+1} = 2\phi_{j,k}^{n}(1-\gamma-\lambda) + \gamma\left(\phi_{j-1,k}^{n} + \phi_{j+1,k}^{n}\right) + \lambda\left(\phi_{j,k-1}^{n} + \phi_{j,k+1}^{n}\right) - \phi_{j,k}^{n-1}, \qquad (3.2)$$

where $\gamma \equiv (c\Delta t/\Delta x)^2$ and $\lambda \equiv (c\Delta t/\Delta y)^2$.

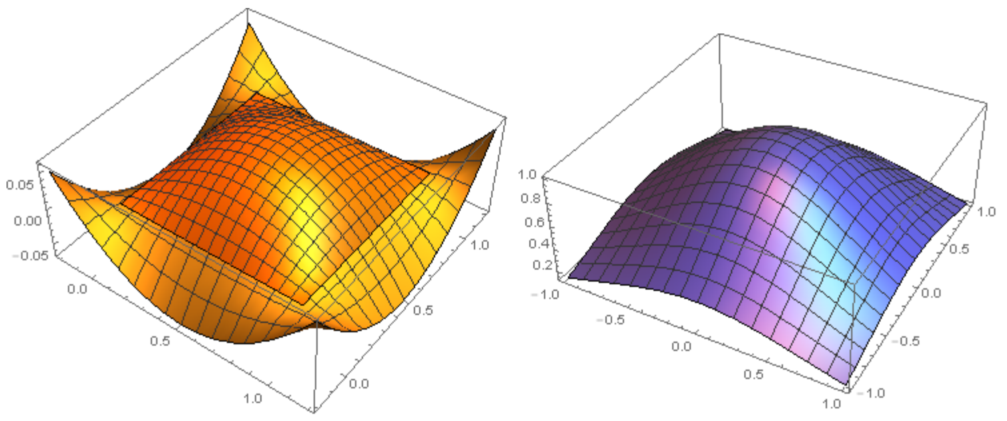

Figure 3.2: Mathematica plots. Bessel Function (left) and Gaussian (right).

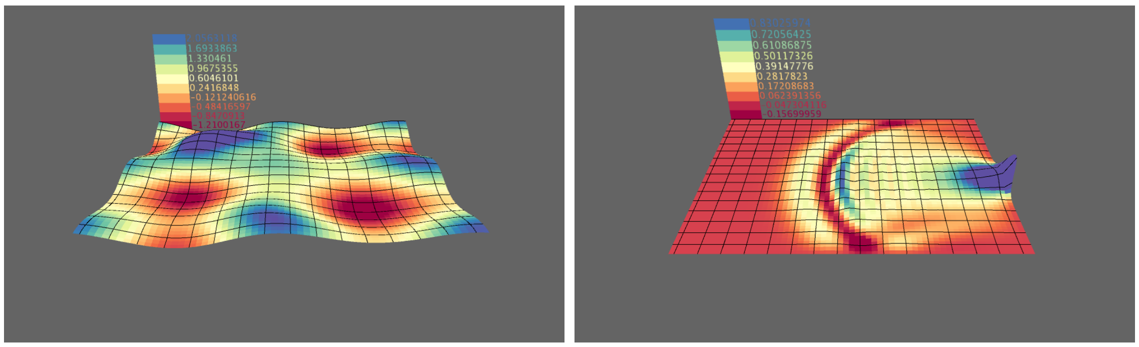

Figure 3.3: Screenshots from Program Two.

3.2 Program Two

The purpose of Program One is to visually communicate the fidelity of numerical models to exact solutions. In the literature, comparing a numerical estimation to a known solution is the most common method of corroborating a numerical technique. Program Two, however, is intended as an interactive wave visualizer. The user interface includes control over the following aspects of the simulation:

1. Initial Conditions: amplitude, standard deviation, and location of a 2D Gaussian function that define the initial surface perturbation (truncated at the boundary). The amplitude and standard deviation have a lower and upper bound for practical reasons.

2. Behavior at the Boundary: The user can either specify that the border be fixed at height zero (Dirichlet BCs), or choose to have the wave's height be fluid at the boundaries (Neumann BCs). The finite difference scheme for the first choice is the same as equation 3.2, while the pseudo-spectral equation for the second choice uses

$$u_j^{n+1} = 2u_j^n + \alpha(\Delta t) IFFT\big(\frac{\partial^2}{\partial x^2} FFT(u_j^n)\big) - u_j^{n-1}, \tag{3.3}$$

where the second derivative is replaced with equation 2.14.

3. Ability to pause, change ICs as desired, and restart simulation (or continue from paused).

Furthermore, at any point the current frame can be saved at the press of a button as a .tiff file in the home directory of the Processing file. This feature can be set to save every frame that can be compiled afterwards into a movie.

