PHYS 581: Assignment #5

Scott Salmon UCID: 30093320

April 11th, 2025

1 - Assignment Details

Unlike previous assignments, this one did not consist of a series of questions, but instead involved solving a single, comprehensive problem. We were tasked with modeling a wave propagating inside a box of length ℓ, where $-\ell/2 \leq x \leq \ell/2$.

We had the choice of modeling either the one-dimensional Schrödinger wave equation or the classical one-dimensional wave equation. The primary boundary condition required that the wave amplitude vanish at the walls, i.e., $f(x = \pm\ell/2, t) = 0$ for all times. This defines a finite domain in which the wave reflects off the walls.

Time propagation was to be handled using the RK45 method (Runge–Kutta–Fehlberg), while spatial discretization was to be implemented using two methods of our own choosing from the following list:

- Finite differencing (beyond lowest order)
- Lagrange interpolating polynomials
- Gaussian quadrature
- Fourier series
- Discrete variable representation
- Finite-element method

Finally, we were required to generate snapshots of the wave at different points in time and visualize the results as an animation. These animations would then serve as the basis for our discussion of the simulation outcomes.

For my implementation, I chose to model the classical one-dimensional wave equation, as it appeared more straightforward than the Schrödinger equation. For the spatial meshing, I selected two methods: fourth-order finite differencing and the Fourier series method. I chose finite differencing because it was the technique we spent the most time developing in class and served as a reliable default. I chose the Fourier series method because it was new to me and I wanted to challenge myself a bit.

2 - Implementation Information

For my submission, I have included two C source files, a Python script for generating the animations, the resulting animation files, and a Jupyter Notebook that documents the development process. Each C file implements the same wave propagation framework, with the only difference being the spatial meshing method used. The Python script was used to convert the simulation output into animations for visual analysis. The Jupyter Notebook served primarily as a workspace to write, test, and run both the C code and animation scripts in a unified environment.

I will now outline how the code works in each file.

2.1 - `finitediff.c`

In *finitediff.c*, the program begins with a list of necessary imports and the definition of several constants. The code is designed to run on my Windows 10 laptop; however, it may require minor modifications on Linux or macOS systems. In particular, some of the directory-creation commands and system-specific headers (e.g., `<direct.h>` for Windows) may differ across operating systems. The box length is set to $\ell = 1.0$, and the grid contains $N_x = 100$ spatial points, including both boundaries.

The first function, *initialize_wave*, is responsible for setting the initial conditions of the simulation. It initializes the wave profile as a Gaussian centered within the domain and assigns zero initial velocity across all spatial points.

The second function, *compute_spatial_derivatives*, implements a fourth-order finite differencing method to approximate the second spatial derivative of the wave. This is based on the standard five-point stencil formula:

$$f''(x_i) \approx \frac{-f(x_{i-2}) + 16f(x_{i-1}) - 30f(x_i) + 16f(x_{i+1}) - f(x_{i+2})}{12\,\Delta x^2}$$

In addition, this function enforces Dirichlet boundary conditions by explicitly setting the second derivatives near the boundaries to zero. This is necessary because the fourth-order stencil cannot be applied at the endpoints of the domain, as doing so would require access to values outside the defined array. Since the boundary condition requires the wave to be zero at $x = \pm\ell/2$ regardless, this approach is both physically consistent and numerically stable.

The third function, *rk45_step*, is the main time evolution routine in the simulation. It performs a single time step of the Runge–Kutta–Fehlberg (RK45) integration method to update both the wave amplitude array f and its velocity array v. This function is called repeatedly throughout the simulation loop to advance the wave forward in time.

The RK45 method is an embedded Runge–Kutta scheme that uses six intermediate stages to compute a highly accurate update of the system. In this implementation, we compute both the displacement (f) and the velocity (v) at the next time step using information from the current state and its spatial second derivative, $f''(x)$, which is passed in as `d2f_dx2`.

Each intermediate stage generates temporary estimates (`kf1`, `kf2`, ..., `kf6` and `kv1`, `kv2`, ..., `kv6`) that approximate the rate of change of f and v at various points within the time step. These estimates are then weighted and combined to compute f_{next} and v_{next}, the updated wave and velocity arrays.

