Numerical Wave Equation Demonstration

Abel Flores Prieto
Partner: Sang Hun Chou
June 12, 2018

This project will work with any standard Raspberry Pi updated for Physics 129.

The Project and How It Works

Our project is supposed to be a type of demonstration/simulation of waves in both 1 and 2 dimensions, as solved from the wave equation in cartesian coordinates. All waves have the same boundary condition that at the end points the solution is 0. Running the main code 'wave_project.py' will start an interactive process where the user will be able to select simulations for waves in 1 and 2 dimensions. Once the user selects the wave he/she wants to see, a window pops up and starts the simulation. This simulation keeps track of the time of the solution telling the user at what time the simulation is currently in. The user can close this window whenever he/she wants to and start another one.

In order to solve the wave equation in an efficient manner, I wrote two classes so that the solution would be able to update instead of storing the solution for a given time range, which would use a lot of memory and would, obviously have limitations regarding how long the simulation would be able to run. The classes Wave1D and Wave2D implement a central differencing scheme to solve the wave equation. In order to obtain the solution at the following time step, one only needs the last two time step solutions using this scheme. For this reason, we only need to store three time step solutions at any given time, making it fast and as efficient as possible.

These classes have a method called 'iteration()' that yields the solution for the next time step 'dt' defined when initializing the class. On that note, the 'iteration()' method takes in an integer 'steps' to yield the answer at 'steps' times dt, which can speed up the time scale of the animation. However, this can make the simulation look disjointed and not as smooth compared to using steps of just dt.

This project makes use of the scientific libraries of NumPy and Matplotlib. It uses Matplotlib's animation library to be able to animate the solutions to the wave equation. To do so it uses the function FuncAnimation. The animation for the 1D wave equation is pretty straight forward, since FuncAnimation works best for 2 dimensional plots. The function takes in the figure to animate, a function that returns the line to plot and other parameters that feed that function. In our case, the wave classes yield the new solution by calling the method 'iteration()' which updates the animation. The 1D wave equation simulation is able to run fast in a Raspberry Pi because it only updates whatever has changed. On the other hand, the 2D wave equation simulation is a bit slower because it updates the whole plot every time.

Results

To run the code, go to a terminal and go to the directory where you unpacked the tar file. Then 'cd' into 'floresprieto_project' and run the command './wave_project.py' or 'python3 wave_project.py' and you are ready to go. Enjoy!

The code runs in the way that we expected it to. We debugged and refined it as much possible. The result is a fast numerical solver in both 1 and 2 dimensions, a reasonably fast animation for the 1D wave equation solution and a not-so fast, but fair animation for the 2D wave equation solution.

To fully understand the results, we must first understand what we are solving. Therefore, I am going to go over some theory in this section. The wave equation takes the form

$$\frac{\partial^2 u}{\partial t^2} = c^2 \nabla^2 u \tag{1}$$

where ∇^2 is just a double derivative w.r.t. x in 1D, and two double derivatives w.r.t. x and y in 2D. As mentioned in the last section, we use a central differencing scheme to solve the wave equation, which reduces 1 to

$$u_j^{i+1} = -u_j^{i-1} + 2(1-\alpha^2)u_j^i + \alpha^2(u_{j+1}^i + u_{j-1}^i) \tag{2}$$

where $\alpha = c\ dt/dx$, i refers to the time step, and j to the spacial step of x in 1D. Similarly, in 2D the wave equation reduces to,

$$u_{j,k}^{i+1} = -u_{j,k}^{i-1} + 2(1-2\alpha^2)u_{j,k}^i + \alpha^2(u_{j+1,k}^i + u_{j-1,k}^i + u_{j,k+1}^i + u_{j,k-1}^i) \tag{3}$$

where we use the same step Δ for x and y so that $\alpha = c\ dt/\Delta$, and k refers now to the spacial step in y. From equations 2 and 3, it is clear that the solution of u at any given time only depends on the last two solutions in time. Applying boundary conditions, and initial conditions we are able to get the first and second time solutions and solve u^{i+1} at any other time. In this way, for every time step dt the wave classes in our code update the solutions u^{i-1}, u^i and u^{i+1} in the following manner,

$$u^{i-1} = u^i \tag{4}$$

$$u^i = u^{i+1} \tag{5}$$

$$u^{i+1} = \text{eq. 2 or 3} \tag{6}$$

$$u|_{\text{boundary}} = 0 \tag{7}$$

Because of this, we are able to generate solutions to the wave equation in no time and for as long as we want to. This is very useful because we want to feed the FuncAnimation function a fast generator and because we don't want the user to be waiting around while the solutions are being generated. A really 'cool' aspect of solving the wave equation in this way is that the simulation will keep on going until the user clicks the exit button on the animation window.

Graphically, we can confirm our results are correct. For example, looking at Fig. 1 we can see that a square pulse initially at the center stars moving in both directions by splitting. The squares moving are not perfect straight lines, they are very small oscillations! The wave

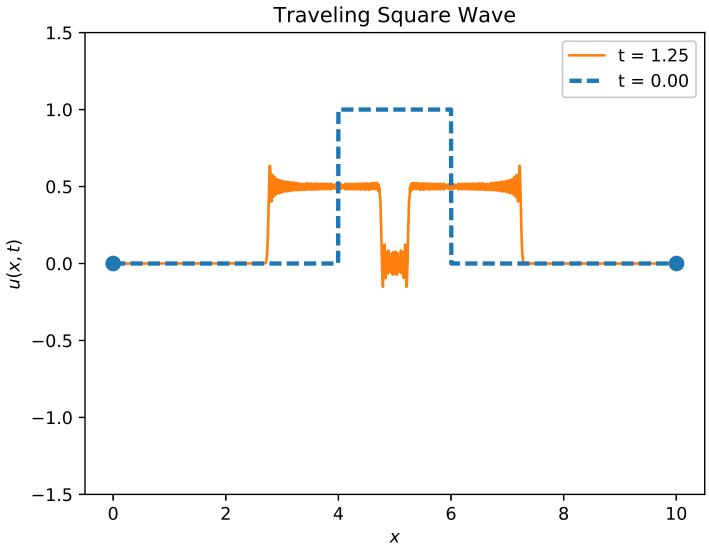

Figure 1: 1D wave equation solution to a square pulse centered on a string with bounds set to 0.

equation in 1D has an analytical solution and that is a Fourier summation, i.e., a sum of many sines and cosines, which is exactly what we are looking at here.

It is worth to note that we made sure our numerical solutions would not diverge by using von Neumann Stability Analysis and incorporating a warning if our parameters would create unstable solution.

Credits

Abel Flores Prieto (me): I was in charge of optimizing the numerical wave equation solution. I had started by solving the wave equation using a function that stored all information in a NumPy array. This way to solve the wave equation was very slow and would also give MemoryError's. This is one of the reasons I created the classes for the wave equation solution, so I would be able to discard past solutions. I tried to use inheretance for Wave2D from Wave1D but it was not needed. Sang and I worked togehter on making the interface.

Sang Hun Chou: Sang was in charge of making the animation work as fast as possible. He wrote the corresponding functions to use in FuncAnimation and to make a working code to animate the 3 dimensional plots. He was also in charge of creating functions to animate for the 3 dimensional plots. He also worked on the interface of the program.