3.3 Extensions

Damping is one feature I believe would improve the simulation's realistic feel. I am not 100% confident how to achieve this effect, but I suspect (and Professor Graham agrees) that if I simply start with the following damped wave equation,

$$\ddot{\phi} = c^2 \nabla^2 \phi - \mu \dot{\phi}, \tag{3.4}$$

the numerical solution will exhibit the expected behavior. Also, while Processing provided satisfactory speed and convenience for this project, further implementations would be greatly enhanced by utilizing WebGL.

3.4 Acknowledgments

Thanks to my thesis advisors, professors Christopher Andrews and Noah Graham of the Middlebury College Computer Science and Physics departments respectively, both of whom have supported me during this process. Thanks also to professor Stephen Ratcliff who participated in my final interview.

3.5 Source Code for Program 1

```
//  ERROR ANALYSIS OF FINITE DIFFERENCE SCHEME FOR 2D WAVE EQN
// written in Processing: https://processing.org/
//  Jake Wood
//  Thesis work for Physics and Computer Science at Middlebury College
//  1/29/15 - 4/26/15
//  Advisors: Noah Graham and Christopher Andrews

int screenWidth = 1000;//this changes the size of window on your screen
int screenHeight = 600;
int gridWidth = 100;//NUMBER OF GRIDS ACROSS
int gridHeight = 100;//NUMBER OF GRIDS VERTICALLY
boolean isFullScreen = false;

float amplitude = 1.0;//USER CAN CHANGE THIS VALUE FOR DIFFERENT INIT CONFIGS
float narrowness = 0.15;//USER CAN CHANGE THIS VALUE FOR DIFFERENT INIT CONFIGS

// dx <= 1 / gridWidth
float dt = 0.0035; //THIS SHOULD BE EXPERIMENTED WITH (FOR STABILITY)
float c = 1.0; //WAVE SPEED

float scaleAmp = 3000.0;
float errorAmp = 100000.0;

final float errColorLow = 0.0;//error range
float errColorHigh = 0.0001;//dynamically increases to reflect actual max value

int DETAIL = 4;//change this as desired. 1 is highest resolution

float theta = 0.3;//legend angle

float dx, dy;
float cX, cY;
float oXerror;//offset for drawing different height fields
float oYexact;
float lowCol;//color bounds
float upperCol;
float time_elapsed;//inits as zero
```

```
boolean pause;

float[] xValues, yValues;//final
Zvalue[][] zVals;

public void setup() {
if (isFullScreen)
size(displayWidth, displayHeight, P3D);
else
size(screenWidth, screenHeight, P3D);
frameRate(60);
pause = false;
oXerror = 0.5 * width;
oYexact = 100.0;
camera(width/2.0, height, (height/2.0),
width/2.0, 0, 0,
0, 1.0, 0);
time_elapsed = 0;
dx = 1.0 / gridWidth;
dy = 1.0 / gridHeight;
cX = sq(c) * sq(dt/dx);
cY = sq(c) * sq(dt/dy);
initXYArrays();
initZvalues();
lowCol = -0.001;
upperCol = 0.00;
//noLoop();
}

public void draw() {
background(180, 180, 180);
camera(width/2.0, height, (height/2.0), width/2.0, 0, 0, 0, 1.0, 0);//sets camera position
if (!pause) {
updateClock();
updateMesh1();
updateMesh2();
}
noStroke();
drawMesh();
drawLegend();
drawMeshError();
stroke(1);
drawMeshExact();
}

//handles key presses
public void keyPressed() {
char letter = key;
switch(letter) {
case 'f':
saveFrame();
break;
case 'p':
pause = !pause;
}
}

//sets x y coordinates in world view
public void initXYArrays() {
xValues = new float [gridWidth + 1];
yValues = new float [gridHeight + 1];
zVals = new Zvalue[gridWidth + 1][gridHeight + 1];

//init x values
for (int i = 0; i <= gridWidth; i++)
xValues[i] = 0.5 * width * i / gridWidth;//width is whatever is set in size()
//int y values
for (int j = 0; j <= gridHeight; j++)
yValues[j] = 0.5 * height * j / gridHeight;
}

//initial config for surface perturbation
public float bessel(float x, float y) {
return (x / gridWidth)*(y / gridWidth)*(1.0 - x / gridWidth)*(1.0 - y / gridWidth);
}

//sets initial hieghts for heightfield
public void initZvalues() {
for (int h = 0; h <= gridHeight; h ++) {
for (int w = 0; w <= gridWidth; w ++) {
//zValues[w][h] = (float)Math.random();
float waveHeight = bessel(w, h);
zVals[w][h] = new Zvalue(waveHeight, waveHeight, 0);
}
}
}

