Finite Difference Methods Applied to the Schrödinger Equation

Nicholas Walker[1]

[1] *Department of Physics and Astronomy, Louisiana State University, Baton Rouge, LA 70803, USA*

In this paper, a numerical approach to solving the Schrödinger equation is taken. Through the use of finite difference methods, the conversion of the problem from a continuous partial differential equation to a discrete matrix equation is detailed along with considerations of the strengths and weaknesses of the method. Strategies and explanations for setting up a system, calculating eigenvalues and eigenvectors, and performing time evolutions are described and analyzed. The choice of the discretization was found to have the largest impact on accuracy and performance. More sample points in the discrete domain return more accurate results since it more closely resembles the continuous spectrum, but this also greatly increases the computational complexity of the problem. For a system with n points in d dimensions, doubling the number of points to $2n$ in each dimension increases the worst-case number of entries in operators from n^{2d} to $4^d n^{2d}$, for instance. This may be a necessary sacrifice, however, as some systems require rather small discretization to maintain stability. The stability of the time-evolution of a state depends not only on the size of the time step, but also on the size of the spatial steps. Furthermore, more sample points result in larger operators, which means more eigenvalues and eigenvectors can be calculated. Higher order difference methods may also be used to control error at the cost of computational complexity. Despite these issues, the method gives satisfactory results and error tolerances for many one-dimensional problems such as the infinite square well and the linear harmonic oscillator. The types of problems that can be considered are limited. Systems with Dirichlet boundary conditions, periodic boundary conditions, or trapped systems can be treated accurately. However, due to the discrete nature of the numerical approach, systems that have access to $\mathbb{R}^n$ cannot be accurately represented. Essentially, the wavefunctions must terminate before reaching the boundaries or the wavefunctions must "wrap around" to the other side of the spatial domain. With these limitations in mind, many different useful systems can be analyzed with this approach. Many such systems are non-trivial or impossible to solve analytically, securing this method's place amongst the tools used in quantum physics.

I. INTRODUCTION

In non-relativistic quantum mechanics, the dynamics of a quantum particle are described by the Schrödinger equation, a partial differential equation that describes the time evolution of a quantum state.

$$\hat{\mathbf{H}}\,|\Psi\rangle = i\hbar\frac{\partial}{\partial t}\,|\Psi\rangle \tag{1}$$

Where $|\Psi\rangle$ is a quantum state that describes the particle. The linear operator $\hat{\mathbf{H}}$ is the Hamiltonian operator, which corresponds to the system energy.

$$\hat{\mathbf{H}} = \frac{\hat{\mathbf{p}}^2}{2m} + V = -\frac{\hbar^2}{2m}\boldsymbol{\nabla}^2 + V \tag{2}$$

Where $\hat{\mathbf{p}}$ is the momentum operator and V is the system potential. Since the Hamiltonian represents the system energy, it's eigenvalues and eigenfunctions correspond to the energies and energy states of the system, respectively. The time-independent form of the Schrödinger equation can then be written as

$$\hat{\mathbf{H}}\,|\Psi\rangle = E\,|\Psi\rangle \tag{3}$$

The Hamiltonian is a Hermitian operator, so the energy eigenvalues are real numbers. In the field of quantum mechanics, it is often useful to know not only the time evolution of a particular quantum state describing a particle, but the spectrum of available energies and corresponding quantum states for a specific choice of potential. In order to fully solve such a system, extra conditions such as the boundary conditions of the system must be provided. Analytically, many methods have been developed for solving these partial differential equations. However, for many systems, this can be very difficult or simply cumbersome. Numerical methods can instead be applied to provide a different approach to the same problem.

Applying numerical methods to solving partial differential equations carries with it many of its own challenges which will be described in this paper. For instance, many problems in quantum mechanics are solved on real spaces $\mathbb{R}^n$. Computationally, the real number line cannot be exactly represented. Some discretization must be chosen. One must be very careful when choosing the discretization, however, as the problem solution may be very sensitive to that choice. Since the domain and consequently the quantum states must be discretized. This then has the consequence that the Hilbert space spanned by the eigenstates found numerically is not the same Hilbert space spanned by the analytical eigenfunctions, as the analytical Hilbert space is of infinite dimension. Additionally, any operator that acts on the state space must similarly be of finite dimension in the numerical formulation. This main feature of discretization carries with it many potential issues that must be explored before committing to applying numerical methods to a quantum

problem. Eigenfunctions will become finite dimensional eigenvectors and linear operators will become finite dimensional matrices. The first question that arises is how these features will be represeted numerically. First, let us explore the nature of numerical differentiation, as differential operators are of course a fundamental feature of partial differential equations in general.

II. FINITE DIFFERENCE SCHEMES

Analytically, derivatives are rather simple. Numerically, however, derivatives can prove rather challenging to calculate. The preferred method in numerical analysis is the use of finite difference methods. Simply put, a finite difference equation is a discrete approximation to a derivative and is thus a discretization method itself. Assuming a function $f(x)$ is differentiable at the point x_0, Taylor's theorem states that the value of the function at the point $x_0 + h$ may be expanded into a Taylor series

$$f(x_0 + h) = f(x_0) + h\frac{\mathrm{d}f(x_0)}{\mathrm{d}x} + \frac{h^2}{2}\frac{\mathrm{d}^2 f(x_0)}{\mathrm{d}x^2} + ... \quad (4)$$

Dividing by h and subtracting the first term from both sides,

$$\frac{f(x_0 + h) - f(x_0)}{h} = \frac{\mathrm{d}f(x_0)}{\mathrm{d}x} + \frac{h}{2}\frac{\mathrm{d}^2 f(x_0)}{\mathrm{d}x^2} + ... \quad (5)$$

Thus, the approximate derivative of the function is

$$\frac{\mathrm{d}f(x_0)}{\mathrm{d}x} \approx \frac{f(x_0 + h) - f(x_0)}{h} \quad (6)$$

So, the first derivative of the function at the point x_0 can be approximated by simply taking the difference of the function at two points separated by h and dividing by the separation h. This particular method is called the two-point forward difference method. Other methods using more points or different approaches to differencing can be derived using Taylor's theorem as well. Note that, analytically, if a limit is taken as the value of h approaches 0, the result should be exact. However, numerical limits prevent an exact answer from being achieved by such methods because of roundoff error [1]. This is because digital computers have size and precision limits on how accurately numbers can be represented. For instance, the value of $f(x_0)$ can be calculated to an accuracy of $Cf(x_0)$, where C is some small number typically on the order of floating point precision, or 10^{-16} [1]. The value of $f(x_0+h)$ is expected to be close to that of $f(x_0)$, so the roundoff error in the calculated derivative, in the worst case, can be represented as

$$\epsilon_{\mathrm{R}} = \frac{2C|f(x_0)|}{h} \quad (7)$$

However, another considerable source of error is the truncation error [1]. Consider the remainder terms that were ignored to make the approximation.

$$R(h) = -\frac{h}{2}\frac{\mathrm{d}^2 f(x_0)}{\mathrm{d}x^2} - ... \quad (8)$$

For some number ξ in the interval $[x_0, x_0 + h]$, the remainder term can be simply taken to be

$$R(h) = -\frac{h}{2}\frac{\mathrm{d}^2 f(\xi)}{\mathrm{d}x^2} \quad (9)$$

In the worst case, this will add on to the error caused by the roundoff [1]. The total error can then be expressed as

$$\epsilon = \frac{2C|f(x_0)|}{h} + \frac{h}{2}\left|\frac{\mathrm{d}^2 f(\xi)}{\mathrm{d}x^2}\right| \quad (10)$$

Upon minimization of the error ϵ with respect to the spacing h, we find that the optimal values are

$$h = \sqrt{4C\left|\frac{f(x_0)}{\frac{\mathrm{d}^2 f(\xi)}{\mathrm{d}x^2}}\right|}; \quad \epsilon = \sqrt{4C\left|f(x_0)\frac{\mathrm{d}^2 f(\xi)}{\mathrm{d}x^2}\right|} \quad (11)$$

This presents some immediately worrying results. If the second derivative of the function being differentiated blows up on the interval, then a very small step size will need to be employed to compensate. This can be computationally expensive, as the number of calculations needed to take the derivative of a function at n points goes as $2n - 1$ (assuming terminating boundaries at the endpoints) under this scheme. Furthermore, the maximum value of the second derivative is not always known. After all, the first derivative is being calculated numerically in this situation. As a rule of thumb, if the value of the function and its second derivative are on the order of 1, the step size should be approximately on the order of 10^{-8} and give an error on the same order. From there, adjustments can be made accordingly. Sufficient testing before relying on such methods for results is thus very important.