This function also enforces Dirichlet boundary conditions, but in a different way than *compute_spatial_derivatives*. Here, the boundary values of f_{next} and v_{next} are explicitly set to zero at the end of the time step. This ensures that, even if numerical errors or the RK integration introduce nonzero values at the endpoints, the wave amplitude remains zero at $x = \pm\ell/2$, as required by the physical problem.

In summary, *compute_spatial_derivatives* enforces boundary conditions by limiting stencil application near the boundaries and zeroing out derivative values, while *rk45_step* enforces the boundary conditions by resetting the physical wave and velocity arrays to zero at the edges after each update step.

The final specialized function in this program is *save_snapshot*, which handles all data output. At each specified time step, this function saves the current wave profile as a plain text file, storing the amplitude values across the spatial domain. These snapshots capture the state of the wave at discrete points in time and are later used to generate animations for visualization and analysis. The function also ensures that the appropriate output directory exists, creating it if necessary.

The *main* function serves as the entry point for the simulation and coordinates the entire execution process. It begins by declaring and initializing all necessary arrays for the spatial grid, the wave amplitude f, its velocity v, their updated versions, and the second spatial derivative array $f''(x)$.

The program then attempts to create a folder named `Finite_Diff_Snapshots` to store the output files. If the folder already exists, the creation is safely ignored; otherwise, it is created to prevent errors during file writing.

Next, the wave is initialized using a Gaussian profile with zero initial velocity via the *initialize_wave* function. The time evolution is carried out over 4000 discrete time steps, using a time step size of $\Delta t = 0.0005$. Within the main simulation loop, the following steps are performed at each iteration:

- Compute the second spatial derivative using the selected spatial meshing method (in this case, finite differencing).
- Advance the solution forward by one time step using the *rk45_step* function.
- Every 20 steps, save the current wave profile using *save_snapshot*.
- Update the wave and velocity arrays with the newly computed values.

This process continues until all 4000 time steps are completed, resulting in a time-resolved simulation of the wave's evolution within the confined domain.

2.2 - `fourier.c`

The second implementation of the simulation is contained in *fourier.c*, which solves the same wave equation using a spectral method instead of finite differencing. While the structure of the program is largely similar, the key difference lies in how the second spatial derivative is computed: this version uses a Fourier-based approach built around the Discrete Sine Transform (DST) to approximate derivatives. The box length remains $\ell = 1.0$, but this implementation uses $N_x = 64$ interior points — not including the boundaries — since the sine transform basis inherently assumes zero values at the endpoints.

The functions *rk45_step*, *save_snapshot*, and *initialize_wave* are implemented identically to those in the finite difference version and serve the same roles: initializing the wave, saving snapshots to disk, and performing time propagation using the RK45 method.

The primary distinction lies in the function *compute_d2f_dx2_sine*, which replaces the finite differencing scheme with a spectral derivative. The process begins by computing the discrete sine transform of the wave profile using a naive implementation of the DST-I algorithm (coded in *dst*). Each mode in the sine basis corresponds to a sine wave that automatically satisfies zero boundary conditions at $x = \pm\ell/2$, making this method ideal for enforcing Dirichlet conditions without needing to manually reset boundary values.

Once the function is transformed into sine space, the second derivative is computed analytically by multiplying each sine coefficient by the square of its associated frequency:

$$\frac{d^2}{dx^2}\sin\left(\frac{n\pi x}{\ell}\right) = -\left(\frac{n\pi}{\ell}\right)^2 \sin\left(\frac{n\pi x}{\ell}\right)$$

This multiplication is carried out in the array `d2f_hat`, where each coefficient is scaled by $-\left(\frac{\pi(k+1)}{\ell}\right)^2$. The resulting array is then inverse-transformed using *idst* to return to physical space, yielding the second derivative $f''(x)$ at each interior point.

The key advantage of this approach is that it respects the zero-boundary conditions at all times due to the orthogonality and vanishing properties of the sine basis. Unlike standard Fourier transforms (which assume periodicity and can wrap around), the sine transform ensures that all functions in the basis go to zero at the

domain edges. This is essential for our problem, where the wave is confined in a box with walls where the amplitude must always be zero.

The main function follows the same structure as in the finite difference version but uses the sine transform-based derivative method instead. It declares the necessary arrays, attempts to create an output directory (`SineFourier_Snapshots`), and initializes the wave. The time loop runs for 4000 steps using a time increment $\Delta t = 0.0005$. At each iteration, the second derivative is computed via the sine spectral method, and the RK45 routine advances the wave forward in time. Every 20 steps, a snapshot of the wave profile is saved to disk. The process repeats until all time steps are completed.

