2D Heat and Linear Wave Equations simulation using PyCUDA

Daniele Fazzari - Unina & UGR

1 Conservation Laws

A **conservation law**, or in other words a law that describes that a quantity is conserved (a quantity like mass, energy, linear and angular momentum etc...), can be mahematically formulated as ***partial differential equations*** (PDEs).

Many natural phenomens can be partially described by mathematical expressions, such as consevation laws. An example of phenomena that can be mathematically described by a conservation law are of course: *Heat propagation* and *waves* (like sound wave, pressure wave, etc).

To solve PDEs (and ODEs) numerically on a computer, we discretize it, replacing the continuous derivatives with **discrete derivatives** (calculated on a discrete grid). So, given initial conditions, the equation can be simulated.

1.0.1 Example of discretization of 1D heat equation

In general the heat equation states that the rate of change in temperature over time is equal to the second derivative of the temperature with respect to space multiplied by the ***heat diffusion coefficient***. The following is the PDE that describes the heat propagation in a 1D object:

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

in wich "**u**" is the temperature, "**k**" is the heat diffusion coefficient, "**t**" is time and **x** is space. We can now descritize it by replacing the continuous derivatives with discrete approximations:

$$\frac{1}{\Delta t}(u_i^n - u_i^{n-1}) = \kappa \left[\frac{1}{\Delta x^2}(u_{i-1}^n - 2u_i^n + u_{i+1}^n) \right]$$

Now we can create an **explicit numerical scheme** (instead of an **implicit scheme**, usually slow and memory hungry on the GPU) by replacing the backward difference, used to discretize the time derivative, with a forward difference (possible if the disturbances travel at a finite speed). Note that with explicit scheme we must have much smaller time-steps, but this scheme is perfectly

suited for executing on the GPU (each grid cell status at time-step n+1 can be computed indipendently of all other grid cells)

$$\frac{1}{\Delta t}(u_i^{n+1} - u_i^n) = \kappa \left[\frac{1}{\Delta x^2}(u_{i-1}^n - 2u_i^n + u_{i+1}^n) \right]$$

$$u_i^{n+1} = \kappa \left[\frac{\Delta t}{\Delta x^2}(u_{i-1}^n - 2u_i^n + u_{i+1}^n) \right] + u_i^n$$

This discrete approximations needs to be calculated on a grid of finite discrete points in space and time. the discretization gives us one equation per grid point wich has to be solved.

However there is a restriction on how large the timestep can be, and is called the Courant-Friedrichs-Levy condition, that for the heat equation is:

$$\frac{1}{2} < \frac{\kappa \Delta t}{\Delta x^2},$$

2 Languages and Libraries Used

Has been used **Python**, an high-level general-purpose programming language. Using Python, we can implement an *explicit simulator* in very few lines of code.

- **Pycuda**: PyCUDA lets you access Nvidia's CUDA parallel computation API from Python language (the one used for these homeworks). It can be better then other CUDA API wrappers for several reasons such as: Automatic Error Checking (automatically translated in python exceptions), execution speed (pycuda's base layer is coded in C), helpful documentation, completeness (access to full power of CUDA API).
- **numpy**: offers comprehensive mathematical functions, random number generators, linear algebra routines. It also offers fast and versitile vectorization (and indexing) for easy array computation.
- **matplotlib**: used in these homeworks to plot the results of the computational simulation of the equations, infact it is a comprehensive library for creating static, animated, and interactive visualizations in Python.
- **math**: This module provides access to the mathematical functions defined by the C standard (like for example the "ceil" function).

