Differential Equations

Computational Notebook

Clancy Andrews

Contents

Introduction

The intention of this notebook is to give a complete overview of the scientific computing methods available for solving ordinary differential equation, partial differential equations, and systems of equations. The idea when solving these equations is to find ways to represent high level mathematics in terms of something that a computer can understand. We will address three specific problems, Van der Pol Oscillator, Quantum Harmonic Oscillator, and Reaction-Diffusion, with observations into the methods employed to solve them.

There are various sources of information on the topic of scientific computing. Textbooks on the topic, including but not limited to those done by Heath (2018) and Kincaid & Cheney (2009), give a deep understanding to mathematical computing topics, with introductory material, to numerical methods for solving differential equations.

While scientific computing has just started taking off with the introduction of the computer, problems and methods for solving complex equations have been identified and approached centuries before. Gustafsson (2018) addressed these topics in the textbook Scientific Computing, A Historical Perspective, where the idea of computation was more, or less, achieved before the invention of computers, and how it evolved after the fact. We can observe a method that was developed and was computationally complex before the utilization of computers, which was the simplex method for linear optimization. Bertsimas and Tsitsiklis (1997) demonstrate the simplex algorithm using a tableau, which can be established and carried out by hand for smaller linear programming problems. The advent of the computer has since allowed this method to be achieved with extreme precision and efficiency for much larger formulations.

Work in the field has not slowed down, with more complex problems being addressed and solved today. The paper "Parallel matching-based AMG preconditioners for elliptic equations discretized by IgA", written by D'Ambra, Durastante, and Filippone (2025), is one example of improvements and capabilities made possible by high-performance computing, and efficient methods for solving problems in the field. In their work, D'Ambra, Durastante, and Filippone (2025) implement Krylov subspace methods, with parallel-computing frameworks, to efficiently solve discretized elliptic boundary value problems. Another example can be found by the work of Li, Di, and Jiang (2025), for which they use a Two-Step Generalized RBF Finite Difference method to solve partial differential equations on manifolds. With both of these examples, it is quite evident that the field of scientific computing has developed substantially following major improvements in methodological refinement and technological advancement.

As we begin to progress through the methods, keep in mind the order of error, the type of method, and any nuance details that set each method apart from the rest. Each method has its own benefits and drawbacks, depending on the use and problem formulation.

Methods

There are various methods we can employ to solve problems. The following sections will be a brief overview of methods for solving initial value problems, boundary value problems, and partial differential equations, with attention to their properties and stability.

Methods for Inital Value Problems

Initial value problems arise when a differential equation is paired with an initial state at a specific starting time. The methods that we will address will work utilize the initial value of the equation at the starting time t_0, which will need to be derived or assumed. There are two main types of methods in this section. Those are single-step and multi-step methods. The idea of single-step methods is to use an adjacent point in time to calculate the next step in the series. Multi-step methods are not restricted to just the immediate, adjacent point in time. They will often use points two or more steps away. The Adams-Bashforth and Adams-Moulton methods are the only two multi-step methods that we will go over, the rest with be single-step.

Forward Euler

The Forward Euler method is defined as follows

$$y_{n+1} = y_n + \Delta t f(t_n, y_n) \tag{1}$$

where $f(t_n, y_n) = \frac{dy_n(t)}{dt_n}$. We can observe that the method is explicit, and once derived with the Taylor Expansion, has local error of $O(\Delta t^2)$ and a global error of $O(\Delta t)$. In terms of stability, let

$$\frac{dy}{dt} = \lambda y$$

with $y(0) = y_0$. If we replace $f(t_n, y_n)$ with λy_n, we get

$$\begin{aligned} y_{n+1} &= y_n + \Delta t \lambda y_n \\ &= y_n(1 + \Delta t \lambda). \end{aligned}$$

As we iterate from 1 to n, we then get

$$\begin{aligned} y_1 &= y_0(1 + \Delta t \lambda) \\ y_2 &= y_0(1 + \Delta t \lambda)^2 \\ &\vdots \\ y_n &= y_0(1 + \Delta t \lambda)^n. \end{aligned}$$

Given a small error ϵ, we have

$$y_n = y_0(1 + \Delta t\lambda)^n + \epsilon(1 + \Delta t\lambda)^n.$$

While ϵ may be small, the error term may blow up over many iterations. Let

$$r_{FE} = 1 + \Delta t\lambda.$$

For $\lambda > 0$, $|r_{FE}|$ will always be greater that 1, resulting in the Forward Euler method being unconditionally unstable. For $\lambda < 0$, $|r_{FE}| < 1$, when $\Delta t < \frac{2}{\lambda}$, the Forward Euler method is stable. Otherwise, when $\Delta t > \frac{2}{\lambda}$, the method is unstable. As a result, our method has limited stability.

Backward Euler

The Backwards Euler method is defined as follows

$$y_n = y_{n+1} - \Delta t f(t_{n+1}, y_{n+1}) \tag{2}$$

where $f(t_{n+1}, y_{n+1}) = \frac{dy_{n+1}(t)}{dt_{n+1}}$. We can observe that this method has the same local and global error as the Forward Euler method. Since the Backward Euler method is implicit, the initial y_{n+1} term can be determined using an different explicit method, i.e., Forward Euler. For the stability of this method, we can again let

$$\frac{dy}{dt} = \lambda y$$

where

$$\begin{aligned} y_n &= y_{n+1} - \Delta t\lambda y_{n+1} \\ &= y_{n+1}(1 - \Delta t\lambda). \end{aligned}$$

With further rearrangements, we arrive at

$$y_{n+1} = y_n(1 - \Delta t\lambda)^{-1}.$$

Letting $r_{BE} = (1 + \Delta t\lambda)^{-1}$, we get unconditional stability for $\lambda < 0$ and $r_{BE} < 1$. The Backward Euler method is also stable when $\Delta t > \frac{2}{|\lambda|}$ for $\lambda > 0$ and $|r_{BE}| > 1$. However, the method is unstable when $\Delta t < \frac{2}{|\lambda|}$ for $\lambda > 0$ and $|r_{BE}| > 1$. Compared to the Forward Euler method, the Backward Euler method has a larger range of stability.

Heun (Average)

The Heun method is defined as follows

$$y_{n+1} = y_n + \frac{\Delta t}{2}\Big[f(t_n, y_n) + f(t_{n+1}, y_{n+1})\Big] \tag{3}$$

where $f(t_n, y_n) = \frac{dy_n(t)}{dt_n}$ and $f(t_{n+1}, y_{n+1}) = \frac{dy_{n+1}(t)}{dt_{n+1}}$. One major difference between the Heun method and the two previous methods mentioned is the global and local error of this method. The Heun method has $O(\Delta t^3)$ and $O(\Delta t^2)$ local and global error, respectively. This results in a higher accuracy when making approximations, especially after many iterations.

