APPM 3050
Spring 2014
Andrew Gordon
Evan Roncevich
5/2/14

The Wave Equation by Finite Difference

"The whole idea of The Wave is that the people in it have to support it."

1. *Introduction*

Schrodinger's wave equation describes the movement of energy, or a wave, moving through a medium. Using calculus we can describe waves of any speed and shape; however, if we wish to simulate these waves using a computer, teaching it calculus poses a problem. Therefore, we use the numerical technique of a finite difference method to approximate the shape of a wave. The finite difference method in one and two dimensions uses a combination of difference quotients to estimate the actual position at a point. We will demonstrate that technique in this paper and provide code in matlab that simulates both one and two dimensional waves.

2. *Waves in 1D: Describing basic motion*

In one dimension, the wave equation is described as:

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

And because we are essentially solving a partial differential equation using a finite difference technique, we require initial conditions. These will vary throughout this project: initially in the Wave1D the initial condition is modeled by a rise and fall on the left boundary based on the function sin(t) where $0 \leq t < \pi$. In addition to the initial condition, there is also a flexible right boundary, where the boundary is free-floating, meaning a wave will be able to "bounce" against it without inverting itself. The left boundary while $t \geq \pi$ becomes fixed at 0. All boundary conditions can be described as:

Initial displacement of zero: $u(x,0) = f(x)$

Initial speed of zero: $u_t(x,0) = 0$

Variable displacement: $u(0,t) = g(t)$

Flexible right boundary: $\alpha u(L,t) + \beta \frac{du(L,t)}{dx} = \gamma$ where $\alpha = 0, \beta = 1, \gamma = 0$

And finally, we rearrange the wave equation to form a constant lambda, which will be used to scale our finite difference:

$$\lambda = \frac{c^2 \Delta T^2}{\Delta X^2}$$

Because we need to solve for the next point in time, we can multiply the finite difference with lambda to find the solution. The Finite difference at a given point takes the values at that point along with the values of the positions next to the point, using a total of 3 points. These points, with the addition of the current and previous values in time will find the next point using the Finite Difference method.

3. *Waves in 1D: Plucking a wire (Fixed boundary condition)*

This portion simulates a wave that travels over a medium with fixed boundaries, and can be thought of as plucking a wire or string. This matlab code is derived from a basic finite difference for second derivatives. The first derivation will lead to our initial array of values for our finite difference:

$$\frac{u_i^{k+1} - 2u_i^k + u_i^{k-1}}{\Delta t^2} = \frac{c^2 u_{i-1}^k - 2u_i^k + u_{i+1}^k}{\Delta x^2}$$

And recalling that we can rearrange to form lambda, we find:

$$u_i^{k+1} = \lambda u_{i-1}^k + 2(1-\lambda)u_i^k + \lambda u_{i+1}^k - u_i^{k-1}$$

On our first iteration of the finite difference, we do not know the values of u_i^2. However, we can create a phantom value, and use this to create a forward difference by setting $u_i^0 = u_i^2$ and solving by dividing by 2. In this way, we can calculate u_i^2 as:

$$u_i^2 = (\lambda u_{i-1}^1 + 2(1-\lambda)u_i^1 + \lambda u_{i+1}^1)/2$$

Once we have this iteration, we can use the standard finite difference. Here they are in matlab code:

```
uCur(i) = (lamb*uInit(i-1) + (2-2*lamb)*uInit(i) + lamb*uInit(i+1))/2;

uFut(i) = (lamb*uCur(i-1) + (2-2*lamb)*uCur(i) + lamb*uCur(i+1) - uInit(i));
```

This code is used in a nested for-loop. The first loop represents entire time steps in terms of each frame; the second will iterate over the length of the wire for each time step and calculate the position at each point using the above code uFut(i). At each step, we will re-plot the position of the wire to form a movie. This technique will be used throughout the project.

4. *Waves in 1D: Moving left boundary*

Now that we have defined our most basic condition of fixed boundaries in 1D, we can move onto a different case: a moving left boundary. We can use any function we desire, but to best describe a wave we use a sin function, in this case $.3sin(2\pi * k * \Delta T)$ where k is the current frame of the movie. This will show a wave slowly rise across the left boundary, and we can adjust the amplitude and frequency to make different conditions. After $k * \Delta T > .5$ the left boundary stays fixed at 0 throughout the rest of the run-time. This function is called by `LeftBoundary.m`. This is a simulation of the real world situation in which, instead of plucking a taut string, we take a string fixed at one end and give it a strong wiggle; or, we are at a point some distance from shore and observe a wave traveling past us. We use the same equations from the last condition, with one difference: we call upon `LeftBoundary.m` to create the value of the first position in our array of position values:

```
uFut(1) = leftBoundary(i, deltaT)
```

Then we iterate from $2 : n$ to calculate our finite difference. This yields a wave moving smoothly across the graph.

