Simple Model of a Tsunami

Jagat Kafle, Yerik Singh

A project report for the course
PHY2410 - Oscillations, Waves, and Optics

Department of Physics
Ashoka University

May, 2022

Abstract

In this paper, we discuss and analyse a simple Tsunami model. The Finite Difference Method is used to solve the two dimensional wave equation with constant velocity. Adapting the same method, the wave equation is then solved for a variable velocity, which is the Tsunami wave model we study. The displacement of the wave is plotted against the surface at different time instants and a physical interpretation of the results is done. Further, the amplitude of the wave at one point of the grid (constant x and y index) is plotted against time index to further analyze the nature of the wave model. The paper then goes on to discuss the further scope of this research and possible outcomes of more appropriate models.

1 Introduction

Tsunamis are large destructive waves with displacement of large volume of water generated by earthquakes, slides, volcanoes, human activities, etc on the ocean bed. These waves have longer wavelength and can propagate over large distances. When these waves travel towards the shore, the amplitude keeps on increasing, thus causing a huge deal of havoc and razing everything in its path. Hence, understanding Tsunamis and its behaviour is critical in identifying vulnerable coastal areas and assist warning systems. When there is a disturbance, a Tsunami is created and the proposed models tracks the movement of the water wave and predicts it position in time, and this information can be vital in studying the characteristic of Tsunamis and saving lives and minimizing damages.

In this paper, we'll go into the numerical modeling of Tsunamis. First, we'll use the Finite Difference Method (FDM) to solve the two dimensional wave equation with uniform velocity. Then, we'll see the solution for the wave equation with varying velocity. Based on the evolution of the system with time, we interpret the results that can signify real world phenomenon.

2 Model

2.1 The Wave equations

This section briefly discusses the one dimensional and two dimensional wave equations, wherein we add complexities to finally give the equation of the Tsunami model we're trying to solve. The one dimensional wave equation is,

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

Here, c is the uniform velocity of the wave. $u(x, t)$ is the displacement of the one dimensional object (say string) from the equilibrium position. t represents the time. This equation approximates the waves in strings (like guitar) assuming

uniform tension, velocity.

Now, consider we have infinite such strings laid next to each other lengthwise. We'll get a continuum which can be idealised into a two-dimensional membrane. If $u(x, y, t)$ is the vertical deflection of the membrane from the equilibrium position, then the two dimensional wave equation is,

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

Here again, c is the uniform velocity with which the wave propagates. Under the assumptions of uniform velocity, uniform membrane density, we can approximate the water waves using this equation.

Now, while we can do the variable velocity in the $1D$ wave equation, we will generalize it by considering the 2D wave equation. In the case of variable velocity, say $\sqrt{\lambda(x, y, t)}$, we can write the wave equation as,

$$\frac{\partial^2 u}{\partial t^2} = \frac{\partial}{\partial x}\left(\lambda(x, y, t)\frac{\partial u}{\partial x}\right) + \frac{\partial}{\partial y}\left(\lambda(x, y, t)\frac{\partial u}{\partial y}\right) - \frac{\partial^2 \lambda}{\partial t^2} \tag{3}$$

This equation looks a little complicated, as the velocity varies on all three variable x, y, and t. We'll simplify this by including the dependence of the velocity only on the x variable, so the wave equation now becomes,

$$\frac{\partial^2 u}{\partial t^2} = \frac{\partial}{\partial x}\left(\lambda(x)\frac{\partial u}{\partial x}\right) + \frac{\partial}{\partial y}\left(\lambda(x)\frac{\partial u}{\partial y}\right) \tag{4}$$

Here again, $\sqrt{\lambda(x)}$ is the variable wave velocity. We take $\lambda(x) = gH(x)$, where $H(x)$ is the still water depth. This comes from the dispersion relation of velocity and height for shallow water, which is $v = \sqrt{gH}$.

Solving equations 1 and 2 analytically – using D'Alembert's method or separation of variables – gives the function u, which can be modeled to see how the wave

evolves. However, what if there are more complexities in the model – like non-uniform velocity in equations 3 and 4. In such cases, solving the equation analytically can be a tedious task. We then turn into numerical solutions, which approximate the function u on different order of errors based on the algorithm and methods we use. In this project, we implement the Finite Difference Method (FDM) with central difference scheme to approximate the solution of the wave equation, which is discussed in the next section.

3 Methods

3.1 The Finite Difference Method

For a function $u(x)$, we can approximate the derivative between two points as,

$$u'(x) \approx \frac{u(x + \delta x) - u(x)}{\delta x} \tag{5}$$

Here, δx is the time-step we take when dividing the curve into finite points. For this approximation of derivative, we assume $|\delta x| \ll 1$. If $\delta x \longrightarrow 0$, this limit gives the derivative at x.

$$u'(x) = \lim_{\delta x \to 0} \frac{u(x + \delta x) - u(x)}{\delta x} \tag{6}$$

Now, assume that $u(x)$ is at least twice continuously differentiable. Then, the first order Taylor expansion is,

$$u(x + \delta x) = u(x) + \delta x u'(x) + \frac{1}{2} u''(\epsilon) \delta x^2 \tag{7}$$

Here, we have used the Cauchy form for the remainder term, where ϵ is a point that lies between x and $x + \delta x$. From equations 5 and 7, we can see the error in the finite difference formula is,

$$\frac{u(x + h) - u(x)}{\delta x} - u'(x) = \frac{1}{2} u''(\epsilon) \delta x \tag{8}$$

As the error is proportional to δx, the finite difference quotient in equation 8 is a first order approximation. We'll now approximate $u''(x)$ by sampling u at the particular points $x - \delta x$, x, and $x + \delta x$. The Taylor expansions can be written,

$$u(x + \delta x) = u(x) + u'(x)\delta x + u''(x)\frac{\delta x^2}{2} + u'''(x)\frac{\delta x^3}{6} + \mathcal{O}(\delta x^4) \tag{9}$$

$$u(x - \delta x) = u(x) - u'(x)\delta x + u''(x)\frac{\delta x^2}{2} - u'''(x)\frac{\delta x^3}{6} + \mathcal{O}(\delta x^4) \tag{10}$$

Adding equations 9 and 10 gives,

$$u(x + \delta x) + u(x - \delta x) = 2u(x) + \delta x^2 u''(x) + \mathcal{O}(\delta x^4) \tag{11}$$

Hence, from equation 11, we can write the centered finite difference approximation for $u''(x)$ as,

$$u''(x) = \frac{u(x + \delta x) - 2u(x) + u(x - \delta x)}{\delta x^2} + \mathcal{O}(\delta x^2) \tag{12}$$

The error in this approximation is proportional to δx^2.