A. Central Differences

The forward difference method, while very easy to derive and compute, is not very accurate, as can be seen from the analysis in the main section. A different technique can be used in order to mitigate this issue to a certain extent. This technique is called the central difference. Consider the following two Taylor expansions

$$f(x_0+h) = f(x_0) + h\frac{\mathrm{d}f(x_0)}{\mathrm{d}x} + \frac{h^2}{2}\frac{\mathrm{d}^2 f(x_0)}{\mathrm{d}x^2} + ... \quad (12)$$

$$f(x_0-h) = f(x_0) - h\frac{\mathrm{d}f(x_0)}{\mathrm{d}x} + \frac{h^2}{2}\frac{\mathrm{d}^2 f(x_0)}{\mathrm{d}x^2} - ... \quad (13)$$

Subtracting the two equations,

$$f(x_0+h) - f(x_0-h) = 2h\frac{\mathrm{d}f(x_0)}{\mathrm{d}x} + \frac{h^3}{3}\frac{\mathrm{d}^3 f(x_0)}{\mathrm{d}x^3} + ... \quad (14)$$

Dividing by $2h$ and rearranging,

$$\frac{\mathrm{d}f(x_0)}{\mathrm{d}x} = \frac{f(x_0+h) - f(x_0-h)}{2h} - \frac{h^2}{6}\frac{\mathrm{d}^3 f(x_0)}{\mathrm{d}x^3} - ... \quad (15)$$

Once again, using the same methods from the last part, the error terms can be collected by evaluating the leading error term at ξ in the interval $[x_0-h, x_0+h]$ so that the total error can be taken to be, in the worst case,

$$\epsilon = \frac{C|f(x_0)|}{h} + \frac{h^2}{6}\left|\frac{\mathrm{d}^3 f(\xi)}{\mathrm{d}x^3}\right| \quad (16)$$

Minimizing with respect to h,

$$h = \sqrt[3]{3C\left|\frac{f(x_0)}{\frac{\mathrm{d}^3 f(\xi)}{\mathrm{d}x^3}}\right|}; \quad \epsilon = \frac{3^{\frac{2}{3}}}{2}\sqrt[3]{C^2\left|f(x_0)^2\frac{\mathrm{d}^3 f(\xi)}{\mathrm{d}x^3}\right|} \quad (17)$$

If the third derivative blows up, there will be trouble. But, this time around, if the function and its derivatives are of the order 1 on the interval, the step size should be on the order of 10^{-6} and give an error on the order of 10^{-11}. This is a great improvement over the forward difference and requires $2n-2$ (assuming terminating boundaries at the endpoints) calculations to calculate the derivative of a function on n sample points. For these reasons, central difference methods will be employed for the remainder of this paper when taking spatial derivatives. The error can also be further reduced through the use of more sample points with which to calculate the derivatives. While this does reduce the error, this also increases the computational complexity. After two sample points comes using four, which doubles the number of calculations that need to be performed to take the derivative of a function.

B. Higher Order Derivatives

For higher order derivatives, similar derivations can be used. For the second derivative, for instance, the two Taylor expansions used to derive the first order derivative can be written out again.

$$f(x_0+h) = f(x_0) + h\frac{\mathrm{d}f(x_0)}{\mathrm{d}x} + \frac{h^2}{2}\frac{\mathrm{d}^2 f(x_0)}{\mathrm{d}x^2} + ... \quad (18)$$

$$f(x_0-h) = f(x_0) - h\frac{\mathrm{d}f(x_0)}{\mathrm{d}x} + \frac{h^2}{2}\frac{\mathrm{d}^2 f(x_0)}{\mathrm{d}x^2} - ... \quad (19)$$

Adding the two equations, rearranging, and collecting the error terms by using $\xi \in [x_0-h, x_0+h]$ returns the following result

$$\frac{\mathrm{d}^2 f(x_0)}{\mathrm{d}x^2} = \frac{f(x_0-h) - 2f(x_0) + f(x_0+h)}{h^2} - \frac{h^2}{12}\frac{\mathrm{d}^4 f(\xi)}{\mathrm{d}x^4} \quad (20)$$

The total error can be written as

$$\epsilon = \frac{4C|f(x_0)|}{h^2} + \frac{h^2}{12}\left|\frac{\mathrm{d}^4 f(\xi)}{\mathrm{d}x^4}\right| \quad (21)$$

Minimization returns the following results

$$h = \sqrt[4]{48C\left|\frac{f(x_0)}{\frac{\mathrm{d}^4 f(\xi)}{\mathrm{d}x^4}}\right|}; \quad \epsilon = \sqrt{\frac{4}{3}C\left|f(x_0)\frac{\mathrm{d}^3 f(\xi)}{\mathrm{d}x^3}\right|} \quad (22)$$

Thus, for first and fourth derivatives on the order of 1, the optimal step size is on the order of 10^{-4} and the error goes as 10^{-8}. This error is comparable to that seen in the forward difference method for the first derivative. For the purposes of he calculations in this paper, this three point method will be accurate enough. In addition to that, since the kinetic energy is proportional to the square momentum and higher orders are rarely needed, only second derivatives will need to regularly be taken. However, the coefficients for higher order derivatives and greater numbers of points used can be calculated using Fornberg's algorithm, which is a recursive scheme for calculating the weights for arbitrary orders of derivatives and associated errors [2].

III. THE EIGENVALUE PROBLEM

Returning to the Schrödinger equation, consider the time-independent equation once again

$$\hat{\mathbf{H}}\,|\Psi\rangle = \left(-\frac{\hbar^2}{2m}\boldsymbol{\nabla}^2 + V\right)|\Psi\rangle = E\,|\Psi\rangle \quad (23)$$

The structure of the problem indicates that this is an eigenvalue problem. When the Hamiltonian operator is applied to the state vector $|\Psi\rangle$, the linear transformation does not change the direction of the state vector. That is to say that the linear transformation is equivalent to multiplication by a scalar. In quantum mechanical terms, this scalar is the observable associated with the operator.

In the case of the Hamiltonian operator, the associated observable is the energy of the state. The equation is presented in the position basis in this case, so the operator is a continuous operator. As mentioned before, this presents a computational issue, as a continuous spatial spectrum cannot be exactly represented. The approximate numerical representation is detailed in the following subsection. For now, the problem is begin considered in one dimension only.

A. Representation

In order to represent a quantum system for a numerical treatment, the domain must be built first. For a spatial domain with grid spacing h and n sample points, this can be expressed in set notation as

$$x = \{ih; i \in \mathbb{N}, i < n\} \tag{24}$$

From a numerical perspective, this would simply be stored as an array. The wavefunction can then simply be an array of values that correspond to each grid point. This method very cleanly allows for the representation of the domain and the wavefunction in a numerical scheme, but does have limitations. The length of an array must be finite. This means that an infinite domain such as the real line $\mathbb{R}$ cannot be represented. This immediately limits the capability of the numerical scheme. The only problems defined on the real numbers that would be treatable by this method would be ones where a particle is bound by a potential. For example, a particle in a harmonic trap would be able to represented as long as the eigenvectors decayed to 0 before the boundaries on the the finite domain were reached. Essentially, even though the problem is defined on the real line, the area of interest would be confined to a finite interval. Otherwise, problems with either Dirichlet boundary conditions or periodic boundary conditions on a finite interval can be treated. The Dirichlet boundary conditions would force the eigenfunctions to be zero at the boundaries of the finite domain, giving the conditions seen for a infinite square well. From there, any potential could be placed within the infinite square well for analysis. One common problem set up in this fashion is a Dirac delta well in an infinite square well. Period boundary conditions, on the other hand, would allow for the system to "wrap" around such that the wavefunction must match on opposite sides of the domain. This is useful for many problems in condensed matter physics, for instance, where periodic boundary conditions are employed to represent an infinitely repeating lattice. Problems that simply cannot be represented, however, include the free particle, the Dirac delta well, various scattering problems, etc.

Now that the representation of the domain and the wavefunction have been established along with the limitations that follow, we must consider how operators will be treated. First off, let us consider how operators that are functions of position will work. Since the domain is the position in this treatment of the problem, all operators and eigenvectors are in the position basis. So, one would expect that a function such as the potential or the position operator would simply be vectors. However, it was already established that such operators will need to be linear transformations, and thus represented by finite matrices in this representation. When multiplying two functions, one can imagine a pairwise product being taken between each corresponding value. This can be done with matrices by simply placing the values of the operator along the diagonal of a matrix. For some function f with discrete values f_i for each point x_i in the domain, it's corresponding matrix operator will be of the form

$$\begin{bmatrix} f_0 & 0 & \dots & \dots & \dots & 0 \\ 0 & f_1 & \ddots & \ddots & & \vdots \\ \vdots & \ddots & f_2 & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & \ddots & \vdots \\ \vdots & & \ddots & \ddots & \ddots & 0 \\ 0 & \dots & \dots & \dots & 0 & f_n \end{bmatrix} \begin{bmatrix} \Psi_0 \\ \Psi_1 \\ \vdots \\ \Psi_n \end{bmatrix} = \begin{bmatrix} f_0\Psi_0 \\ f_1\Psi_1 \\ \vdots \\ f_n\Psi_n \end{bmatrix} \tag{25}$$

Note that since most of the values in the matrix are simply 0, the use of sparse matrices is highly desirable for numerical calculations to save on both memory usage and computational time. For the potential operator, for instance, this means that

$$V(x)\Psi(x) \to \begin{bmatrix} V_0 & 0 & \dots & \dots & \dots & 0 \\ 0 & V_1 & \ddots & \ddots & & \vdots \\ \vdots & \ddots & V_2 & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & \ddots & \vdots \\ \vdots & & \ddots & \ddots & \ddots & 0 \\ 0 & \dots & \dots & \dots & 0 & V_n \end{bmatrix} \begin{bmatrix} \Psi_0 \\ \Psi_1 \\ \vdots \\ \Psi_n \end{bmatrix} \tag{26}$$

For differential operators, the approach is a little different. Recall the central difference equation for the first order derivative.