5. *Waves in 1D: Flexible boundary on right*

Previously, we have used a fixed point of zero to describe the right boundary. The idea of the fixed point was that the wave was a string which vibrated after reaching the send. In this case, we will model the wave more realistically by doing away with an assumption and actually creating a boundary condition at the right side to reflect the waves coming from the left. This is invoked by `RightBoundary.m`. In this case we add to check to see if $x = L$, and if so, fill in this position with our boundary function based on position, gamma, alpha, and beta, with $\alpha = 0, \beta = 1, \gamma = 0$. In matlab code, with all previous conditions remaining the same, we add a check to our for-loop:

```
    if i == n
        if beta == 0
            uCur(n) = gamma/alpha;
        else
            phantom = rightBoundary(n, alpha, beta, gamma, deltaX, uCur);
            uCur(n) = ( lamb*uInit(n-1) + (2-2*lamb)*uInit(n)...
                                          + lamb*phantom ) / 2;
```

6. *Waves in 1D: Variable Depth*

For our most realistic simulation of a wave, we will now add the concept of variable depth. This is most similar to a wave crashing into a shoreline. The equation for standard finite difference is modified to yield:

$$\frac{1}{c^2}\frac{\partial^2 u}{\partial t^2} = \frac{\partial}{\partial x}\left(h(x)\frac{\partial u}{\partial x}\right) = \frac{dh}{dx}\frac{\partial u}{\partial x} + h(x)\frac{\partial^2 u}{\partial x^2}$$

This additional of a depth function $h(x)$ yields by chain rule the above function. The way this is implemented is create two one-dimensional vectors representing the depth and rate of change of the depth, which can be used to modify our finite difference at any point along the wave. As shown above, the actual depth $h(x)$ is multiplied by the second derivative of position, acceleration; also, the rate of change of the ground is related to the first derivative of position, velocity. Therefore, if depth is great, the wave travels faster. This is in fact how waves travel in real water. A wave in the middle of the ocean travels much faster than a wave approaching shore. Also, as the front of the wave approaches shore, that portion will decelerate while the portion behind it travels faster. When the back of the wave is traveling faster, it increases the amplitude of the wave. This is what causes waves to become taller as they approach shore and crash upon the beach. In our code, we modify the initial and current finite difference as such to access the depth $h(x)$ at a given point:

```
uCur(i) = (lamb*(hx(i)*deltaX*((uInit(i+1)-uInit(i-1))/2)...
            + h(i)*(uInit(i-1)-2*uInit(i)+uInit(i+1)))...
            +2*uInit(i))/2;

uFut(i) = lamb*( hx(i)*deltaX *((uCur(i+1)-uCur(i-1))/2)...
            +h(i)*(uCur(i-1)-2*uCur(i)+uCur(i+1)))...
            +2*uCur(i)-uInit(i);
```

As shown above, we input the finite difference that was the result of solving for u_i^{k+1}, which will be our `uFut(i)`. The video produced by running this code with all the above conditions is a wave that begins with an amplitude of 0.25 and increases in size until it crashes up on the far edge. The depth is described as $h = 1 - (x/6) * 0.8$, so $h(7.5) = 0$. At this point, the waves become chaotic if we run our simulation past it; however this does not really happen in real life because as a wave travels over a shoreline, there is still water traveling under it, and so it travels up the shore it eventually reaches 0 velocity and falls back.

7. *Waves in 2D: The Finite Difference*

For the wave equation to work in two spatial dimensions, we simply add the second derivative of motion with respect to the equation to yield:

$$\frac{1}{c^2}\frac{\partial^2 u}{\partial t^2} = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}$$

This leads to variables that in our code we refer to as `lambX` and `lambY`. By rearranging the equation we yield:

$$lambX = c(\frac{\partial t^2}{\partial x^2})$$
$$lambY = c(\frac{\partial t^2}{\partial y^2})$$