3.2 FDM Wave Equations

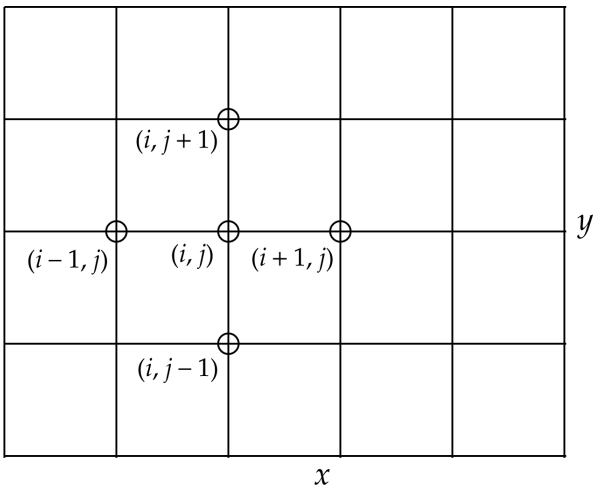


Figure 1: The x-y grid

We assume that the water surface lies in a rectangular region. The region is $R = [0, a] \times [0, b]$. The boundary conditions for the problem are defined on the

edges of this rectangle. We create a rectangular grid (x, y) with $x_i = i\delta x$ and $y_j = j\delta y$. The time is discretised as $t_l = l\delta t$. We thus have the spatial function $f(x, y) = f(i\delta x, j\delta y)$.

We have the displacement function $u(x, y, t)$. So, using equation 12, we can now approximate the t derivative as,

$$\frac{\partial^2 u}{\partial t^2} \approx \frac{u_{i,j}^{l+1} - 2u_{i,j}^{l} + u_{i,j}^{l-1}}{\delta t^2} \tag{13}$$

Similarly, the x and y derivatives are,

$$\frac{\partial^2 u}{\partial x^2} \approx \frac{u_{i+1,j}^{l} - 2u_{i,j}^{l} + u_{i-1,j}^{l}}{\delta x^2} \quad \text{and} \quad \frac{\partial^2 u}{\partial y^2} \approx \frac{u_{i,j+1}^{l} - 2u_{i,j}^{l} + u_{i,j-1}^{l}}{\delta y^2} \tag{14}$$

Hence, using equations 13 and 14, we can write the FDM scheme of the 2D wave equation as,

$$\frac{1}{c^2}\frac{u_{i,j}^{l+1} - 2u_{i,j}^{l} + u_{i,j}^{l-1}}{\delta t^2} = \frac{u_{i+1,j}^{l} - 2u_{i,j}^{l} + u_{i-1,j}^{l}}{\delta x^2} + \frac{u_{i,j+1}^{l} - 2u_{i,j}^{l} + u_{i,j-1}^{l}}{\delta y^2} \tag{15}$$

We apply some simplifications to the approximation. Take $\delta y = \delta x$ and $C = c\frac{\delta t}{\delta x}$, then, at time $l + 1$, we can approximate u as,

$$u_{i,j}^{l+1} = 2u_{i,j}^{l} - u_{i,j}^{l-1} + C^2\left(u_{i+1,j}^{l} + u_{i-1,j}^{l} + u_{i,j+1}^{l} + u_{i,j-1}^{l} - 4u_{i,j}^{l}\right) \tag{16}$$

For the model to be numerically stable, the condition is,

$$\delta t \leq \frac{1}{c}\left(\frac{1}{\delta x^2} + \frac{1}{\delta y^2}\right)^{-1/2} \implies c\delta t \leq \frac{\delta x}{\sqrt{2}} \implies \sqrt{2}C \leq 1$$

The above is the convergence condition by Courant-Friedrichs-Lewy (CFL condition). C is dimensionless and called the Courant number.

For the wave equation with variable velocity, given by equation 4, we can

use a two-step discretisation process. The outer operator can be discretised as,

$$\frac{\partial}{\partial x}\left(\lambda(x)\frac{\partial u}{\partial x}\right) \approx \frac{1}{\delta x}\left(\left(\lambda\frac{\partial u}{\partial x}\right)\Bigg|_{x=x_{i+1/2}} - \left(\lambda\frac{\partial u}{\partial x}\right)\Bigg|_{x=x_{i-1/2}}\right)$$

And, the inner operator can be discretised as,

$$\left(\lambda\frac{\partial u}{\partial x}\right)\Bigg|_{x=x_{i+1/2}} \approx \lambda_{i+1/2}\frac{u_{i+1,j} - u_{i,j}}{\delta x}$$

Hence, the overall two-step discretization can be written as,

$$\frac{\partial}{\partial x}\left(\lambda(x)\frac{\partial u}{\partial x}\right) \approx \frac{\lambda_{i+1/2}\left(u_{i+1,j} - u_{i,j}\right) - \lambda_{i-1/2}\left(u_{i,j} - u_{i-1,j}\right)}{\delta x^2}$$

We see that $\lambda_{i+1/2}$ and $\lambda_{i-1/2}$ are points defined on mid-points, which we haven't defined in our grid. So, we use the grid points to approximate these points. To do that we can either use Arithmetic mean, Harmonic mean or Geometric mean. In our case, as the height is a linear function, we suppose the arithmetic mean best fits the model. So, these points are defined as,

$$\lambda_{i+1/2} = \frac{1}{2}\left(\lambda_{i+1} + \lambda_i\right) \quad \text{and} \quad \lambda_{i-1/2} = \frac{1}{2}\left(\lambda_{i-1} + \lambda_i\right)$$

Hence, for the wave equation with variable velocity, we can write the FDM wave equation as,

$$\begin{aligned}\frac{u_{i,j}^{l+1} - 2u_{i,j}^{l} + u_{i,j}^{l-1}}{\delta t^2} = &\frac{\lambda_{i+1/2}\left(u_{i+1,j} - u_{i,j}\right) - \lambda_{i-1/2}\left(u_{i,j} - u_{i-1,j}\right)}{\delta x^2} \\ &+ \lambda_i\left(\frac{u_{i,j+1}^{l} - 2u_{i,j}^{l} + u_{i,j-1}^{l}}{\delta y^2}\right)\end{aligned} \tag{17}$$

Thus, using equation 17, we can find the displacement function u at time $l + 1$.