$$\frac{\mathrm{d}f(x_0)}{\mathrm{d}x} = \frac{f(x_0 + h) - f(x_0 - h)}{2h} \tag{27}$$

Now, since the wavefunctions $|\Psi\rangle$ is going to be differentiated, assuming that it is defined on an equally spaced grid of spacing h, the equation for the derivative at the i^{th} grid point can be written as

$$\frac{\mathrm{d}\Psi_i}{\mathrm{d}x} = \frac{\Psi_{i+1} - \Psi_{i-1}}{2h} \tag{28}$$

This differential operator can then be represented in matrix form rather simply as

$$\mathbf{D}^1 = \frac{1}{2h}\begin{bmatrix} 0 & +1 & 0 & \dots & \dots & 0 \\ -1 & 0 & +1 & \ddots & & \vdots \\ 0 & -1 & 0 & +1 & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & \ddots & 0 \\ \vdots & & \ddots & \ddots & \ddots & +1 \\ 0 & \dots & \dots & 0 & -1 & 0 \end{bmatrix} \tag{29}$$

Where $\mathbf{D}^1$ simply denotes the first order derivative operator. Since there are no points beyond the boundaries, the first and last points do not have preceding or following values, respectively, to use in the expression of the derivative. Since they are neglected, they are simply taken to be 0 automatically, fulfilling Dirichlet boundary conditions. Periodic boundary conditions can be incorporated by simply using the last and first points, respectively, for the needed points in the difference formula, like so

$$\mathbf{D}^1_{\text{periodic}} = \frac{1}{2h}\begin{bmatrix} 0 & +1 & 0 & \dots & 0 & -1 \\ -1 & 0 & +1 & \ddots & & 0 \\ 0 & -1 & 0 & +1 & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & \ddots & 0 \\ 0 & & \ddots & \ddots & \ddots & +1 \\ +1 & 0 & \dots & 0 & -1 & 0 \end{bmatrix} \tag{30}$$

Returning to the standard Dirichlet conditions, under this scheme, the momentum operator would then become

$$\hat{\mathbf{p}} = -i\hbar\frac{\partial}{\partial x} \to -\frac{i\hbar}{2h}\begin{bmatrix} 0 & +1 & 0 & \dots & \dots & 0 \\ -1 & 0 & +1 & \ddots & & \vdots \\ 0 & -1 & 0 & +1 & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & \ddots & 0 \\ \vdots & & \ddots & \ddots & \ddots & +1 \\ 0 & \dots & \dots & 0 & -1 & 0 \end{bmatrix} \tag{31}$$

For the kinetic energy, the second derivative formula can be used such that

$$\frac{\hat{\mathbf{p}}^2}{2m} = -\frac{\hbar^2}{2m}\frac{\partial^2}{\partial x^2} \to \frac{\hbar^2}{2mh^2}\begin{bmatrix} 2 & -1 & 0 & \dots & \dots & 0 \\ -1 & 2 & -1 & \ddots & & \vdots \\ 0 & -1 & 2 & -1 & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & \ddots & 0 \\ \vdots & & \ddots & \ddots & \ddots & -1 \\ 0 & \dots & \dots & 0 & -1 & 2 \end{bmatrix} \tag{32}$$

With these examples, it is now possible to define the Hamiltonian operator

$$\hat{\mathbf{H}} = -\frac{\hbar^2}{2m}\boldsymbol{\nabla}^2 + V(x) \to -\frac{\hbar^2}{2m}\mathbf{D}^2 + \mathbf{V} \tag{33}$$

Where $\mathbf{V}$ is the potential operator detailed earlier and $\mathbf{D}^2$ is the second derivative difference operator detailed earlier. The Hamiltonian matrix operator in the position basis has now been fully defined. Note that in order to keep the Hamiltonian matrix Hermitian, central differencing must be used for the differential operators that are components of the Hamiltonian.

B. Multiple Dimensions

The matrix operators defined in the previous section only work for one spatial dimension. This presents an issue when representing multiple dimensions. Fundamentally, it is desirable to retain the same structure of the problem, so the wavefunction should still be represented as a vector and the operators should still be represented as matrices. For the wavefunctions, one could imagine a matrix structure for the two-dimensional case. In a 2D system where the x coordinate is indexed as a row up to m and the y coordinate is indexed as a column up to n, the wavefunction would appear like so

$$\Psi = \begin{bmatrix} \Psi_{00} & \Psi_{01} & \dots & \dots & \dots & \Psi_{0n} \\ \Psi_{10} & \Psi_{11} & \ddots & \ddots & & \vdots \\ \vdots & \ddots & \ddots & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & \ddots & \vdots \\ \vdots & & \ddots & \ddots & \ddots & \vdots \\ \Psi_{m0} & \dots & \dots & \dots & \dots & \Psi_{mn} \end{bmatrix} \tag{34}$$

A simple way to construct this as a column vector would be to simply append each column on to the first one in order, like so

$$\Psi = \begin{bmatrix} \Psi_{00} \\ \Psi_{10} \\ \vdots \\ \Psi_{m0} \\ \Psi_{01} \\ \vdots \\ \Psi_{0n} \\ \vdots \\ \Psi_{mn} \end{bmatrix} \tag{35}$$

This resolves the problem by retaining a sensible ordering for the points from 2D in a 1D structure that can be used in the same way as the points from a 1D system.

In general, the standard matrix reshape function works in exactly this manner. A matrix of arbitrary dimension representing a function on defined over an arbitrary number of spatial dimensions can simply be reshaped into a vector with a number of elements equal to the total number of points in the system. For instance, in 3D, one could imagine another matrix like the 2D one shown for Ψ for each z coordinate, like a "stack" of 2D matrices. Reshaping this structure into a column vector would simply consist of appending the reshaped 2D matrices from each z coordinate onto each other. This procedure can then be extended to even higher dimensions, though that should not be necessary since there are only 3 spatial dimensions in real life.

However, this still leaves the problem of how to deal with operators in multiple dimensions. As it turns out, this is actually very simple given the structure chosen for the wavefunction. Kronecker tensor products can be used to construct product spaces for each operator. For instance, first derivatives for each dimension in 3D can be constructed in the following fashion

$$\mathbf{D}^1_{3x} = \mathbf{D}^1_{1x} \otimes \mathbf{I}_y \otimes \mathbf{I}_z \tag{36}$$

$$\mathbf{D}^1_{3y} = \mathbf{I}_x \otimes \mathbf{D}^1_{1y} \otimes \mathbf{I}_z \tag{37}$$

$$\mathbf{D}^1_{3z} = \mathbf{I}_x \otimes \mathbf{I}_y \otimes \mathbf{D}^1_{1z} \tag{38}$$

Where the superscript of 1 denotes a first derivative, the number in the subscript denotes the dimensions of the system with which the operator is constructed for, and the letter in the subscript denotes the direction. Due to the structure of the column vector representing the wavefunction, the ordering of the tensor products is very important.

An important multidimensional operator is the Laplacian operator. It can be calculated in the following manner in 3D dimensions.

$$\mathbf{L} = \mathbf{D}^2_{1x} \otimes \mathbf{I}_y \otimes \mathbf{I}_z + \mathbf{I}_x \otimes \mathbf{D}^2_{1y} \otimes \mathbf{I}_z + \mathbf{I}_x \otimes \mathbf{I}_y \otimes \mathbf{D}^2_{1z} \tag{39}$$

While this method of incorporating multiple dimensions is mathematically sound, it does get very complex very quickly. For instance, in an d-dimensional system where each dimension is defined with n discrete points, the linear operators for the system will be of the size $n^d \times n^d$. So, they will have, in the worst case, n^{2d} matrix elements. For three dimensions, this is n^6 matrix elements. So, for a 10 spatial points in each direction, there are already 1 million matrix elements. This is the same as a 1D system with 1 thousand spatial points. Thus, the computational complexity grows very quickly to the point where a large multidimensional system may simply be too complex to treat numerically using this scheme.

C. Eigenvalues & Eigenvectors

Now that the matrix operators for an n-dimensional system have been determined, there is the question of how to determine the eigenvalues and eigenvectors of the system. This can actually be accomplished with any number of well-established algorithms. The algorithm used by the solver used in this paper was the Implicitly Restarted Lanczos Method [3]. The method is numerically stable and guaranteed to produce a set of orthogonal eigenvectors, no spurious eigenvalues, and does not require more memory upon increased iterations [3]. However, numerical error is still present to machine precision. Consider the Hamiltonian eigenvalue problem

$$\mathbf{H}\Psi = \left(-\frac{\hbar^2}{2m}\mathbf{L} + \mathbf{V}\right)\Psi = E\Psi \tag{40}$$

Where $\mathbf{L}$ is the Laplacian operator. If we consider this Hamiltonian matrix, the approximation in the derivative operator through the use of finite differencing introduces further error. So while the eigenvalues and eigenvectors calculated may indeed be very accurate for the given matrix, they may be different than the analytical eigenvalues and eigenfunctions to a great degree simply because the Hamiltonian matrix is a poor approximation of the true Hamiltonian. Great care must be taken to ensure that the discretization is valid within the error tolerance necessary for the problem in order to ensure that the calculated eigenvalues and eigenvectors are valid for the operators in the system. Furthermore, since the matrix operators are finite, there is a finite number of eigenvalues and eigenvectors that can be calculated. This presents an issue since the continuous operators have an infinite number of eigenvalues and eigenfunctions, though the eigenvalues may be discrete. So, the choice of the discretization has a direct impact on the highest energy and energy state of the system, for instance when finding eigenvalues and eigenvectors of the Hamiltonian operator. So, if high energies need to be known, then generous numbers of points in the discretization must be provided. This proves even more inhibiting with multidimensional systems, as degeneracies in the energies that produce numerically different eigenvalues and produce different eigenvectors will provide even more obstacles to reaching high energy states. This, on top of the issue with adding more points to a multidimensional system means that studying the energy eigenvalue problem for multidimensional systems is a very difficult numerical task.