3 2D Heat Equation

The **heat equation in 2D** can be written as follows (where u is the temperature, and κ is the material specific heat diffusion constant):

$$\frac{\partial u}{\partial t} = \kappa \nabla^2 u = \kappa \left[\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \right]$$

By approximating the temporal derivative with a forward difference, the spatial derivative with a central difference, and gathering $u^n + 1$ on the left hand side and u^n on the right, we obtain:

$$\frac{1}{\Delta t}(u_{i,j}^{n+1} - u_{i,j}^n) = \kappa \left[\frac{1}{\Delta x^2}(u_{i-1,j}^n - 2u_{i,j}^n + u_{i+1,j}^n) + \frac{1}{\Delta y^2}(u_{i,j-1}^n - 2u_{i,j}^n + u_{i,j+1}^n) \right]$$

$$u_{i,j}^{n+1} = u_{i,j}^n + \frac{\kappa \Delta t}{\Delta x^2}(u_{i-1,j}^n - 2u_{i,j}^n + u_{i+1,j}^n) + \frac{\kappa \Delta t}{\Delta y^2}(u_{i,j-1}^n - 2u_{i,j}^n + u_{i,j+1}^n)$$

The discretization is stable if the following CFL conditions are met:

$$\frac{1}{2} < \frac{\kappa \Delta t}{\Delta x^2}, \quad \frac{1}{2} < \frac{\kappa \Delta t}{\Delta y^2},$$

or equivalently the following CFL condition:

$$\Delta t < min\left(\frac{\Delta x^2}{2k}, \frac{\Delta y^2}{2k} \right)$$

3.0.1 Code solution description

First of all are imported all the necessary libraries, already discussed in a previous paragraph, with the following code snippet (figure 1):

```
#IMPORTING LIBRARIES

import pandas as pd
from mpl_toolkits.mplot3d import Axes3D
from pylab import figure, cm
import pycuda
import pycuda.driver as cuda
from pycuda.compiler import SourceModule
import pycuda.autoinit
import numpy as np
import matplotlib.pyplot as plt
```

Figure 1: libraries

Then are defined the **problem parameters** (2) and stored into python variables in order to make the code parameter-dependent and thus easily modifiable, mantainable, understandable and flexible. In particular are defined the block x-axis and y-axis dimensionality (we are talking about bidimensional blocks), the problem dimensionality in terms of x-axis (nx) and y-axis (ny) dimensions of the wave (in general of the object wich behaviour we are trying to simulate), the material dependent "k" heat diffusion constant, the time and space discretization steps (**dx/dy** and **dt**) used respectively to: determine the distance between two successive simulation steps and the number of point in wich we discretize the object, in other words the "heat propagation grid" dimensionalities, in order to simulate it's behaviour in time and space. Note that the timestep is calculated in order to verify the Courant-Friedrichs-Levy condition for the specifi problem; in particular by multiplying by 0.4 the result of $min(\frac{\Delta x^2}{2k}, \frac{\Delta y^2}{2k})$ we ensure that the following CLF condition is met: $\Delta t < min(\frac{\Delta x^2}{2k}, \frac{\Delta y^2}{2k})$

```
#DEFINING PROBLEM PARAMETERS

BLOCK_SIZE_X=8 #x-dimensionality of the blocks
BLOCK_SIZE_Y=8 #y-dimensionality of the blocks
nx=16 #x-dimensionaly of the object
ny=16 #y-dimensionaly of the object
kappa=1 #kappa, object diffusion coefficient
dx=1 #discretization step of the x-dimensionality of the object grid
dy=1 #discretization step of the y-dimensionality of the object grid
dt=0.4*min(dx*dx / (2.0*kappa), dy*dy / (2.0*kappa)) #timestep, calcluted like that to make sure that the evolution is stable
```