Runge-Kutta 2 (Midpoint) & Runge-Kutta 4

The Runge-Kutta 2 method is defined as follows

$$y_{n+1} = y_n + \Delta t\left(f\left[t_n + \frac{\Delta t}{2}, y\left(t_n + \frac{\Delta t}{2}\right)\right]\right) \tag{4}$$

where $f(t_n, y_n) = \frac{dy_n(t)}{dt_n}$. The Runge-Kutta 4 method is defined as follows

$$y_{n+1} = y_n + \frac{\Delta t}{6}\Big(F_1 + 2F_2 + 2F_3 + F_4\Big) \tag{5}$$

where

$$\begin{aligned}
F_1 &= f(t_n, y_n) = \frac{dy_n(t)}{dt_n}, \\
F_2 &= f\left(t_n + \frac{\Delta t}{2}, y_n + \frac{\Delta t}{2}F_1\right), \\
F_3 &= f\left(t_n + \frac{\Delta t}{2}, y_n + \frac{\Delta t}{2}F_2\right), \\
F_4 &= f\Big(t_n + \Delta t, y_n + \Delta t F_3\Big).
\end{aligned}$$

Both of the Runge-Kutta methods are derived from the Talyor series, much like Forward Euler was, but in these cases, we did not limit ourselves to the first three terms of the series. We can continue to expand these methods, and gain better accuracy in doing so. However, it may not always be feasible to calculate the nth derivative, so we must accept trade offs when making approximations.

Adams-Bashforth & Adams-Moulton

As mentioned at the beginning of this section, the Adams-Bashforth and Adams-Moulton methods are multi-step methods, with the Adams-Bashforth method being explicit and the Adams-Moulton method being implicit. The Adams-Bashforth method is defined as follows

$$y_{n+1} = y_n + \frac{\Delta t}{2}\Big(3f_n - f_{n-1}\Big) \tag{6}$$

where $f_n = f(t_n, y_n) = \frac{dy_n(t)}{dt_n}$ and $f_{n-1} = f(t_{n-1}, y_{n-1}) = \frac{dy_{n-1}(t)}{dt_{n-1}}$. The Adams-Moulton method is defined as follows

$$y_{n+1} = y_n + \frac{\Delta t}{2}\Big(f_{n+1} - f_n\Big) \tag{7}$$

where $f_n = f(t_n, y_n) = \frac{dy_n(t)}{dt_n}$ and $f_{n+1} = \frac{dy_{n+1}(t)}{dt_{n+1}}$. The combination of these methods are what we refer to as predictor-corrector methods. The idea behind predictor-corrector methods is to use a fast, simple predictor (explicit) method to approximate the next solution point, then utilize that approximation in the corrector (implicit) method. This allows us to efficiently converge to a solution while maintaining stability. As it relates to the "Adams' family" methods, the Adams-Bashforth method (6) is our predictor, and the Adams-Moulton method (7) is our corrector.

Methods for Boundary Value Problems

Another set of problems are boundary value problems. Unlike initial value problems, BVPs require two or more initial conditions, which will need to be derived or assumed. Working with differential equations allows us to implement different boundary conditions, some of which are Neumann boundary conditions, Dirichlet boundary conditions, and Robin (mixed) boundary conditions. Neumann boundary conditions have us use the boundary values of our derivative function $f'(t_0) = \alpha$ and $f'(t_n) = \beta$ where Dirichlet boundary conditions $f(t_0) = \alpha$ and $f(t_n) = \beta$. For Robin boundary conditions, we get to use a linear combination of the Neumann and Dirichlet boundary conditions, and it will be defined as

$$\alpha_1 f'(t_0) + \beta_1 f(t_0) = \gamma_1$$

and

$$\alpha_2 f'(t_n) + \beta_2 f(t_n) = \gamma_2.$$

With these notions of boundary conditions set, we can proceed into the methods that can be employed to solve boundary valued problems.

Shooting Method

The Shooting method is a method that can be used to solve boundary value problems, especially problems where it may be difficult to get boundary conditions. It transforms the boundary value problem into an initial value problem, and then "shoots" toward a solution, with the goal of getting closer to the target with every consecutive shot. To achieve this, the method starts with taking an initial condition, and then solving the problem to see where the solution "lands" relative to the target, which in this case is our boundary condition. Once we determine the distance the "shot" was from the target, we adjust the initial guess using a root-finding method, such as the bisection or Newton-Raphson root-finding methods. The Newton-Raphson method is defined as

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}. \tag{8}$$

In order to achieve success with this method, by the intermediate value theorem, the continuous function f must have at least one root on the interval (a, b) where $f(a)f(b) < 0$. That is, $f(a)$ must have a different sign then $f(b)$.

Direct Method for Linear Boundary Value Problems

The goal for the direct method is to transform the linear boundary value problem into a system of linear equations, which can be solved via $A\vec{x} = \vec{b}$. Suppose we are trying to solve the linear boundary value problem

$$y'' = p(t)y' + q(t)y + r(t).$$

We can populate our A matrix by using the central difference method to approximate rows $2, \ldots, n-1$, and using forward and backward difference for rows 1 and n, respectively. Central difference for the first derivative is defined as

$$y'(t) = \frac{y(t + \Delta t) - y(t - \Delta t)}{2\Delta t} \tag{9}$$

where the second derivative is defined as

$$y''(t) = \frac{y(t + \Delta t) - 2y(t) + y(t - \Delta t)}{\Delta t^2}. \tag{10}$$

For Forward Difference, we have

$$y'(t) = \frac{-3y(t) + 4y(t + \Delta t) - y(t + 2\Delta t)}{2\Delta t} \tag{11}$$

and for Backward Difference, we have

$$y'(t) = \frac{3y(t) - 4y(t - \Delta t) + y(t - 2\Delta t)}{2\Delta t}. \tag{12}$$

Once those methods are used to populate A, we get

$$A = \begin{bmatrix} 1 & 0 & 0 & 0 & \cdots & 0 \\ a_1 & b_1 & c_1 & 0 & \cdots & 0 \\ 0 & a_2 & b_2 & c_2 & \cdots & 0 \\ 0 & 0 & a_3 & b_3 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & 1 \end{bmatrix}$$

where

$$\begin{aligned} a_i &= 1 + \frac{1}{2}p(t_i)\Delta t, \\ b_i &= -2 - q(t_i)\Delta t^2, \\ c_i &= 1 - \frac{1}{2}p(t_i)\Delta t \end{aligned}$$

for $i = 1, \ldots, n$. Our $\vec{b}$ will be the boundary values with $\vec{b}_0 = \alpha$ and $\vec{b}_n = \beta$. All other values will be $\vec{b}_i = r(t_i)\Delta t$ for $i = 1, \ldots, n-1$. It is important to note that as our problem gets bigger, so does A and $\vec{b}$, making the solution more costly for the computer. The next section will discuss methods for solving systems of equations, with methods that account for computational complexity.

Methods for Systems of Equations

While we might know some methods for solving systems of equations, i.e., Gaussian Elimination, we will also address some other methods that perform with better efficiency for problems with higher computational complexity.