//returns exact height of wave at position (w,h) and time t
public float calcExactZ(float w, float h, float t) {
float u = 0.0;
int lim = 10;
for (int m = 1; m < lim; m+=2) {
for (int n = 1; n < lim; n+=2) {
u+=64.0/pow(PI,6)*1.0/pow(m*n,3)*sin(dx*w*PI*m)*sin(dy*h*PI*n)*cos(t*PI*sqrt(sq(m)+sq(n)));
}
}
return u;
}

//absolute error
public float calcError(float experiment, float theory) {
//df is first time derivitive at spatial location
return abs(experiment - theory)*errorAmp;
}

//updates clock for calcExactZ time input
public void updateClock() {
time_elapsed += dt + dt;//each redrawing happens after 2 time steps
```

```
}

//updateMesh 1 and 2 trade which arrays are holding which values
//to cut down on overhead of copying arrays uneccessarily each time step
public void updateMesh1() {
//loops over interior points only
for (int h = 1; h < gridHeight; h++) {
for (int w = 1; w < gridWidth; w++) {
Zvalue cell = zVals[w][h];
cell.setTemp(2.0f*cell.getZ()-cell.getPrev()+cX*(zVals[w+1][h].getZ()-2.0f*cell.getZ()+
zVals[w-1][h].getZ())+cY*(zVals[w][h-1].getZ()-2.0f*cell.getZ()+zVals[w][h+1].getZ()));
cell.setPrev(cell.getZ());
}
}
}

public void updateMesh2() {
//loops over interior points only
for (int h = 1; h < gridHeight; h ++) {
for (int w = 1; w < gridWidth; w ++) {
//only need exact z in 2nd of two updates
Zvalue cell = zVals[w][h];

cell.setExactZ(calcExactZ(w, h, time_elapsed));

cell.setZ(2.0f*cell.getTemp() - cell.getZ() + cX*(zVals[w+1][h].getTemp() -
2.0f*cell.getTemp() + zVals[w-1][h].getTemp()) + cY*(zVals[w][h-1].getTemp() -
2.0f*cell.getTemp() + zVals[w][h+1].getTemp()));
cell.setPrev(cell.getTemp());
cell.setError(calcError(cell.getZ(), cell.getExactZ()));
}
}
}

//draws the 3 distinct legends and text labels
public void drawLegend() {
float rigthEdge = width/4;
float inc = 1.1/9;
float scale = 0.4;
textSize(height*inc*scale);
pushMatrix();
rotateX(theta);
float indexedColor = ((upperCol - lowCol) + lowCol);
pushMatrix();
rotateX(-PI/2.0f);
fill(1);
text("Wave Amplitude", 0, -height*(1.1+inc/4)*scale, 1);
text("(Absolute Error)/(Amplitude Range of PDE Wave)",
height*0.7, -height*(1.1+inc/4)*scale, 1);
popMatrix();
for (float i=0; i<1.1; i+=inc){

//legend labels
pushMatrix();
```

```
rotateX(-PI/2.0f);
String textLabel = str(i*(upperCol - lowCol) + lowCol);
fill((255 - i*255)/2.0);
text(textLabel.substring(0, 6), rigthEdge + 2, -height*(i+inc/4)*scale, 1);
String errorLabel = str(i*errColorHigh/(upperCol - lowCol)/errorAmp);
text(errorLabel, 2*rigthEdge + rigthEdge/2, -height*(i+inc/4)*scale, 1);
popMatrix();

//purple to green
colorChooser(0.0, 1.0, i, 0);
beginShape(QUAD);
vertex(0, 0, height*i*scale);
vertex(rigthEdge/2, 0, height*i*scale);
vertex(rigthEdge/2, 0, height*(i+inc)*scale);
vertex(0, 0, height*(i+inc)*scale);
endShape();

//red to blue
colorChooser(0.0, 1.0, i, 1);
beginShape(QUAD);
vertex(rigthEdge/2, 0, height*i*scale);
vertex(rigthEdge, 0, height*i*scale);
vertex(rigthEdge, 0, height*(i+inc)*scale);
vertex(rigthEdge/2, 0, height*(i+inc)*scale);
endShape();

//red error color scale
colorChooser(0.0, 1.0, i, 2);
beginShape(QUAD);
//stroke(0, 150);
vertex(rigthEdge*2, 0, height*i*scale);
vertex(rigthEdge*2+rigthEdge/2, 0, height*i*scale);
vertex(rigthEdge*2+ rigthEdge/2, 0, height*(i+inc)*scale);
vertex(rigthEdge*2, 0, height*(i+inc)*scale);
endShape();
}
popMatrix();
}

//draws height field surface for error
public void drawMeshError() {
for (int h = 0; h < gridHeight; h++) {
for (int w = 0; w < gridWidth; w++) {
float wh = zVals[w][h].getError();
float w1h = zVals[w + 1][h].getError();
float w1h1 = zVals[w + 1][h + 1].getError();
float wh1 = zVals[w][h + 1].getError();
beginShape(QUAD);
//System.out.println(0.25*(wh + w1h + w1h1 + wh1)/errorAmp);
colorChooser(0.0, errColorHigh, (wh + w1h + w1h1 + wh1)/4.0, 2);
vertex(xValues[w]+oXerror, yValues[h], wh);
vertex(xValues[w + 1]+oXerror, yValues[h], w1h);
vertex(xValues[w + 1]+oXerror, yValues[h + 1], w1h1);
vertex(xValues[w]+oXerror, yValues[h + 1], wh1);
```