Figure 2: Parameters

The main CPU program (fig 10) first of all, using "***cuda.mem_alloc***" function, allocates in the **GPU memory** the necessary space for 2 vectors of nx*ny elements (nx*ny elements because when calling the main program we pass as the first parameter a pointer to a vector of nx*ny elements). We need 2 vectors because the "n+1" timestep of the simulation is computed based on timestep "n" (because of the first order time partial derivative in the left side of the equation). In particular u0 is a pointer to the initial state grid (with the initial temperatures of all the discrete points of the 2D object) that is used for the time-step "n", u1 is the wave state vector used for the time-step "n+1" (has the same dimensionality of n0) thus at the final time-step is the updated temperatures state that we want to compute with the simulation.

```
#MAIN CPU PROGRAM

def heatEquationGPU(u0,kappa,dt,dx,dy,num_timesteps):

    assert(u0.dtype == np.float32) #assert that we are using single-precision floats
    u1_gpu = cuda.mem_alloc(u0.nbytes) #making space for u1 on GPU memory
    u0_gpu=cuda.mem_alloc(u0.nbytes) #making space for u0 on GPU memory
    cuda.memcpy_htod(u0_gpu,u0) #copying u0 initial temperatures vector on GPU memory

    block=(BLOCK_SIZE_X, BLOCK_SIZE_Y,1) #computing block size. Bidimensional blocks.
    grid=(int(np.ceil(nx/BLOCK_SIZE_X)),int(np.ceil(ny/BLOCK_SIZE_Y)),1) #computing grid size. Bidimensional grid.

    for n in range(num_timesteps):  #n is the time-index
        heatEqnGPU(u1_gpu, u0_gpu, np.float32(kappa), np.float32(dt), np.float32(dx),np.float32(dy),np.uint32(nx),np.uint32(nx), block=block, grid=grid)
        u0_gpu, u1_gpu = u1_gpu, u0_gpu  #Swap the new and old temperatures

    u_updated = np.empty_like(u0)
    cuda.memcpy_dtoh(u_updated, u0_gpu) #retrieving from GPU memory the updated temperatures at timestep n=num_timesteps
    return u_updated  #Return the updated temperatures
```

Figure 3: Main CPU Program

Then are computed the block and the grid dimensionalities and stored into "block" and "grid" variables. In particular the blocks will be bidimensional with **BLOCK_SIZE_X*BLOCK_SIZE_Y** threads each, and the grid will be as well bidimensional with $\lceil \frac{nx}{\text{BLOCK\_SIZE\_X}} \rceil * \lceil \frac{ny}{\text{BLOCK\_SIZE\_Y}} \rceil$ blocks.

Then is used a for loop that ranges from 0 to num_timesteps to launch the CUDA GPU threads to solve num_timesteps times the wave equation. Note that the at the end of a timestep ***u1_gpu*** is pointing to the updated wave state, and note that at the end of the execution of the threads (thus after calling waveEqnGPU) we can use the same pointers by swapping them for the next iteration: in particular the vector representing the state "i" will be representing state "i+1" in the next iteration (in wich state "i+2" will be computed). Thus

we need to make u0_gpu pointing to what is pointed by ***u1_gpu*** and ***u0_gpu*** to a location of memory that can be modified (in this case what was pointed by u0_gpu can be used). At the end of the for loop is retrieved from the GPU memory the final state of the wave and it's returned.

for what concerns about the kernel implementation (fig 4), we first need to compute 2 indexes (that's because we are simulating the 2D heat equation, so we have a 2D "heat grid" of temperatures to compute) that we need to access u0 and u1. In particular we use **blockIdx.y/.x**, **blockDim.x/.y** and **threadIdx.x/.y** to compute them, and are computed in order to access all the cells of the object state matrix in a way that consecutive threads in a block are computing consecutive cells of the object state matrix at the next timestep. We can talk about "consecutive" elements of the matrix because the vectors u0_gpu and u1_gpu are the result of the linearization of the object state matrix, thus we can access the matrix using uni-dimensional indexing (indexing computed using **nx**, **j** and **i**). Then we compute the index of the element (point of the object) wich next temperature has to be computed, that is "center": basically we first compute the "raw" in the object grid with **nx*j** (moving in "rows" along the y-axis, imagining to put the object in a Cartesian plane) and then we move to the correct element adding "**i**" (moving in "columns" along the x-axis). Following the same method we compute the indexes in the array of the discrete points of the object that are above (**"north" index**), under (**"south" index**), at left (**"west" index**) and at right (**"east" index**)of the center element.

```
#KERNEL FUNCTION

heat_eqn_kernel_src = """
__global__ void heatEqn2D( float* u1, float* u0, float kappa, float dt,
            float dx, float dy, unsigned int nx, unsigned int ny)
{
    int i = blockIdx.x*blockDim.x + threadIdx.x;
    int j = blockIdx.y*blockDim.y + threadIdx.y;

    int center = j*nx + i;
    int north =(j+1)*nx+i;
    int south=(j-1)*nx+i;
    int east=j*nx + (i+1);
    int west=j*nx + (i-1);

            if(i>0 && i<nx-1 && j>0 && j<ny-1){
                u1[center] = u0[center]
                + (kappa*dt/(dx*dx)) * (u0[west] - 2*u0[center] + u0[east])
                + (kappa*dt/(dy*dy)) * (u0[south]-2*u0[center]+u0[north]);
            }

    else{
        u1[center] = u0[center];
    }
}
"""

mod = SourceModule(heat_eqn_kernel_src)
heatEqnGPU = mod.get_function("heatEqn2D")
```