3.3 Algorithm

To solve the wave equation, boundary conditions and initial conditions are crucial as they define how the system behaves with evolving time. In our model, we define the initial condition by giving a function dependent on x and y at time $t = 0$.

$$u(x, y, 0) = I(x, y) \longrightarrow u^0_{i,j} = I(x_i, y_i) \tag{18}$$

Similarly, while we can define the function at boundaries, we need the function at $t = -1$ to evaluate it at $t = 1$. We constraint the model with one more condition – the wave is travelling with some velocity. For that condition

$$u_t(x, y, 0) = v \implies \frac{u^1_{i,j} - u^{-1}_{i,j}}{\delta t} = v \implies u^1_{i,j} = u^{-1}_{i,j} + v\delta t \tag{19}$$

In the case when the velocity is zero, we will have $u^1_{i,j} = u^{-1}_{i,j}$.

For the boundary condition, we try to check for different kinds of boundaries, physically implying different nature of the system we are trying to study. In the case of a fixed boundary (all the points in the edges of the rectangular grid have zero displacement), we can write,

$$u(x, y, t) = 0$$

This holds $\forall\, x, y$ in the boundary points. This boundary condition is implemented for the wave model given by equation 16. For the Tsunami model, we fix the boundary at the shoreline (where $y = 0\ \forall x$) while keeping free ends on other edges of the rectangle.

We now write out the algorithm of numerically solving the wave equation using the Finite Difference Method (FDM).

1. Define the initial conditions
2. Set the array u of displacements at time $l + 1 \longrightarrow up$, $l \longrightarrow u$, $l - 1 \longrightarrow um$

3. Create a mask for inside points and boundary points
4. Set the boundary conditions
5. Update up using the centered FDM for the defined points in the mask
6. $um = u$ and $u = up$ for the next step

4 Results

4.1 The simple 2D wave equation

We model the 2D wave equation 16 using Python programming language, implementing the algorithm in the previous section. The plots are shown in the next page.

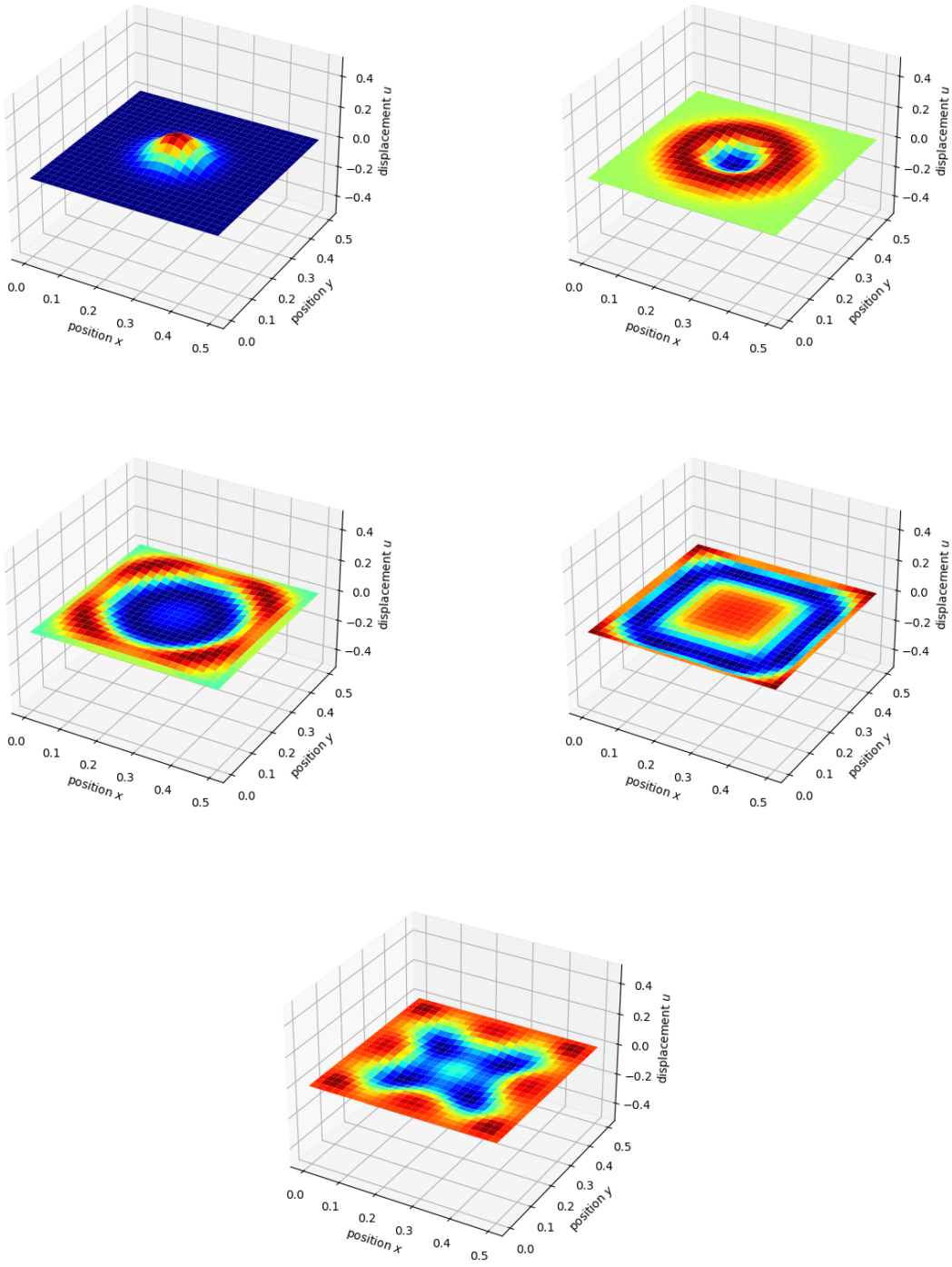
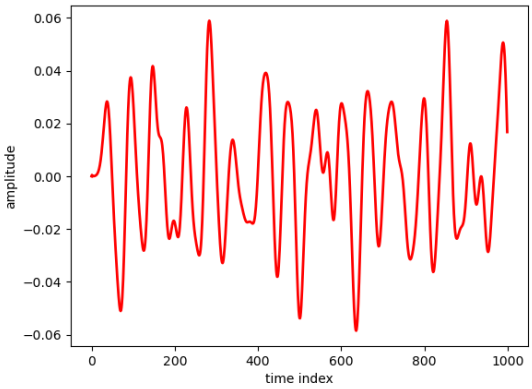
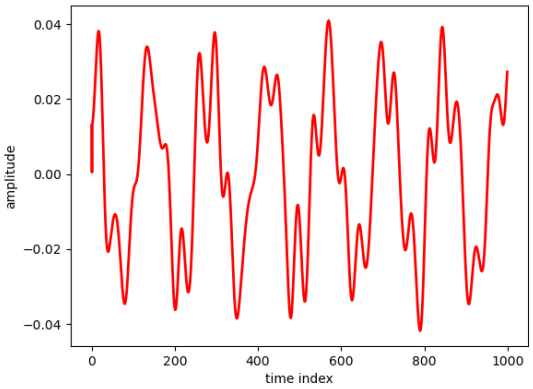