```
endShape();
}
}
}

//draws heightfield for exact solution
public void drawMeshExact() {

for (int h = 0; h < gridHeight-DETAIL; h+=DETAIL) {
for (int w = 0; w < gridWidth-DETAIL; w+=DETAIL) {
float wh = zVals[w][h].getExactZ();
float w1h = zVals[w + DETAIL][h].getExactZ();
float w1h1 = zVals[w + DETAIL][h + DETAIL].getExactZ();
float wh1 = zVals[w][h + DETAIL].getExactZ();
beginShape(QUAD);
colorChooser(lowCol, upperCol, 0.25f * (wh + w1h + w1h1 + wh1), 0);
vertex(xValues[w], yValues[h], scaleAmp*wh - oYexact);
vertex(xValues[w + DETAIL], yValues[h], scaleAmp*w1h - oYexact);
vertex(xValues[w + DETAIL], yValues[h + DETAIL], scaleAmp*w1h1 - oYexact);
vertex(xValues[w], yValues[h + DETAIL], scaleAmp*wh1 - oYexact);
endShape();
}
}
}

//draws heightfield for FDM numerical solution
public void drawMesh() {
//updates coloring info for both exact and numerical mesh
float middleHeight = zVals[gridWidth/2][gridHeight/2].getZ();
//the following conditions dynamically update the colortable to accurately reflect range
if (middleHeight > upperCol) {
upperCol = middleHeight;
} else if (middleHeight < lowCol) {
lowCol = middleHeight;
}

float middleError = zVals[gridWidth/2][gridHeight/2].getError();
if (middleError > errColorHigh) {
errColorHigh = middleError;
}

for (int h = 0; h < gridHeight-DETAIL-1; h+=DETAIL+1) {
for (int w = 0; w < gridWidth-DETAIL-1; w+=DETAIL+1) {
float wh = zVals[w][h].getZ();
float w1h = zVals[w + DETAIL][h].getZ();
float w1h1 = zVals[w + DETAIL][h + DETAIL].getZ();
float wh1 = zVals[w][h + DETAIL].getZ();
beginShape(QUAD);
colorChooser(lowCol, upperCol, 0.25 * (wh + w1h + w1h1 + wh1), 1);
vertex(xValues[w], yValues[h], scaleAmp*wh);
vertex(xValues[w + DETAIL], yValues[h], scaleAmp*w1h);
vertex(xValues[w + DETAIL], yValues[h + DETAIL], scaleAmp*w1h1);
vertex(xValues[w], yValues[h + DETAIL], scaleAmp*wh1);
endShape();
```

```
}
}
}

//does the appropriate color fill for the given range, value, and type
void colorChooser(float low, float high, float val, int colormode) {
int index = min(8, floor( 9 * (val - low)/(high - low)));
if (colormode == 0) {
//purple to green
switch(index) {
case 0:
fill(118, 42, 131);
break;
case 1:
fill(153, 112, 171);
break;
case 2:
fill(194, 165, 207);
break;
case 3:
fill(231, 212, 232);
break;
case 4:
fill(247, 247, 247);
break;
case 5:
fill(217, 240, 211);
break;
case 6:
fill(166, 219, 160);
break;
case 7:
fill(90, 174, 97);
break;
case 8:
fill(27, 120, 55);
break;
}
} else if (colormode == 1) {
switch(index) {
//red to blue
case 0:
fill(215, 48, 39);
break;
case 1:
fill(244, 109, 67);
break;
case 2:
fill(253, 174, 97);
break;
case 3:
fill(254, 224, 144);
break;
case 4:
```

```
fill(255, 255, 191);
break;
case 5:
fill(224, 243, 248);
break;
case 6:
fill(171, 217, 233);
break;
case 7:
fill(116, 173, 209);
break;
case 8:
fill(69, 117, 180);
break;
}
} else {
switch(index) {
//reds
case 0:
fill(255, 255, 204);
break;
case 1:
fill(255, 237, 160);
break;
case 2:
fill(254, 217, 118);
break;
case 3:
fill(254, 178, 76);
break;
case 4:
fill(253, 141, 60);
break;
case 5:
fill(252, 78, 42);
break;
case 6:
fill(227, 26, 28);
break;
case 7:
fill(189, 0, 38);
break;
case 8:
fill(128, 0, 38);
break;
default:
println("unindexed value " +index);
}
}
}


//z value class, contains all the information we store in each quad of heightfield.
class Zvalue {
```

```
float prevZ;
float z;
float tempZ;
float exactZ;
float error;

Zvalue(float prev, float zval, float temp) {
prevZ = prev;
z = zval;
tempZ = temp;
}

public float getZ() {
return z;
}

public void setZ(float newZ) {
z = newZ;
}

public float getError() {
return error;
}

public void setError(float newError) {
error = newError;
}

public float getExactZ() {
return exactZ;
}

public void setExactZ(float newZ) {
exactZ = newZ;
}

public float getPrev() {
return prevZ;
}

public void setPrev(float newPrev) {
prevZ = newPrev;
}

public float getTemp() {
return tempZ;
}

public void setTemp(float newTemp) {
tempZ = newTemp;
}
}
```