Gaussian Elimination

Gaussian Elimination is an algorithm that performs elementary row operations on an augmented matrix $[A|b]$, resulting in a simplified form that can then be solved (or determine solvability). The following example will walk us through the steps taken for Gaussian Elimination. We first need to perform diagonalization on $[A|b]$, putting the augmented matrix into row echelon form. Suppose we have

$$[A|b] = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 2 & 4 & -1 \\ 1 & 3 & 9 & 1 \end{bmatrix}.$$

We can reduce the matrix by starting with adding R_2 by $-R_1$ and R_3 by $-R_1$. We get

$$[A|b] = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 0 & 1 & 3 & -2 \\ 0 & 2 & 8 & 0 \end{bmatrix}.$$

We can reduce the matrix again by adding R_3 with $-2R_2$. We then get

$$[A|b] = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 0 & 1 & 3 & -2 \\ 0 & 0 & 2 & 4 \end{bmatrix}.$$

Now that our augmented matrix is in row echelon form, we can perform backward substitution on the matrix. We have the following equations:

$$\begin{aligned} 2x_3 = 4 &\Rightarrow x_3 = 2 \\ x_2 + 3x_3 = -2 &\Rightarrow x_2 + 3(2) = -2 \\ &\Rightarrow x_2 = -8 \\ x_1 + x_2 + x_3 = 1 &\Rightarrow x_1 + (-8) + (2) = 1 \\ &\Rightarrow x_1 = 7. \end{aligned}$$

The resulting solution is $\vec{x}^T = [7, -8, 2]$.

While this method works great on paper and is easy to program, as our problems get larger, computational run time takes a hit. For our diagonalization step, it takes n pivots, for n rows, for n columns. This results in $O(n^3)$ computational time complexity. Our backward substitution step only takes $O(n^2)$, which results in a total worst case computational time complexity of $O(n^3) + O(n^2) = O(n^3)$. While this does not look great for trying to efficiently solve systems of equations, there are alternative methods that provide better time complexity.

LU Decomposition

Another method for solving systems of equations, by which the matrix A is decomposed into the product of two triangular matrices, is known as LU Decomposition. As stated, the goal is to break A into two triangular matrices L and U, such that $A = LU$. In this case, L is our lower triangular matrix, which we define as

$$L = \begin{bmatrix} 1 & 0 & 0 & \cdots & 0 \\ c_{21} & 1 & 0 & \cdots & 0 \\ c_{31} & c_{32} & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ c_{n1} & c_{n2} & c_{n3} & \cdots & 1 \end{bmatrix}$$

where c_{ij} is our scaling factor that we would perform during Gaussian Elimination at row i and column j in our matrix. We define our upper triangular matrix U as

$$U = \begin{bmatrix} x_{11} & x_{12} & x_{13} & \cdots & x_{1n} \\ 0 & x_{22} & x_{23} & \cdots & x_{2n} \\ 0 & 0 & x_{33} & \cdots & x_{2n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & x_{nn} \end{bmatrix}$$

where x_{ij} is the ith row and jth column element of A. We can then let $U\vec{x} = \vec{y}$, and solving $A\vec{x} = \vec{b}$, we get

$$L\vec{y} = LU\vec{x} = A\vec{x} = \vec{b}.$$

The worst case computational time complexity is now down to $O(n^2)$ from $O(n^3)$ performed by Gaussian Elimination. It is worthwhile to mention a similar method to LU Decomposition, which is Cholesky's factorization. While similar to LU, Cholesky's makes use of the L and U matrices when $U = L^T$. By one theorem on the Cholesky factorization method, if A is a real, symmetric, and positive definite matrix, then it has a unique factorization, $A = LL^T$, in which L is lower triangular with a positive diagonal. When all of the conditions are met for Cholesky factorization, the method will only take half as many steps as LU Decomposition does.

Jacobi & Gauss-Seidel

Unlike the previous methods we explored to solve systems of equations, which were direct methods, we will now explore two indirect methods. Indirect methods produce a sequence of vectors that converge toward a solution. We will begin by defining the Jacobi method as follows

$$\begin{aligned}
x_1^{k+1} &= (-a_{12}x_2^k - a_{13}x_3^k - \ldots - a_{1n}x_n^k + b_1)/a_{11} \\
x_2^{k+1} &= (-a_{21}x_1^k - a_{23}x_3^k - \ldots - a_{2n}x_n^k + b_2)/a_{22} \\
&\vdots \\
x_n^{k+1} &= (-a_{n1}x_1^k - a_{n2}x_2^k - \ldots - a_{nn-1}x_{n-1}^k + b_n)/a_{nn}
\end{aligned}$$

Given an initial value $\vec{x}^0$, the Jacobi method is convergent if and only if A is strictly diagonal dominant. That is, if the diagonal element is strictly larger than the sum of all other element in that row, then it is convergent. If we let $A = D + R$ where D is the diagonal of A and R is all other elements of A, then we get

$$A\vec{x} = (D + R)\vec{x} = \vec{b}$$

Once we have our D, R, and $\vec{b}$, we can solve for $\vec{x}^{k+1}$ as follows:

$$\vec{x}^{k+1} = D^{-1}(-R\vec{x}^k + \vec{b}).$$

An alternative to this is to let $A = L + U^s$ where L is the lower triangular matrix, including the diagonal, and U^s is the strict upper triangular matrix. We then get

$$\vec{x}^{k+1} = L^{-1}(-U^s\vec{x}^k + \vec{b}).$$

While $A = L + U^s$ is generally faster sequentially than letting $A = D + R$, the latter formulation allows for parallel execution of the method, resulting in a greater speedup.

The Gauss-Seidel method is quite similar to the Jacobi method, with one slight change. Once we calculate x_1^{k+1}, we can plug the estimation into for formula for x_2^{k+1}, and continue this process until we reach x_n^{k+1}. The formulation is as follows

$$\begin{aligned}
x_1^{k+1} &= (-a_{12}x_2^k - a_{13}x_3^k - \ldots - a_{1n}x_n^k + b_1)/a_{11} \\
x_2^{k+1} &= (-a_{21}x_1^{k+1} - a_{23}x_3^k - \ldots - a_{2n}x_n^k + b_2)/a_{22} \\
x_3^{k+1} &= (-a_{31}x_1^{k+1} - a_{32}x_2^{k+1} - \ldots - a_{3n}x_n^k + b_3)/a_{33} \\
&\vdots \\
x_n^{k+1} &= (-a_{n1}x_1^{k+1} - a_{n2}x_2^{k+1} - \ldots - a_{nn-1}x_{n-1}^{k+1} + b_n)/a_{nn}
\end{aligned}$$

The altered method will allow for a faster convergence to the solution than what is achieved by the Jacobi method.