Because the DST method used here only considers interior points (i.e., it excludes the boundary values), there is no need to manually apply Dirichlet boundary conditions in this version of the code. The boundary behavior is inherently satisfied by the properties of the sine basis, which distinguishes this method from the finite difference approach, where zeroing out the ends explicitly is required.

Overall, this implementation offers a more mathematically elegant and spectrally accurate alternative to finite differencing, especially for problems with smooth initial conditions and strict boundary constraints. While it is computationally slower due to the naive DST implementation, it naturally enforces the boundary conditions without any artificial corrections and results in smooth, globally accurate second derivatives.

2.3 - `wave_animation.py`

To visualize the wave propagation in both implementations, I created a Python script named *wave_animation.py*. This script reads the simulation output files generated by each C program and creates animated GIFs showing the time evolution of the wave amplitude across the spatial domain. It uses the `matplotlib` library for plotting, along with `FuncAnimation` to generate the animations.

The script is designed to be flexible and reusable. It defines a function, `make_animation`, that takes three arguments: the path to the directory containing snapshot files, the desired name of the output GIF, and a title to display on the animation. This function:

- Locates and loads all snapshot text files using `glob` and `natsort` to ensure the frames are ordered by time step,
- Constructs a spatial grid using the number of entries in the first snapshot file and a fixed domain length $\ell = 1.0$,
- Loads all wave profiles into a list for animation,
- Sets up a plot and defines an `update` function to animate each frame,
- Saves the resulting animation as a GIF using `PillowWriter` at 20 frames per second.

Two calls to `make_animation` are made at the bottom of the script: one for the finite difference output (`Finite_Diff_Snapshots`) and one for the sine Fourier output (`SineFourier_Snapshots`). This produces two separate GIFs named `wave_animation_fd.gif` and `wave_animation_fourier.gif`, each showing the wave evolution for the corresponding method. A message is printed to confirm successful completion.

3 - Discussion

After running both simulations and generating animations, we can now evaluate the accuracy, behavior, and qualitative performance of each spatial meshing method. The animations show the time evolution of a wave initially centered in the box with a Gaussian profile and zero initial velocity. In both simulations, the wave evolves symmetrically, propagates toward the boundaries, reflects off the walls, and interferes with itself,

forming recognizable standing wave-like behavior over time.

Overall behavior: Both simulations exhibit physically realistic wave propagation and reflection. The wave reflects off the boundaries at $x = \pm\ell/2$ without visible loss of energy or artificial damping. The finite difference method produces slightly more jagged reflections near the boundaries, while the sine Fourier method results in a cleaner, more sinusoidal profile throughout the simulation. This is consistent with expectations: finite differencing introduces local truncation errors, while the Fourier approach models the solution globally using smooth basis functions. That said, both animations are smooth and visually convincing — to the human eye, the difference in numerical accuracy is subtle.

Boundary enforcement: One of the most important distinctions between the two methods is how boundary conditions are handled. In the finite difference implementation, the Dirichlet boundary condition $f(x = \pm\ell/2) = 0$ is enforced manually by explicitly zeroing the values at each time step. In the sine Fourier method, the basis functions themselves vanish at the endpoints, satisfying the boundary condition by construction. However, because we do not explicitly overwrite the boundaries in the Fourier version, and because we only compute the solution at interior points, the wave is not exactly zero at the edges — especially when it comes close to the boundary. This is a subtle but noticeable effect in the animation, and highlights the distinction between implicit and explicit enforcement of constraints.

Amplitude conservation: Neither method showed significant amplitude drift or blow-up over time. The wave remains bounded and oscillates consistently across the 4000 time steps. Although the classical wave equation does not require amplitude normalization in the same way the Schrödinger equation does, it is reassuring that both methods preserve stability across a long simulation.

Performance and efficiency: A major difference between the two methods is runtime performance. The finite differencing method runs nearly instantaneously — a fraction of a second — as expected for a compiled C program using only local operations. In contrast, the sine Fourier method took approximately 12 seconds to run in the Jupyter Notebook environment, according to the built-in timer. This is a surprisingly long time for compiled C code, and reflects the computational cost of repeatedly performing naive discrete sine transforms and inverse transforms at every time step. This highlights a practical tradeoff: spectral methods offer elegance and precision, but at a higher computational cost.

