AM205 Project: Numerical Methods of the Shallow Water Equations and their applications

Silin Zou, Lihong Zhang, Ruoxi Yang
Harvard John A. Paulson School of Engineering and Applied Sciences
Fall 2019

Abstract

The shallow water equations are a system of basic hyperbolic partial differential equations that describe the shallow water flow. Solving shallow water equations analytically is expensive, but many numerical methods are found to solve these equations effectively. Thus, in the study of both Computational Mathematics and Earth Sciences, it is a critical issue to figure out what methods are accurate and efficient for solving shallow water equations. In this project, we analyze different numerical methods applied in solving shallow water equations, discuss their properties and explore their applications.

Contents

1 Introduction **1**
- 1.1 Motivation . . . 1
- 1.2 Two Forms of Shallow Water Equations . . . 1
  - 1.2.1 Conservative Shallow Water Equations . . . 1
  - 1.2.2 Non-conservative Shallow Water Equations . . . 2
- 1.3 Numerical Methods . . . 3
- 1.4 Applications . . . 3

2 Methods and Results **4**
- 2.1 Forward Euler Method . . . 5
- 2.2 Leapfrog Central Difference . . . 6
- 2.3 Adams-Bashforth Method . . . 13
- 2.4 Runge-Kutta . . . 14

3 Applications **17**

4 Conclusions and Discussion **17**

A The attached compressed file contains all the pictures and videos shown in this report. **21**

B The following are all the codes used to implement this project. **21**

1 Introduction

1.1 Motivation

Geologists are interested in the wave travelling of water, atmosphere, seismic activities and so on, by which they can predict environmental changes, such as sea wave movements, weather fluctuations, and earthquakes. The Shallow water equations(SWE) are derived from Navier-Stokes equations, in which the horizontal dimension scales are much larger than the vertical dimension scale [8]. Under this condition, the SWE are a classical system of hyperbolic partial differential equations(PDEs) describing the ideal wave propagations of water underneath a pressure surface (e.g. atmospheric pressure) in shallow basins where the height of water is negligible compared to its horizontal dimensions. It means that the average height of water is almost constant, and the vertical velocity is also negligible compared to horizontal velocities[7]. Figure 1 shows an example of shallow water basin where horizontal dimension scales are set 10^6, but the height of water is set 10^2, which is very small compared to the horizontal ones. In this case, the vertical velocities can be negligible.

However, in practice, the vertical velocity is not zero. When water height changes, or water fluctuates vertically, the vertical velocity is nonzero. The assumption that the vertical velocity is zero only holds with the SWE, and after the solutions of the SWE are determined, the vertical velocity can be obtained by the continuity equation [9].

The SWE are a system of 3 PDEs contaning 3 undetermined variables, the height, h and two horizontal velocities, u and v, of fluids. In this case, solving the SWE can be expensive, and the final solutions of h, u, and v may be functions of each other, which makes it hard to plot the movements analytically. Considering that the SWE are typical to describe the wave movements of not only water, but also other fluids, it is important to find exact and accurate solutions of SWE. People tried different mathematical methods to tackle this issue, and found the numerical methods are fast and accurate [2]. Even though some numerical methods do not give the exact expression of the SWE solutions, they can provide exact data for wave movement modelling. In this project, we discuss the various numerical methods we learned in AM 205, apply them to solving the SWE, and analyze their effects on the SWE.

1.2 Two Forms of Shallow Water Equations

To be explicit, for a shallow water area, the velocities and height of water can be described by the conservative form or non-conservative form of SWE [7].

1.2.1 Conservative Shallow Water Equations

The conservative form of SWE is derived from the conservation laws of mass and linear momentum [7]. This form is still valid when the assumption that the average height of

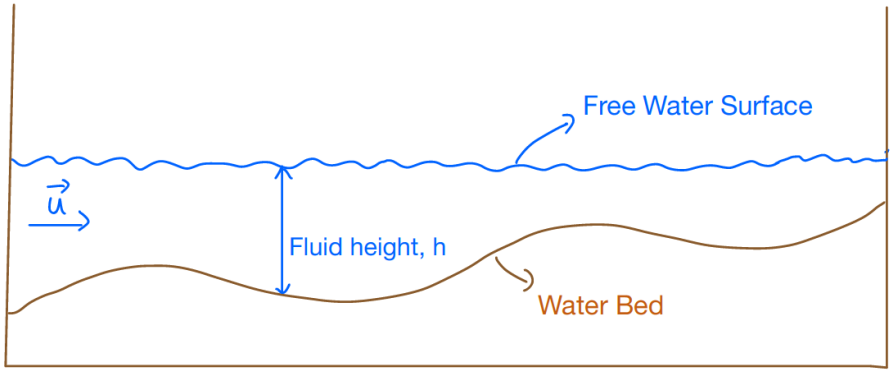


Figure 1: Shallow Water Example

water is unchanged does not hold [8]. Without considering viscous forces, Coriolis forces, and viscous forces, A water with horizontal bed has the SWE as the following [7].

$$\begin{aligned} \frac{\partial(\rho h)}{\partial t} + \frac{\partial(\rho hu)}{\partial x} + \frac{\partial(\rho hv)}{\partial y} &= 0 \\ \frac{\partial(\rho hu)}{\partial t} + \frac{\partial(\rho hu^2 + \frac{1}{2}\rho gh^2)}{\partial x} + \frac{\partial(\rho huv)}{\partial y} &= 0 \\ \frac{\partial(\rho hv)}{\partial t} + \frac{\partial(\rho huv)}{\partial x} + \frac{\partial(\rho hv^2 + \frac{1}{2}\rho gh^2)}{\partial y} &= 0 \end{aligned} \tag{1}$$

where x and y are the horizontal dimensions;
h is the water height as a function of x, y and t;
u and v are the vertically averaged velocity of water in the direction of x and y respectively;
ρ is the water density;
g is the gravitational acceleration.

1.2.2 Non-conservative Shallow Water Equations

The non-conservative SWE are obtained by expanding the conservative SWE in equation (1), and including the viscous forces, Coriolis forces, and viscous forces. The following is a typical expression of non-conservative SWE [7].

$$\begin{aligned} \frac{\partial h}{\partial t} + u\frac{\partial h}{\partial x} + v\frac{\partial h}{\partial y} &= -h(\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y}) \\ \frac{\partial v}{\partial t} + u\frac{\partial v}{\partial x} + v\frac{\partial v}{\partial y} + fu &= -g\frac{\partial h}{\partial y} - bv + \sigma(\frac{\partial^2 v}{\partial^2 x} + \frac{\partial^2 v}{\partial^2 y}) \\ \frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} + v\frac{\partial u}{\partial y} - fv &= -g\frac{\partial h}{\partial x} - bu + \sigma(\frac{\partial^2 u}{\partial^2 x} + \frac{\partial^2 u}{\partial^2 y}) \end{aligned} \tag{2}$$

where x and y are the horizontal dimensions;
h is the water height as a function of x, y and t;
u and v are the vertically averaged velocity of water in the direction of x and y respectively;
f is the Coriolis coefficient;
b is the viscous drag coefficient;
σ is the kinematic viscosity; g is the gravitational acceleration;
the water density is assumed to be constant;

If we reduce the horizontal dimension to 1D with only x scale, then h, u, v are still functions of time, but only at x direction do these variables vary, and the variables at y direction are constants with respect to space. Since u and $\frac{\partial u}{\partial x}$ are both very small, $u\frac{\partial u}{\partial x}$ is negligible, and other similar terms are also negligible. To make equations simple, we ignore the viscous forces which are tiny in shallow water, but the Cariolis force should be kept. The PDEs in equation 2 become

$$\begin{aligned} \frac{\partial u}{\partial t} &= fv - g\frac{\partial h}{\partial x} \\ \frac{\partial h}{\partial t} &= -H\frac{\partial u}{\partial x} \\ \frac{\partial v}{\partial t} &= -fu \end{aligned} \tag{3}$$

1.3 Numerical Methods

There are many numerical methods to solve the SWE. For finite difference methods, we can use forward, backward, or central schemes of difference. Furthermore, finite volume method is a powerful tool of solving 2D wave equations. To increase the accuracy, many complicated time and space stepping sizes are applied to the basic finite difference methods, which gives numerical methods with better accuracy, such as leapfrog method, Adams-Bashforth method, and Runge Kutta method [8]. Although many numerical methods match the forms of the SWE, not all of them can give accurate results. Some of the above methods are ill-conditioned and very sensitive to small input changes, thus do not fit the SWE. Some methods are more efficient than others. In this project, we analyze the mathematical principles of different numerical methods, apply the methods to solving the SWE, and discuss their feasibility.

1.4 Applications

SWE describe not only the water movements under ideal conditions, but also the wave propagations of other fluids under the consumption of no vertical velocities. Even though the SWE are simple equations with ideal conditions, they have a lot of applications in geophysical sciences, especially the turbulence of the atmospheric layer, the ocean currents near seashores, and so on [9]. For example, the Ocean-Land-Atmosphere Model(OLAM) describes the global-scale dynamics represented by the SWE, and provides the wave movement solutions by finite volume methods [13].

To understand the SWE deeply, we can start from the simplest case: 1D equations which are describe the movements in 1 horizontal dimension and the vertical dimension. Adhémar Jean Claude Barré de Saint-Venant derived the one-dimensional(1D) form of SWE [8], which is the fundamental formula for any more complicated cases. In this project, We start from applying numerical methods to solving 1D SWE, then discuss the 2D forms with the numerical methods that work well in 1D, and finally explore their basic applications.

In the following section, different numerical methods are discussed, and the corresponding solutions of SWE are shown.

2 Methods and Results

Any system of evolution equations can be written as [6]

$$\frac{\partial y}{\partial t} = F(y) \tag{4}$$

where y is a list of all dependent variables at all points in space and F describes how they evolve. We now examine how different numerical difference schemes perform in solving the SWE system.
The function F in the Equation 4 of h, u, v are

$$\begin{aligned} F_u &= fv - g\frac{\partial h}{\partial x} \\ F_h &= -H\frac{\partial u}{\partial x} \\ F_v &= -fu \end{aligned} \tag{5}$$

where H is the average water height which we assume to be a constant under the assumption above that the vertical scale is much smaller than the horizontal scales.
With respect to the initial conditions, we have the following settings:

$$\begin{aligned} L &= 10^6 m & N &= 200, \\ \Delta t &= 10s & \Delta x &= L/N, \\ T &= 500000s & f &= 0.0001, \\ g &= 9.8m/s^2 & iteration &= int(T/\Delta t) \end{aligned}$$

We set the horizontal magnitude L as $10^6 m$ and vertical magnitude h as about $100m$. The real ocean movement is also affected by Coriolis Force, so we set the value of Coriolis Force as f. The grid number N is 200, and the space step $\Delta x = L/N = 5000$. In order to satisfy the CFL condition, i.e. $c\frac{\Delta t}{\Delta x} <= 1$, we set Δt as $10s$. The speed c can be derived by the equation 6 when the rotation is considered i.e. f exists[12]:

$$c = \frac{\omega}{k} = \sqrt{\frac{f^2}{k^2} + g * H} \tag{6}$$

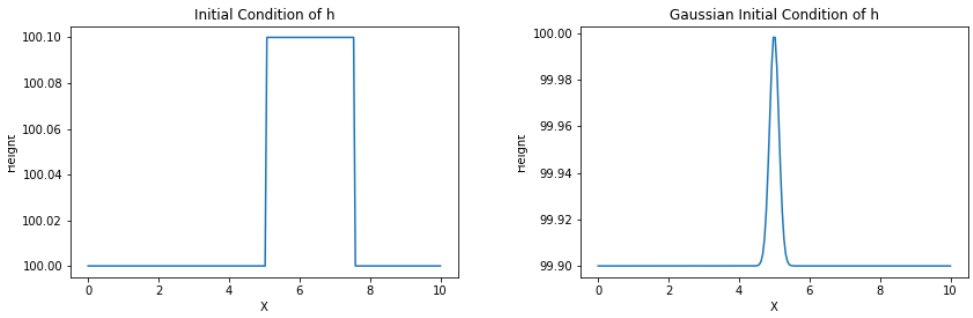


Figure 2: 2 Initial Conditions of h

where k is the wave number and c is the speed caused by gravity wave and inertial wave. In our case, the maximum of c is $100m/s$, which renders $c\frac{\Delta t}{\Delta x} = 0.2$. For the water height h, we set two initial conditions, as shown in Figure 2, both of which have little perturbation in the middle of the water initially. In the following sections, We show how the shallow water system evolves under these conditions. The difference between these two conditions are also analyzed in the following sections.

For boundary conditions, there are two prevailing choices. One is the periodic boundary condition, i.e. the previous point of the first grid point is the last grid point, and the next point of the last grid point is the first grid point [4]. The other one is the Dirichlet boundary condition, where the boundaries of the system act like walls and the velocity perpendicular to the wall is 0 [1]. Unless specified otherwise, we use periodic boundary in this project.