Methods of Lines

The goal of Methods of Lines is to discretize more than one dimension of a given partial differential equation. Once we have reduced our equations down to a continuous, single dimension, we can go about solving them as we have mentioned previously. In this instance, we discretize our spacial domain, and leave our time domain continuous. For each method, there will be an ordered pair followed behind the name of the method. We read this, (m, n), as mth order in time, nth order in space. This is similar to our local/global errors we discussed in previous sections, but this time it applies to space and time, since these methods address partial differential equations. For our discussion of stability for each of the following methods, let $\lambda_1 = \frac{c\Delta t}{\Delta x}$ and $\lambda_2 = \frac{c\Delta t}{(\Delta x)^2}$. Our stability criterion is such that the method is stable if $|g| < 1$ and unstable if $|g| > 1$.

Finite Difference (1,2)

The Finite Difference method is defined as follows

$$u_n^{m+1} = u_n^m + \frac{c\Delta t}{2\Delta x}\left(u_{n+1}^m - u_{n-1}^m\right). \tag{13}$$

With our definition, we can address method stability for the Finite Difference method. We will start with stability for solving partial differential equations of the form $\frac{du}{dt} = c\frac{du}{dx}$. Let $u_n^m = g^m e^{ik_n x}$ where $k_n = \frac{n\pi}{L}$. Plugging $g^m e^{ik_n x}$ in for all u_n^m values, we get

$$g^{m+1}e^{ik_n x} = g^m e^{ik_n x} + \frac{\lambda_1}{2}\left(g^m e^{ik_{n+1}x} - g^m e^{ik_{n-1}x}\right).$$

Multiplying both side of the equation by $g^{-m}e^{-ik_n x}$, we arrive at

$$\begin{aligned} g &= 1 + \frac{\lambda_1}{2}\left(e^{i\Delta x} - e^{-i\Delta x}\right) \\ &= 1 + \lambda_1 i \sin(\Delta x). \end{aligned}$$

We then have

$$|g|^2 = gg^* = 1 + \lambda_1^2 \sin^2(\Delta x)$$

which results in $|g| > 1$, making this method unstable.

For solutions of partial differential equations of the form $\frac{du}{dt} = c\frac{d^2u}{dx^2}$, we can go through the same process as we did previously, arriving at stability, as long as $\lambda_2 < \frac{1}{2}$.

Backwards Difference (1,2)

The Backwards Difference method is defined as follows

$$u_n^m = u_n^{m+1} - \frac{c\Delta t}{2\Delta x}\left(u_{n+1}^{m+1} - u_{n-1}^{m+1}\right). \tag{14}$$

For our discussion of stability for Backwards Difference, let $u_n^m = g^m e^{ik_n x}$ where $k_n = \frac{n\pi}{L}$. Plugging $g^m e^{ik_n x}$ in for all u_n^m values, we get

$$g^m e^{ik_n x} = g^{m+1} e^{ik_n x} - \frac{\lambda_1}{2} g^{m+1}\left(e^{ik_{n+1}x} - e^{ik_{n-1}x}\right).$$

Multiplying both side of the equation by $g^{-m}e^{-ik_n x}$, we arrive at

$$\begin{aligned} 1 &= g\left(1 - \lambda_1 i \sin(\Delta x)\right) \\ g &= \left(1 - \lambda_1 i \sin(\Delta x)\right)^{-1}. \end{aligned}$$

We then have

$$|g|^2 = gg^* = \left(1 + \lambda_1^2 \sin^2(\Delta x)\right)^{-1} < 1$$

which results in $|g| < 1$, making this method stable.

In terms of solving partial differential equations of the form $\frac{du}{dt} = c\frac{d^2u}{dx^2}$, one perk to the Backwards Difference method is that it is unconditionally stable, similar to the Backwards Euler method. However, as it is similar to the Backwards Euler method in stability, it is also an implicit methods. An explicit scheme will need to be used to determine initial conditions.

Leap Frog (2,2) & Leap Frog (2,4)

The Leap Frog (2,2) method is defined as follows

$$u_n^{m+1} = u_n^{m-1} - \frac{c\Delta t}{\Delta x}\left(u_{n+1}^m - u_{n-1}^m\right). \tag{15}$$

The Leap Frog (2,4) method is defined as follows

$$u_n^{m+1} = u_n^{m-1} - \frac{c\Delta t}{6\Delta x}\left(-u_{n+2}^m + 8u_{n+1}^m - 8u_{n-1}^m + u_{n+2}^m\right). \tag{16}$$

With our definition, we can address method stability for the Leap Frog (2,2) method. We will start with stability for solving partial differential equations of the form $\frac{du}{dt} = c\frac{du}{dx}$. Let $u_n^m = g^m e^{ik_n x}$ where $k_n = \frac{n\pi}{L}$. Plugging $g^m e^{ik_n x}$ in for all u_n^m values, we get

$$g^{m+1}e^{ik_n x} = g^{m-1}e^{ik_n x} + \lambda_1 g^m \left(e^{ik_{n+1}x} - e^{ik_{n-1}x}\right).$$

Multiplying both side of the equation by $g^{-m}e^{-ik_n x}$, we arrive at

$$g^2 = 1 + i2\lambda_1 g \sin(k_n \Delta x).$$

We then have

$$g_{1,2} = i\lambda_1 \sin(k_n \Delta x) \pm \sqrt{1 - \lambda_1^2 \sin^2(k_n \Delta x)}$$

which results in $|g| < 1$ for $\lambda_1 < 1$, making this method stable.

For solutions of partial differential equations of the form $\frac{du}{dt} = c\frac{d^2u}{dx^2}$, we can go through the same process as we did previously, arriving at unconditional instability, regardless of our value of λ_2.

Spectral Methods

Alternative to solving differential equations by representing functions as polynomials, spectral methods represent functions as the sums of basis functions. As a result, we are able to calculate solutions more efficiently. We will discuss two methods, Fast Fourier Transform and Chebyshev Polynomials, that we can use to accurately approximate solutions.

Fast Fourier Transform

The Fourier Transform is a technique that transforms a function of time to a function of frequency. We define the Fourier Transform as follows

$$\widehat{f(x)} = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} e^{-ikx} f(x) dx \tag{17}$$

where $x \in (-\infty, \infty)$. If we take the Fourier Transform again, i.e., $\widehat{\widehat{f(x)}}$, we get

$$\widehat{\widehat{f(x)}} = f(x) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} e^{-ikx} \widehat{f(x)} dx \tag{18}$$

where $k \in (-\infty, \infty)$. We will also use the Fourier Series, which is easier for a computer to understand. That definition is

$$f(x) = b_0 + \sum_{n=1}^{N} a_n \sin\left(\frac{n\pi x}{L}\right) + \sum_{n=1}^{N} b_n \cos\left(\frac{n\pi x}{L}\right). \tag{19}$$

One property that is important to us, which allows us to determine a solution more easily is the derivative property for the Fourier Tranform:

$$\widehat{f'(x)} = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} e^{-ikx} f'(x) dx$$

Solving this equation, we can use integration by parts, with $u = e^{-ikx}$, $du = -ike^{-ikx}$, $v = f(x)$, and $dv = f'(x)dx$, we get

$$\widehat{f'(x)} = ik\ \widehat{f(x)}.$$

In fact, for any nth derivative of f, we have the following generalized formula

$$\widehat{f^{(n)}(x)} = (ik)^n \widehat{f(x)}. \tag{20}$$

The Fast Fourier Transform is a fast algorithm for computing the Discrete Fourier Transform, that has a worst case time computational complexity of $O(n \log n)$, as well as periodic boundaries, with $x \in [-L, L]$. We also want $N = 2^p$ where $p \in \mathbb{N}$.