Figure 2: Snapshot of wave at time index 0, 20, 40, 60, 80, with time step $dt = 10^{-5}$ and spatial step $\delta x = \delta y = 0.01$

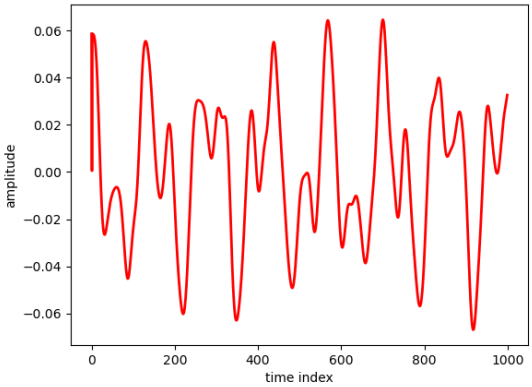
As seen from the 3D plot of the wave, we take a Gaussian initial condition, and then the wave propagates, reflects off along the fixed boundaries and the propagation continues. To understand the displacement of the wave, we look at the amplitude of the wave at particular x and y index with changing time.



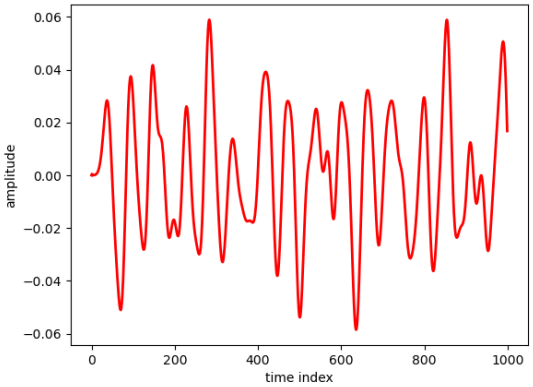
(a) Amplitude at the index $[x, y] = [10, 10]$



(b) Amplitude at the index $[x, y] = [20, 15]$



(c) Amplitude at the index $[x, y] = [20, 30]$



(d) Amplitude at the index $[x, y] = [40, 40]$

Figure 3: The amplitude plot with $\delta x = \delta y = 0.01$ for particular points (x, y).

We see that the amplitude of the wave at a particular point is changing throughout, as also seen in the 3D plot. Physically, as the wave travels and reflects off the boundaries, the amplitude keeps on increasing and decreasing. This wave model can simulate water waves with constant velocity, such as perturbation in the middle of a small rectangular pond. However, to simulate Tsunamis, as already discussed, we use the FDM Tsunami wave equation 17 with variable velocity.

4.2 The 2D Tsunami Model

We then model the 2D Tsunami wave equation 17 using the same algorithm. Here, however, the boundary conditions are changed, with the lines $x = 0$ and $y = 0$ being the lines of fixed boundary. The shoreline is given by the line $y = 0$.

The initial condition is again a Gaussian function, which we can physically attribute to a disturbance caused by an earthquake or volcano.

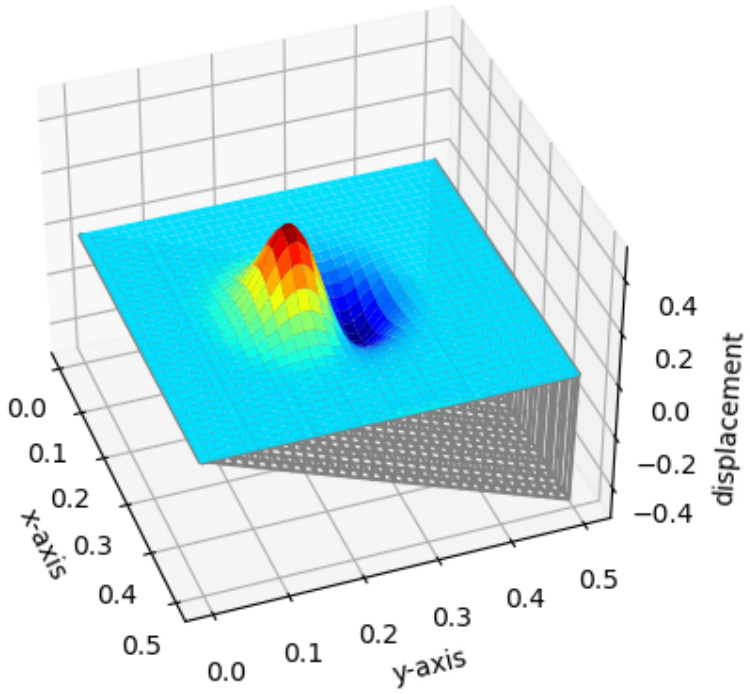


Figure 4: The snapshot of the wave with the wire-frame showing the depth of the sea, which is increasing as we move away from the shore. The shoreline is where the depth is at zero.

Now, we plot the wave at different time conditions and see how it is propagating. The wave-packet itself has some velocity, hence the initial Gaussian shape also travels with the velocity we have given. We can see from the plots (see Figure 5) that as the Gaussian moves towards the shoreline, the amplitude of the wave increases. This is the characteristic feature of a Tsunami wave, suggesting that the model represents some behaviour of Tsunamis. However, after the time index 250, the model starts to become numerically unstable and explodes after some time.