2.1 Forward Euler Method

First, we tried one of the most basic explicit method—— Forward Euler Method. As described in the AM205 class, Forward Euler numerical method can solve 1st-order degree differential equations with a given initial value in the 1st order accuracy.
In general, the forward Euler scheme can be written as:

$$y^{(n+1)} = y^{(n)} + \Delta t F(y^{(n)}) \tag{7}$$

where y^n is the value of y at the n_{th} time grid;
Δt is the time scheme unit;

Specifically, by applying forward Euler method to SWE systems 3, we have

$$\begin{aligned} h_j^{i+1} &= h_j^i - H\Delta t \frac{u_{j+1}^i - u_{j-1}^i}{2\Delta x} \\ u_j^{i+1} &= u_j^i + \Delta t(-g\frac{h_{j+1}^i - h_{j-1}^i}{2\Delta x} + f v_j^i) \\ v_j^{i+1} &= v_j^i - f\Delta t u_j^i \end{aligned} \tag{8}$$

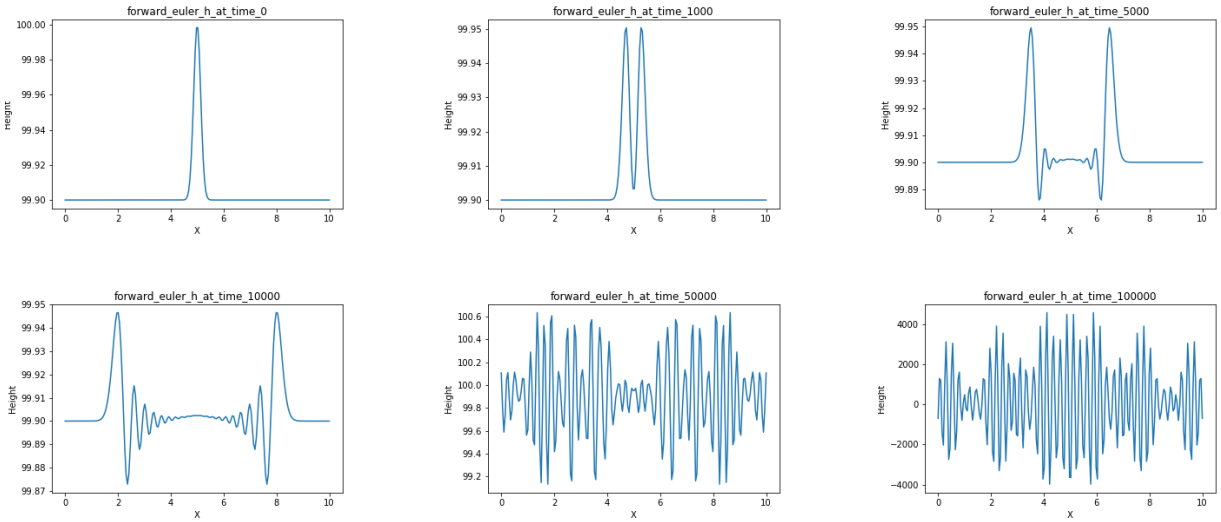


Figure 3: The results of Forward Euler Method

where we use central difference method in space dimension.

The simulation results under the Gaussian initial condition are shown in Figure 3 and the video (video link:https://drive.google.com/file/d/1qiSiFZqhzKryYNHKCz9uoy4bixqi9j91/view?usp=sharing). From these figures, we find that the Forward Euler Method is unstable after 2000 time steps (20000s), although it gives reasonable simulation at the beginning. This method is also proved unstable for hyperbolic wave equation in the AM205 class. According to the slides of lecture 15 (Page 15) [10] in AM205, if we apply Fourier stability analysis to this method, it yields $|\lambda(k)| > 1$ except for several wave number values, which means the Fourier modes are amplified in most wave number and this method is unstable.

Regardless of the numerical instability, we can still get some movement features of SWE from this method. For example, in Figure 3, the little perturbation propagates along the x-axis towards the edges, then the water height in the middle falls down. In the following sections, we will explore some stable finite difference methods and compare the results of them.

2.2 Leapfrog Central Difference

Leapfrog integration is an explicit method for solving differential equations numerically. It is a second-order method, and stable for the SWE. The stability can also be proven by using Fourier Stability Analysis in the Lecture15 [10].

With the PDE equation $\frac{\partial u}{\partial t} + c\frac{\partial u}{\partial x} = 0$, we consider the ansatz as $U_j^n(k) \equiv \lambda(k)^n e^{ikx_j}$.

By applying the Leapfrog Central Difference method, we get the finite differential equation:

$$\frac{U_j^{n+1} - U_j^{n-1}}{2\Delta t} + c\frac{U_{j+1}^n - U_{j-1}^n}{2\Delta x} = 0$$

Then we have:

$$\frac{\lambda(k)^{n+1}e^{ikj\Delta x} - \lambda(k)^{n-1}e^{ikj\Delta x}}{2\Delta t} + c\frac{\lambda(k)^n e^{ikj\Delta x+ik\Delta x} - \lambda(k)^n e^{ikj\Delta x-ik\Delta x}}{2\Delta x} = 0$$

$$\frac{\lambda(k)^2 - 1}{2\Delta t} + c\frac{\lambda(k)e^{ik\Delta x} - \lambda(k)e^{-ik\Delta x}}{2\Delta x} = 0$$

$$\lambda(k)^2 - 1 + c\frac{\Delta t}{\Delta x}\lambda(k)(\cos k\Delta x + i\sin k\Delta x - \cos k\Delta x + i\sin k\Delta x) = 0$$

$$\lambda(k)^2 - 1 + c\frac{\Delta t}{\Delta x}\lambda(k)(2 * i\sin k\Delta x) = 0$$

$$\lambda(k) = -i * c\frac{\Delta t}{\Delta x} * \sin k\Delta x \pm \sqrt{-(c\frac{\Delta t}{\Delta x})^2 * (\sin k\Delta x)^2 + 1}$$

Then we have :

$$\begin{aligned} |\lambda(k)|^2 &= -(c\frac{\Delta t}{\Delta x})^2 * (\sin k\Delta x)^2 + 1 + (c\frac{\Delta t}{\Delta x})^2 * (\sin k\Delta x)^2 \\ &= 1 \end{aligned}$$

What's more, the equations above should satisfy CFL condition that $c\frac{\Delta t}{\Delta x} \leq 1$ to make sure the solutions for the equations exist. From the derivation above, we can infer that the leapfrog method used in this project is stable for the SWE problem since the Fourier modes are not amplified in any wave number k.
Another advantage of leapfrog integration when applied to mechanics problem is it could conserve the energy of dynamical systems[15]. So it can give more accurate wave simulation than other methods.
In general, the leapfrog scheme can be written as:

$$y^{(n+1)} = y^{(n-1)} + 2\Delta t F(y^{(n)}) \tag{9}$$

where y^n is the value of y at the nth time grid;
Δt is the time scheme unit;
Specifically, by using leapfrog to solve 1-D SWE systems 3, we have

$$\begin{aligned} h_j^{i+1} &= h_j^{i-1} - 2H\Delta t\frac{u_{j+1}^i - u_{j-1}^i}{2\Delta x} \\ u_j^{i+1} &= u_j^{i-1} + 2\Delta t(-g\frac{h_{j+1}^i - h_{j-1}^i}{2\Delta x} + fv_j^i) \\ v_j^{i+1} &= v_j^{i-1} - 2f\Delta t u_j^i \end{aligned} \tag{10}$$

where i is the time grid index, and j is the space grid index. Note that since the central difference method cannot be applied to the first time grid point, we use the forward

difference method for the first Δt in time.

The results of simulation using leapfrog method are shown in Figure 4. This method is stable in our project's experimental time length (500000s) and is expected to keep stable for longer time period. We can see a water flow with weak perturbations in Figure 4: Beginning with a Gaussian perturbation, the water propagates towards two edges with wavy amplitudes. When it collides with the walls, the energy accumulates at the boundaries under the assumption of periodic boundary, and then the wave propagates back towards the center from two edges. Along with this report, we have a video for these simulation, which shows more intuitively how the water moves and how the wave propagates in the shallow water model (video link:https://drive.google.com/file/d/1JLyF_DLYNsemoz8ZtmJOkrpq5yKAHM1h/view?usp=sharing).
If we use square wave perturbation as the initial condition, we can get a similar water movement to the that of the Gaussian initial condition above. However, the final simulation results in a long time period become pretty rough, as shown in the Figure 5 and the video, and are prone to be unstable (video link: https://drive.google.com/file/d/1RmnvIinM7isiFOcCaPkHXnynNfbvVgDv/view?usp=sharing). We think it is because there is a vertical steep change in water height at the edge of perturbation which has no derivative analytically. Therefore, the steep change may result in a huge or even infinite derivative value, which reduces the smoothness of water flow in the simulation results with the square wave initialization.

Regardless of the disadvantage of square wave initialization, we use it to initialize the 2-D simulation since it is easy to set up and keeps the basic characteristic of the SWE. Based on the 1-D initial conditions, we set the 2-D conditions in the following expressions:

$$\begin{aligned} L_x &= 10^6 m & L_y &= 10^6 m, \\ \Delta t &= 50s & N &= 100, \\ T &= 100000s \end{aligned}$$

In order to avoid the possible overtime running, we set low-resolution time and space steps, with $N = 100, \Delta t = 50s$, and $T = 100000s$. Besides, we choose the Dirichlet boundary to imitate the case of wall collision.

And the 2-D finite difference equations are in the following.

$$\begin{aligned} \frac{h_{i,j}^{n+1} - h_{i,j}^{n-1}}{2\Delta t} &= -H\left(\frac{u_{i+1,j}^{n} - u_{i-1,j}^{n}}{2\Delta x} + \frac{v_{i,j+1}^{n} - v_{i,j-1}^{n}}{2\Delta y}\right) \\ \frac{u_{i,j}^{n+1} - u_{i,j}^{n-1}}{2\Delta t} - f * v_{i,j}^{n} &= -g\frac{h_{i+1,j}^{n} - h_{i-1,j}^{n}}{2\Delta x} \\ \frac{v_{i,j}^{n+1} - v_{i,j}^{n-1}}{2\Delta t} + f * u_{i,j}^{n} &= -g\frac{h_{i,j+1}^{n} - h_{i,j-1}^{n}}{2\Delta y} \end{aligned} \tag{11}$$

where i, j are the space grid along x axis and y axis, and n is the time index.