Chebyshev Polynomials

This methods is spectral, much like FFT, with worst case computational complexity of $O(n \log n)$. The equation is defined as

$$\sqrt{1-x^2}\frac{d}{dx}\left(\sqrt{1-x^2}\frac{dT_n(x)}{dx}\right) + n^2 T_n(x) = 0 \tag{21}$$

where $T_n(x)$ is the Chebyshev Polynomial, and $x \in [-1, 1]$. Properties include haveing real valued eigenvalues and eigenfunctions, as well as using orthogonal polynomials. When determining the Chebyshev Polynomials, we can use the following equation

$$T_{n+1}(x) = 2xT_n(x) - T_{n-1}(x). \tag{22}$$

With our final method reviewed, we can now observe a few applications of our methods. It is important to make note of the fact that we can use more than one method to solve a given problem. In each of the following cases, we will consider more than one method, and discuss implementation for each.

Problems

With the methods established above, we can now proceed with solving the three problems mentioned at the beginning: Van der Pol Oscillator, the Quantum Harmonic Oscillator, and the Reaction-Diffusion equations.

Van der Pol Oscillator

The Van der Pol Oscillator is a second-order ordinary differential equation, defined as follows

$$\frac{d^2y(t)}{dt^2} + \epsilon\Big[y^2(t) - 1\Big]\frac{dy(t)}{dt} + y(t) = 0$$

with ϵ being our dampening parameter. By observation, we can see that the equation is nonlinear, and our ϵ parameter works to help amplify or decay our nonlinear portion $y^2(t) - 1$. Since we want ϵ to be small, we can set $\epsilon = 0.1$. We also want to setup our discretization of t, as well as our initial conditions $y_0' = y(0) = 2$ and $y_0' = \frac{dy(t_0=0)}{dt} = \pi^2$.

```
import numpy as np
from scipy.integrate import solve_ivp
import matplotlib.pyplot as plt

### Initial Conditions
epsilon = 0.1
tspan = np.arange(0, 32 + 1e-6, 0.1)
y0 = [2, np.pi**2]
```

In order to start solving this problem, we will want to put the equation in canonical form. We can start by defining the following:

$$\begin{aligned} w_1 &= y'(t), \\ w_2 &= y''(t) = w_1'(t). \end{aligned}$$

With our new variables, we can rearrange the original equation:

$$\begin{aligned} \frac{d^2y(t)}{dt^2} + \epsilon\Big[y^2(t) - 1\Big]\frac{dy(t)}{dt} + y(t) &= 0 \\ \frac{d^2y(t)}{dt^2} &= -\epsilon\Big[y^2(t) - 1\Big]\frac{dy(t)}{dt} - y(t) \\ \frac{dw_2}{dt} &= -\epsilon\Big[w_1^2 - 1\Big]w_2 - w_1. \end{aligned}$$

With the canonical form defined, we can establish our right-hand side function.

```
def Van_der_Pol(t, y, epsilon):
  """
  Implementation of the Van der Pol Oscillator
  """
  w1, w2 = y
  return w2, -epsilon * ((w1**2)-1) * w2 - w1
```

Having our initial conditions established, and our right-hand side function setup, we can now go about solving our initial value problem using the `scipy.integrate.solve_ivp` package.

```
### Solution of the Van der Pol Oscillator over t
sol = solve_ivp(Van_der_Pol, [tspan[0], tspan[-1]], y0, \
                args = [epsilon], t_eval = tspan)

t = sol.t
y1 = sol.y[0,:]        # Grabs the solutions to y'(t)
y2 = sol.y[1,:]        # Grabs the solutions to y''(t)

plt.figure(1)
plt.plot(t, y1, label = r"$y'(t)$")
plt.plot(t, y2, label = r"$y''(t)$")
plt.title(r"Van der Pol Oscillator with $\epsilon = 0.1$")
plt.xlabel(r"$t$", fontsize = 12)
plt.ylabel(r"$y(t)$", fontsize = 12)
plt.legend()
plt.show()
```

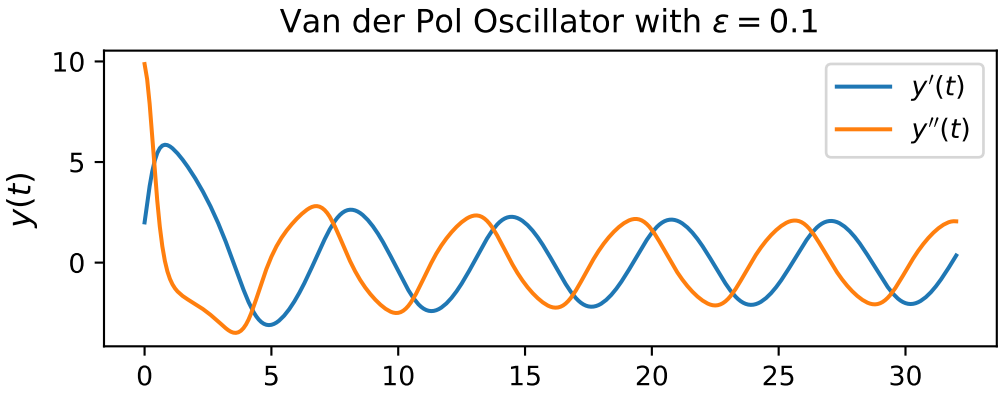

Figure 1: Solution of the Van der Pol Oscillator with $t \in [0, 32]$ and $\epsilon = 0.1$. Notice the instability for t on the interval $[0, 4]$.

We can observe in Figure 1 that, between $t = 0$ and $t = 4$, there is a bit of inconsistency in the approximation of $y''(t)$. To counteract this inconsistency, we can try decreasing our value of ϵ to 0.01.

```
### Same solution with different epsilon
epsilon = 0.01
sol = solve_ivp(Van_der_Pol, [tspan[0], tspan[-1]], y0, \
                args = [epsilon], t_eval = tspan)

t = sol.t
y1 = sol.y[0,:]        # Grabs the solutions to y'(t)
y2 = sol.y[1,:]        # Grabs the solutions to y''(t)

plt.figure(2)
plt.plot(t, y1, label = r"$y'(t)$")
plt.plot(t, y2, label = r"$y''(t)$")
plt.title(r"Van der Pol Oscillator with $\epsilon = 0.01$")
plt.xlabel(r"$t$", fontsize = 12)
plt.ylabel(r"$y(t)$", fontsize = 12)
plt.legend()
plt.show()
```