IV. TIME PROPAGATION

Analytically, solving the time dependent component of the Schrödinger equation returns the time propagation operator, defined like so

$$\hat{\mathbf{U}}(t) = e^{-\frac{i\hat{\mathbf{H}}}{\hbar}t} \tag{41}$$

This is immediately concerning, as exponentiating matrices, while certainly possible, is a very computationally expensive task. Especially for the sizes of the systems we are dealing with. However, approximations can be made to assuage this issue. Some form of this operator will be needed to take some initial state forward in time. This is necessary to study the dynamics of the problem. So far, only methods for solving the time-independent features of the system have been explored.

A. First Order Approximation

A very simply approximation to the propagation operator is to simply take the first order approximation for some small step τ, assuming the Hamiltonian is not itself dependent on time

$$\hat{\mathbf{U}}(\tau) \approx \mathbf{U} = \mathbf{I} - \frac{i\mathbf{H}}{\hbar}\tau \tag{42}$$

However, this has a considerable weakness due to the fact that the propagation operator must be unitary. If the propagation operator is not unitary, then probability will not be conserved.

$$\langle\Psi|\Psi\rangle = 1; \langle\Psi(t)|\Psi(t)\rangle = \langle\Psi|\,\hat{\mathbf{U}}^\dagger(t)\hat{\mathbf{U}}(t)\,|\Psi\rangle = 1 \tag{43}$$

This directly implies that

$$\hat{\mathbf{U}}^\dagger(t)\hat{\mathbf{U}}(t) = 1 \tag{44}$$

For the first order approximation,

$$\mathbf{U}^\dagger\mathbf{U} = \left(\mathbf{I} + \frac{i\mathbf{H}}{\hbar}\tau\right)\left(\mathbf{I} - \frac{i\mathbf{H}}{\hbar}\tau\right) = \mathbf{I} + \frac{\mathbf{H}^2}{\hbar^2}\tau^2 \tag{45}$$

This approximation will then only be unitary if $\mathbf{H}^2$ terminates, which cannot be taken to be true in general. Going to higher order terms will require more and more matrix multiplications of large matrices, but there will always be an error term proportional to the Hamiltonian matrix to some power. Even if an extremely small time step is used, this lack of unitarity simply introduces far too much error into the time evolution, even if higher order terms are included which also become more and more computationally demanding to calculate. This may seem hopeless, but there is a way to enforce unitarity detailed in the next subsection.

B. Cayley Form

Taking a Crank-Nicolson approach to the problem, the time evolution can be written, using the first order approximation, as a step forward in time and a step backward in time like so with respect to the wavefunction at times indexed by i

$$\left(\mathbf{I} + \frac{i\mathbf{H}}{\hbar}\tau\right)\Psi_{i+1} = \left(\mathbf{I} - \frac{i\mathbf{H}}{\hbar}\tau\right)\Psi_{i-1} \tag{46}$$

The step backwards on the left hand side takes the state Ψ_{i+1} to Ψ_i. Similarly, the step forward on the right hand side takes the state Ψ_{i-1} to Ψ_i. This establishes the equivalence. Using the Cayley form, this equation can be rewritten as

$$\Psi_{i+1} = \left(\mathbf{I} + \frac{i\mathbf{H}}{\hbar}\tau\right)^{-1}\left(\mathbf{I} - \frac{i\mathbf{H}}{\hbar}\tau\right)\Psi_{i-1} \tag{47}$$

Reducing the time-step by a factor of 2 and assuming the time intervals are equally spaces, this can be written as

$$\Psi_{i} = \left(\mathbf{I} + \frac{i\mathbf{H}}{2\hbar}\tau\right)^{-1}\left(\mathbf{I} - \frac{i\mathbf{H}}{2\hbar}\tau\right)\Psi_{i-1} \tag{48}$$

This would then suggest that

$$e^{-\frac{i\mathbf{H}}{\hbar}\tau} \approx \frac{\mathbf{I} - \frac{i\mathbf{H}}{2\hbar}\tau}{\mathbf{I} + \frac{i\mathbf{H}}{2\hbar}\tau} \tag{49}$$

Since this is composed of first-order approximations, the method still has error that is second order in time [4]. However, it is unitary, so probability will be conserved by this method

$$\left(\frac{\mathbf{I} - \frac{i\mathbf{H}}{2\hbar}\tau}{\mathbf{I} + \frac{i\mathbf{H}}{2\hbar}\tau}\right)^\dagger\left(\frac{\mathbf{I} - \frac{i\mathbf{H}}{2\hbar}\tau}{\mathbf{I} + \frac{i\mathbf{H}}{2\hbar}\tau}\right) = \left(\frac{\mathbf{I} + \frac{i\mathbf{H}}{2\hbar}\tau}{\mathbf{I} - \frac{i\mathbf{H}}{2\hbar}\tau}\right)\left(\frac{\mathbf{I} - \frac{i\mathbf{H}}{2\hbar}\tau}{\mathbf{I} + \frac{i\mathbf{H}}{2\hbar}\tau}\right) = \mathbf{I} \tag{50}$$

Care must still be taken to choose sufficiently small timesteps since the error is only second order in the timestep. But, any problems with the unitarity will be caused only by round-off error assuming the Hamiltonian is accurate. Since the accuracy of the Hamiltonian depends on the spatial discretization, the interdependence of the accuracy in time and space becomes immediately clear. For better results in time integration, a reduction in the time step may not be sufficient by itself, a reduction in the spatial step may be required as well.

V. OBSERVABLES

The final question that remains is how to extract the observables of a system from the wavefunction. This is

done in essentially the same manner as it is in the analytical sense, but numerically. Analytically, the mean value of the observable associated with the operator $\hat{\mathbf{A}}$ is expressed as

$$\left\langle \hat{\mathbf{A}} \right\rangle = \langle \Psi | \, \hat{\mathbf{A}} \, | \Psi \rangle = \int \mathrm{d}^n x \, \Psi^* \hat{\mathbf{A}} \Psi \tag{51}$$

Numerically, the integrand is simply the element-wise product of the matrix operator form of the continuous operator acting on the wavefunction vector and the conjugate of the wavefunction vector. The operation is shown below

$$\Psi^* \cdot (\mathbf{A}\Psi) = \begin{bmatrix} \Psi_0^* \\ \Psi_1^* \\ \vdots \\ \Psi_n^* \end{bmatrix} \cdot \left(\begin{bmatrix} A_{00} & A_{01} & \dots & \dots & \dots & A_{0n} \\ A_{10} & A_{11} & \ddots & \ddots & & \vdots \\ \vdots & \ddots & \ddots & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & \ddots & \vdots \\ \vdots & & \ddots & \ddots & \ddots & \vdots \\ A_{n0} & \dots & \dots & \dots & \dots & A_{nn} \end{bmatrix} \begin{bmatrix} \Psi_0 \\ \Psi_1 \\ \vdots \\ \Psi_n \end{bmatrix} \right) \tag{52}$$

Where the dot denotes an element-wise multiplication. A standard integration method such a the trapezoidal method or Simpson's method can then be applied to extract the mean value of the observable. With that also comes the typical errors associated with numerical integration. Note that the initial wavefunction should always be normalized using a numerical integration technique as well.

VI. SELECTED PROBLEMS

In this section, the numerical schemes presented thus far will be applied to a selection of common problems in quantum mechanics and compared with analytical results and expectations.

A. Infinite Square Well

In this simple problem, the potential is defined as

$$V(x) = \begin{cases} \infty & x < 0 \\ 0 & 0 < x < L \\ \infty & x > L \end{cases} \tag{53}$$

The solutions are well-known to be sinusoidal in nature.

$$\Psi_n(x) = \sqrt{\frac{2}{L}} \sin\left(\frac{n\pi}{L} x\right); \quad E_n = \frac{n^2 \pi^2 \hbar^2}{2mL^2} \tag{54}$$

The numerical treatment provides the following results for the first few energy eigenstates, shown in Figure 1. The results are very clearly sinusoidal in nature and obey the boundary conditions.

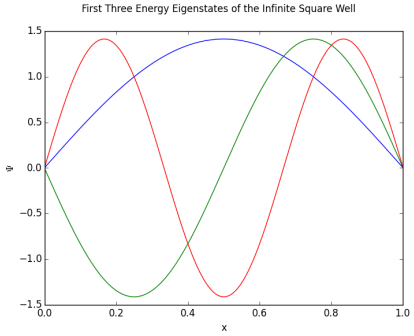

FIG. 1. Energy eigenstates for $n = 1, 2, 3$. The results are sinusoidal and obey Dirichlet boundary conditions.