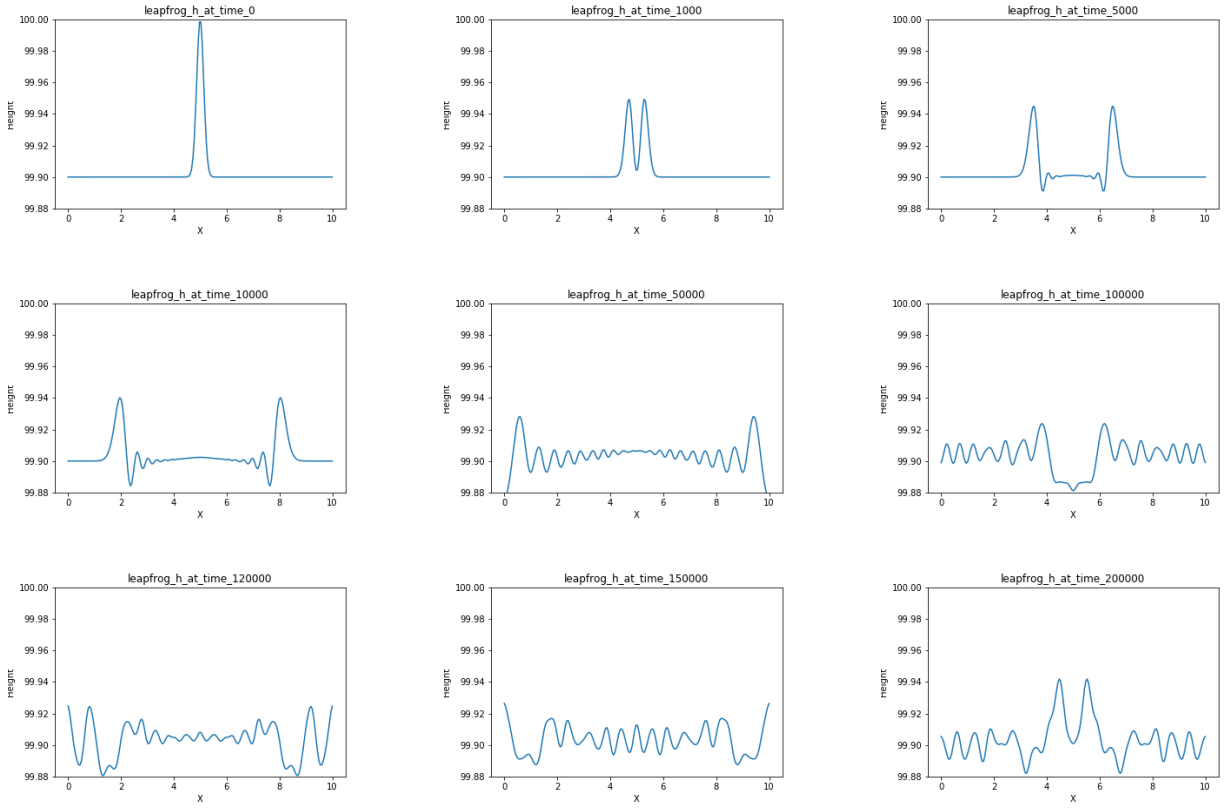


Figure 4: The results of Leapfrog Method with Gaussian Initial Conditions

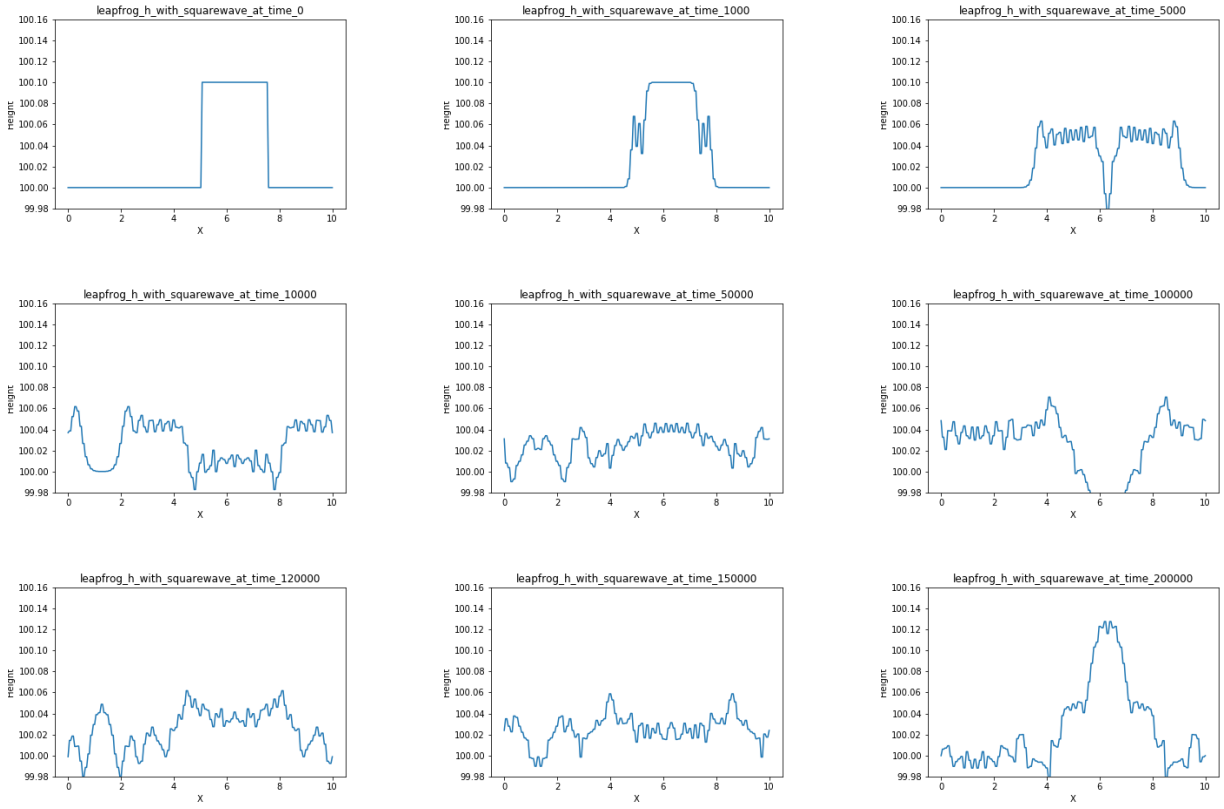


Figure 5: The Results of Leapfrog Method with the Square Wave Initial Conditions

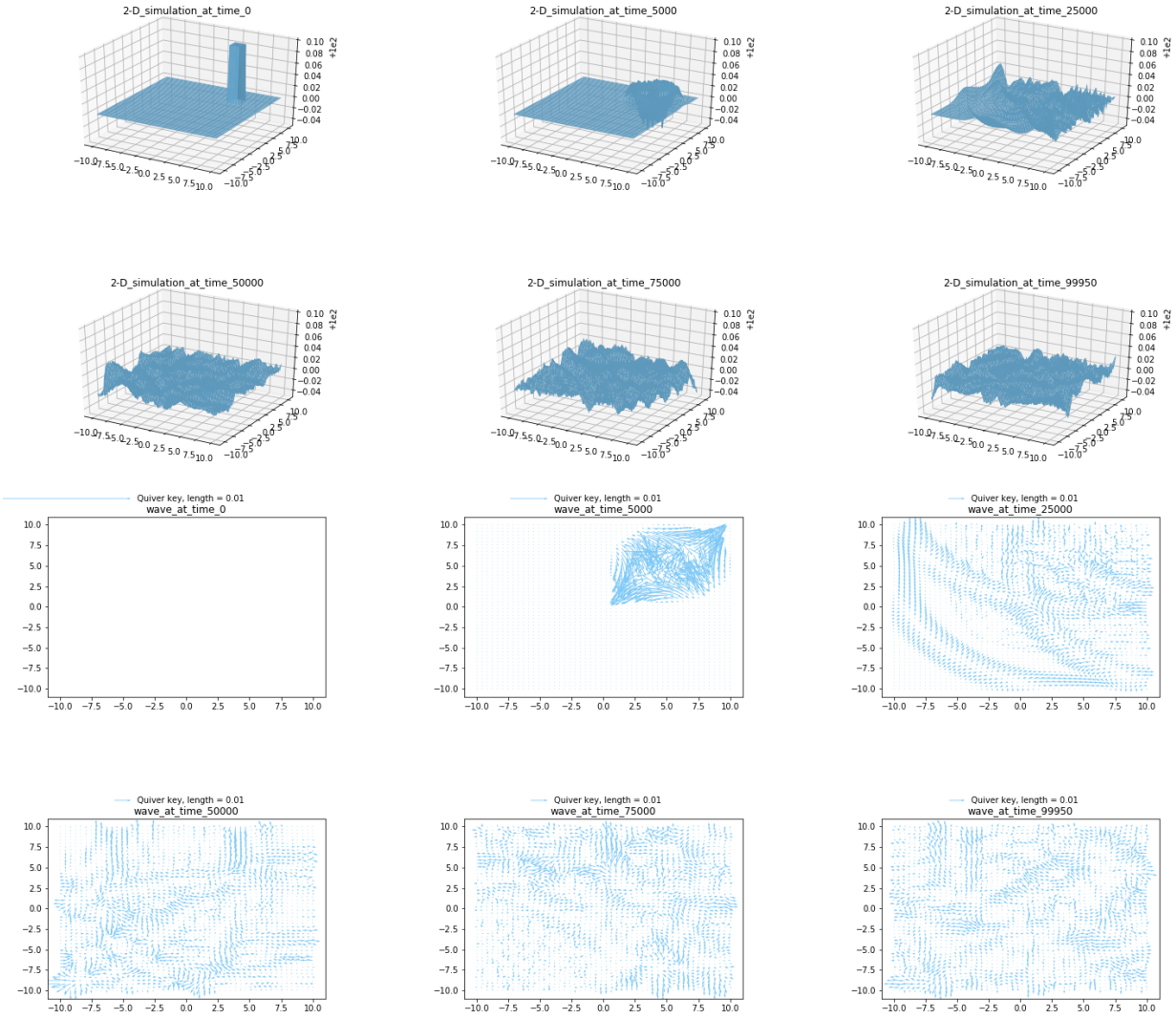


Figure 6: The 2-D Simulation of Shallow Water Equation

We have a video to show the movement of water under the effect of inertial gravity in Figure 6 (video link:https://drive.google.com/file/d/1pKaDg79azYJFbBpBzDuXax9iruzPo9YV/view?usp=sharing). The first 6 images represent the evolution of water heights, and the last 6 images show the wave speed distribution in the process. From the images, we can draw a similar description to that of 1-D shallow water movement: the central perturbation falls down due to its gravity which makes the wave spread around. The wave maintains its shape when propagating, however, the solid wall on the boundary changes its movement with the reflected waves.

If we consider viscous forces and horizontal advection, the 2D SWE becomes Equation 12

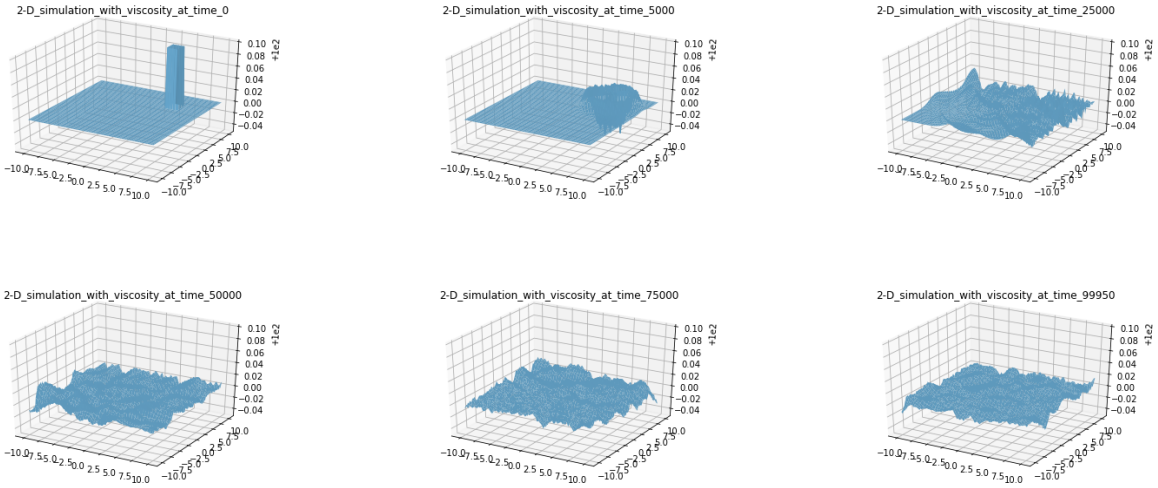


Figure 7: The 2-D Simulation of Shallow Water Equation with Viscosity and Advection

which is closer to the complete form of SWE.

$$\begin{aligned} \frac{\partial h}{\partial t} + u\frac{\partial h}{\partial x} + v\frac{\partial h}{\partial y} &= -h(\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y}) \\ \frac{\partial v}{\partial t} + u\frac{\partial v}{\partial x} + v\frac{\partial v}{\partial y} + fu &= -g\frac{\partial h}{\partial y} - bv \\ \frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} + v\frac{\partial u}{\partial y} - fv &= -g\frac{\partial h}{\partial x} - bu \end{aligned} \tag{12}$$

where the parameter b equals to 10^{-5}.
The simulation with with viscosity and advection considered are shown in Figure 7 and a video (video link:https://drive.google.com/file/d/1ihS6kgIhigNDFcZmmmbbYzdJQCLb2qvN/view?usp=sharing). In Figure 7, the waves are pretty similar to those in Figure 6 in the beginning time periods. The difference between these two figures is that, because viscous flowing water has damping energy consumption, the waves become weak and damped as time goes by in Figure 7.

In order to figure out why there are damped waves and explore the effect of viscous force, we did two experiments: we have already known that b is the viscous parameter, so we set b equals to 10^{-5} and 0 respectively, which defines the water with and without the viscous force respectively. Then we calculate the energy of the simulated area. In terms of energy, we get two types of energy here: kinetic energy (K) and potential energy (P). These energy can be calculated by the Equation 13.

$$\begin{aligned} K &= \frac{1}{2}m * V^2 \\ P &= m * g * h' \\ m &= \rho * S * h \end{aligned} \tag{13}$$

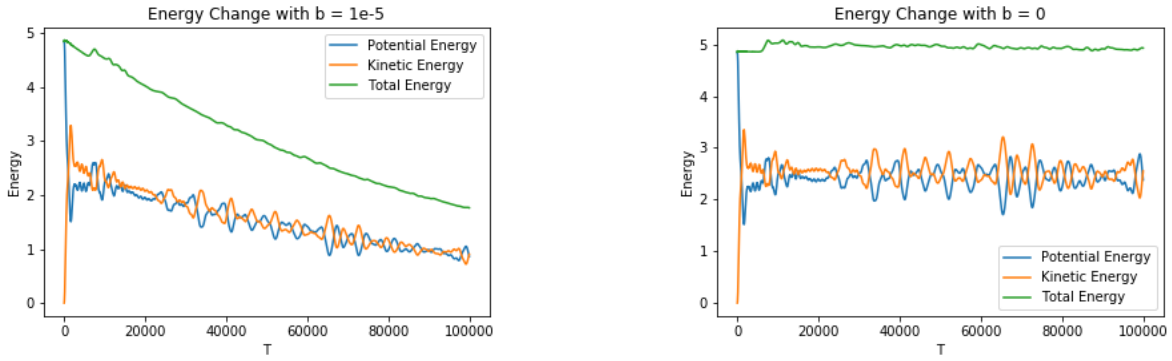


Figure 8: The Effect of Viscosity

Because the density of water ρ and the area of simulated region S are constant, we drop these two parameters. Also, for the unit volume of water, the average h' equals to $h/2$, so the Equation 13 becomes Equation 14. We first calculate the energy of every grid, then sum them up. Note that for potential energy, we subtract the average initial potential energy at every time step to make sure these two types of energy are at same magnitude. Our results for different b are shown in Figure 8, where the x-axis is the time scale and the y-axis is the energy.

$$\begin{aligned} K &= \frac{1}{2} h * (u^2 + v^2) \\ P &= \frac{g * h^2}{2} \end{aligned} \tag{14}$$

We can see from the Figure 8 that the kinetic energy (KE) and potential energy (PE) are mutually transformed. If the KE increases, the PE decreases, and vice versa. This implicates our model complies the law of conservation of energy and explains the wave propagation in Figure 6 and Figure 7. The most obvious feature between the two images in Figure8 is if $b = 0$, then the total energy is almost constant in our shallow water model, while if $b = 1e-5$, the total energy is dissipated as time goes by. That means the viscous force plays an important role in this model since it reduces the usable energy (KE and PE) and transforms energy into heat. And that's why we get damped waves in shallow water if we consider the viscosity.