3.6 Source Code for Program 2

```
//  FINITE DIFFERENCE AND PSEDUO-STRECTRAL SCHEME FOR 2D WAVE EQN
//  Jake Wood
// written in Processing: https://processing.org/
//  Thesis work for Physics and Computer Science at Middlebury College
//  1/29/15 - 4/26/15
//  Advisors: Noah Graham and Christopher Andrews

final int screenWidth = 1000;//this changes the size of window on your screen
final int screenHeight = 600;

final int gridWidth = (int)pow(2, 6);//NUMBER OF GRIDS ACROSS, must be power of 2
final int gridHeight = (int)pow(2, 6);//NUMBER OF GRIDS VERTICALLY

boolean isFullScreen = false;//fullscreen not supported for webpage

// dx <= 1 / gridWidth; //imnplicity defined as such
final float dt = 0.001; //Can be experimented to see effects on stability
final float c = 4.0; //WAVE SPEED, 4 is arbitrary

final float kSQ = -sq(2 * PI);//fourier differentiation coefficient

//initial configs for surface perturbation
float amp1 = 20.0;
float narrow1 = 0.15;
float xcenter1 = 0.5;
float ycenter1 = 0.5;


float lowCol;//bounds on amplitude, for coloring
float upperCol;

int DETAIL;//1 is highest resolution

float theta;//legend angle
float pitch;//camera angle

float dx, dy;
float cfl;

boolean pause;//is simulation paused?
boolean input;//will program accept input now?
boolean record;//is program recording? not supported for web
boolean usingFourier;//using Pseduospectral method?

float[] xValues, yValues;//unchanging (x,y) coordinates in world view
Zvalue[][] zVals;

float[][] fourierDX;
float[][] fourierDY;

public void setup() {
if (isFullScreen)
size(displayWidth, displayHeight, P3D);
else
size(screenWidth, screenHeight, P3D);
frameRate(40);
pause = true;
input = false;
record = false;
usingFourier = false;
DETAIL = 1 + 2;//3 is good init for web
theta = 0.4;
pitch = PI*7/20;
setCam();
dx = 1.0 / gridWidth;
dy = 1.0 / gridHeight;
cfl = sq(c) * sq(dt/dx);
initXYArrays();
initZvalues();
lowCol = -0.01;//arbitrary but must be small on given
//scale and < uppercol. vise versa for upperCol
upperCol = 0.01;
}

public void reset() {
DETAIL = 3;
initZvalues();
if (pause)
input = false;
else
input = true;
lowCol = -0.01;
upperCol = 0.01;
}

public void draw() {
background(100, 100, 100);
setCam();
if (usingFourier) {
if (!pause) {
doFourierStep();
}
} else {
if (!pause) {
updateMesh1();
updateMesh2();
}
}
drawLegend();
drawMesh();
if (!pause && record)//records every frame, not supported on web
saveFrame();
}


//handles key presses
```

```
public void keyPressed() {
char letter = key;
userInput(letter);
switch(letter) {
case '+':
saveFrame();
println("saving frame");
break;
case 'z':
pitch = min(PI*9.9/20.0, pitch + PI/20.0);
break;
case 'x':
pitch = max(PI/10, pitch - PI/20.0);
break;
case '/':
reset();
break;
case 'q':
theta += 0.1;
break;
case 'w':
theta -= 0.1;
break;
case '.':
DETAIL = min(12, DETAIL+1);
break;
case '0':
DETAIL = max(1, DETAIL-1);
break;
case '5':
input = true;
pause = !pause;
break;
case '-':
record =! record;
break;
}
setCam();
}

//sets the camera view
void setCam() {
camera(width/2.0, 2*height * sin(pitch), 2*height * cos(pitch),
width/2.0, height/2.0, 0, 0, 1.0, 0);
}

//handles key presses after a reset screen
public void userInput(char k) {
if (!input) {
switch(k) {
case '4':
xcenter1 = max(0, xcenter1-1.0/20.0);
initZvalues();
break;
case ’6’:
xcenter1 = min(1, xcenter1+1.0/20.0);
initZvalues();
break;
case ’2’:
ycenter1 = min(1, ycenter1+1.0/20.0);
initZvalues();
break;
case ’8’:
ycenter1 = max(0, ycenter1-1.0/20.0);
initZvalues();
break;
case ’7’:
amp1 = max(-40.0, amp1 - 1);
initZvalues();
break;
case ’9’:
amp1 = min(40.0, amp1 + 1);
initZvalues();
break;
case ’1’:
narrow1 = max(0.05, narrow1 - 0.025);
initZvalues();
break;
case ’3’:
narrow1 = min(0.5, narrow1 + 0.025);
initZvalues();
break;
case ’*’:
usingFourier =! usingFourier;
reset();
break;
}
}
}

//initializes (X,Y) array coordinates
public void initXYArrays() {
xValues = new float [gridWidth];
yValues = new float [gridHeight];
zVals = new Zvalue[gridWidth][gridHeight];
fourierDX = new float[gridWidth][gridHeight];
fourierDY = new float[gridWidth][gridHeight];
//init x and y values
for (int i = 0; i < gridWidth; i++)
xValues[i] =  width * i / gridWidth;//width is whatever is set in size()
for (int j = 0; j < gridHeight; j++)
yValues[j] = height * j / gridHeight;
}

//init surface purturbation
public float multiGaussian(float x, float y) {
return amp1 * exp(-1.0f * ( sq( x / gridWidth - xcenter1) +
sq( y / gridHeight - ycenter1)) / sq(narrow1));