Figure 4: Kernel Function

then the thread actually computes the "center" element of the temperatures vector at timestep n+1 (in other words the heat equation is computed). Note that all the equation is computed only for the internal elements/points of the object temperatures matrix, and that's because for the element in the "frame" some of the necessary points are not defined (for example for the first raw of the matrix the element ubove is not defined). Thus we need to implement an else branch that computes these **boundary conditions**: In particular the boundaries temperatures at timestep n+1 will be the same as the initial boundaries temperatures (u1[center] = u0[center]).

```
#CALLING MAIN PROGRAM

np.random.seed(50)
initial_temp=np.random.rand(nx*ny).astype(np.float32) #initial pseudo-random temperatures
u_tot_1 = heatEquationGPU(u0=initial_temp.copy(), kappa=kappa,dt=dt,dx=dx,dy=dy,num_timesteps=100)
```

Figure 5: Calling Main Program

Then the program is executed starting with a pseudo-random inital state configuration (using the same seed will give us the same simulation, so it's replicable). In particular it's executed 50 timesteps. Then the iniitial temperatures and the updated temperatures are plotted (respectively fig 6 and 7)

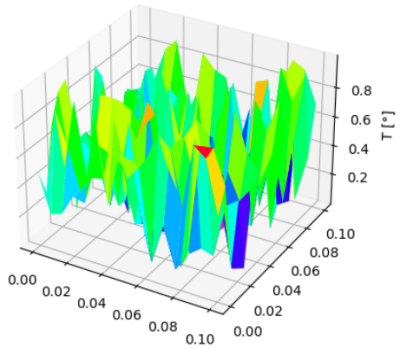

Figure 6: Initial temperatures

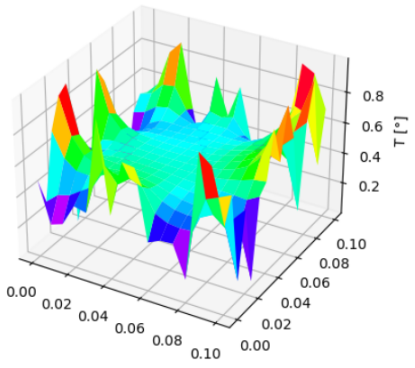

Figure 7: Updated temperatures

4 2D Linear Wave Equation

The **linear wave equation in 2D** can be written as follows (where u is the string displacement from the rest position and c is a material dependent constant):

$$\frac{\partial^2 u}{\partial t^2} = c^2 \nabla^2 u = c^2 \left[\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \right]$$