According to the reference [5], the leapfrog scheme has the tendency to increase the amplitude of the computational mode with time in nonlinear examples, so we follow the suggestion from the reference and add a Robert-Asselin time filter every 5 time steps.

2.3 Adams-Bashforth Method

The leapfrog time differencing method above is a widely used method of solving ODEs and PDEs of low-viscosity fluids, and is robust in various time grids [3]. Its robustness cannot always hold since the phase-speed error of the scheme usually makes the leapfrog method unstable [3]. This problem can be avoided by using Adams-Bashforth method [3]. This section is to analyze the effects of Adams-Bashforth method in PDEs of low-viscosity fluids,

and compare it with the leapfrog method.

The general N-th order Adams-Bashforth form for the classical PDE 4 is[3]

$$\frac{y^{n+1}-y^n}{\Delta t} = \sum_{i=0}^{N-1} a_i F(y^{n-i}) \tag{15}$$

where y^n is the value of y at the nth time grid;
Δt is the time scheme unit;
and the coefficients a_i can be determined by applying the Taylor series expansion on both sides of the equation 15.

In this project, we use the 2nd-order Adams-Bashforth method, which has the form [14]

$$y^{n+1} = y^n + \frac{\Delta t}{2}(3F(y^n) - F(y^{n-1})) \tag{16}$$

By applying the Equation 16 to Equation 5, we get the finite difference expressions of h, u, and v

$$\begin{aligned}
u_j^i &= u_j^{i-1} + \frac{\Delta t}{2}(3(-fv_j^{i-1} - g\frac{h_{j+1}^{i-1} - h_{j-1}^{i-1}}{2\Delta x}) - (-fv_j^{i-2} - g\frac{h_{j+1}^{i-2} - h_{j-1}^{i-2}}{2\Delta x})) \\
h_j^i &= h_j^{i-1} + \frac{\Delta t}{2}(3(-H\frac{u_{j+1}^{i-1} - u_{j-1}^{i-1}}{2\Delta x}) - (-H\frac{u_{j+1}^{i-2} - u_{j-1}^{i-2}}{2\Delta x}) \\
v_j^i &= h_j^{i-2} - \frac{\Delta t}{2}(3fu_j^{i-1} - fu_j^{i-2})
\end{aligned} \tag{17}$$

where i is the time grid index, and j is the space grid index.

The phase-speed errors in the Adams-Bashforth method can be examined by applying it in the oscillation equations. This is not the focus of this project, so we do not discuss much about the examination process here. We choose the initial condition of water height as a Gaussian function, as shown in the first image in Figure 9. By applying the Equation 15 to Equation 3 with the central spatial difference method, and set Δt = 2, then we get the solutions of water heights shown in the Figure 9 and the video (video link:https://drive.google.com/file/d/12B5tlkx8y3O6tJ9bGyasRBxkMf3HR0rC/view?usp=sharing).

We should notice that, although the leapfrog method is always stable regardless of Δt and Δx as long as the CFL condition is satisfied, Adams-Bashforth method is very sensitive to Δt. That's to say, the Adams-Bashforth method has a high requirement for CFL condition.

2.4 Runge-Kutta

Runge-Kutta (RK) methods are a series of multi-step discretization methods, which use the intermediate values of y between the nth and (n+1)th time grid [11]. Both RK and Adams-Bashforth methods use the derivative function $F(y^n)$ to determine the finite difference values.

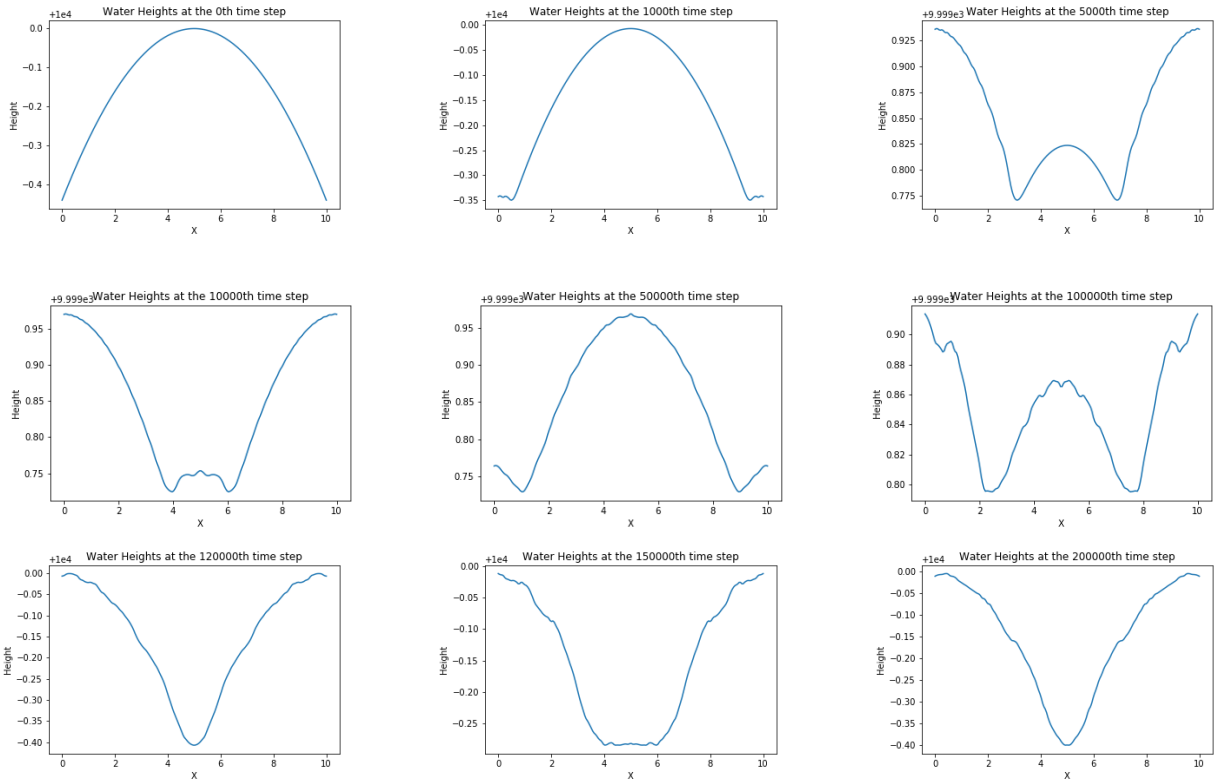


Figure 9: The results of Adams-Bashforth Method with Gaussian Initial conditions

But the RK method uses k values which are the increments based on the slopes at different points in the finite difference interval, shown in the following formulas.
The number of k levels leads to different RK methods. If 4 k values are used, then the RK method is called RK4 method. In general, the RK4 formulas can be written as:

$$\begin{aligned} k_1 &= \Delta t F(y^{(n)}) \\ k_2 &= \Delta t F(y^{(n)} + \frac{1}{2}k_1) \\ k_3 &= \Delta t F(y^{(n)} + \frac{1}{2}k_2) \\ k_4 &= \Delta t F(y^{(n)} + k_3) \\ y^{(n+1)} &= y^{(n)} + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4) \end{aligned} \tag{18}$$

where k_i is the ith level k value;
y^n is the value of y at the nth time grid;
Δt is the time scheme unit;

If only k_1 and k_2 are used, then it is the 2nd order RK method, i.e. RK2 method, whose formulas are shown as the following.

$$\begin{aligned} k_1 &= \Delta t F(y^{(n)}) \\ k_2 &= \Delta t F(y^{(n)} + \frac{1}{2}k_1) \\ y^{(n+1)} &= y^{(n)} + k_2 \end{aligned} \tag{19}$$

Specifically, by using RK2 on time-stepping and forward Euler on space-stepping to solve SWE in Equation 3, we have intermediate steps as:

$$\begin{aligned} k_1 &= (-g\frac{h_j^i - h_{j-1}^i}{\Delta x} + fv_j^i)\Delta t \\ l_1 &= -fu_j^i \Delta t \\ m_1 &= -H\Delta t \frac{h_j^i - h_{j-1}^i}{\Delta x} \\ k_2 &= (-g\frac{h_j^i - h_{j-1}^i}{\Delta x} + fv_j^i + 0.5l_1)\Delta t \\ l_2 &= -f(u_j^i + 0.5k_1)\Delta t \\ m_2 &= m_1 \end{aligned} \tag{20}$$

Thus,

$$\begin{aligned} h_j^{(i+1)} &= h_j^{(i)} + \frac{1}{2}(m_1 + m_2) \\ u_j^{(i+1)} &= u_j^{(i)} + \frac{1}{2}(k_1 + k_2) \\ v_j^{(i+1)} &= v_j^{(i)} + \frac{1}{2}(l_1 + l_2) \end{aligned} \tag{21}$$

where i is the time grid index, and j is the space grid index.

It is worth noting that, the problem with a square wave as the initial condition cannot be solved by the RK4 method, because the corners of step functions do not have derivatives, or we can say the the corner curves have infinite derivatives, which will lead to the blow-ups of values.

When we try using the Gaussian wave as the initial condition, the RK4 method still has value blow-ups in the long time period. We guess it may be because the RK4 method has 4 k values, and errors may happen in each step of calculating k. Since the value of k2 depends on k1, the value of k3 depends on k2, and the value of k4 depends on k3, thus the calculation errors are accumulated when in the process of calculating the 4 k values. From the results of this project, the RK4 method is very sensitive to initial inputs, and is prone to amplify the errors. In this project, not only different initial conditions, we also tried many different time and space step sizes, but did not find the proper time and space scheming yet. If possible, in the future research work, We can study on the mathematical principles behind the RK4 method more deeply, and try to find the suitable finite difference scheming of SWE for RK4 method.

3 Applications

The finite difference methods for the SWE can be applied to solve many different problems, such as the status analysis of cloud layers, the movements of ocean currents, and the flow conditions of rivers [12]. Through the above analysis of different numerical methods for SWE, we found that the leapfrog central difference method is the most stable one, and the Adams-Bashforth method is also a good alternative. RK4 is sensitive to initial conditions, and prone to produce blow-up values.

In this section, we use the leapfrog central difference method to simulate a pond with continuous water input from a waterfall. We simplify the waterfall model by a continuous rectangular flow of water from outside. As shown in Figure 10, the waterfall produces larger and larger waves as more and more water are poured into the pond. The Figure 11 shows the plane graphs of waves in the pond, and the 2nd graph shows an obviously fast water diffusion due to the falling of water. The video shows the whole process of water movement (video link:https://drive.google.com/file/d/1v9g_Tk4lNE5b-eJcu5ZJrseK55JcjPID/view?usp=sharing).