```

```
}

//initilizes height field values --> 2d gaussian
public void initZvalues() {
for (int h = 0; h < gridHeight; h ++) {
for (int w = 0; w < gridWidth; w ++) {
float waveHeight = multiGaussian(w, h);
zVals[w][h] = new Zvalue(waveHeight, waveHeight, 0);
}
}
}

//does the appropriate color fill for the given range, value, and type
void colorChooser(float low, float high, float val) {
int index = min(8, floor( 9 * (val - low)/(high - low)));
switch(index) {
//red to blue
case 0:
fill(215, 48, 39);
break;
case 1:
fill(244, 109, 67);
break;
case 2:
fill(253, 174, 97);
break;
case 3:
fill(254, 224, 144);
break;
case 4:
fill(255, 255, 191);
break;
case 5:
fill(224, 243, 248);
break;
case 6:
fill(171, 217, 233);
break;
case 7:
fill(116, 173, 209);
break;
case 8:
fill(69, 117, 180);
break;
}
}

//draws the colorscale and labels
public void drawLegend() {
float inc = 0.1f;
float scale = 0.6f;
float rigthEdge = width/8;
textSize(height*inc*scale);
pushMatrix();
```

```
rotateX(theta);
for (float i=0; i<1; i+=inc) {
colorChooser(0, 1, i);
pushMatrix();
rotateX(-PI/2.0f);
text(str(i*(upperCol - lowCol) + lowCol), rigthEdge + 2, -height*(i+inc/4)*scale, 1);
popMatrix();
beginShape(QUAD);
vertex(0, 0, height*i*scale);
vertex(rigthEdge, 0, height*i*scale);
vertex(rigthEdge, 0, height*(i+inc)*scale);
vertex(0, 0, height*(i+inc)*scale);
endShape();
}
popMatrix();
}

//returns second spatial x derivitive at location (fourier method)
public float getFdx(int w, int h) {
return fourierDX[w][h];
}

//returns second spatial y derivitive at location (fourier method)
public float getFdy(int w, int h) {
return fourierDY[h][w];
}


//updateMesh 1 and 2 trade which arrays are holding which
//values to cut down on overhead of copying arrays uneccessarily each time step
public void updateMesh1() {
//loops over interior points only
for (int h = 1; h < gridHeight - 1; h++) {
for (int w = 1; w < gridWidth - 1; w++) {
Zvalue cell = zVals[w][h];
cell.setTemp(2.0f*cell.getZ() - cell.getPrev() + cfl*(zVals[w+1][h].getZ() -
2.0f*cell.getZ() + zVals[w-1][h].getZ()) + cfl*(zVals[w][h-1].getZ() -
2.0f*cell.getZ() + zVals[w][h+1].getZ()));
cell.setPrev(cell.getZ());
}
}
}

public void updateMesh2() {
//loops over interior points only
for (int h = 1; h < gridHeight - 1; h ++) {
for (int w = 1; w < gridWidth - 1; w ++) {
//only need exact z in 2nd of two updates
Zvalue cell = zVals[w][h];
cell.setZ(2.0f*cell.getTemp() - cell.getZ() + cfl*(zVals[w+1][h].getTemp() -
2.0f*cell.getTemp() + zVals[w-1][h].getTemp()) + cfl*(zVals[w][h-1].getTemp() -
2.0f*cell.getTemp() + zVals[w][h+1].getTemp()));
cell.setPrev(cell.getTemp());
}
}
}

//draws each rectangle on the surface of the height field
public void drawMesh() {
float middleHeight = zVals[gridWidth/2][gridHeight/2].getZ();
//the following conditions dynamically update the bounds
//of color so the legend color syncs with arbitrary height field
if (middleHeight > upperCol) {
upperCol = middleHeight;
} else if (middleHeight < lowCol) {
lowCol = middleHeight;
}
for (int h = 0; h < gridHeight-DETAIL; h+=DETAIL) {
//removed +1 from conditions for fourier mesh draw
for (int w = 0; w < gridWidth-DETAIL; w+=DETAIL) {
float wh = zVals[w][h].getZ();
float w1h = zVals[w + DETAIL][h].getZ();
float w1h1 = zVals[w + DETAIL][h + DETAIL].getZ();
float wh1 = zVals[w][h + DETAIL].getZ();
//draws the vertical meshing
if (h%4 == 0) {
stroke(1);
line(xValues[w], yValues[h], amp1*wh, xValues[w + DETAIL], yValues[h], amp1*w1h);
noStroke();
}
//draws the horizontal
if (w%4 == 0) {
stroke(1);
line(xValues[w + DETAIL], yValues[h], amp1*w1h, xValues[w + DETAIL],
yValues[h + DETAIL], amp1*w1h1);
noStroke();
}
beginShape(QUAD);
colorChooser(lowCol, upperCol, (0.25 * (wh + w1h + w1h1 + wh1)));
vertex(xValues[w], yValues[h], amp1*wh);
vertex(xValues[w + DETAIL], yValues[h], amp1*w1h);
vertex(xValues[w + DETAIL], yValues[h + DETAIL], amp1*w1h1);
vertex(xValues[w], yValues[h + DETAIL], amp1*wh1);
endShape();
}
}
}

//copies heightfield in row order and column order for fourier step
public void updateFourierLists() {
for (int h = 0; h < gridHeight; h ++) {
for (int w = 0; w < gridWidth; w ++) {
float fieldHeight = zVals[w][h].getZ();
fourierDX[w][h] = fieldHeight;
fourierDY[h][w] = fieldHeight;
}
}
}
```

```
//rows (x dir) fourier transformed and differentiated
public void fourierUpdateDX() {
for (int w = 0; w < gridWidth; w ++) {
realft(fourierDX[w], 1);//transform into fourier space
for (int h = 0; h <= gridHeight/2; h ++) {