The largest error was in the largest calculated eigenvector, $n = 32$, shown with the true eigenfunction in Figure 2. The mean relative error of the data was $\approx 7.52 \cdot 10^{-12}\%$

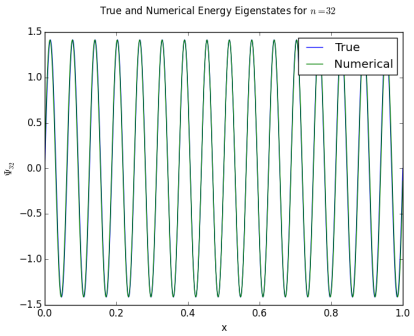

FIG. 2. Numerical and true energy eigenstates for $n = 32$.

The energy eigenvalues, scaled by the ground state energy, are shown in Figure 3. The greatest error was in the largest calculated energy eigenvalues and was $\approx 0.0801\%$

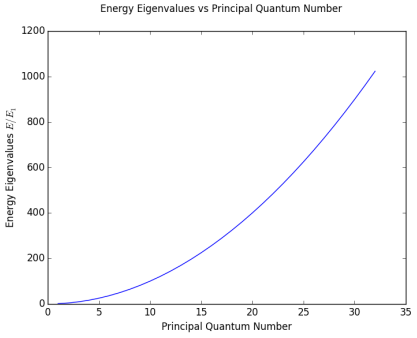

FIG. 3. Energy eigenvalues for $n = 1, 2, .., 32$, scaled by the ground state energy. The results are quadratic, as expected.

In a time evolution of the even superposition of the ground state and the first excited state, the mean energy over time is shown in Figure 4. The drift in the

energy tends towards energy loss, but on a scale of 11 orders of magnitude lower than the energy of the system. This shows energy conservation within a very acceptable tolerance.

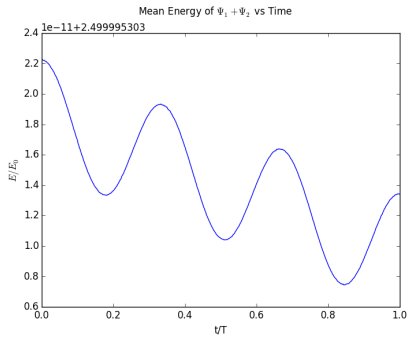

FIG. 4. Mean energy as a function of time.

Overall, the numerical scheme did a very well in finding solutions to the infinite square well problem. The energy eigenvalues and eigenfunctions were calculated to a high degree of accuracy and the time evolution of a superposition experienced very small energy drift well within the tolerance needed.

B. Linear Harmonic Oscillator

The linear harmonic oscillator problem carries a potential of the following form

$$V(x) = \frac{1}{2}m\omega^2 x^2 \tag{55}$$

Once again, the analytical results are well-known.

$$\Psi_n(x) = \frac{1}{\sqrt{2^n n!}}\left(\frac{m\omega}{\pi\hbar}\right)^{\frac{1}{4}} e^{-\frac{m\omega x^2}{2\hbar}} H_n\left(\sqrt{\frac{m\omega}{\hbar}}x\right) \tag{56}$$

$$E_n = \hbar\omega\left(n+\frac{1}{2}\right) \tag{57}$$

The first few eigenstates of the linear harmonic oscillator, calculated numerically, are shown in Figure 5. The results are as expected, with the ground state being Gaussian in shape and the excited states being products of Hermite polynomials with a Gaussian wave.

The energy eigenvalues are shown in Figure 6. As expected, the energy levels are linear with respect to the principal quantum number and there is a small offset for the ground state energy. The greatest error is in the largest energy eigenvalue with a relative error of $\approx .193\%$.

The phase diagram of the time evolution of the even superposition of the ground state and the first excited state is shown in Figure 7. The shape of the plot is ellipsoidal, just as expected for a harmonic oscillator.

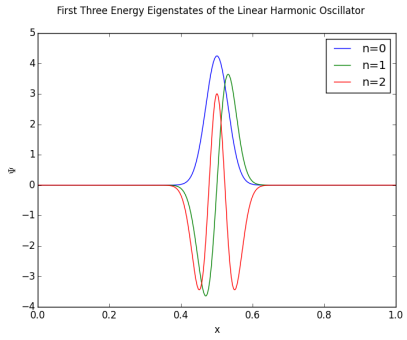

FIG. 5. Energy eigenvvectors for $n = 0, 1, 2$.

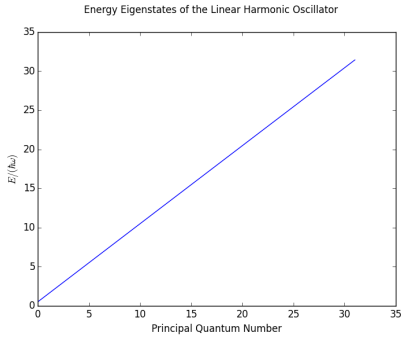

FIG. 6. Energy eigenvalues for $n = 0, 1, .., 31$, scaled by $\hbar\omega$. The results are linear, as expected.

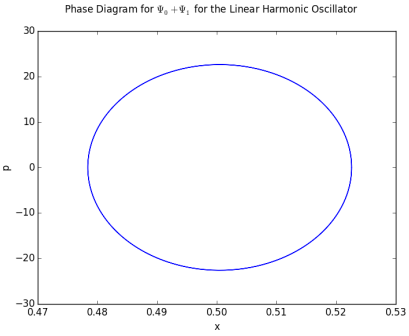

FIG. 7. Phase diagram for the superposition $|0\rangle + |1\rangle$.

For the same time evolution, the mean energy is shown as a function of time in Figure 8. Once again, the scale on the plot itself is 11 orders of magnitude smaller than the energy. Even then, there appears to be very little variation in the energy over time. This shows astounding energy conservation. In fact, the standard deviation of the scaled energy is $\approx 2.14 \cdot 10^{-15}$.

The numerical solution to the quantum harmonic oscillator shows incredible qualitative and quantitative agreement with the theory.

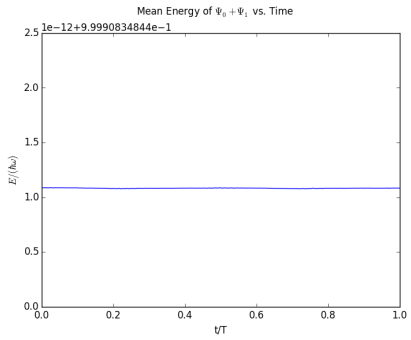

FIG. 8. Mean energy as a function of time.

C. Harmonic Oscillator in Two Dimensions

For a harmonic oscillator in two dimensions, the potential can be written in the form

$$V(x,y) = \frac{1}{2}m\omega^2(x^2+y^2) \tag{58}$$

Numerically, calculations in two dimensions take a very long time. Modest domain sizes often do not produce workable results. For instance, in a system with 51 sample points in each direction, the energy eigenvalues ae shown in Figure 9. For small energies, the results are accurate and even show the degeneracy that is introduced in the multi-dimensional system. However, this quickly breaks down as the discretization puts harsh limits on the accuracy of the higher energy states.

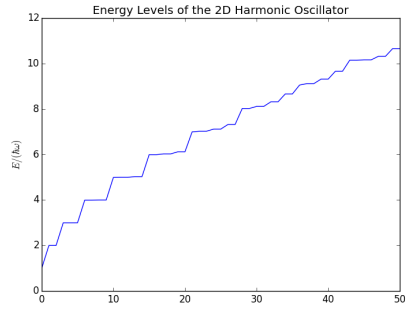

FIG. 9. Energy eigenvalues for the 2D harmonic oscillator.

The smaller eigenvalues are still rather workable and so are the corresponding eigenvectors. A superposition of the first two states is shown in Figure 10.

The phase plots for the time evolution of this system also produce the desired elliptical shapes, as shown in Figure 11.

The mean energy of the time evolution is also shown in Figure 12. The variation in the energy is once again 12 orders of magnitude lower than the energy, showing excellent conservation of energy during the time evolution.

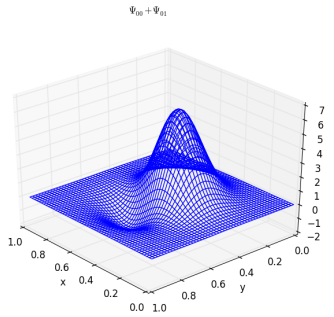

FIG. 10. Wireframe plot of $|00\rangle + |01\rangle$.

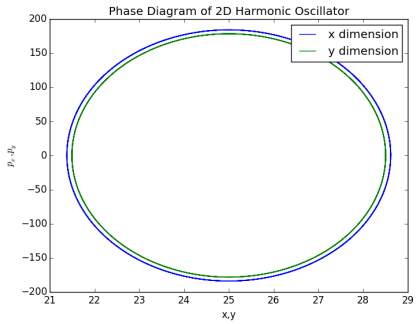

FIG. 11. Phase diagram for time evolution of $|00\rangle + |01\rangle$.

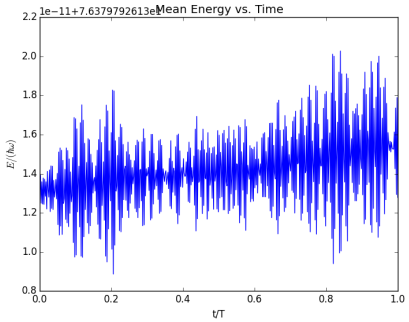

FIG. 12. Mean energy for time evolution of $|00\rangle + |01\rangle$.