4 Conclusions and Discussion

In this project, we use four classic numerical differential methods to find accurate and stable solutions for shallow water models. We expand the finite difference expressions from 1-D simplified linear shallow water equations, to 2-D complete shallow water equations. We code to simulate a shallow water model in a large basin with a waterfall and draw the following

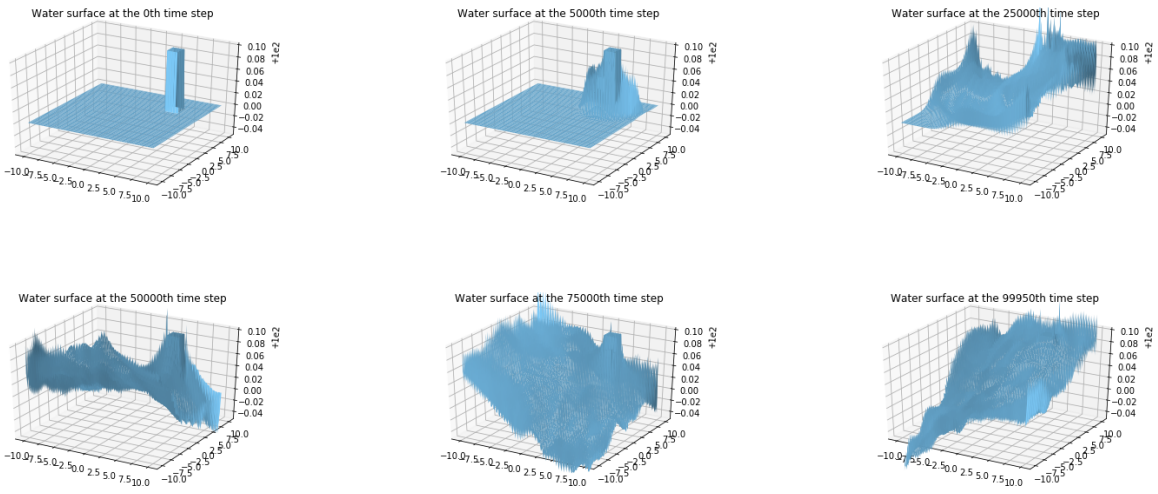


Figure 10: The 2-D Simulation of a Pond with Waterfall

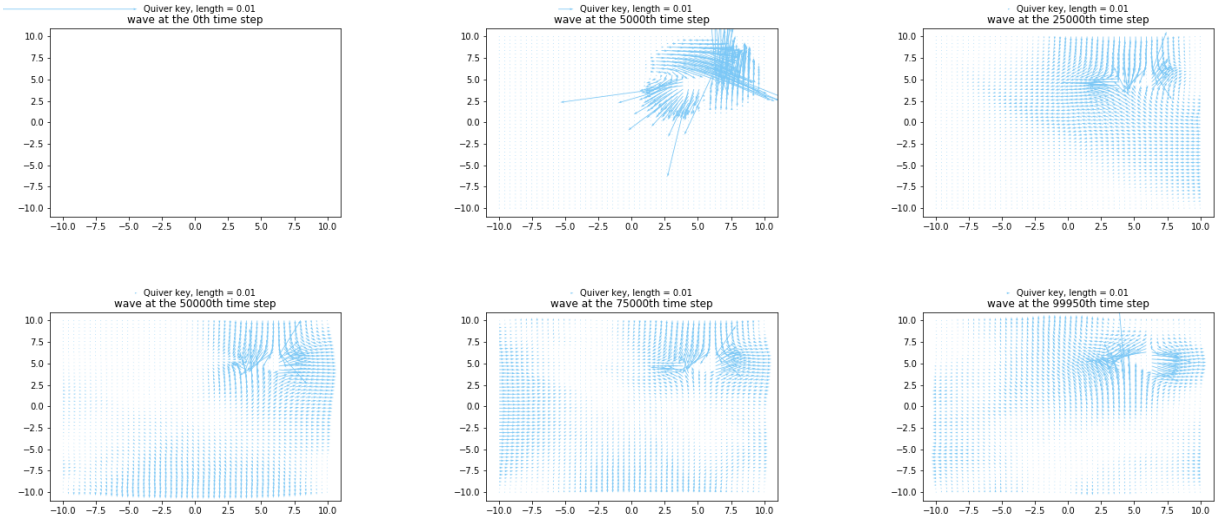


Figure 11: The Plane Graphs of Waves in a Pond with Waterfall

conclusions:

1. The Forward Euler method is a simple and straightforward method [10], but it is unstable for hyperbolic PDEs because the Fourier Stability Analysis shows that Fourier modes are amplified in most wave number using this method. So even though the initial condition satisfies CFL condition, the long-term simulation explodes.

2. Leapfrog Central Difference method is a simple and stable method for the shallow water equations model [10]. We can use Fourier Stability Analysis to prove the stability [10]. The water movement are perfectly shown in our figures and videos using this method: Begin with a little perturbation, the water wave propagates towards two sides with lower height, then it comes back with different patterns if we use different boundary conditions. The initial condition also has a great effect on this method. For example, the square wave perturbation will lead to a rough and serrated simulation.

3. Adams-Bashforth method (AB method) is an explicit method in the 2nd order of accuracy regards time scheming [3]. The water height changes are shown in our figures using the method of spatial central finite difference and timing AB method. AB method is very sensitive to time step sizes since it uses the derivative of heights and velocities with respect to time. If the time step size is too large and the heights or velocities vary too much, then the finite difference method with respect to time may give the results far away from the real time derivative. In this project, if the time step $\Delta t > 10$, the results of the AB method will blow up. The less the time step is, the more accurate the result is. It is also sensitive to boundary conditions, and it does not work for Dirichlet conditions.

4. Runge-Kutta 4 method is an explicit method in the 4th order of accuracy regards time scheming [11]. Similar to AB method, it is also sensitive to time step sizes and boundary conditions. In this project, we tried both Dirichlet boundary condition and periodic boundary condition and many different ways of time and step scheming, but the RK4 method failed. We guess it may be because we did not find the appropriate boundary conditions or correct scheming methods.

5. The energy is conserved in the ideal shallow water model without considering frictions, so the viscous force is important for energy dissipation in real-world shallow water models. Compared to the model without viscosity, the model with viscosity has the waves of which the oscillation amplitude decreases with time. This is because the total energy is dissipated due to the viscous force. Besides, our simulation represents a perfect transformation between Kinetic Energy and Potential Energy.

The above shallow water results have many applications. They can be applied to circular or rectangular shallow pools. They can also be applied to the cases of shallow rivers with different shapes of river beds. It can also be applied to simulate Waterfall if we initialize the water heights as continuous water walls, as shown above. The SWE are the foundation of many ecological models which have similar features to shallow waters. Therefore, the study on the finite difference method of SWE is critical for ecological and environmental research. With the finite difference method mentioned in this project, it is expected that people can solve complicated SWE under different conditions accurately and efficiently.

References

[1] Ashyralyyev, C. and Dedeturk, M. (2013). A finite difference method for the inverse elliptic problem with the dirichlet condition. *Contemporary Analysis and Applied Mathematics*, 1(2):132–155.

[2] Didenkulova, I. and Pelinovsky, E. (2011). Rogue waves in nonlinear hyperbolic systems (shallow-water framework). *Nonlinearity*, 24(3):R1.

[3] Durran, D. R. (1991). The third-order adams-bashforth method: An attractive alternative to leapfrog time differencing. *Monthly weather review*, 119(3):702–720.

[4] Harms, P., Mittra, R., and Ko, W. (1994). Implementation of the periodic boundary condition in the finite-difference time-domain algorithm for fss structures. *IEEE Transactions on Antennas and Propagation*, 42(9):1317–1324.

[5] Kalnay, E. (2003). *Atmospheric modeling, data assimilation and predictability*. Cambridge university press.

[6] Kato, T. and Tanabe, H. (1962). On the abstract evolution equation. *Osaka Mathematical Journal*, 14(1):107–133.

[7] Kinnmark, I. (2012). *The shallow water wave equations: formulation, analysis and application*, volume 15. Springer Science & Business Media.

[8] Liggett, J. A. and Woolhiser, D. A. (1967). *Difference solutions of the shallow-water equation*. Cornell University Water Resources Center.

[9] Lindborg, E. and Mohanan, A. V. (2017). A two-dimensional toy model for geophysical turbulence. *Physics of fluids*, 29(11):111114.

[10] Rycroft, C. H. (2019a). Harvard AM 205, lecture notes 12: Runge Kutta. URL: http://iacs-courses.seas.harvard.edu/courses/am205/slides/am205_lec15.pdf. Last visited on 2019/12/23.

[11] Rycroft, C. H. (2019b). Harvard AM 205, lecture notes 12: Runge Kutta. URL: http://iacs-courses.seas.harvard.edu/courses/am205/slides/am205_lec12.pdf. Last visited on 2019/12/23.

[12] Vallis, G. K. (2017). *Atmospheric and oceanic fluid dynamics*. Cambridge University Press.

[13] Walko, R. L. and Avissar, R. (2008). The ocean–land–atmosphere model (olam). part i: Shallow-water tests. *Monthly Weather Review*, 136(11):4033–4044.

[14] Wicker, L. J. (2009). A two-step adams–bashforth–moulton split-explicit integrator for compressible atmospheric models. *Monthly Weather Review*, 137(10):3588–3595.

[15] Wikipedia contributors (2019). Leapfrog integration — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Leapfrog_integration&oldid=914962842. [Online; accessed 30-December-2019].

Appendix A The attached compressed file contains all the pictures and videos shown in this report.

Appendix B The following are all the codes used to implement this project.

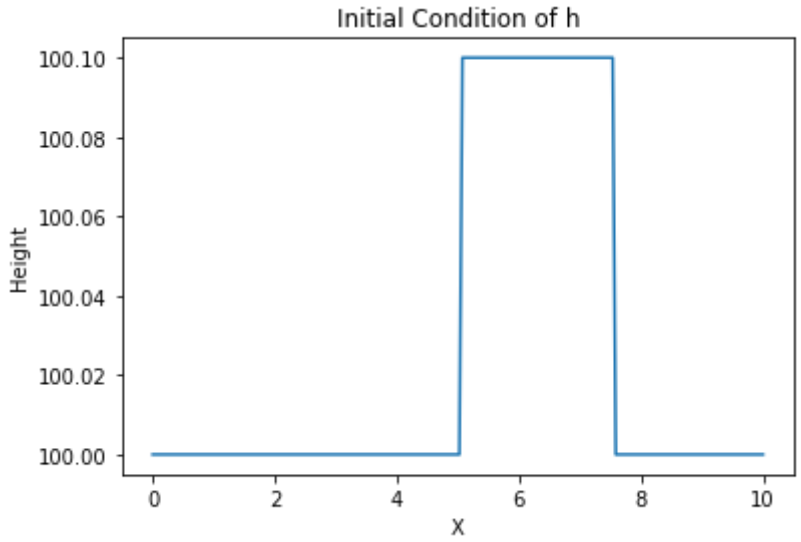
In [2]:
```
import numpy as np
import math
import copy
from  matplotlib import pyplot as plt
from matplotlib import animation
from IPython.display import HTML
```

AM205 - Final Project

Initial Conditions of 1D

In [7]:
```
# initialization
def initial_1d():
    L = 10**6
    N = 200  # grid number in x axis

    del_t = 10
    del_s = L/N
    T = 500000
    itr = int(T/del_t)

    f = 0.0001 # Coriolis force
    g = 9.8

    u = np.zeros((N,itr),dtype='float')
    v = np.zeros((N,itr),dtype='float')
    h = np.zeros((N,itr),dtype='float')

    for i in range(N):
        if 100 <i <= 150:
            h[i,0] = 100.1
        else:
            h[i,0] = 100
    H = np.mean(h[:,0],axis=0)
    return H, N, del_t, del_s, itr, f, g, u, v, h
```

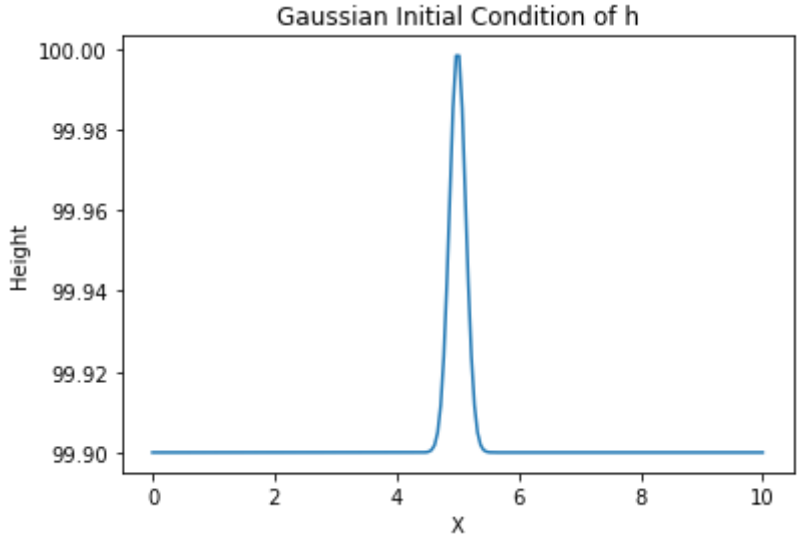
In [4]:
```
H, N, del_t, del_s, itr, f, g, u_ini, v_ini, h = initial_1d()
plt.plot(np.linspace(0,10,N),h[:,0])
plt.title('Initial Condition of h')
plt.xlabel('X')
plt.ylabel('Height')
plt.savefig('./pictures/initial_h_rec.png')
```



In [5]:

```
#guassian
def gaussian(x, mu, sig):
    return np.exp(-np.power(x - mu, 2.) / (2 * np.power(sig, 2.)))

def initial_guassian_1d():
    L = 10**6
    N = 200  # grid number in x axis

    del_t = 10
    del_s = L/N
    T = 500000
    itr = int(T/del_t)

    f = 0.0001 # Coriolis force
    g = 9.8

    u = np.zeros((N,itr),dtype='float')
    v = np.zeros((N,itr),dtype='float')
    h = np.zeros((N,itr),dtype='float')

#     for i in range(N):
#         if 100 <i <= 150:
#             h[i,0] = 100.1
#         else:
#             h[i,0] = 100
    x_values = np.linspace(-3, 3, N)
    y_values = gaussian(x_values, 0, 0.08)*0.1 + 99.9
    for i in range(N):
        h[i,0] = y_values[i]
    H = np.mean(h[:,0],axis=0)
    return H, N, del_t, del_s, itr, f, g, u, v, h
```

In [21]:
```
H, N, del_t, del_s, itr, f, g, u_ini, v_ini, h = initial_guassian_1d()
plt.plot(np.linspace(0,10,N),h[:,0])
plt.title('Gaussian Initial Condition of h')
plt.xlabel('X')
plt.ylabel('Height')
plt.savefig('./pictures/initial_h_gauss.png')
```