We can now discretize the law and thus calculate explicitily the state of the wave at timestep "n+1" (so, for a fixed and finite number of discrete timesteps, we can simulate the behaviour of the 2D wave):

$$\frac{1}{\Delta t^2}(u_{i,j}^{n+1} - 2u_{i,j}^n + u_{i,j}^{n-1}) = c \left[\frac{1}{\Delta x^2}(u_{i-1,j}^n - 2u_{i,j}^n + u_{i+1,j}^n) + \frac{1}{\Delta y^2}(u_{i,j-1}^n - 2u_{i,j}^n + u_{i,j+1}^n) \right]$$

$$u_{i,j}^{n+1} = 2u_{i,j}^n - u_{i,j}^{n-1} + \frac{c\Delta t^2}{\Delta x^2}(u_{i-1,j}^n - 2u_{i,j}^n + u_{i+1,j}^n) + \frac{c\Delta t^2}{\Delta y^2}(u_{i,j-1}^n - 2u_{i,j}^n + u_{i,j+1}^n)$$

IMPORTANT: This discretization is unstable if the following CFL condition is not met:

$$\frac{1}{2} < \frac{c\Delta t}{\Delta x^2}, \frac{1}{2} < \frac{c\Delta t}{\Delta y^2}$$

$$\Delta t < min(\frac{\Delta x^2}{2c}, \frac{\Delta y^2}{2c})$$

4.0.1 Code solution description

First of all are imported all the necessary libraries (8), already discussed in a previous paragraph, with the following code snippet (figure 8):

```
#IMPORTING LIBRARIES

import pycuda
import pycuda.driver as cuda
from pycuda.compiler import SourceModule
import pycuda.autoinit
import numpy as np
import matplotlib.pyplot as plt
import math
```

Figure 8: Libraries

Then are defined the block x-axis and y-axis dimensionality (we are talking about bidimensional blocks), the problem dimensionality in terms of x-axis (nx) and y-axis (ny) dimensions of the wave (in general of the object wich behaviour we are trying to simulate), the material dependent "c" constant, the time and space discretization steps used respectively to determine the distance between two successive simulation steps and the number of point in wich we discretize the wave (in other words the "wave grid" dimensionalities) in order to simulate it's behaviour in time and space. Note that the timestep is calculated in order to verify the Courant-Friedrichs-Levy condition for the specifi problem; in particular by multiplying by 0.4 the result of $min(\frac{\Delta x^2}{2c}, \frac{\Delta y^2}{2c})$ we ensure that the following CLF condition is met: $\Delta t < min(\frac{\Delta x^2}{2c}, \frac{\Delta y^2}{2c})$

```
#DEFINING PROBLEM PARAMETERS

BLOCK_SIZE_X=4 #x-dimensionality of the blocks
BLOCK_SIZE_Y=4 #y-dimensionality of the blocks
nx=16 #x-dimensionaly of the wave
ny=16 #y-dimensionaly of the wave
c=1 #c constant of the wave equation
dx=1 #discretization step of the x-dimensionality of the wave
dy=1 #discretization step of the y-dimensionality of the wave
dt=0.4*min(dx*dx / (2.0*c), dy*dy / (2.0*c)) #timestep, calcluted like that to make sure that the evolution of the wave is stable
```