D. Dirac Comb

As a final example, consider a potential that consists of repeated Dirac delta potentials in a periodic 1D spatial domain. Such a system is called a Dirac comb. The Dirac deltas may seem questionable at first, as they are only defined on a continuous domain. However, they can be represented nicely simply as functions that exist only at on spatial point that are normalized to 1. For this particular system, ten Dirac delta potentials were prepared inside a period domain. It is expected that in such a periodic system that there will be band gaps, which are of great interest in condensed matter physics. A plot of the energy spectrum is shown in Figure 13. The desired

behavior is clearly seen as the energy spectrum appears continuous in between large jumps in the energy.

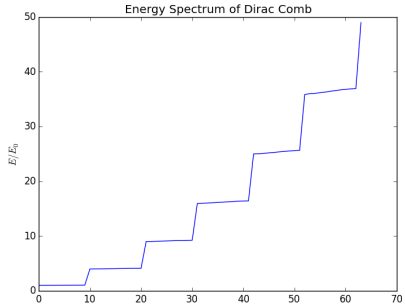

FIG. 13. Energy eigenvalues of Dirac comb.

VII. CONCLUSIONS

The numerical methods presented in this paper are powerful tools for solving many problems in quantum mechanics expressible with the Schrödinger equation. There are many advantages and disadvantages to its use.

The main advantage is that the numerical scheme can be used to quickly and effectively extract eigenvalues, eigenvectors, and time evolutions for many different types of systems, many of which would be prohibitively difficult to solve analytically. The user essentially has the freedom to set the value of the potential at every point on the domain to whatever real value they desire. The potential need not even be expressible in terms of elementary functions.

However, the discrete nature of the numerical strategies presented come with many drawbacks. The stability of the solutions are heavily impacted by the choice of discretization. If steps in the spacial dimension are too large or there are too few points in the spatial domains, solutions become innacurate very quickly. Specifically, the eigenvalues and eigenfunctions of operators defined for the system become very innacurate for all but the smallest eigenvalues if the discretization is too small. This can render many solutions completely useless as the higher energy features become completely unreliable. Furthermore, the time evolution of the system is highly sensitive to the spatial discretization even if a sufficiently small time step is chosen.

Larger numbers of sample points allow for the calculation of larger eigenvalues and the corresponding eigenvalues in addition to calculating them to a higher degree of accuracy. Smaller spatial discretizations also allow for more accurate time evolutions. However, this results in an increase in the complexity of the calculations at a worrying rate, especially for multidimensional systems. For a system with n points in d dimensions, doubling the number of points to $2n$ in each dimension increases the worst-case number of entries in operators from n^{2d} to $4^d n^{2d}$.

Higher order difference methods for spatial derivatives can be employed to allow for more accurate calculations of eigenvalues, eigenvectors, and time evolutions. However, this still does not allow for the calculation of more eigenvalues and eigenvectors.

The types of problems that can be sufficiently treated by these numerical methods are also limited. Since $\mathbb{R}^n$ cannot be represented on a computer, the system has to be limited in some way such that all of the features are contained in a finite space that can be accurately represented as a discrete domain. This includes systems that are contained in an infinite square well, systems with periodic boundary conditions, and systems that are bound by a potential.

Despite these limitations, the numerical solutions to select problems can be shown to exhibit astounding accuracy. Results for 1D systems that fulfill the criteria of being in a box, periodic, or bound are very satisfying, for example. This feature secures the finite difference approach's place amongst the tools that can be used to find solutions to problems in quantum mechanics.

VIII. ACKNOWLEDGEMENTS

The author acknowledges Parampreet Singh, Ph.D. for personal correspondence and classroom instruction relevant to the material presented in this paper.

IX. REFERENCES

[1] Mark Newman. "Computational Physics." 2nd ed., 2013. Print.

[2] Bengt Fornberg. "Generation of Finite Difference Formulas on Arbitrary Spaced Grids." Mathematics of Computation Volume 51, Number 184: 699-706, October 1988.

[3] D. Calvetti, L. Reichel, and D.C. Sorenson. "An Implicitly Restarted Lanczos Method for Large Symmetric Eigenvalue Problems." Electronic Transactions on Numerical Analysis. Volume2, 1-21, March 1994.

[4] F.L. Dubeibe. "Solving the Time-Dependent Schrödinger Equation with Absorbing Boundary Conditions and Source Terms in Mathematica 6.0." International Journal of Modern Physics C, 2 December 2010.

Appendix A: Integrate

```
from __future__ import division, print_function
import numpy as np
import scipy.integrate as spint


def integrate(dx, f, k):
    """ given sample separation dx (array-like), function f (array-like), and
        dimensions to be integrated over k (array-like), integrates function f
        over the dimensions in k """
    # array of dimensions to integrate over
    k = np.array(k)
    # alias of integrand function
    g = f
    # for loop to step through each integration axis
    for i in xrange(0, len(k)):
        #integrate over dimension
        g = spint.simps(g, None, dx[i], k[i])
        # reduce the dimensions by 1
        k = k - 1
    # return the integral
    return g
```

Appendix B: Normalize

```
from __future__ import division, print_function
import numpy as np
import copy
from integrate import integrate


def normalize(dx, n, f):
    """ given sample separation dx (array-like), sample number (array-like),
        and function f (array-like), normalizes the function f """
    # make copy of the dimension of the system
    m = copy.copy(n)
    # append -1 so that the dimension of g is inferred in the last dimension
    np.append(m, -1)
    # calculate PDF and reshape to integrate along distinct dimensions
    g = np.reshape(np.multiply(np.conj(f), f), m)
    # integrate over PDF
    g = integrate(dx, g, np.arange(0, len(n), 1))
    # cases to habdle the possible types of the integration constant
    if type(g) == np.ndarray:
        f = f / np.sqrt(g[0])
    else:
        f = f / np.sqrt(g)
    # return normalized function
    return f
```

Appendix C: Fornberg

```
from __future__ import division, print_function
from numpy import zeros, arange
```

```
def fornberg(u, v, k):
    """ given the index u (int), the relative indices v (array-like), and the
        order k (int), calculates an array of coefficients for derivatives
        through order k

        Generation of Finite Difference Formulas on Arbitrarily Spaced Grids
        Bengt Fornberg

        http://amath.colorado.edu/faculty/fornberg/
        Docs/MathComp_88_FD_formulas.pdf """

    n = len(v)
    C = zeros([k+1, n])
    c1 = 1
    c4 = v[0]-u
    C[0,0] = 1
    for i in xrange(2, n+1):
        mn = min(i, k+1)
        c2 = 1
        c5 = c4
        c4 = v[i-1]-u
        for j in xrange(1, i):
            c3 = v[i-1]-v[j-1]
            c2 = c2*c3
            if j == (i-1):
                C[1:mn, i-1] = c1*(arange(1, mn)*C[0:(mn-1), i-2]-
                               c5*C[1:mn, i-2])/c2
                C[0, i-1] = -c1*c5*C[0, i-2]/c2
            C[1:mn, j-1] = (c4*C[1:mn, j-1]-arange(1, mn)*C[0:(mn-1), j-1])/c3
            C[0, j-1] = c4*C[0, j-1]/c3
        c1 = c2
    return C
```

Appendix D: EigenVectors

```
from __future__ import division, print_function
import scipy.sparse.linalg as spsplin
import numpy as np
from normalize import normalize


def eigenVectors(n, dx, A):
    """ given sample numbering n (array-like), sample separation dx
        (array-like), and an operator A (array-like), calculates the
        eigenvalues (array-like) and the eigenvectors (array-like) and returns
        them in a list """
    # calculate total number of samples
    N = np.prod(n)
    # calculate the first sqrt(N) eigenvalues/vectors
    (u, v) = spsplin.eigsh(A, int(np.sqrt(N)), None, None, 'SM')
    # calculate shape of v
    m = np.shape(v)
    # for each eigenvector
    for i in xrange(0, m[1]):
        # normalize each eigenvector
        v[:, i] = normalize(dx, n, v[:, i])
```

```
    # return eigenvalues and eigenvectors in a list
    ev = [u, v]
    return ev
```