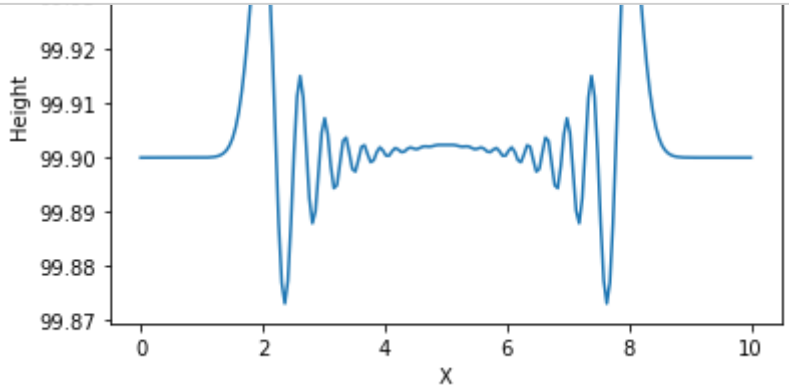


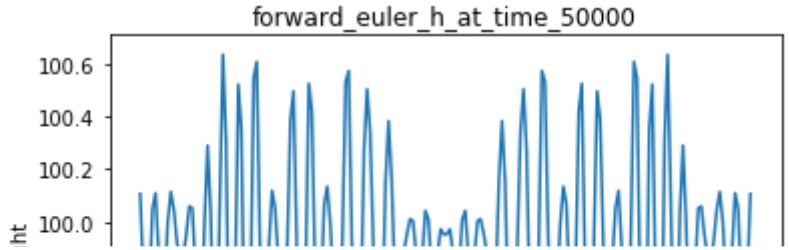
Forward Euler Method

In [61]:
```
def forward_central_scheme(H, N, del_t, del_s, itr, f, g, u_, v_, h_):
    h = copy.deepcopy(h_)
    u = copy.deepcopy(u_)
    v = copy.deepcopy(v_)
    for i in range(1,itr):
        for j in range(N):
            if j == 0:
                h[j,i] = del_t*(-H*(u[j+1,i-1]-u[-1,i-1])/(2*del_s)) + h[j,i-1]
                u[j,i] = del_t*(-g*(h[j+1,i-1]-h[-1,i-1])/(2*del_s) + f*v[j,i-1]) + u[j,i-1]
                v[j,i] = -f*u[j,i-1]*del_t + v[j,i-1]
            elif j == N-1:
                h[j,i] = del_t*(-H*(u[0,i-1]-u[j-1,i-1])/(2*del_s)) + h[j,i-1]
                u[j,i] = del_t*(-g*(h[0,i-1]-h[j-1,i-1])/(2*del_s) + f*v[j,i-1]) + u[j,i-1]
                v[j,i] = -f*u[j,i-1]*del_t + v[j,i-1]
            else:
                h[j,i] = del_t*(-H*(u[j+1,i-1]-u[j-1,i-1])/(2*del_s)) + h[j,i-1]
                u[j,i] = del_t*(-g*(h[j+1,i-1]-h[j-1,i-1])/(2*del_s) + f*v[j,i-1]) + u[j,i-1]
                v[j,i] = -f*u[j,i-1]*del_t + v[j,i-1]
    return u,v,h
```

In [ ]:
```
H, N, del_t, del_s, itr, f, g, u_ini, v_ini, h = initial_guassian_1d()
U,V,H = forward_central_scheme(H, N, del_t, del_s, itr, f, g, u_ini, v_ini, h)
```

In [39]:
```
temp = [0, 100, 500, 1000, 5000, 10000]
for i in temp:
    plt.figure()
    plt.plot(np.linspace(0,10,N),H[:,i], c =u'#1f77b4')
    plt.title('forward_euler_h_at_time_{}'.format(i*10))
    plt.xlabel('X')
    plt.ylabel('Height')
    plt.savefig('./pictures/forward_euler_h_at_time_{}.png'.format(i*10))
```





In [26]:
```
H.shape
```

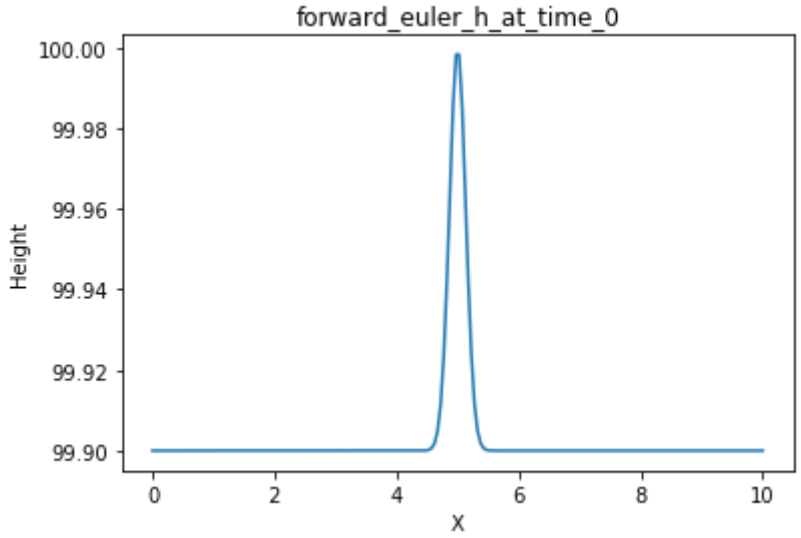
Out[26]:
```
(200, 50000)
```

In [44]:

```
fig, ax = plt.subplots()

x = np.linspace(0,10,N)
line, = ax.plot(x, H[:,0])


def init():  # only required for blitting to give a clean slate.
    line.set_ydata(H[:,0])
    ax.set_title('forward_euler_h_at_time_0')
    ax.set_xlabel('X')
    ax.set_ylabel('Height')
#     ax.set_ylim(99,101)
    return line,


def animate(i):
    line.set_ydata(H[:,i*20])  #  the data.
#     ax.set_ylim((99,101))
    ax.set_title('forward_euler_h_at_time_{}'.format(i*20*10))
    ax.set_xlabel('X')
    ax.set_ylabel('Height')
    return line,


ani = animation.FuncAnimation(
    fig, animate, init_func=init, frames = 500, interval=20, blit=False)

HTML(ani.to_html5_video())
```

Out[44]:

0:00


forward_euler_h_at_time_0
100.00
99.98
99.96
99.94
99.92
99.90
Height
0
2
4
6
8
10
X

Leapfrog Method in 1D without Horizontal Advection and Viscosity

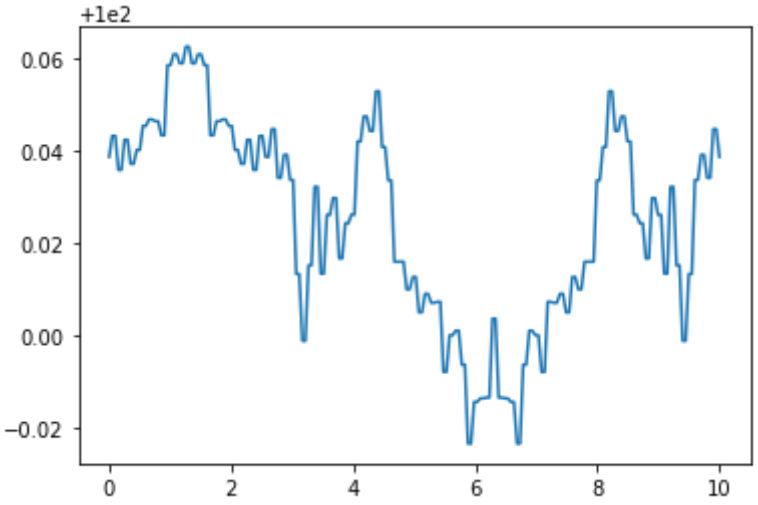
In [64]:

```
def leap_frog_central_scheme(H, N, del_t, del_s, itr, f, g, u_, v_, h_):
    h = copy.deepcopy(h_)
    u = copy.deepcopy(u_)
    v = copy.deepcopy(v_)
    for i in range(1,itr):
        if i == 1 : # forward difference
            for j in range(N):
                if j == 0:
                    h[j,i] = del_t*(-H*(u[j+1,i-1]-u[-1,i-1])/(2*del_s)) + h[j,i-1]
                    u[j,i] = del_t*(-g*(h[j+1,i-1]-h[-1,i-1])/(2*del_s) + f*v[j,i-1]) + u[j,i-1]
                    v[j,i] = -f*u[j,i-1]*del_t + v[j,i-1]
                elif j == N-1:
                    h[j,i] = del_t*(-H*(u[0,i-1]-u[j-1,i-1])/(2*del_s)) + h[j,i-1]
                    u[j,i] = del_t*(-g*(h[0,i-1]-h[j-1,i-1])/(2*del_s) + f*v[j,i-1]) + u[j,i-1]
                    v[j,i] = -f*u[j,i-1]*del_t + v[j,i-1]
                else:
                    h[j,i] = del_t*(-H*(u[j+1,i-1]-u[j-1,i-1])/(2*del_s)) + h[j,i-1]
                    u[j,i] = del_t*(-g*(h[j+1,i-1]-h[j-1,i-1])/(2*del_s) + f*v[j,i-1]) + u[j,i-1]
                    v[j,i] = -f*u[j,i-1]*del_t + v[j,i-1]
        else:
            for j in range(N):
                if j == 0: # central difference in time and space, conditionally stable
                    h[j,i] = 2*del_t*(-H*(u[j+1,i-1]-u[-1,i-1])/(2*del_s)) + h[j,i-2]
                    u[j,i] = 2*del_t*(-g*(h[j+1,i-1]-h[-1,i-1])/(2*del_s) + f*v[j,i-1]) + u[j,i-2]
                    v[j,i] = -f*u[j,i-1]*del_t*2 + v[j,i-2]
                elif j == N-1:
                    h[j,i] = 2*del_t*(-H*(u[0,i-1]-u[j-1,i-1])/(2*del_s)) + h[j,i-2]
                    u[j,i] = 2*del_t*(-g*(h[0,i-1]-h[j-1,i-1])/(2*del_s) + f*v[j,i-1]) + u[j,i-2]
                    v[j,i] = -f*u[j,i-1]*del_t*2 + v[j,i-2]
                else:
                    h[j,i] = 2*del_t*(-H*(u[j+1,i-1]-u[j-1,i-1])/(2*del_s)) + h[j,i-2]
                    u[j,i] = 2*del_t*(-g*(h[j+1,i-1]-h[j-1,i-1])/(2*del_s) + f*v[j,i-1]) + u[j,i-2]
                    v[j,i] = -f*u[j,i-1]*del_t*2 + v[j,i-2]
#             if i%5 == 0: # smooth
#                 h[:,i-1] = h[:,i-1] + 0.01*(h[:,i]-2*h[:,i-1]+h[:,i-2])
#                 u[:,i-1] = u[:,i-1] + 0.01*(u[:,i]-2*u[:,i-1]+u[:,i-2])
#                 v[:,i-1] = v[:,i-1] + 0.01*(v[:,i]-2*v[:,i-1]+v[:,i-2])

    return u,v,h
```

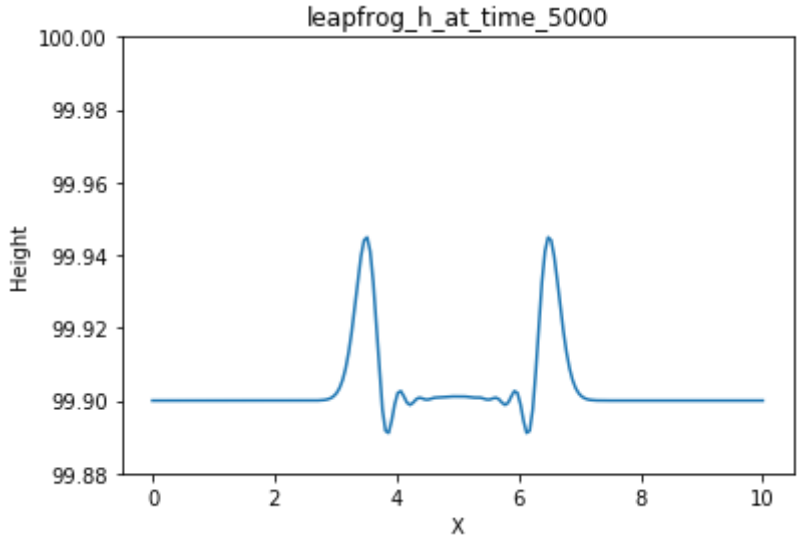
In [65]:
```
H, N, del_t, del_s, itr, f, g, u_ini, v_ini, h = initial_guassian_1d()
U,V,H = leap_frog_central_scheme(H, N, del_t, del_s, itr, f, g, u_ini, v_ini, h)
```

In [53]:
```
plt.plot(np.linspace(0,10,N),H[:,-1])
```

Out[53]:
```
[<matplotlib.lines.Line2D at 0x11d5f1ba8>]
```



In [67]:
```
temp = [0, 100, 500, 1000, 5000, 10000, 12000,15000, 20000]
for i in temp:
    plt.figure()
    plt.plot(np.linspace(0,10,N),H[:,i], c =u'#1f77b4')
    plt.title('leapfrog_h_at_time_{}'.format(i*10))
    plt.ylim((99.88,100.0))
    plt.xlabel('X')
    plt.ylabel('Height')
    plt.savefig('./pictures/leapfrog_h_at_time_{}.png'.format(i*10))
```