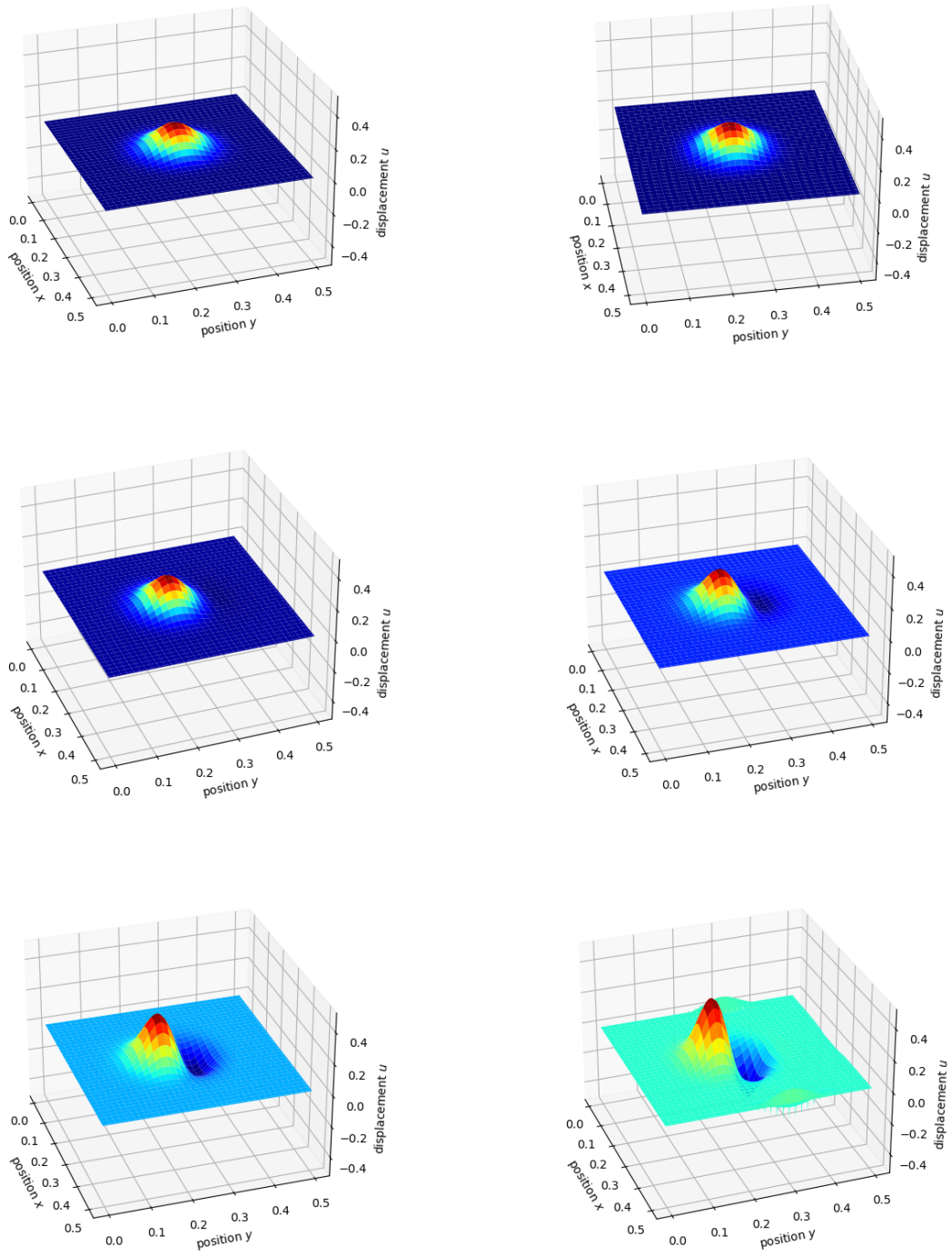
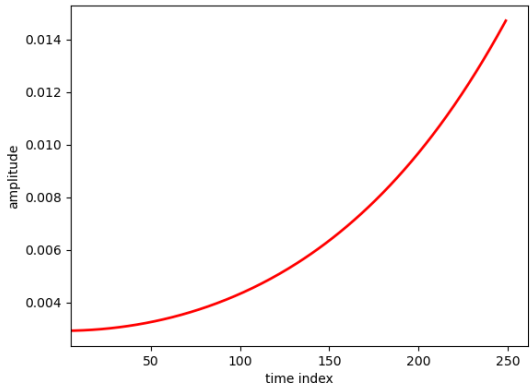
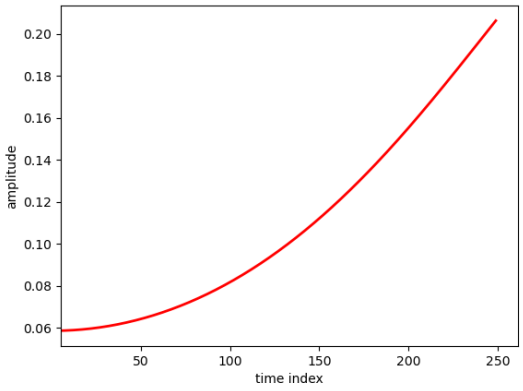


Figure 5: Snapshot of the Tsunami wave at time index 0, 50, 100, 150, 200, 250, with time step $dt = 10^{-4}$ and spatial step $\delta x = \delta y = 0.005$.

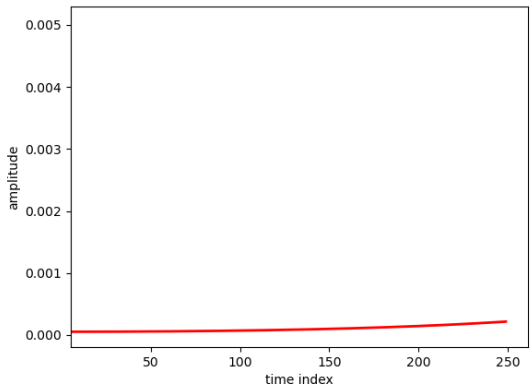
To understand more about the nature of the wave, we plot the amplitude versus time for particular points (x, y) in the water surface till the time index where it is numerically stable.