Figure 9: Problem Parameters

The main CPU program (fig 10) first of all, using "***cuda.mem_alloc***" function, allocates in the **GPU memory** the necessary space for 3 vectors of nx*ny elements. We need 3 vectors because the "n+1" timestep of the simulation is computed based on "n" and "n-1" timesteps (because of the second order time partial derivative in the left side of the equation). In particular u0, passed as the first parameter of the function, is a pointer to the initial state grid (with the initial displacements from the rest position of all the discrete points of the wave grid) that is used for the time-step "n-1", u1 is the wave state vector used for the time-step "n" (has the same dimensionality of n0) and u2 is the wave state vector used for the time-step "n+1" thus at the final time-step is the updated wave state that we want to calculate with the simulation.

```
def waveEquationGPU(u0,c,dt,dx,dy,num_timesteps):

    assert(u0.dtype == np.float32) #assert that we are using single-precision floats
    u2_gpu = cuda.mem_alloc(u0.nbytes) #making space for u1 on GPU memory
    u1_gpu = cuda.mem_alloc(u0.nbytes) #making space for u1 on GPU memory
    u0_gpu = cuda.mem_alloc(u0.nbytes) #making space for u0 on GPU memory
    cuda.memcpy_htod(u0_gpu, u0) #copying u0 initial state vector on GPU memory

    block=(BLOCK_SIZE_X, BLOCK_SIZE_Y,1) #computing block size. Bidimensional blocks.
    grid=(int(np.ceil(nx/BLOCK_SIZE_X)),int(np.ceil(ny/BLOCK_SIZE_Y)),1) #computing grid size. Bidimensional grid.

    for n in range(num_timesteps): #n is the time-index
        waveEqnGPU(u2_gpu, u1_gpu, u0_gpu, np.float32(c), np.float32(dt), np.float32(dx),np.float32(dy),np.uint32(nx),np.uint32(ny), block=block, grid=grid)
        u0_gpu, u1_gpu, u2_gpu = u1_gpu, u2_gpu, u0_gpu #instruction to swap the pointers for the next iteration

    u_updated = np.empty_like(u0) #reserving space for the updated state of the wave
    cuda.memcpy_dtoh(u_updated, u2_gpu) #retrieving from GPU memory the updated wave state at timestep n=num_timesteps
    return u_updated #return updated wave state
```

Figure 10: Main CPU Program

Then are computed the block and the grid dimensionalities and stored into "block" and "grid" variables. In particular the blocks will be bidimensional with **BLOCK_SIZE_X*BLOCK_SIZE_Y** threads each, and the grid will be as well bidimensional with $\lceil \frac{nx}{\text{BLOCK\_SIZE\_X}} \rceil * \lceil \frac{ny}{\text{BLOCK\_SIZE\_Y}} \rceil$ blocks.n

Then is used a for loop that ranges from 0 to num_timesteps to launch the CUDA GPU threads to solve num_timesteps times the wave equation. Note that at the end of a timestep u2 is pointing to the updated wave state, and note that at the end of the execution of the threads (thus after calling waveEqnGPU) we can use the same pointers by swapping them for the next iteration: in particular the vector representing the state "i" will be representing state "i+1" in the next iteration (every pointer is moved forward of a timestep). Thus we need to make u0_gpu pointing to what is pointed by ***u1_gpu***, ***u1_gpu*** pointing to what ***u2_gpu*** is pointing and ***u2_gpu*** to a location of memory that can be modified (in this case what was pointed by u0_gpu can be used). At the end of the for loop is retrieved from the GPU memory the final state of the wave and it's returned.

For what concerns about the kernel implementation (fig 11 and fig 12), we first need to compute 2 indexes that we need to access u0, u1 and u2, in the same way we did for the heat equation (because the vectors u0_gpu and u1_gpu are the result of the linearization of the wave matrix, thus we can access it matrix using uni-dimensional indexing (using "center","north", "south", "west" and "east" indexes).

```
#KERNEL FUNCTION

wave_eqn_kernel_src = """
__global__ void waveEqn2D(float* u2, float* u1, float* u0, float c, float dt, float dx, float dy, int nx, int ny)
{

    int i = blockIdx.x*blockDim.x + threadIdx.x;
    int j = blockIdx.y*blockDim.y + threadIdx.y;

    int center = j*nx + i;
    int north =(j+1)*nx+i;
    int south=(j-1)*nx+i;
    int east=j*nx + (i+1);
    int west=j*nx + (i-1);

    if(i>0 && i<nx-1 && j>0 && j<ny-1)
    {
        u2[center] = 2.0f*u1[center] - u0[center]
        + ((c*dt*dt)/(dx*dx)) * (u1[west] - 2.0f*u1[center] + u1[east])
        + ((c*dt*dt)/(dy*dy)) * (u1[south]-2.0f*u1[center]+u1[north]);
    }
```