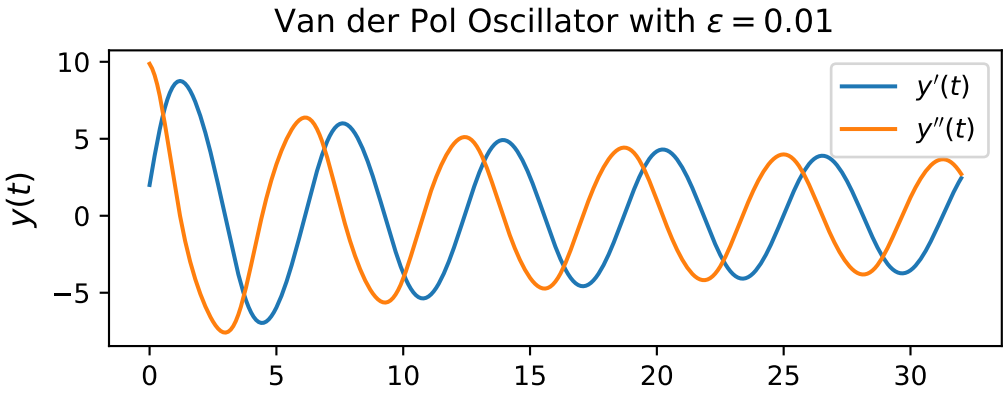

Figure 2: Solution of the Van der Pol Oscillator with $t \in [0, 32]$ and $\epsilon = 0.01$. We can now notice a more stable solution for t on the interval $[0, 4]$.

It is important to note that the `solve_ivp` function uses the Runge-Kutta 4 method by default. Changing the `method` parameter will allow us to utilize different methods, some of which may be better to use, depending on the problem. For this problem, we were able to give examples on applications of methods for solving initial value problems. The next problem will allow us to utilize methods for solving boundary value problems and systems of equations.

Quantum Harmonic Oscillator

We start with the probability density evolution in a one-dimensional harmonic trapping potential, which is governed by the partial differential equation

$$i\hbar\psi_t + \frac{\hbar^2}{2m}\psi_{xx} - V(x)\psi = 0 \tag{23}$$

where ψ is the probability density and $V(x) = \frac{kx^2}{2}$ is the harmonic confining potential. Let us assume a solution to equation (23) is as follows

$$\psi(x,t) = \sum_{n=1}^{N} a_n \phi_n(x) e^{\left(-i\frac{E_n}{2\hbar}t\right)}, \tag{24}$$

which is the eigenfunction expansion solution. Plugging equation (24) into equation (23), we get

$$\frac{d^2\phi_n}{dx^2} - \left[Kx^2 - \epsilon_n\right]\phi_n = 0 \tag{25}$$

where $\phi(x)$ goes to 0 as x goes to $\pm\infty$, and $\epsilon_n > 0$ is the quantum energy. In equation (25), it is also worth noting that $K = \frac{km}{\hbar^2}$ and $\epsilon_n = \frac{E_n m}{\hbar^2}$. With this problem established, we can now work on constructing a computational solution. We want to start with imports and initial values for the problem.

```
import numpy as np
from scipy.integrate import solve_ivp, trapezoid
from scipy.linalg import eig
import matplotlib.pyplot as plt

TOL = 1e-4
L = 4
K = 1
A = 1
epsilon = 0.1

# Setup initial conditions
dx = 0.1
xspan = np.arange(-L, L + dx, dx) #-L to L with step size 0.1
N = xspan.size
```

For this setup, we will assume that $K = 1$ and $\epsilon_n = 0.1$. We will also want to setup the right-hand side function for equation (25).

```
def quantum_harmonic_rhs(x, phi, beta, eps):
    """
    Implementation of the right hand side of the
    quantum harmonic oscillator function
    """
    y1, y2 = phi
    dy2 = (beta - eps) * y1    # beta = K * (x**2)
    return [y2, dy2]           # Returns y'1, y'2
```

For this first attempt at a solution, we will want to implement the shooting method to solve our equation. The following is a psuedo-implementation of that method.

```
def shooting_method(epsilon):
    """
    Psuedo-implementation of the shooting method for
    iterating over eps_n
    """
    beta0 = epsilon
    for mode in range(1,6):
        beta = beta0
        dbeta = beta0 / 100
        for _ in range(1000):
            solution = solve_ivp(quantum_harmonic_rhs,
                                 [xspan[0], xspan[-1]],
                                 [A, (beta - epsilon_start)*A],
                                 args=[beta,
                                       epsilon_start],
                                 t_eval=xspan)

            right_hand_boundary = (beta - epsilon)*A
            if np.abs(solution.y[:,-1] - right_hand_boundary < TOL:
                print(f"Eps = {beta}")
                norm = np.sqrt(trapezoid(np.abs(solution.y)**2,
                                         solution.t))
                return solution.y / norm

            if (-1) ** (mode + 1) * solution.y[:,-1] > 0:
                beta -= dbeta
            else:
                beta += (dbeta/2)
                dbeta /= 2
```

Once we have established our initial condition, the right-hand side of our Quantum Harmonic Oscillator equation, and the function for the shooting method, we can now observe the solution to

the problem. In this instance, we will want to normalize such that $\int_{\infty}^{\infty} |\phi_n|^2 dx = 1$, which we performed in the shooting method function. We normalize in this step to ensure meaningful probability densities for our solution.

```
solution = shooting_method(epsilon)
```

We can also solve the problem using the Direct method, coupled with methods for solving systems of equations.

```
import numpy as np
from scipy.integrate import solve_ivp, trapezoid
from scipy.linalg import eig
import matplotlib.pyplot as plt

TOL = 1e-4
L = 4
K = 1
A = 1
epsilon_start = 0.1

# Setup initial conditions
dx = 0.1
xspan = np.arange(-L, L + dx, dx) #-L to L with step size 0.1
N = xspan.size


# Initialize A
A_matrix = np.zeros((N,N))

def forward_difference():
    """
    Implementation of the forward difference scheme
    """
    A_matrix[0,0] = (2/3) + (dx**2)*K*(xspan[0]**2)
    A_matrix[0,1] = -2/3


def backward_difference():
    """
    Implementation of the backward difference scheme
    """
    A_matrix[N-1,N-2] = -2/3
    A_matrix[N-1,N-1] = (2/3) + (dx**2)*K*(xspan[-1]**2)
```

```
def central_difference():
    """
    Implementation of the central difference scheme
    """
    for i in range(1,N-1):
        A_matrix[i,i-1] = -1
        A_matrix[i,i+1] = -1
        A_matrix[i,i] = 2 + (dx**2)*K*(xspan[i]**2)


# Populate the values of A

forward_difference()
backward_difference()
central_difference()

# Solve Ax=ex
eigen_values, eigen_vectors = eig(A_matrix)

# Normalize first 5 eigenvalues and vectors
A_vals = eigen_values[:5]
A_vects = [np.abs(eigen_vectors[i].reshape(-1,1)) for i in range(5)]

for i in range(5):
    norm = np.sqrt(trapezoid(np.abs(A_vects[i].flatten())**2,
                                    xspan,
                                    dx))
    A_vects[i] /= norm
```

In summary, the Quantum Harmonic Oscillator can prove to be an ideal benchmark for testing boundary value problem methods. We are able to show emphasis on the iterative approach via the Shooting method, where we set our boundary conditions to be some value L, which the computer can represent. We can also encode the problem as a matrix, and perform different methods for solving systems of equations, allowing for efficient and fast execution to obtain the solution. The next problem will explore spectral methods for solving partial differential equations.