In [57]:
```
H.shape
```

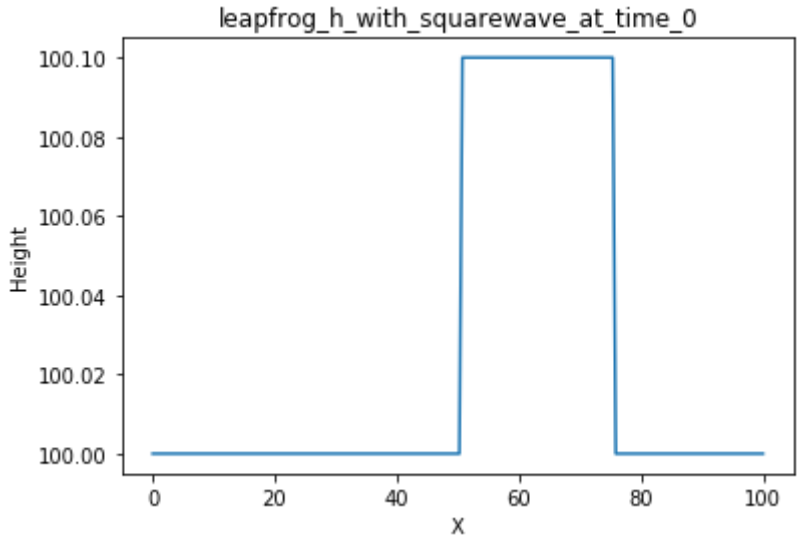
Out[57]:
```
(200, 50000)
```

In [59]:

```
fig, ax = plt.subplots()

x = np.linspace(0,100,N)
line, = ax.plot(x, H[:,0])



def init():  # only required for blitting to give a clean slate.
    line.set_ydata(H[:,0])
#     ax.set_ylim((99.5,100.5))
    ax.set_title('leapfrog_h_with_squarewave_at_time_0')
    ax.set_xlabel('X')
    ax.set_ylabel('Height')
    return line,


def animate(i):
    line.set_ydata(H[:,i*20])  #  the data.
#     ax.set_ylim((99.5,100.5))
    ax.set_title('leapfrog_h_with_squarewave_at_time_{}'.format(i*20*10))
    ax.set_xlabel('X')
    ax.set_ylabel('Height')
    return line,

ani = animation.FuncAnimation(
    fig, animate, init_func=init, frames = 500, interval=30, blit=False)

HTML(ani.to_html5_video())
```

Out[59]:

0:00


leapfrog_h_with_squarewave_at_time_0
100.10
100.08
100.06
100.04
100.02
100.00
Height
0
20
40
60
80
100
X

In [15]:

Leapfrog Method in 2D without Horizontal Advection and Viscosity

In [3]:
```
# initialization
def initial_2d():
    X = 10**6
    Y = 10**6
    N = 100  # grid number in

    del_t = 50
    del_x = X/N
    del_y = Y/N
    T = 100000

    itr = int(T/del_t)

    f = 0.0001 # Coriolis force
    g = 9.8

    h = np.zeros((N,N,itr),dtype='float')

    h[:,:,0] = 100
    h[70:80,70:80,0] = 100.1
    H = np.mean(h[:,:,0])
    return H, N, del_t, del_x, del_y, itr, f, g, h
```

In [4]:

```
def leap_frog_central_scheme_3d_zeroboundary(H, N, del_t, del_x, del_y, itr, f, g, h_):
    h = copy.deepcopy(h_)
    v = np.zeros((N,N,itr),dtype='float')
    u = np.zeros((N,N,itr),dtype='float')

    for t in range(1,itr):
        if t == 1 : # forward difference
            for i in range(N):
                for j in range(N):
                    if i == 0 and 0 < j < N-1: # boudary condition
                        h[i,j,t] = del_t*(-H*((u[i+1,j,t-1]-u[i,j,t-1])/(del_x) + (v[i,j+1,t-1]-v[i,j-1,
                        u[i,j,t] = 0
                        v[i,j,t] = (-f*u[i,j,t-1]-g*(h[i,j+1,t-1] - h[i,j-1,t-1])/(2*del_y))*del_t + v[i
                    elif j == N-1 and 0< i <N-1:
                        h[i,j,t] = del_t*(-H*((u[i+1,j,t-1]-u[i-1,j,t-1])/(2*del_x) + (v[i,j,t-1]-v[i,j-
                        u[i,j,t] = del_t*(-g*(h[i+1,j,t-1]-h[i-1,j,t-1])/(2*del_x) + f*v[i,j,t-1]) + u[i
                        v[i,j,t] = 0
                    elif i == N-1 and 0< j <N-1:
                        h[i,j,t] = del_t*(-H*((u[i,j,t-1]-u[i-1,j,t-1])/(del_x) + (v[i,j+1,t-1]-v[i,j-1,
                        u[i,j,t] = 0
                        v[i,j,t] = (-f*u[i,j,t-1]-g*(h[i,j+1,t-1] - h[i,j-1,t-1])/(2*del_y))*del_t + v[i
                    elif j ==0 and 0< i < N-1:
                        h[i,j,t] = del_t*(-H*((u[i+1,j,t-1]-u[i-1,j,t-1])/(2*del_x) + (v[i,j+1,t-1]-v[i,
                        u[i,j,t] = del_t*(-g*(h[i+1,j,t-1]-h[i-1,j,t-1])/(2*del_x) + f*v[i,j,t-1]) + u[i
                        v[i,j,t] = 0
                    elif (i==0 and j==0):
                        u[i,j,t] = 0
                        v[i,j,t] = 0
                        h[i,j,t] = del_t*(-H*((u[i+1,j,t-1]-u[i,j,t-1])/(del_x) + (v[i,j+1,t-1]-v[i,j,t-
                    elif (i==0 and j == N-1):
                        u[i,j,t] = 0
                        v[i,j,t] = 0
                        h[i,j,t] = del_t*(-H*((u[i+1,j,t-1]-u[i,j,t-1])/(del_x) + (v[i,j,t-1]-v[i,j-1,t-
                    elif (i==N-1 and j==0):
                        u[i,j,t] = 0
                        v[i,j,t] = 0
                        h[i,j,t] = del_t*(-H*((u[i,j,t-1]-u[i-1,j,t-1])/(del_x) + (v[i,j+1,t-1]-v[i,j,t-
                    elif (i ==N-1 and j==N-1):
                        u[i,j,t] = 0
                        v[i,j,t] = 0
                        h[i,j,t] = del_t*(-H*((u[i,j,t-1]-u[i-1,j,t-1])/(del_x) + (v[i,j,t-1]-v[i,j-1,t-
                    else:
```

```
                    h[i,j,t] = del_t*(-H*((u[i+1,j,t-1]-u[i-1,j,t-1])/(2*del_x) + (v[i,j+1,t-1]-v[i,
                    u[i,j,t] = del_t*(-g*(h[i+1,j,t-1]-h[i-1,j,t-1])/(2*del_x) + f*v[i,j,t-1]) + u[i
                    v[i,j,t] = (-f*u[i,j,t-1]-g*(h[i,j+1,t-1] - h[i,j-1,t-1])/(2*del_y))*del_t + v[i

        else:
            for i in range(N):
                for j in range(N):
                    if i == 0 and 0 < j < N-1: # boudary condition
                        h[i,j,t] = 2*del_t*(-H*((u[i+1,j,t-1]-u[i,j,t-1])/(del_x) + (v[i,j+1,t-1]-v[i,j-
                        u[i,j,t] = 0
                        v[i,j,t] = (-f*u[i,j,t-1]-g*(h[i,j+1,t-1] - h[i,j-1,t-1])/(2*del_y))*del_t*2 + v
                    elif j == N-1 and 0< i <N-1:
                        h[i,j,t] = 2*del_t*(-H*((u[i+1,j,t-1]-u[i-1,j,t-1])/(2*del_x) + (v[i,j,t-1]-v[i,
                        u[i,j,t] = 2*del_t*(-g*(h[i+1,j,t-1]-h[i-1,j,t-1])/(2*del_x) + f*v[i,j,t-1]) + u
                        v[i,j,t] = 0
                    elif i == N-1 and 0< j <N-1:
                        h[i,j,t] = 2*del_t*(-H*((u[i,j,t-1]-u[i-1,j,t-1])/(del_x) + (v[i,j+1,t-1]-v[i,j-
                        u[i,j,t] = 0
                        v[i,j,t] = (-f*u[i,j,t-1]-g*(h[i,j+1,t-1] - h[i,j-1,t-1])/(2*del_y))*del_t*2 + v
                    elif j ==0 and 0< i < N-1:
                        h[i,j,t] = 2*del_t*(-H*((u[i+1,j,t-1]-u[i-1,j,t-1])/(2*del_x) + (v[i,j+1,t-1]-v[
                        u[i,j,t] = 2*del_t*(-g*(h[i+1,j,t-1]-h[i-1,j,t-1])/(2*del_x) + f*v[i,j,t-1]) + u
                        v[i,j,t] = 0
                    elif (i==0 and j==0):
                        u[i,j,t] = 0
                        v[i,j,t] = 0
                        h[i,j,t] = 2*del_t*(-H*((u[i+1,j,t-1]-u[i,j,t-1])/(del_x) + (v[i,j+1,t-1]-v[i,j,
                    elif (i==0 and j == N-1):
                        u[i,j,t] = 0
                        v[i,j,t] = 0
                        h[i,j,t] = 2*del_t*(-H*((u[i+1,j,t-1]-u[i,j,t-1])/(del_x) + (v[i,j,t-1]-v[i,j-1,
                    elif (i==N-1 and j==0):
                        u[i,j,t] = 0
                        v[i,j,t] = 0
                        h[i,j,t] = 2*del_t*(-H*((u[i,j,t-1]-u[i-1,j,t-1])/(del_x) + (v[i,j+1,t-1]-v[i,j,
                    elif (i ==N-1 and j==N-1):
                        u[i,j,t] = 0
                        v[i,j,t] = 0
                        h[i,j,t] = 2*del_t*(-H*((u[i,j,t-1]-u[i-1,j,t-1])/(del_x) + (v[i,j,t-1]-v[i,j-1,
                    else:
                        h[i,j,t] = 2*del_t*(-H*((u[i+1,j,t-1]-u[i-1,j,t-1])/(2*del_x) + (v[i,j+1,t-1]-v[
                        u[i,j,t] = 2*del_t*(-g*(h[i+1,j,t-1]-h[i-1,j,t-1])/(2*del_x) + f*v[i,j,t-1]) + u
                        v[i,j,t] = (-f*u[i,j,t-1]-g*(h[i,j+1,t-1] - h[i,j-1,t-1])/(2*del_y))*del_t*2 + v
```

```
            if t%5 == 0: # smooth
                h[:,:,t-1] = h[:,:,t-1] + 0.01*(h[:,:,t]-2*h[:,:,t-1]+h[:,:,t-2])
                u[:,:,t-1] = u[:,:,t-1] + 0.01*(u[:,:,t]-2*u[:,:,t-1]+u[:,:,t-2])
                v[:,:,t-1] = v[:,:,t-1] + 0.01*(v[:,:,t]-2*v[:,:,t-1]+v[:,:,t-2])

    return u,v,h
```

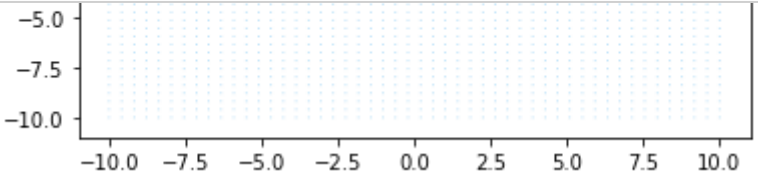
In [72]:
```
# U,V,H = leap_frog_central_scheme_3d(del_t,del_x,del_y,itr,u,v,h)
```

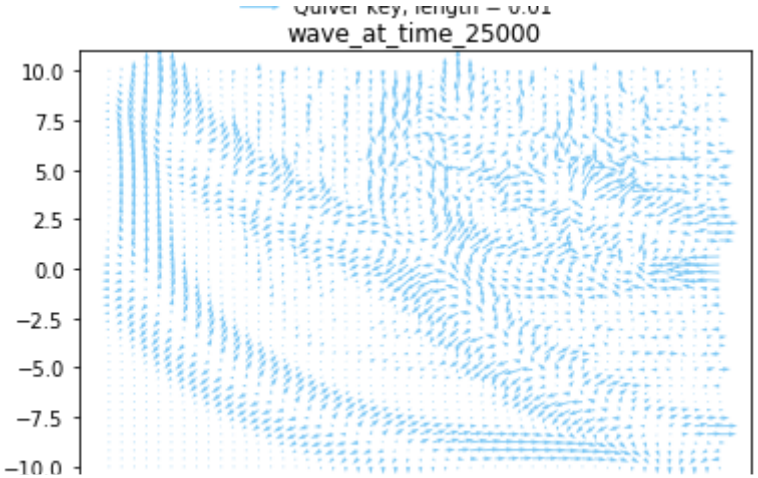
In [5]:
```
H, N, del_t, del_x, del_y, itr, f, g, h = initial_2d()
U,V,H = leap_frog_central_scheme_3d_zeroboundary(H, N, del_t, del_x, del_y, itr, f, g, h)
```