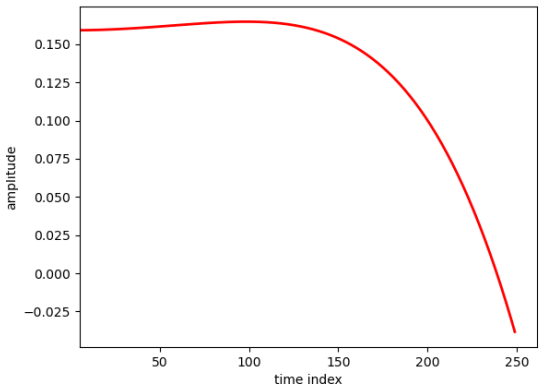
(a) Amplitude at the index $[x, y] = [30, 30]$



(b) Amplitude at the index $[x, y] = [40, 40]$



(c) Amplitude at the index $[x, y] = [50, 10]$



(d) Amplitude at the index $[x, y] = [50, 50]$

Figure 6: The amplitude plot with $\delta x = \delta y = 0.005$ for particular points (x, y).

From the amplitude graph, we can see that the amplitude at a particular point near the shore is increasing as we increase the time. However, as the wave propagates, the amplitude at points where it initially started off decreases, since the wave itself is moving towards the shore. This suggests that our model is physically feasible one for a Tsunami wave.

5 Discussion and Further research

While we obtain a model showing characteristic of a Tsunami wave, the model is far from being an accurate representation of the Tsunami. The first aspect the model falls short is that it becomes numerically unstable after sometime, characterised by the distortion in the wave. This suggests that

the parameters and method we have chosen to build the model have some nonphysical significance or numerical error. An area of improvement is to remove this numerical instability and see the complete behaviour of the wave as it evolves and reaches the shore. While we have scaled the size of the rectangular grid and velocity of the wave based on it, further improvements can be done by taking a large grid and velocities Tsunamis take physically. The boundary conditions can also be changed to see how the wave behaves under different physical boundaries. We can also vary the height function additionally on y and t, which allows to model the situations like underwater slide (moving bottom). The depth of the sea floor as a function of time is a crucial aspect of modelling a Tsunami and incorporating this into the model will greatly improve the accuracy and efficiency. We attempted solving the model with height as a function of time, but the FDM wave equation became much complicated to solve.

6 Conclusion

We have studied the nature of water waves using 2D wave equations with constant velocity and variable velocity. For the constant velocity model, we see that the wave reflects off the boundaries and propagates throughout – like ripples in a rectangular pool do. The wave with variable velocity has increasing amplitude as we move towards the shore. This model is supposed to imitate the Tsunami waves, which is successfully does to a certain reasonable extent.

7 Acknowledgements

We are grateful to our graduate teaching assistant for the course, Philip Cherian for his constant guidance throughout the project. We also thank Professor Garima Mishra and fellow students in the course Oscillations, Waves and Optics (Spring 2022) for their encouragement in learning.

References

[1] Morten Hjorth-Jensen (2009), *Computational Physics*, University of Oslo, 432 – 436.

[2] Peter J. Olver (2008), *Finite Difference Methods for Partial Differential Equations*, Numerical Analysis Lecture Notes, University of Minnesota.

[3] Knut–Andreas Lie (2005), *The Wave Equation in 1D and 2D*, Department of Informatics, University of Oslo.

[4] Oliver Beckstein (2020), *Computational Methods in Physics* Department of Physics, Arizona State University.

Appendix 1

The code which implements the Finite Difference Method used in building the models for this project can be accessed in the given Github link. Code link

The code contains the animation function which plays the animation of the evolution of the wave. The code is also attached with this document for reference.

Appendix - Code

May 15, 2022

Authors: Jagat Kafle, Yerik Singh

Course: Oscillations, Waves, and Optics (2022)

This jupyter notebook document is a merged document which solves two dimensional wave equation with constant velocity and variable velocity. The title is given before the start of each instance.

1 The 2D Wave Equation with constant velocity

[ ]:
```
# importing the required libraries
import matplotlib.pyplot as plt
import numpy as np
from mpl_toolkits.mplot3d import Axes3D
from matplotlib.animation import FuncAnimation
```

[ ]:
```
# L - Variable for size (in the x-axis)
# N - Number of grid box (in the x-axis)
# Dt - Time step
# Nt - End time
# aspect - ratio between the horizontal and the veritcal
# wave2D - solves two dimensional wave equation and returns u_t, X, Y, delta,␣
 ↪Dt, step

def wave2D(L=0.5, N=50, Dt=1e-5, Nt=1500, step=1, aspect=1):

    Lx = L # Length of the horizontal component of the grid
    Ly = aspect*L # Length of the vertical component of the grid
    Nx = N # Number of grid boxes in x
    Ny = int(aspect*N)  # Number of grid boxes in Y
    delta = L/N # the grid spacing step


    c = 500 # Speed of the wave

    C = c*Dt/delta # Courant number
    Csq = C**2 # The sq of the Courant number

    # the points in the meshgrid
```

```
    x = np.linspace(0, Lx, Nx+1)  # we need N+1 points for N spacing
    y = np.linspace(0, Ly, Ny+1)  # we need N+1 points for N spacing
    # note: we use u[i*delta, j*delta] = u(x, y)
    # (which is the transpose of the numpy row-column layout so we transpose X␣
↪and Y)
    X, Y = np.meshgrid(x, y)
    X, Y = X.T, Y.T

   # gaussian - set up the initial form of the wave
    def gaussian(x, um=0.05, x0=None, sigma=0.1*L):
        x0 = np.mean(x) if x0 is None else x0
        g = um/np.sqrt(2*np.pi*sigma**2) * np.exp(-(x-x0)**2 / (2*sigma**2))
        return g

    #def sin(x):
    #return np.sin(x)

    #Creates a two dimensional array for u_{-1}, u, u_{+1}
    um = np.zeros((Nx+1, Ny+1))
    u = np.zeros_like(um)
    up = np.zeros_like(um)
    #H = np.zeros_like(um)
    # Three dimensional array that stores, Time, X points, Y points
    u_t = np.zeros((int(np.ceil(Nt/step)), Nx+1, Ny+1))

    # Used to determine whether the point under consideration is on the␣
↪boundary or inside it.
    # If it is inside the bundary - True
    # If it is on the boundary - False
    inside = np.zeros_like(um, dtype= bool)
    inside[1:-1, 1:-1] = True
    boundary = np.logical_not(inside)

    # Establishing boundary conditions
    def set_boundary_conditions(u):
        u[boundary] = 0 # Anything that is on the boundary is equated to 0.

    # Boundaries of um, u, and up is taken as 0.
    set_boundary_conditions(um)
    set_boundary_conditions(u)
    set_boundary_conditions(up)

    # Initially there is no difference in the inside of um and u.
    # The shape of both of them is set up by the guassian in a symmetric manner.
    um[inside] = u[inside] = (gaussian(X)*gaussian(Y))[inside]
    #um[inside] = u[inside] = (np.sin(2*np.pi*X)*np.sin(2*np.pi*Y))[inside]
    #um[inside] = u[inside] = (X**2*Y)[inside]
```

```
    # initialising
    t_index = 0
    u_t[t_index, :, :] = um
    if step == 1:
        t_index += 1
        u_t[t_index, :, :] = u_t[t_index,:,:] + 50*Dt

#     def H(x,y):
#         return np.sin(x)*np.sin(y)

    # vectorized form to calculate the wave in next time step
    for nt in range(2, Nt):
        up[1:-1, 1:-1] = 2*(1-2*Csq)*u[1:-1, 1:-1] - um[1:-1, 1:-1] + Csq*(u[2:
 ↪, 1:-1]
                                                + u[:-2, 1:-1] + u[1:-1, 2:] + u[1:
 ↪-1, :-2])
        um[:, :], u[:, :] = u, up

        if nt % step == 0 or nt == Nt-1:
            t_index += 1
            u_t[t_index, :, :] = up
            print("Iteration {0:5d}".format(nt), end="\r")

    return u_t, X, Y, delta, Dt, step
```