Figure 11: Kernel function part 1

```
    if (i==0) {
        u1[center] = u1[east];
        u2[center] = u1[east];
    }
    else if (i == nx-1) {
        u1[center] = u1[west];
        u2[center] = u1[west];
    }
    else if (j == 0) {
        u1[center] = u1[north];
        u2[center] = u1[north];
    }
    else if (j == ny-1) {
        u1[center] = u1[south];
        u2[center] = u1[south];
    }
}
"""

mod = SourceModule(wave_eqn_kernel_src)
waveEqnGPU = mod.get_function("waveEqn2D")
```

Figure 12: Kernel function part 2

Then the thread computes the "center" element of the displacements vector at timestep n+1 (in other words the linear wave equation is computed). Note that all the equation is computed only for the internal discrete points of the wave (both at state "n" and "n+1"), just like in the heat equation kernel function. Thus we need to implement that computes these **boundary conditions** for both u2_gpu and u1_gpu, in particular we compute the boundaries "center" elements using just a defined near point (for the first column of the matrix we use the right element, for the last column the left element, for the first row the point underneath and for the last row the element above).

Then the program is executed starting with a pseudo-random inital state configuration, In particular it's executed 3 times for increasing number of timesteps in order to show the evolution of the wave. Then the initial state and the updated states are plotted (respectively fig 14, 15, 16, 17)

```
#CALLING MAIN CPU PROGRAM

np.random.seed(50)
initial_state=np.random.rand(nx*ny).astype(np.float32) #initial pseudo-random wave state

u_tot_1 = waveEquationGPU(u0=initial_state.copy(), c=c,dt=dt,dx=dx,dy=dy,num_timesteps=150)
u_tot_2 = waveEquationGPU(u0=initial_state.copy(), c=c,dt=dt,dx=dx,dy=dy,num_timesteps=500)
u_tot_3 = waveEquationGPU(u0=initial_state.copy(), c=c,dt=dt,dx=dx,dy=dy,num_timesteps=2500)
```

Figure 13: Calling Main Program

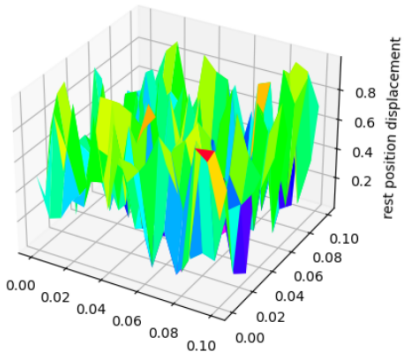

Figure 14: Initial Wave State

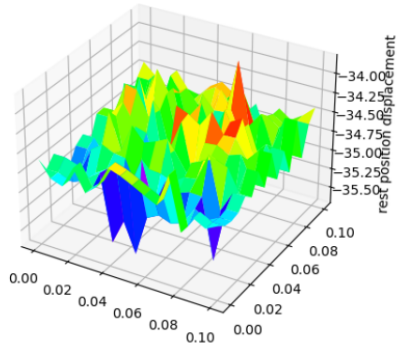

Figure 15: Updated Wave State 1

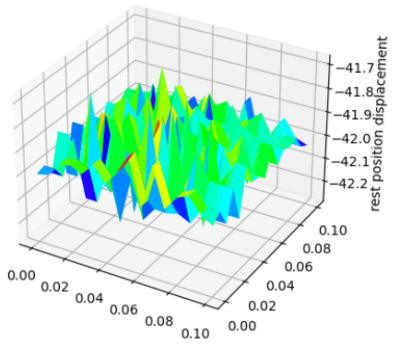

Figure 16: Updated Wave State 2

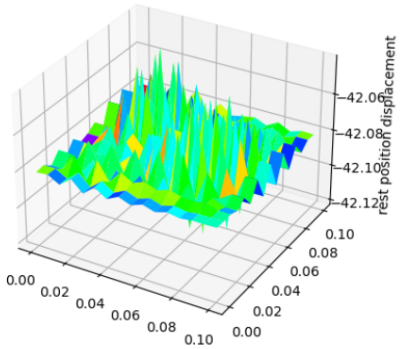

Figure 17: Updated Wave State 3