Ease of implementation: In terms of programming effort and conceptual understanding, the finite differencing method is significantly easier to implement. Its logic is straightforward, the stencil is intuitive, and boundary handling is direct. The sine Fourier method, while more elegant mathematically, requires a deeper understanding of basis functions, spectral transforms, and careful indexing. This made it more challenging to implement... Despite this, it was a rewarding experience to construct the spectral method from scratch.

Which method performs better? From a purely numerical perspective, the sine Fourier method offers smoother results and more precise boundary behavior. However, from a practical perspective, the finite differencing method is faster, easier to understand, and still produces high-quality output. The choice between the two depends on the problem: for exploratory simulations or rapid prototyping, finite differencing is ideal. For high-precision applications with strict boundary conditions, Fourier-based methods are worth the extra complexity.

4 - Conclusion

This assignment was a fantastic opportunity to explore two powerful numerical techniques for solving partial differential equations. While both the finite differencing and sine Fourier methods successfully simulated wave propagation in a confined domain, they highlighted the different strengths of local and global approaches. Seeing these mathematical concepts come to life through time-evolving animations was a genuinely rewarding experience.

What stood out most to me was the elegance of the sine-based Fourier method. The idea that a wave could be represented as a sum of sine functions, each of which inherently satisfies the boundary conditions, was deeply satisfying. Even though it was slower and more complex to implement, it gave me a new appreciation for spectral methods and their power in solving physics problems where precision near the boundaries is critical. At the same time, the finite differencing approach proved that simple, efficient tools can still produce beautiful and accurate results — especially when performance and clarity are key.

More broadly, this assignment — and the course as a whole — has shown me just how far we've come from early numerical methods like Euler's method. Each assignment introduced a new and more sophisticated tool: we began by deriving Lagrange interpolating polynomials and constructing integration rules; moved into Gaussian quadrature using interpolation and the Golub-Welsch algorithm; explored orthogonal polynomials like Hermite and Chebyshev; and wrapped up with dynamical simulations using finite differencing and Fourier series. Every task required building the method from the ground up, understanding its derivation, and evaluating its performance. That combination of mathematical rigor and computational creativity made the learning process both challenging and deeply rewarding.

I genuinely enjoyed this course. It was hands-on, intellectually engaging, and gave me the opportunity to build real, working simulations of complex systems. More than anything, it reinforced how powerful numerical methods are in the modern computational physics toolkit. I leave this class not just with a better understanding of the math, but with practical skills I can use in future projects, research, and beyond.

It was also really rewarding to apply techniques from PHYS 381 and PHYS 481 while creating the Python animation script for this assignment. I used `matplotlib.pyplot` to generate smooth, frame-by-frame visualizations of wave propagation, and then leveraged `FuncAnimation` and `PillowWriter` to dynamically update plots and export the results as GIFs. These are techniques I first learned years ago, and it's funny that I'm still using them today to solve new problems. In fact, I used the exact same tools to generate key figures and animations for my undergraduate honours thesis, including visualizations of spin correlations and lattice dynamics.

Being able to take raw simulation output from C and turn it into polished visuals in Python felt like a culmination of everything I've learned across the computational physics stream. Taken together, PHYS 381, 481, and 581 not only made me a better scientific programmer, but also helped me grow as a physicist — especially in terms of data analysis, presentation, and communicating results clearly and effectively.

Learning Mathematica in PHYS 581 was also a huge highlight. I genuinely wish I had been introduced to it earlier in my degree — it would have been incredibly useful in so many upper-year theory and lab courses. It's such a powerful tool for both symbolic and numerical work, and being able to use it fluently now feels like unlocking a whole new level of productivity. Better late than never, I guess.

Finally, I'd like to sincerely thank Dr. David Feder — not only for making this class so engaging and impactful, but also for supervising my research thesis project this year. His guidance has shaped both my understanding of computational physics and my confidence as an independent researcher. I am truly grateful to have had him as a mentor for this final chapter of my undergraduate degree. I don't know how I would have gotten through this year without his support. I also really appreciated the way this class was structured — the grading scheme was generous, but the content was still challenging. That balance is rare and refreshing. It made the course feel like it was actually about learning, not just about surviving. I wish more physics classes — and university classes in general — were like this.