[ ]:
```
# Running the solver function
u_t, X, Y, delta, Dt, step = wave2D(Nt=5000)
```

[ ]:
```
# 3d wireframe plot
def plot_wireframe(X, Y, u, limits=None, ax=None):
    if ax is None:
        fig = plt.figure()
        ax = fig.add_subplot(111, projection="3d")
    ax.plot_wireframe(X, Y, u, color='blue')
    ax.set_xlabel(r"position $x$ (m)")
    ax.set_ylabel(r"position $y$ (m)")
    ax.set_zlabel(r"displacement $u$ (m)")
    ax.set_zlim(limits)
    ax.figure.tight_layout()
    return ax
```

[ ]:
```
# 3d surface plot
def plot_surface(X, Y, u, limits=None, ax=None):
    if ax is None:
        fig = plt.figure()
        ax = fig.add_subplot(111, projection="3d")
```

```
    ax.plot_surface(X, Y, u, cmap='jet')
    ax.set_xlabel(r"position $x$")
    ax.set_ylabel(r"position $y$")
    ax.set_zlabel(r"displacement $u$")
    ax.set_zlim(limits)
    ax.figure.tight_layout()
    return ax
```

[ ]:
```
#plot the surface
for i in range(0, 100, 20):
    plot_surface(X, Y, u_t[i], limits=(-0.5,0.5))
```

[ ]:
```
# The following cell is used to animate the results obtained previously.
def animate_3d(array, save_animation=False, save_name='video.mp4',␣
 ↪save_dpi=300, save_fps=100):

    #Sets up plot shape
    fig = plt.figure()
    ax = fig.add_subplot(111,projection='3d')

   # Initialise the whole plot
    def init():
        ax.plot_wireframe(X,Y,array[0])
        ax.set_zlim(-0.25,0.25) # The limits on the vertical
        return [ax]


    def animate(frame):
        # Clears all the previous frames
        plt.cla()
        #Plot the surface of the wave
        ax.plot_surface(X,Y,array[frame], cmap='jet')
        ax.set_xlabel('x-axis')
        ax.set_ylabel('y-axis')
        ax.set_zlabel('z-axis')
        ax.set_zlim(-0.25,0.25)
        return [ax]
        #return graph

    # Most important line, this is what actually handles the animation by␣
 ↪calling the `animate' function with a frame number (integer)
    ani = FuncAnimation(fig, animate, init_func=init, blit=True, frames=1000,␣
 ↪interval=20, repeat=True)   # Code to create animations

    if(save_animation):
        ani.save(save_name, dpi=save_dpi, fps=save_fps)
```

```
    return ani
```

[ ]:
```
%matplotlib notebook
animate_3d(u_t)
```

[ ]:
```
def amplitude(u_t, a, b):
    u = []
    for i in range(0, 1000):
        u.append(u_t[i][a][b])
    return u

# for i in range(0, 1000):
#     u.append(u_t[i][50][20])
#plt.plot(amplitude(u_t, 10, 10), color='red', linewidth=2)
#plt.plot(amplitude(u_t, 20, 15), color='red', linewidth=2)
#plt.plot(amplitude(u_t, 20, 30), color='red', linewidth=2)
plt.plot(amplitude(u_t, 40, 40), color='red', linewidth=2)
plt.xlabel('time index')
plt.ylabel('amplitude')
plt.xlim(5, 1000)
```