Appendix E: DiscreteCartesianDomain

```
from __future__ import division, print_function
import numpy as np


class DiscreteCartesianDomain:
    """ a class used to establish a basic discrete Cartesian coordinate
        system. three paramters are required for initialization. the
        sampleNumber is a list or numpy array that established the number of
        equally spaced discrete samples in each direction. the
        sampleSeparation is a list or array of the same length as the length of
        sampleNumber that establishes exactly what the equal spacing is between
        sample points in each dimension. the boundaryCondition is a string of
        the same length as the length of the sampleNumber and establishes
        the edge behavior of the coordinates. the two conditions supported
        are 'd' for dirichlet boundary conditions and 'p' for periodic boundary
        conditions. mixed conditions in each dimension is supported.

        initialization of a 1D system with 1000 sample points, a spacing of 1,
        and periodic boundary conditions:

        DiscreteCartesianDomain([1000], [1], 'p')

        initialization of a 2D system with 50 sample points in the x direction,
        100 sample points in the y direction, a spacing of 1 in the x
        direction, a spacing of .5 in the y direction, dirichlet conditions in
        the x direction, and periodic conditions in the y direction:

        DiscreteCartesianDomain([50, 100], [1, .5], 'dp') """

    def __init__(self, sampleNumber, sampleSeparation, boundaryCondition):
        """ takes three parameters. sampleNumber (array-like), sampleSeparation
            (array-like), and boundaryCondition (string) """
        # number of sample points
        self.n = np.array(sampleNumber)
        # spacing between sample points
        self.dx = np.array(sampleSeparation)
        # boundary conditions
        self.bc = boundaryCondition
        # construct coordinate system
        self.constructDomain()

    def returnSampleNumber(self):
        """ when called, returns the sample number of the coordinate system
            (array) """
        # return sample number
        return self.n

    def returnSampleSeparation(self):
        """ when called, returns the sample separation of the coordinate
            system (array) """
        # return sample separation
```

```
    return self.dx

def returnBoundaryCondition(self):
    """ when called, returns the boundary conditions of the coordinate
        coordinate system (string) """
    # return boundary condition
    return self.bc

def returnDimensionLength(self):
    """ when called, returns the length of each direction in the coordinate
        system (array) """
    # return length
    return self.l

def returnDomain(self):
    """ when called, returns the domain of the coordinate system (list of
        arrays) """
    # return domain
    return self.x

def constructDomain(self):
    """ when called, constructs the domain of the system using the
        information from the number of sample points, the sample point
        separation, and the boundary conditions """
    # each dimension has n-1 links of length dx, l = dx*(n-1)
    self.l = self.dx*(self.n-1)
    # empty list to store arrays of allowed discrete positions
    self.x = []
    # for each dimension in n
    for i in xrange(0, len(self.n)):
        # periodic boundary conditions will add an extra link
        if self.bc[i] == 'p':
            self.l[i] += self.dx[i]
        # construct dimension-specific array of allowed discrete positions
        d = np.array(np.linspace(0, (self.n[i]-1)*self.dx[i], self.n[i]))
        # append array of allowed positions
        self.x.append(d)

def setSampleNumber(self, sampleNumber):
    """ when called, takes in the parameter sampleNumber (array-like) and
        uses it to construct a new domain accordingly """
    # reassign sample number
    self.n = np.array(sampleNumber)
    # construct new domain
    self.constructDomain()

def setSampleSeparation(self, sampleSeparation):
    """ when called, takes in the parameter sampleSeparation (array-like)
        and uses it to construct a new domain accordingly """
    # reassign sample separation
    self.dx = np.array(sampleSeparation)
    # construct new domain
    self.constructDomain()

def setBoundaryCondition(self, boundaryCondition):
    """ when called, takes in the parameter boundaryCondition (string) and
        uses it to construct a new domain accordingly """
    # reassign boundary condition
```

```
        self.bc = boundaryCondition
        # construct new domain
        self.constructDomain()

    def expand(self, sampleNumber, sampleSeparation, boundaryCondition):
        """ when called, takes in the parameters sampleNumber (array-like),
            sampleSepration (array-like), and boundarycondition (string) in
            order to extend the dimension of the coordinate system and
            construct a new domain accordingly """
        # extend the sample number to include the new desired dimension
        self.n = np.concatenate((self.n, np.array(sampleNumber)), 1)
        # extend the sample separation to include the new desired dimension
        self.dx = np.concatenate((self.dx, np.array(sampleSeparation)), 1)
        # extend the boundary conditions to include the new desired dimension
        self.bc = ''.join([self.bc, boundaryCondition])
        # construct the new domain
        self.constructDomain()
```

Appendix F: DifferentiableDomain

```
from __future__ import division, print_function
import numpy as np
import scipy.sparse as spsp
from DiscreteCartesianDomain import DiscreteCartesianDomain
from fornberg import fornberg


class DifferentiableDomain(DiscreteCartesianDomain):
    """ a class that inherits from the DiscreteCartesianDomain and carries with
        it the exact same data members. two methods are added to encapsulate
        the desired differentiable nature of the coordinate system: the
        derivativeOperator and the laplacianOperator.

        initialization of a 1D system with 1000 sample points, a spacing of 1,
        and periodic boundary conditions:

        DifferentiableDomain([1000], [1], 'p')

        initialization of a 2D system with 50 sample points in the x direction,
        100 sample points in the y direction, a spacing of 1 in the x
        direction, a spacing of .5 in the y direction, dirichlet conditions in
        the x direction, and periodic conditions in the y direction:

        DifferentiableDomain([50, 100], [1, .5], 'dp') """

    def __init__(self, sampleNumber, sampleSeparation, boundaryCondition):
        DiscreteCartesianDomain.__init__(self,
                                         sampleNumber,
                                         sampleSeparation,
                                         boundaryCondition)

    def derivativeOperator(self, v = [-1, 0, 1], o = 1, d = 0):
        """ when called, returns a derivative matrix operator using the
            relative sample points in v, the order o, and the direction d.

            a first-order derivative in the x-direction using only
            nearest-neighbors would be expressed as:
```

```
        derivativeOperator([-1, 0, 1], 1, 0)

        a second-order derivative in the y direction using the second-
        nearest neighbors would be expressed as:

        derivativeOperator([-2, -1, 0, 1, 2], 2, 1) """
    # number of samples in direction d
    n = self.n[d]
    # find difference weights (w) on indices (v) centered on 0 of order o
    # using fornberg algorithm
    w = fornberg(0, v, o)
    # prepare on-filled array with same cardinality as dimension
    a = np.ones(n)
    # prepare a list to be filled with arrays of the weights
    u = []
    # loop through weights
    for i in xrange(0, len(v)):
        u.append(a*w[o, i])
    # prepare derivative operator as sparse 1D derivative operator
    D = spsp.spdiags(u, v, n, n, format = 'csc')
    # if periodic boundary conditions
    if self.bc[d] == 'p':
        # find node index
        i = v.index(0)
        # for each row, fill in periodic terms
        for j in xrange(0,i):
            D[j, (n-i+j):n] = w[o, 0:(i-j)]
            D[n-j-1, 0:(i-j)] = w[o, (len(v)-i+j):len(v)]
    # if the domain is of only one dimension
    if len(self.n) != 1:
        # prepare empty list to store each dimension matrix
        A = []
        # loop through dimensions
        for i in xrange(0, len(self.n)):
            # if dimension is being differentiated, use derivative operator
            if i == d:
                A.append(D)
            # else, simply use identity matrix to preserve the dimension
            # structure
            else:
                A.append(spsp.eye(self.n[i], self.n[i]))
        # take first kronecker tensor product to compose spaces
        D = spsp.kron(A[1], A[0], format = 'csc')
        # iterate through remaining tensor products to construct full
        # operator
        for i in xrange(2, len(self.n)):
            D = spsp.kron(A[i], D, format = 'csc')
    # return derivative operator, dividing by sample separation to the
    # of the order of the derivative
    return D/self.dx[d-1]**o


def laplacianOperator(self, v = [-1, 0, 1]):
    """ when called, returns a laplacian matrix operator calculated by
        taking the sum of the second order derivative operators in each
        direction using the relative sample points in v
```

```
            a laplacian operator using nearest-neighbors would be expressed as:

            laplacianOperator([-1, 0, 1])

            a laplacian operator using second-nearest-neighbors would be
            expressed as:

            laplacianOperator([-2, -1, 0, 1, 2]) """
        # total number of points
        N = np.prod(self.n)
        # empty sparse matrix to intialize operator
        L = spsp.csc_matrix((N,N))
        # loop through each spatial dimension
        for i in xrange(0, len(self.n)):
            L = L+self.derivativeOperator(v, 2, i)
        return L
```