As for our finite difference, in addition to dealing with the difference quotient of u_i^k in the left-right direction, we must also add in the up-down direction of u_j^k, u_{j+1}^k, and u_{j-1}^k. As before, we need to add phantom points and then can work with the general formula of a normal time step. For our initial and general time steps, we yield:

$$u_{i,j}^2 = [\lambda_X u_{i-1,j}^1 - 2 * \lambda_X u_{i,j}^1 + \lambda_X u_{i+1,j}^1 + \lambda_Y u_{i,j-1}^1 - 2 * \lambda_Y u_{i,j}^1 + \lambda_Y u_{i,j+1}^1 + 2 * u_{i,j}^1]/2$$

and

$$u_{i,j}^{k+1} = [\lambda_X u_{i-1,j}^k + 2 * \lambda_X u_{i,j}^k + \lambda_X u_{i+1,j}^k + \lambda_Y u_{i,j-1}^k + 2 * \lambda_Y u_{i,j}^k + \lambda_Y u_{i,j+1}^k] + 2u_{i,j}^k - u_{i,j}^{k-1}$$

These will be incorporated into the existing code to create a grid of i, j values for our new wave simulation. In addition the formula uses the central difference to find the first derivative in relation to x and y. The finite difference formulas with height included are below:

$$u_{i,j}^2 = [\lambda_Y[hy(j,i)\tfrac{u_{i,j+1}^1 + u_{i,j-1}^1}{2}\Delta Y + h(i,j) * (u_{i,j+1}^1 - 2u_{i,j}^1 + u_{i,j-1}^1)] + \lambda_X[(h_x(i,j)\tfrac{u_{i-1,j}^1 + u_{i+1,j}^1}{2}\Delta X.....$$
$$...+h(i,j) * (u_{i-1,j}^1 - 2u_{i,j}^1 + u_{i+1,j}^1)] + 2u_{i,j}^1]/2$$

and

$$u_{i,j}^{k+1} = (\lambda_Y[h_y(i,j)\tfrac{u_{i,j+1}^k - u_{i,j-1}^k}{2}\Delta Y + h(i,j) * (u_{i,j+1}^k - 2u_{i,j}^k + u_{i,j-1}^k)] + \lambda_X[hx(i,j) * \tfrac{u_{i+1,j}^k - u_{i-1,j}^k}{2}\Delta X...$$
$$...+h(i,j) * (u_{i-1,j}^k - 2u_{i,j}^k + u_{i+1,j}^k)] + 2u_{i,j}^k - u_{i,j}^{k-1}$$

8. *Waves in 2D: Constant Depth*

Unlike `Tsunami1D.m`, we assume that there are no boundary conditions on the right side. Instead it is a fixed-point boundary. Therefore, at $x = n$ the value for all uFut(:,Lx) is 0, which makes waves vibrate and invert when hitting the end. This simulation shows a wave travelling smoothly across the graph and bouncing back after hitting the end. When the wave reaches its original position, it will continue off the graph, as if flowing back into the ocean. This is done by setting each value uFut(:,1)= uCur(:,2). This allows the wave to continue moving left without vibrating off a fixed point. Another modification was to make the graph continuous between the top and bottom of the graph, so that any waves flowing from the top will continue to the bottom. This presents a more realistic depiction of a wave hitting the shore with the ends hitting one another. The code has boundary conditions for when y=1 and y=Ly so the surrounding points can be grabbed without an index error. Below is the code for the general case of $u_{i,j}^{k+1}$:

```
uFut(j,i) = lambY*( uCur(j+1,i)-2*uCur(j,i)+uCur(j-1,i)) ...
            + lambX*(uCur(j,i-1)-2*uCur(j,i) + uCur(j,i+1))...
            + 2*uCur(j,i)-uInit(j,i);
```

In this, we can see a graph of (i, j) values plotted with a colormap to form a wave moving across the water. With constant depth, the wave moves at constant speed until it reaches the far boundary.

9. *Waves in 2D: Variable Depth*

For our final simulation, we will incorporate all previous iterations of the finite difference to create a final, 2-D plot with varying depth. This uses the formulation shown in the section Waves in 2D. Now as the depth decreases, the waves will gradually become taller and slower than those in deep waters. Below is the code for the general case of $u_{i,j}^{k+1}$ (see appendix for full code):

```
uFut(j,i) = lambY*( hy(j,i)*(uCur(j+1,i)-uCur(j-1,i))/2*deltaY+(uCur(j+1,i)...
            -2*uCur(j,i) + uCur(j-1,i))*h(j,i))+ lambX*(hx(j,i)*(uCur(j,i+1)...
            -uCur(j,i-1))/2*deltaX + (uCur(j,i-1)-2*uCur(j,i) + ...
            uCur(j,i+1))*h(j,i)) + 2*uCur(j,i)-uInit(j,i);
```