Reaction-Diffusion Equations

The Reaction-Diffusion equation helps to solve for, and find, phenomena when one or more substances change over space and time. Those equations are defined as follows

$$U_t = \lambda(A)U - \omega(A)V + D_1\nabla^2 U$$
$$V_t = \omega(A)U - \lambda(A)V + D_2\nabla^2 V$$

where $A^2 = U^2 + V^2$ and $\nabla^2 = \partial_x^2 + \partial_y^2$. In this formulation, we will look at the system where

$$\lambda(A) = 1 - A^2,$$
$$\omega(A) = -\beta A^2.$$

We will now investigate the solutions to this system formulation using two different methods, those being the spectral Fast Fourier Transform method with periodic boundary conditions and the Chebyshev Polynomial method with Dirichlet boundary conditions.

For the first solution, we will being by implementing the initial conditions for the spectral fast fourier transform. We will assume that $x, y \in [-10, 10]$, $n = 64$, $\beta = 1$, $D_1 = D_2 = 0.1$, $m = 1$, and time spanning 0 to 4 with time steps of 0.5.

```
import numpy as np
from scipy.integrate import solve_ivp
from numpy.fft import fft2, ifft2
from math import pi

n = 64
beta = 1
D1 = D2 = 0.1
m = 1            # Number of spirals
L = 10

tspan = np.arange(0, 4 + 1e-6, 0.5)
xspan = np.linspace(-L, L, n+1)
x = xspan[:n]
y = x

X,Y = np.meshgrid(x,y, indexing='xy')

u = np.tanh(np.sqrt(X**2 + Y**2)) \
    * np.cos(m * np.angle(X + 1j*Y) - (np.sqrt(X**2 + Y**2)))
v = np.tanh(np.sqrt(X**2 + Y**2)) \
    * np.sin(m * np.angle(X + 1j*Y) - (np.sqrt(X**2 + Y**2)))

ut = fft2(u)
vt = fft2(v)

n2 = n*n

ut_flat = ut.reshape(n2, order='F')
vt_flat = vt.reshape(n2, order='F')

uv = np.concatenate([
    np.real(ut_flat), np.imag(ut_flat),
```

```
    np.real(vt_flat), np.imag(vt_flat)
])

# Spectral domain initialization
kx = (2*np.pi/(2*L)) \
      * np.concatenate((np.arange(0, n//2),
                        np.arange(-n//2, 0)))
kx[0] = 1e-6
ky = kx

KX, KY = np.meshgrid(kx, ky, indexing='xy')

K = KX**2 + KY**2
```

We want to get the real and imaginary parts of U and V separated appropriately, so when we put the initial conditions into our right-hand side function, we don't interfere with processing the function's input. We can now define our right-hand side function, which will take the `uv` vector in the frequency domain as an input.

```
def spectral_rhs(t, uv_vec, n, beta, D1, D2, K):
    """RHS for FFT spectral version

    Accepts a real vector y containing
    real/imag parts:
      y = [Re(ut); Im(ut); Re(vt); Im(vt)] of length 4*n^2

    Returns derivatives stacked the same way
    """
    n2 = n * n

    Re_ut = uv_vec[0:n2]
    Im_ut = uv_vec[n2:2 * n2]
    Re_vt = uv_vec[2 * n2:3 * n2]
    Im_vt = uv_vec[3 * n2:4 * n2]

    ut = (Re_ut + 1j * Im_ut).reshape((n, n), order='F')
    vt = (Re_vt + 1j * Im_vt).reshape((n, n), order='F')

    # Physical fields (take real part;
    # should remain real if symmetry holds)
    u = np.real(ifft2(ut))
    v = np.real(ifft2(vt))

    A2 = u ** 2 + v ** 2
```

```
    lam = 1 - A2
    omega = -beta * A2

    dudt_hat = fft2(lam * u - omega * v) - (K * D1) * ut
    dvdt_hat = fft2(omega * u + lam * v) - (K * D2) * vt

    dudt_flat = dudt_hat.reshape(n * n, order='F')
    dvdt_flat = dvdt_hat.reshape(n * n, order='F')

    # stack into real-imag parts
    dRe_ut = np.real(dudt_flat)
    dIm_ut = np.imag(dudt_flat)
    dRe_vt = np.real(dvdt_flat)
    dIm_vt = np.imag(dvdt_flat)

    return np.concatenate([dRe_ut, dIm_ut, dRe_vt, dIm_vt])
```

Now that our right-hand side has be defined, we can finish by solving our partial differential equations via the `solve_ivp` method.

```
spectral_sol = solve_ivp(
    spectral_rhs,
    [tspan[0], tspan[-1]],
    uv,
    args=[n, beta, D1, D2, K],
    t_eval=tspan
)
```

For our second solution, we will implement the initial conditions for the Chebyshev polynomial method. We again assume that $x, y \in [-10, 10]$, $\beta = 1$, $D_1 = D_2 = 0.1$, $m = 1$, and time spanning 0 to 4 with time steps of 0.5. However, we will now let $n = N = 30$.

```
import numpy as np
from scipy.integrate import solve_ivp
from math import pi
import matplotlib.pyplot as plt
import imageio

N = 30
L = 10
m = 1      #Number of spirals
beta = 1
D_1 = D_2 = 0.1
tspan = np.arange(0, 4 + 1e-6, 0.5)
```

We now want to go about implementing the right-hand side function and Chebyshev function that we can use to solve the equations. Like the spectral solution, we will be stacking our u and v vectors, so the dimension of the new vector will be 1922×1.

```
def cheb(N):
    if N == 0:
        return np.array([[0.]]), np.array([1.])
    x = np.cos(pi * np.arange(0, N + 1) / N).reshape(-1, 1)# column
    c = np.hstack(([2.],
                   np.ones(N - 1), [2.])) \
                   * ((-1) ** np.arange(0, N + 1))
    X = np.tile(x, (1, N + 1))
    dX = X - X.T
    C = c.reshape(-1, 1) @ (1.0 / c).reshape(1, -1)
    D = C / (dX + np.eye(N + 1))
    D = D - np.diag(np.sum(D, axis=1))
    return D, x.flatten()

def chebyshev_rhs(t, uv, beta, D_1, D_2, Lap):
    """
    Implementation of the right hand side of the
    reaction-diffusion equation

    uv is the horizontally stacked U and V flattened matrices
    Lap is the Laplacian d^2_x + d^2_y

    Returns the Ut and Vt as n**2 horizontal stacked vector
    """
    U,V = np.hsplit(uv, 2)  # Splits the uv vec into u and v vectors

    A2 = U**2 + V**2        # A^2 = U^2 + V^2

    lam = 1.0 - A2          # lambda(A) = 1 - A^2
    w = -beta * A2          # w(A) = -beta A^2

    Ut = np.array(lam * U - w * V + D_1 * (Lap @ U))# Solution to Ut
    Vt = np.array(w * U + lam * V + D_2 * (Lap @ V))# Solution to Vt

    return np.hstack((Ut, Vt))
```