fourierDX[w][h] = fourierDX[w][h] * kSQ * sq(h);
}
for (int h = gridHeight/2 + 1; h < gridHeight; h ++) {
fourierDX[w][h] = fourierDX[w][h] * kSQ * sq(h - gridHeight);
}
realft(fourierDX[w], -1);//transform back into real space
}
}

//columns (y dir) get fourier transformed and differentiated
public void fourierUpdateDY() {
for (int h = 0; h < gridHeight; h ++) {
realft(fourierDY[h], 1);//transform into fourier space
for (int w = 0; w <= gridWidth/2; w ++) {
fourierDY[h][w] = fourierDY[h][w] * kSQ * sq(w);
}
for (int w = gridWidth/2 + 1; w < gridWidth; w ++) {
fourierDY[h][w] = fourierDY[h][w] * kSQ * sq(w - gridWidth);
}
realft(fourierDY[h], -1);//transform back into real space
}
}

//performs FDM with differentiated, fourier transformed, spatial components
public void fourierTimeStep() {
float cft = sq(dt);
for (int h = 0; h < gridHeight; h ++) {
for (int w = 0; w < gridWidth; w ++) {
Zvalue cell = zVals[w][h];
cell.tempAssignTimeStep(cft * (getFdx(w, h) + getFdy(w, h)));//forward difference update
cell.setZtoPrev();
cell.setZasTemp();
}
}
}

//grouped fourier steps
public void doFourierStep() {
updateFourierLists();
fourierUpdateDX();
fourierUpdateDY();
fourierTimeStep();
}

//class holds all the important infomation held in each array cell
class Zvalue {
float prevZ;
float z;
float tempZ;

Zvalue(float prev, float zval, float temp) {
prevZ = prev;
z = zval;
tempZ = temp;
}

public float getZ() {
return z;
}

public void setZ(float newZ) {
z = newZ;
}

public float getPrev() {
return prevZ;
}

public void setPrev(float newPrev) {
prevZ = newPrev;
}

public float getTemp() {
return tempZ;
}

public void setTemp(float newTemp) {
tempZ = newTemp;
}

public void setZtoPrev() {
prevZ = z;
}

public void setZasTemp() {
z = tempZ;
}

public void tempAssignTimeStep(float cdx2) {
tempZ = 2.0f * z - prevZ + cdx2;//forward difference update
}
}

/*
*  I DID NOT WRITE the below code.
*  The following implements a naively optimized real FFT
*  The original source code can be found at:
*  https://code.google.com/p/scalalab/wiki/JavaFFTvsNative
*/
public final  static void swap(float x[], int a, int b) {
float t = x[a];
x[a] = x[b];
```

```
x[b] = t;
}

public static void four1(final float[] data, final int n, final int isign) {
int nn, mmax, m, j, istep, i;
float wtemp, wr, wpr, wpi, wi, theta, tempr, tempi;
if (n<2 || (n&(n-1))!= 0) throw new IllegalArgumentException("n must be power of 2 in four1");
nn = n << 1;
j = 1;
for (i=1; i<nn; i+=2) {
if (j > i) {
swap(data, j-1, i-1);
swap(data, j, i);
}
m=n;
while (m >= 2 && j > m) {
j -= m;
m >>= 1;
}
j += m;
}
mmax=2;
while (nn > mmax) {
istep=mmax << 1;
theta=isign*(6.283185307179586f/mmax);
wtemp=sin(0.5f*theta);
wpr = -2.0f*wtemp*wtemp;
wpi=sin(theta);
wr=1.0f;
wi=0.0f;
for (m=1; m<mmax; m+=2) {
for (i=m; i<=nn; i+=istep) {
j=i+mmax;
tempr=wr*data[j-1]-wi*data[j];
tempi=wr*data[j]+wi*data[j-1];
data[j-1]=data[i-1]-tempr;
data[j]=data[i]-tempi;
data[i-1] += tempr;
data[i] += tempi;
}
wr=(wtemp=wr)*wpr-wi*wpi+wr;
wi=wi*wpr+wtemp*wpi+wi;
}
mmax=istep;
}
}

/**
* Calculates the Fourier transform of a set of n real-valued data points.
* Replaces these data (which are stored in array data[0..n-1]) by the
* positive frequency of half of their complex Fourier transform. The real-valued
* first and last components of the complex transform are returned as elements
* data[0] and data[1], respectively. n must be a power of 2. This routine
* also calculates the inverse transform of a complex data array if it is the
* transform of real data. (Result in this case must be multiplied by 2/n.)
*
* @param data
* @param isign
*/
public static void realft(final float[] data, final int isign) {
int i, i1, i2, i3, i4, n=data.length;
float c1=0.5f, c2, h1r, h1i, h2r, h2i, wr, wi, wpr, wpi, wtemp;
float theta=PI/(n>>1);
if (isign == 1) {
c2 = -0.5f;
four1(data, n/2, 1);
} else {
c2=0.5f;
theta = -theta;
}
wtemp=sin(0.5f*theta);
wpr = -2.0f*wtemp*wtemp;
wpi=sin(theta);
wr=1.0f+wpr;
wi=wpi;
for (i=1; i< (n>>2); i++) {
i2=1+(i1=i+i);
i4=1+(i3=n-i1);
h1r=c1*(data[i1]+data[i3]);
h1i=c1*(data[i2]-data[i4]);
h2r= -c2*(data[i2]+data[i4]);
h2i=c2*(data[i1]-data[i3]);
data[i1]=h1r+wr*h2r-wi*h2i;
data[i2]=h1i+wr*h2i+wi*h2r;
data[i3]=h1r-wr*h2r+wi*h2i;
data[i4]= -h1i+wr*h2i+wi*h2r;
wr=(wtemp=wr)*wpr-wi*wpi+wr;
wi=wi*wpr+wtemp*wpi+wi;
}
if (isign == 1) {
data[0] = (h1r=data[0])+data[1];
data[1] = h1r-data[1];
} else {
data[0]=c1*((h1r=data[0])+data[1]);
data[1]=c1*(h1r-data[1]);
four1(data, n/2, -1);
}
}
```