2 The 2D Tsunami Model with variable velocity

[ ]:
```
#The variables and methods remain the same.

def wave2D(L=0.5, N=100, Dt=1e-4, Nt=1500, step=1, aspect=1):

    Lx = L
    Ly = aspect*L
    Nx = N
    Ny = int(aspect*N)
    delta = L/N

    c = 50

    C = c*Dt/delta
    Csq = C**2
    m = Dt/delta
    msq = m**2


    x = np.linspace(0, Lx, Nx+1)
    y = np.linspace(0, Ly, Ny+1)

    X, Y = np.meshgrid(x, y)
    X, Y = X.T, Y.T
```

```
    def gaussian(x, um=0.05, x0=None, sigma=0.1*L):
        x0 = np.mean(x) if x0 is None else x0
        g = um/np.sqrt(2*np.pi*sigma**2) * np.exp(-(x-x0)**2 / (2*sigma**2))
        return g

    h = np.zeros(Nx+1)
    g = np.zeros_like(h)
    for i in range(Nx+1):
        h[i:] = - x[i]
    g = 9.8*h

    #Creates a two dimensional array for u_{-1}, u, u_{+1}
    um = np.zeros((Nx+1, Ny+1))
    u = np.zeros_like(um)
    up = np.zeros_like(um)
    base = np.zeros_like(um)

    for i in range(Nx):
        for j in range(Ny):
            base[i, j] = h[i]


    u_t = np.zeros((int(np.ceil(Nt/step)), Nx+1, Ny+1))


    inside = np.zeros_like(um, dtype= bool)
    inside[0:-1, 0:-1] = True
    boundary = np.logical_not(inside)

    # Establishing boundary conditions
    def set_boundary_conditions(u):
        u[boundary] = 0
#     def set_boundary_conditions(u, um, up):
#         j = 0
#         while j < 51:
#             up[1, j] = um[i, j] + msq*((0.5*(g[2:] + g[1:-1])*(u[2:, 1:-1] -␣
 ↪u[1:-1, 1:-1]) - 0.5*(g[1:-1] + g[:-2])*(u[1:-1, 1:-1]-u[:-2, 1:-1]))+ g[1:
 ↪-1]*(u[1:-1,2:] - 2*u[1:-1, 1:-1] + u[1:-1, :-2]))
#             j = j + 1

    # Boundaries of um, u, and up is taken as 0.
    set_boundary_conditions(um)
    set_boundary_conditions(u)
    set_boundary_conditions(up)
```

```
    um[inside] = u[inside] = (gaussian(X)*gaussian(Y))[inside]
    #um[inside] = u[inside] = (np.sin(2*np.pi*X)*np.sin(2*np.pi*Y))[inside]
    #um[inside] = u[inside] = ((X)**2*(Y))[inside]
    #um[inside] = u[inside] = ((X)**2)[inside]

    # initialising
    t_index = 0
    u_t[t_index, :, :] = um
    if step == 1:
        t_index += 1
        u_t[t_index, :, :] = u_t[0,:,:] + 50*Dt


    for nt in range(2, Nt):
        up[i, j] = 2*u[i, j]

    #Almost the same function, with just an addition of g
    for nt in range(2, Nt):
        up[1:-1, 1:] = 2*u[1:-1,1:] - um[1:-1, 1:] + msq*((0.5*(g[1:] + g[0:
↪-1])*(u[2:, 1:] - u[1:-1, 1:]) - 0.5*(g[0:-1] + g[1:])*(u[1:-1, 1:]-u[:-2, 1:
↪]))+ g[1:]*(u[1:-1,1:] - 2*u[1:-1, 1:] + u[1:-1, :-1]))
        um[:, :], u[:, :] = u, up

        if nt % step == 0 or nt == Nt-1:
            t_index += 1
            u_t[t_index, :, :] = up
            print("Iteration {0:5d}".format(nt), end="\r")

    return u_t, X, Y, delta, Dt, step, h,x, y, base
```

[ ]:
```
#Running the solver function
u_t, X, Y, delta, Dt, step, h, x, y, base= wave2D(Nt=1000)
```

[ ]:
```
# 3d wireframe plot
def plot_wireframe(X, Y, u, limits=None, ax=None, color=None):
    if ax is None:
        fig = plt.figure()
        ax = fig.add_subplot(111, projection="3d")
    ax.plot_wireframe(X, Y, u, color='blue')
    ax.set_xlabel(r"position $x$ (m)")
    ax.set_ylabel(r"position $y$ (m)")
    ax.set_zlabel(r"displacement $u$ (m)")
    ax.set_zlim(limits)
    ax.figure.tight_layout()
    return ax
```

[ ]:
```
# 3d surface plot
def plot_surface(X, Y, u, limits=None, ax=None):
    if ax is None:
        fig = plt.figure()
        ax = fig.add_subplot(111, projection="3d")
    ax.plot_surface(X, Y, u, cmap='jet')
    ax.set_xlabel(r"position $x$")
    ax.set_ylabel(r"position $y$")
    ax.set_zlabel(r"displacement $u$")
    ax.set_zlim(limits)
    ax.figure.tight_layout()
    return ax
```

[ ]:
```
# # plot the water surface
for i in range(0, 300, 50):
    plot_surface(X, Y, u_t[i], limits=(-0.5,0.5))
```

[ ]:
```
# Running the animate function again, as done earlier.
def animate_3d(array, save_animation=False, save_name='video.mp4',␣
 ↪save_dpi=300, save_fps=100):

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

    #Initialising the Plot
    def init():
        ax.plot_wireframe(X,Y,array[0])
        ax.set_zlim(-0.5,0.5)
        return [ax]


    def animate(frame):
        plt.cla() # Clearing the previous frames
        #ax.plot_wireframe(X,Y,array[frame])
        ax.plot_wireframe(Y,X,base, color='grey')
        ax.plot_surface(X,Y,array[frame], cmap='jet')
        #ax.plot_wireframe(Y,X,base, color='grey')
        ax.set_xlabel('x-axis')
        ax.set_ylabel('y-axis')
        ax.set_zlabel('displacement')
        ax.set_zlim(-0.5,0.5)
        return [ax]
        #return graph

    # Most important line, this is what actually handles the animation by␣
 ↪calling the `animate' function with a frame number (integer)
```

```
    ani = FuncAnimation(fig, animate, init_func=init, blit=True, frames=1000,␣
 ↪interval=20, repeat=True)    # Code to create animations

    if(save_animation):
        ani.save(save_name, dpi=save_dpi, fps=save_fps)

    return ani
```

[ ]:
```
%matplotlib notebook
animate_3d(u_t)
```

[ ]:
```
def amplitude(u_t, a, b):
    u = []
    for i in range(0, 250):
        u.append(u_t[i][a][b])
    return u

# for i in range(0, 1000):
#     u.append(u_t[i][50][20])
#plt.plot(amplitude(u_t, 10, 10), color='red', linewidth=2)
#plt.plot(amplitude(u_t, 20, 20), color='red', linewidth=2)
#plt.plot(amplitude(u_t, 30, 30), color='red', linewidth=2)
#plt.plot(amplitude(u_t, 40, 40), color='red', linewidth=2)
#plt.plot(amplitude(u_t, 50, 50), color='red', linewidth=2)
plt.plot(amplitude(u_t, 50, 10), color='red', linewidth=2)
#plt.plot(amplitude(u_t, 20, 15), color='red', linewidth=2)
plt.xlabel('time index')
plt.ylabel('amplitude')
plt.xlim(5,)
```