Now that the Chebyshev and right-hand side functions have been implemented, we can setup the rest of the initial conditions before solving. For our initial U and V conditions, we will define them the same as we did with the spectral method.

```
D,x = cheb(N)
D = D / L           # Scale since x,y in [-L,L] -> [-1,1]
D2 = D @ D
D2[0,:] = 0         # Dirichlet boundary conditions (zero first row)
D2[-1,:] = 0        # Dirichlet boundary conditions (zero last row)

x = L*x             # Scale since x in [-1,1] -> [-L,L]
y = x
X,Y = np.meshgrid(x, y, indexing='xy')

u = np.tanh(np.sqrt(X**2 + Y**2)) \
    * np.cos(m * np.angle(X + 1j*Y) - (np.sqrt(X**2 + Y**2)))
u = u.reshape((N+1)*(N+1), order="F")

v = np.tanh(np.sqrt(X**2 + Y**2)) \
    * np.sin(m * np.angle(X + 1j*Y) - (np.sqrt(X**2 + Y**2)))
v = v.reshape((N+1)*(N+1), order="F")

uv = np.hstack((u,v)).T

I = np.eye(D.shape[0])
Lap = np.kron(I, D2) + np.kron(D2, I)
```

With all of the initial conditions and right-hand side function established, we can now solve the reaction diffusion equations via Chebyshev.

```
cheb_sol = solve_ivp(
    chebyshev_rhs,
    [tspan[0], tspan[-1]],
    uv,
    args=[beta,D_1,D_2,Lap],
    t_eval=tspan
)

frames = []

fig = plt.figure()
ax = fig.add_subplot(111,projection = "3d")

for j in range(len(cheb_sol.t)):
  curU = cheb_sol.y[: (N+1)*(N+1), j].reshape((N+1, N+1), order="F")

  ax.clear()
  ax.plot_surface(X, Y, curU, rstride=1, cstride=1, cmap="viridis")
```

```
  ax.set_xlim(-L, L)
  ax.set_ylim(-L, L)
  ax.set_zlim(curU.min(), curU.max())
  ax.set_xlabel("x")
  ax.set_ylabel("y")
  ax.set_zlabel("U(x,y)")
  ax.set_title(f"U(x,y) at t = {cheb_sol.t[j]:.2f}")
  ax.view_init(elev=45, azim=-45)

  plt.tight_layout()
  plt.savefig("frame.png", dpi=80)
  frames.append(imageio.imread("frame.png"))

plt.close(fig)
imageio.mimsave("chebyshev_reaction_diffusion.gif",
                frames,
                fps = 1.85
)
```

We can observe the spiral of $m = 1$ in the plot below (one frame from the visualization that was output):

```
## <matplotlib.colorbar.Colorbar object at 0x0000024F019FF290>
```

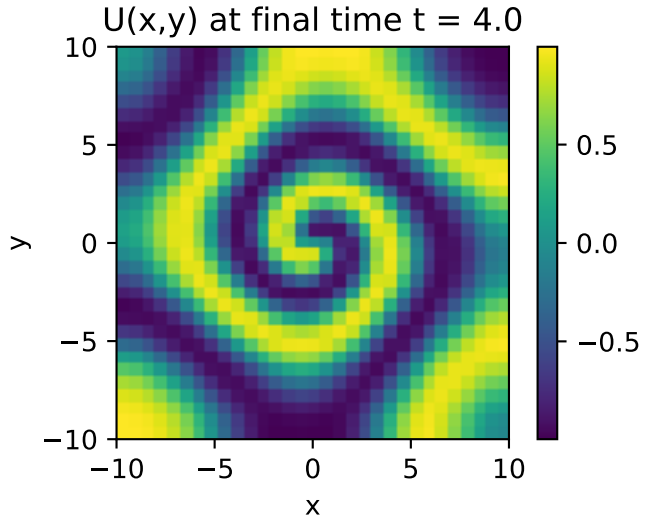

Figure 3: Solution of the Reaction-Diffusion equations with $t \in [0, 4]$ and $m = 1$. It is worth noting that we can increase m and observe the changes to the visualization.

It is important to note that we can increase m to be however many spirals we want in the solution. That can be a task for the reader to try and change parameters as they wish, and

observe the changes in the visualization. As a result of our computations in solving the Reaction-Diffusion equations, we were able to demonstrate the effectiveness of both the Fast Fourier Transform and Chebyshev Polynomials in terms of efficiency in calculating accurate approximations in both our space and time domains. Note that these are not the only ways to solve partial differential equations, and other methods can be explored in turn.

Conclusion

Throughout this notebook, we were able to discuss the various methods that can be employed to solve differential equations in areas of initial value problems, boundary value problems, and partial differential equations. While touching on methods such as spectral methods and methods of lines to solve partial differential equations, we also addressed a few methods for solving systems of equations with higher efficiency.

Once our discussion on methods was completed, we were able to look at three different problems that could be solved utilizing those methods. We were able to solve the Van der Pol Oscillator using a variety of the methods for solving initial value problems. When solving the Quantum Harmonic Oscillator, we utilized both the Shooting and Direct methods for solving boundary value problems. Lastly, we used both of our spectral methods to approximate solutions for the Reaction-Diffusion equations.

While the topics discussed just touched on the essentials needed to solve differential equations, additional resources exist for readers interested in exploring these methods in greater depth. Though some of the topics are quite "new" compared to classical finite difference methods, they arise from well-established research in the field. By pursuing these resources, the reader can expect to gain invaluable insight into both the mathematical theory and practical application that can be achieved through scientific computing.

References

Bertsimas, D. & Tsitsiklis, J.N. (1997). *Introduction to Linear Optimization (6th Edition).* Athena Scientific.

D'Ambra, P. & Durastante, F. & Filippone, S. (2025). *Parallel matching-based AMG preconditioners for elliptic equations discretized by IgA [Preprint].* arXiv. https://arxiv.org/pdf/2511.21268

Gustafsson, B. (2018). *Scientific Computing: A Historical Perspective.* Springer. https://doi.org/10.1007/978-3-319-69847-2.

Heath, M.T. (2018). *Scientific Computing: An Introductory Survey (2nd Edition).* SIAM. https://epubs.siam.org/doi/book/10.1137/1.9781611975581.

Kincaid, D.R. & Cheney, E.W. (2009). *Numerical Analysis: Mathematics of Scientific Computing (3rd Edition).* American Mathematical Society.

Li, R. & Di, H. & Jiang, S.W. (2025). *Two-Step Generalized RBF-Generated Finite Difference Method on Manifolds [Preprint].* arXiv. https://arxiv.org/pdf/2511.18049