Bibliography

[1] Steven G. Johnson. *Notes on FFT-based differentiation.* MIT Applied Mathematics, 2011.

[2] Nikos Drakos & Ross Moore. *Von Neumann Stability Analysis.* University of Leeds, 1996 & Macquarie University, Sydney, 1999. http://www.physics.drexel.edu/courses/Comp_Phys/Physics-307/von-neumann/index.html

[3] Feys, Jan. "Math317 Twenty-Second & Twenty-Fourth Tutorial." Web. 13 Apr. 2015. <http://www.math.mcgill.ca/feys/documents/tutnotesR22n24.pdf>.

[4] J. Olver, Peter. "Numerical Analysis Lecture Notes." 2008. Web. 14 Apr. 2015.<http://www.math.umn.edu/∼olver/num_/lnp.pdf>.

[5] Sigal Gottlieb and David Gottlieb (2009) Spectral methods. Scholarpedia, 4(9):7504.

[6] Heiner Igel. *Pseudospectral Methods*, Lecture Notes. Munich University. web. 22 Apr. 2015.

[7] H. Isliker. *A tutorial on the pseudo-spectral method.* University of Thessaloniki, September 2004. web. web. 22 Apr. 2015.

[8] M. Causon, D., and C. G. Mingham. "Introduction." Introductory Finite Difference Methods for PDEs. 1st ed. Ventus Aps, 2010. 144. Print.

[9] Ryan C. Daileda. "The Two Dimensional Wave Equation". Trinity University. March 1, 2012. web. 26 Apr. 2015.