Also, $h(x)$ is now 2-dimensional, which necessitated changing `Depth1D.m`. We need to have a shape which incorporates both the x direction and y direction. The depth function we chose started at 1 and decreased with a slope of 2/15 as x increased, while the change in y was 0 throughout. This meant the waves would travel slower and taller as x increased, but would remain uniform throughout y. Along with that, two flat, shallow shelves are added diagonally to the graph. This was accomplished using the following matlab code:

```
Depth2D.m

function [h, hx, hy] = depth2D(x,y)

nx = length(x);
ny=length(y);
h = zeros(ny,nx);
hx =zeros(ny,nx);
hy= zeros(ny,nx);
for j = 1:ny
    for i= 1:nx
         if ((abs(x(i)-y(j))<.5)||...
         (abs(x(i)-2-y(j))<.5) ||...
         (abs(x(i)+2-y(j))<.5))&& (x(i)>1)
            h(j,i) =  .2;%1-x(i)*0.2;
            hx(j,i) = 0 ;
            hy(j,i) = 0;
        else
            %otherwise steadily decrease the depth by 2/15
            h(j,i)=1-x(i)*2/15;
            hx(j,i)= -2/15;
            hy(j,i)=0;
        end
    end
end
%---------------------------------
```

Similar to 1D, the wave increases in height as it traverses the graph because of the shallower depth. With the shallow shelves pointing diagonally, the waves will flow in the positive x-y direction because the waves travel slower at shallower depths, making moving around the shelves quicker. After the waves hit the x boundary, the waves will flow back to the beginning and go out to the ocean.

10. *Extra features*

We can simulate things other than waves given this code: imagine a rock being thrown into a pond. We can re-create this condition by selecting a point (i_{mid}, j_{mid}) in the middle of the grid and set it equal to some value β. For $(i_{mid}, j_{mid}) = \beta$, we can see that for a given value ripples will spread outward from this point. If β is large, the ripples are large. If β is huge, the ripples become huge and the graph becomes chaotic. Or, we can use the following code allows us to simulate drops of rain on the surface of water:

```
for k = 2 : nFrames
       if mod(k,40) == 0
           uCur(int8(150*rand+1),int8(rand*150+1))=.05*rand;
       end
```

```
%continue water simulation here
%note uCur in this case is 151X151
```

This code causes a random area in the grid to be a value $0 \leq \beta \leq .05$ to appear in the grid uCur. This gives the appearance of rain upon the surface of the pond. Increase the value of β and you have a rock-throwing simulation.

We also determined the ratio of ΔT to ΔX required to for the simulation to work without having numbers go to infinite. Lamba of both X and Y are formed by taking $c * \frac{\Delta T^2}{\Delta X^2}$. For 2 dimensional simulations, a point cannot displace itself more than those surrounding it. As noted by the formula under constant depth:

$$u_{i,j}^{k+1} = [\lambda_X u_{i-1,j}^k + 2 * \lambda_X u_{i,j}^k + \lambda_X u_{i+1,j}^k + \lambda_Y u_{i,j-1}^k + 2 * \lambda_Y u_{i,j}^k + \lambda_Y u_{i,j+1}^k] + 2u_{i,j}^k - u_{i,j}^{k-1}$$

if λ_X and λ_Y are greater than 2, $-2u_{ij}^k - 2u_{ij}^k + 2u_{ij}^k$ will make the absolute value greater than before, but waves do not work that way. Because we need lamba to be less than 2, $\Delta T/\Delta X$ must be less than $2^{1/2}$.

The last feature deals with the boundary conditions as noted in other sections. While the edge at Lx remains 0 during run-time, the upper and lower edges of y=0, and y=Ly are continuous. This means that the value of along that edge is determined by both y=0 and y=Ly. This make the graph continuous along that y, allowing the more accurately represent a wave hitting the shore. The edge where x=0 after the initial wave, allows waves to continue backwards, so a wave hitting the shore will bounce back and continue to the ocean.

11. *Conclusion*

Fluid Dynamics is complicated with complicated partial differential equations to solve; this simulation is a relatively simple and fast way to represent the tides. For the simple case of an (i, j) grid each with a corresponding value representing the level of the water, a finite difference is adequate to represent a wave traveling across a perfectly level surface of water. However, as we increase the size of our simulation, we can quickly see that it becomes difficult to process and render. Compare this to an actual section of water on a lake or ocean, with very large depth and a varying surface, and we can easily realize that the gap between this simulation and reality is very large. Given that we had the processing power to render it all, we would have to at least add surface conditions, boundary conditions on all sides, and more. However, with this approach we can simulate basic waves and other conditions and gives an excellent framework to work with.