Appendix G: QuantumSystem

```
from __future__ import division, print_function
from DifferentiableDomain import DifferentiableDomain
from eigenVectors import eigenVectors
from normalize import normalize
from integrate import integrate
import numpy as np
import scipy.sparse.linalg as spsplin
import scipy.sparse as spsp


class QuantumSystem(DifferentiableDomain):
    """ a class that inherits form the DifferentiableDomain and shares its data
        members. additional data members are the potential (array-like), the
        value of the mass of the particle (float), and the value of hbar
        (float)

        initialization of a 1D system with 1000 sample points, a spacing of 1,
        periodic boundary conditions, a null potential, a mass of 1, and hbar
        set to 1:

        QuantumSystem([1000], [1], 'p', np.zeros(1000), 1, 1)

        initialization of a 2D system with 50 sample points in the x direction,
        100 sample points in the y direction, a spacing of 1 in the x
        direction, a spacing of .5 in the y direction, dirichlet conditions in
        the x direction, periodic conditions in the y direction, a null
        potential, an electron mass, and hbar set to its SI value:

        QuantumSystem([50, 100], [1, .5], 'dp',
                      np.zeros([50,100]), 9.10938356e-31, 1.0545718e-34) """

    def __init__(self, sampleNumber, sampleSeparation,
                 boundaryCondition, potential, mass, hbar):
        """ takes six parameters. sampleNumber (array-like), sampleSeparation
            (array-like), boundaryCondition (string), potential (array-like),
            mass (float), and hbar (float) """
        DifferentiableDomain.__init__(self,
                                      sampleNumber,
```

```
                                    sampleSeparation,
                                    boundaryCondition)
        self.V = potential
        self.m = mass
        self.h = hbar

    def returnPotential(self):
        """ when called, returns the system potential (array-like) """
        return self.V

    def returnMass(self):
        """ when called, returns the particle mass (float) """
        return self.m

    def returnHbar(self):
        """ when called, returns the value of hbar (float) """
        return self.hbar

    def setPotential(self, V):
        """ when called, takes in the potential parameter (array-like) and
            updates the system potential accordingly """
        self.V = V

    def setMass(self, m):
        """ when called, takes in the parameter mass (float) and updates the
            particle mass accordingly """
        self.m = m

    def setHbar(self, hbar):
        """ when called, takes in the parameter hbar (float) and updates the
            system value accordingly """
        self.hbar = hbar

    def positionOperator(self, o = 1, d = 0):
        """ when called, returns the position operator of order (power) o (int)
            and a direction d (int)

            the position operator in the x direction would be expressed as:

            positionOperator(1, 0)

            the square position operator in the y direction would be expressed
            as:

            positionOperator(2, 1) """
        X = spsp.spdiags(self.x[d]**o, 0, self.n[d], self.n[d], format = 'csc')
        if len(self.n) != 1:
            # prepare empty list to store each dimension matrix
            A = []
            # loop through dimensions
            for i in xrange(0, len(self.n)):
                # if dimension is being differentiated, use derivative operator
                if i == d:
                    A.append(X)
                # else, simply use identity matrix to preserve the dimension
                # structure
                else:
                    A.append(spsp.eye(self.n[i], self.n[i], format = 'csc'))
```

```
        # take first kronecker tensor product to compose spaces
        X = spsp.kron(A[1], A[0])
        # iterate through remaining tensor products to construct full
        # operator
        for i in xrange(2, len(self.n)):
            X = spsp.kron(A[i], X)
    return X

def momentumOperator(self, v = [-1, 0, 1], o = 1, d = 0):
    """ when called, returns the momentum operator using the relative
        vertices in v (array-like), an order o (int), and a direction d
        (int)

        the momentum operator in the x direction using nearest neighbors
        would be expressed as:

        momentumOperator([-1, 0, 1], 1, 0)

        the square momentum operator in the y direction using
        second-nearest neighbors would be expressed as:

        momentumOperator([-2, -1, 0, 1, 2], 2, 1) """
    return (-1j*self.h)**o*self.derivativeOperator(v, o, d)

def kineticOperator(self, v = [-1, 0, 1]):
    """ when called, returns the kinetic energy operator using the relative
        vertices in v (array-like)

        the kinetic energy operator using nearest neighbors would be
        expressed as:

        kineticOperator([-1, 0, 1])

        the kinetic energy operator using second-nearest neighbors would
        be expressed as:

        kineticOperator([-2, -1, 0, 1, 2]) """
    return -1*self.h**2/(2*self.m)*self.laplacianOperator(v)

def potentialOperator(self):
    """ when called, returns the potential energy operator """
    # number of sample points in system
    N = np.prod(self.n)
    # potential values are placed along the diagonal
    return spsp.spdiags(np.reshape(self.V, N), 0, N, N, format = 'csc')

def hamiltonOperator(self, v = [-1, 0, 1]):
    """ when called, returns the Hamiltonian using the relative vertices in
        v (array-like)

        the Hamiltonian using nearest neighbors would be expressed as:

        hamiltonOperator([-1, 0, 1])

        the Hamiltonian using second-nearest neighbors would be expressed
        as:

        hamiltonOperator([-2, -1, 0, 1, 2]) """
```

```
    return self.kineticOperator(v)+self.potentialOperator()

def propagationOperator(self, dt, v = [-1, 0, 1]):
    """ when called, returns the time porpagation operator using the
        timestep dt (float) and the relative vertices v (array-like)

        the propagation operator using a timestep of 1 and nearest
        neighbors would be expressed as:

        propagationOperator(1, [-1, 0, 1])

        the propagation operator using a timestep of .5 and second-nearest
        neighbors would be expressed as:

        propagationOperator(.5, [-2, -1, 0, 1, 2]) """
    # total number of samples in the system
    N = np.prod(self.n)
    # hamiltonian operator
    H = self.hamiltonOperator(v)
    # hamiltonian operator made anti-hermitian and divided by hbar
    AH = H / (1j * self.h)
    # first order porpagation operator
    UF = spsp.eye(N, N, format = 'csc')+(dt/2)*AH
    # Cayley's form to enforce unitarity
    # http://arxiv.org/pdf/physics/0011068.pdf
    U = spsplin.spsolve(UF, UF.conj())
    return U

def eigenStates(self, A):
    """ when called, takes in the operator A (array-like) and returns its
        eigenvalues (array-like) and corresponding eigenvectors
        (array-like) in a list """
    return eigenVectors(self.n, self.dx, A)

def energyEigenStates(self, v = [-1, 0, 1]):
    """ when called, uses the relative vertices in v (array-like) to
        calculate the energy eigenvalues (array-like) and eigenvectors
        (array-like) returned in a list """
    return eigenVectors(self.n, self.dx, self.hamiltonOperator(v))

def timeEvolution(self, f, dt, k, v = [-1, 0, 1]):
    """ when called, uses an initial wavefunction f (array-like), the
        timestep dt (float), the time sampling k (int), and the relative
        vertices in v (array-like) to calculate the full
        time-evolution of the wavefunction (array-like) """
    # calculate the time propagation operator if one is not provided
    U = self.propagationOperator(dt, v)
    # normalize the wavefunction
    f = normalize(self.dx, self.n, f)
    # prepare the full time-evolution array
    F = np.zeros([len(f), k], dtype = 'complex128')
    # initialize time-evolution array with initial wavefunction
    F[:, 0] = f
    # loop through time samples to generate time-evolution
    for i in xrange(1, k):
        F[:, i] = U*F[:, i-1]
    return F
```

```
def probDens(self, F):
    """ when called, uses a wavefunction F (array-like) to calculate the
        probability density (array-like) """
    return np.multiply(np.conj(F), F)

def meanValue(self, A, F):
    """ when called, uses an operator A (array-like) and a wavefunction F
        (array-like) to calculate the mean value of the observable
        associated with the operator A with respect to the wavefunction
        F """
    return integrate(self.dx, np.multiply(np.conj(F), A*F), [0])

def meanPosition(self, F, o = 1, d = 0):
    """ when called, uses a wavefunction F (array-like), an order o (int),
        and a direction d(int) to calculate the mean value of the position
        of order o with respect to the wavefunction F """
    X = self.positionOperator(o, d)
    return self.meanValue(X, F)

def meanMomentum(self, F, v = [-1, 0, 1], o = 1, d = 0):
    """ when called, uses a wavefunction F (array-like), relative vertices
        v (array-like), an order o (int), and a direction d (int) to
        calculate the mean momentum of order o with respect to the
        wavefunction F """
    P = self.momentumOperator(v, o, d)
    return self.meanValue(P, F)

def meanKinetic(self, F, v = [-1, 0, 1]):
    """ when called, uses a wavefunction F (array-like) and relative
        vertices v (array-like) to calculate the mean kinetic energy with
        respect to the wavefunction F """
    T = self.kineticOperator(v)
    return self.meanValue(T, F)

def meanPotential(self, F):
    """ when called, uses a wavefunction F (array-like) to calculate the
        mean potential with repsect to the wavefunction F """
    V = self.potentialOperator()
    return self.meanValue(V, F)

def meanEnergy(self, F, v = [-1, 0, 1]):
    """ when called, uses a wavefunction F (array-like) and relative
        vertices v (array-like) to calculate the mean energy with respect
        to the wavefunction F """
    H = self.hamiltonOperator(v)
    return self.meanValue(H, F)

def virial(self, F, v = [-1, 0, 1]):
    """ when called, uses a wavefunction F (array-like) and relative
        vertices v (array-like) to calculate the mean kinetic energy and
        the mean potential energy with respect to the wavefunction F """
    T = self.meanKinetic(F, v)
    V = self.meanPotential(F)
    return T, V

def phaseSpace(self, F, v = [-1, 0, 1], d = 0):
    """ when called, uses a wavefunction F (array-like), relative vertices
        v (array-like), and direction d (int) to calculate the mean
```

```
            position and the mean momentum with repsect to the wavefunction F
            along the direction d """
        X = self.meanPosition(F, 1, d)
        P = self.meanMomentum(F, v, 1, d)
        return X, P
```

Appendix H: HarmonicOscillator

```
from __future__ import division, print_function
import numpy as np
from QuantumSystem import QuantumSystem

print('Starting..')
n = [1024]
dx = [1/(ni-1) for ni in n]
b = 'd'
V0 = np.zeros(n)
m = 1
h = 1
print('Intialization Complete..')

S = QuantumSystem(n, dx, b, V0, m, h)
x = S.x[0]
Vh = .5*m*(1/dx[0])**2*(x-x[int(n[0]/2)])**2
S.setPotential(Vh)
print('System Constructed..')
[u, v] = S.energyEigenStates()
print('Eigenvalues and Eigenvectors Calculated..')

k = 512
dt = (2*np.pi*h/u[0])/(k-1)
t = np.linspace(0, (k-1)*dt, k)
print('Temporal Domain Constructed..')

f = v[:, 0]+v[:, 1]
print('Initial Wave Defined..')
F = S.timeEvolution(f, dt, k)
P = S.probDens(F)
print('Time Evolution Complete..')
mX, mP = S.phaseSpace(F)
mT, mV = S.virial(F)
mE = mT+mV
print('Mean Values Calculated..')
```