In [6]:
```
U[0:5:-1,0:5:-1,500]
```

Out[6]:
```
array([], shape=(0, 0), dtype=float64)
```

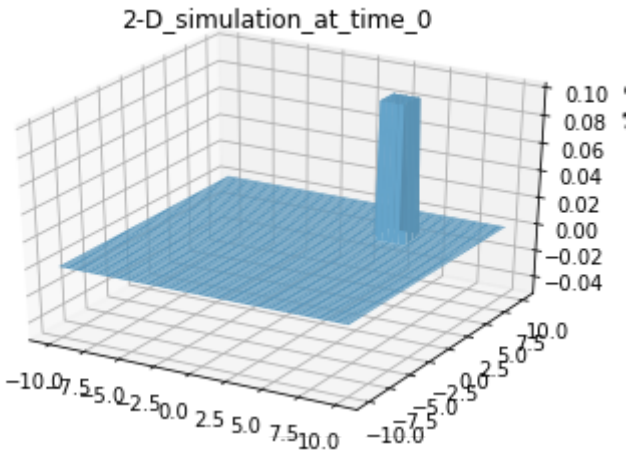
In [7]:
```
X = np.linspace(-10, 10, int(N/2))
Y = np.linspace(-10, 10, int(N/2))
X, Y = np.meshgrid(X, Y)

temp = [0, 100, 500, 1000, 1500, 1999]
for i in temp:
    fig, ax = plt.subplots()
    q = ax.quiver(X, Y, U[::2,::2,i], V[::2,::2,i],color = 'xkcd:lightblue')
    ax.quiverkey(q, X=0.3, Y=1.1, U=0.01,
                 label='Quiver key, length = 0.01', labelpos='E')
    ax.set_title('wave_at_time_{}'.format(i*50))
    plt.savefig('./pictures/wave_at_time_{}.png'.format(i*50))
```

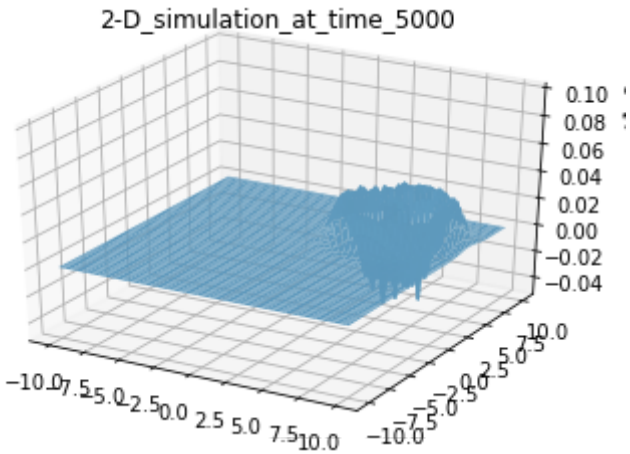


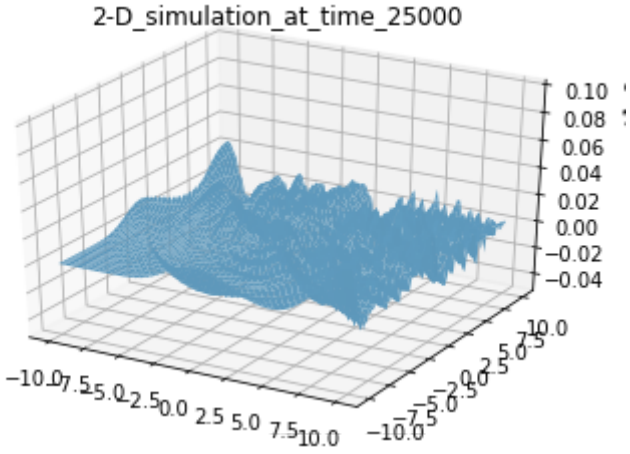


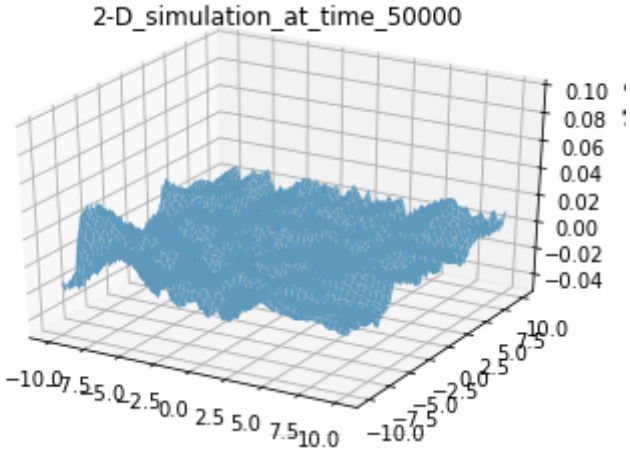
In [113]:

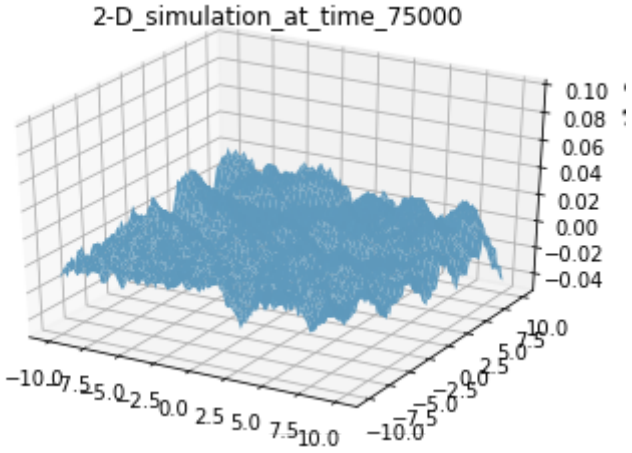
```
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.pyplot as plt
from matplotlib import cm

temp = [0, 100, 500, 1000, 1500, 1999]
for i in temp:
    fig = plt.figure()
    ax = fig.gca(projection='3d')

    # Make data.
    X = np.linspace(-10, 10, N)
    Y = np.linspace(-10, 10, N)
    X, Y = np.meshgrid(X, Y)
    # Plot the surface.
    ax.set_title('2-D_simulation_at_time_{}'.format(i*50))
    ax.set_zlim(99.95,100.1)
    surf = ax.plot_surface(X, Y, H[:,:,i],color = 'xkcd:lightblue')
    plt.savefig('./pictures/2-D_simulation_at_time_{}.png'.format(i*50))
```

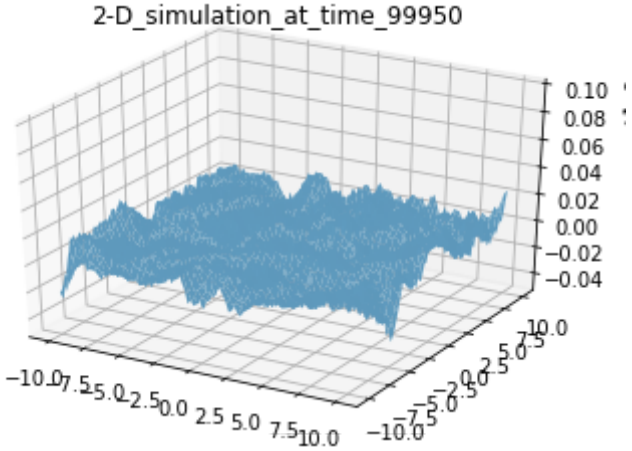



2-D_simulation_at_time_5000
0.10
0.08
0.06
0.04
0.02
0.00
−0.02
−0.04
−10.0
−7.5
−5.0
−2.5
0.0
2.5
5.0
7.5
10.0


2-D_simulation_at_time_25000
0.10
0.08
0.06
0.04
0.02
0.00
−0.02
−0.04
−10.0
−7.5
−5.0
−2.5
0.0
2.5
5.0
7.5
10.0


2-D_simulation_at_time_50000
0.10
0.08
0.06
0.04
0.02
0.00
−0.02
−0.04
−10.0
−7.5
−5.0
−2.5
0.0
2.5
5.0
7.5
10.0


2-D_simulation_at_time_75000
0.10
0.08
0.06
0.04
0.02
0.00
−0.02
−0.04
−10.0
−7.5
−5.0
−2.5
0.0
2.5
5.0
7.5
10.0

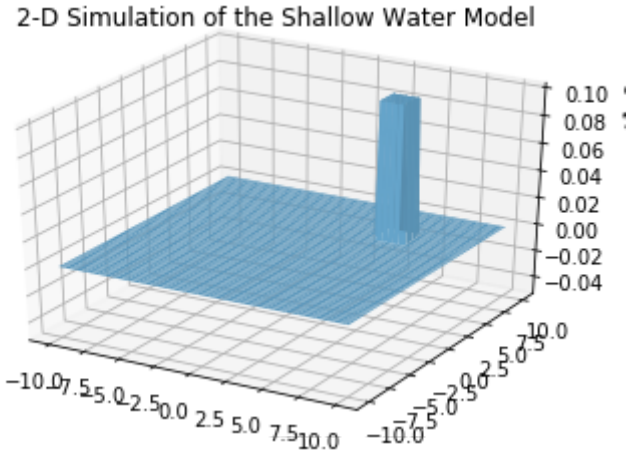

2-D_simulation_at_time_99950
0.10
0.08
0.06
0.04
0.02
0.00
−0.02
−0.04
−10.0
−7.5
−5.0
−2.5
0.0
2.5
5.0
7.5
10.0

In [129]:

```
fig = plt.figure()
ax = fig.gca(projection='3d')
ax.set_title('2-D Simulation of the Shallow Water Model')
# Make data.
X = np.linspace(-10, 10, N)
Y = np.linspace(-10, 10, N)
X, Y = np.meshgrid(X, Y)
# Plot the surface.

surf = None
def animate(i):
    global surf
    # If a line collection is already remove it before drawing.
    if surf:
        ax.collections.remove(surf)
    surf = ax.plot_surface(X, Y, H[:,:,i*5],color = 'xkcd:lightblue')
    ax.set_zlim(99.95,100.1)

    return surf,


ani = animation.FuncAnimation(
    fig, animate, frames = 399,interval=30, blit=False)

HTML(ani.to_html5_video())
```

Out[129]:

0:00



In [ ]:

Leapfrog Method in 2D with Horizontal Advection and Viscosity

In [55]:
```
# initialization
def initial_2d_v():
    X = 10**6
    Y = 10**6
    N = 100  # grid number in

    del_t = 50
    del_x = X/N
    del_y = Y/N
    T = 100000

    itr = int(T/del_t)

    f = 0.0001 # Coriolis force
    g = 9.8
    b = 1e-5
#     b = 0
    h = np.zeros((N,N,itr),dtype='float')

    h[:,:,0] = 100
    h[70:80,70:80,0] = 100.1
    H = np.mean(h[:,:,0])
    return H, N, del_t, del_x, del_y, itr, f, g, h, b
```

In [56]:
```
def leap_frog_central_scheme_3d_zeroboundary_advection(H, N, del_t, del_x, del_y, itr, f, g, h_, b):
    h = copy.deepcopy(h_)
    v = np.zeros((N,N,itr),dtype='float')
    u = np.zeros((N,N,itr),dtype='float')

    for t in range(1,itr):
        if t == 1 : # forward difference
            for i in range(N):
                for j in range(N):
                    if i == 0 and 0 < j < N-1: # boudary condition
                        v_y = (v[i,j+1,t-1]-v[i,j-1,t-1])/(2*del_y)
                        u_y = (u[i,j+1,t-1]-u[i,j-1,t-1])/(2*del_y)
                        h_y = (h[i,j+1,t-1]-h[i,j-1,t-1])/(2*del_y)
                        h[i,j,t] = (- u[i,j,t-1]*(h[i+1,j,t-1]-h[i,j,t-1])/(del_x)\
                                    - h[i,j,t-1]*(u[i+1,j,t-1]-u[i,j,t-1])/(del_x)\
                                    - v[i,j,t-1]*h_y - h[i,j,t-1]*v_y)*del_t\
                                    + h[i,j,t-1]
                        u[i,j,t] = 0
                        v[i,j,t] = (-f*u[i,j,t-1]-g*h_y-b*v[i,j,t-1]-u[i,j,t-1]*(v[i+1,j,t-1]-v[i,j,t-1]
                    elif j == N-1 and 0< i <N-1:
                        v_x = (v[i+1,j,t-1]-v[i-1,j,t-1])/(2*del_x)
                        u_x = (u[i+1,j,t-1]-u[i-1,j,t-1])/(2*del_x)
                        h_x = (h[i+1,j,t-1]-h[i-1,j,t-1])/(2*del_x)
                        h[i,j,t] = (-u[i,j,t-1]*h_x - h[i,j,t-1]*u_x - v[i,j,t-1]*(h[i,j,t-1]-h[i,j-1,t-1
                        u[i,j,t] = (f*v[i,j,t-1] - g*h_x - b*u[i,j,t-1] - u[i,j,t-1]*u_x - v[i,j,t-1]*(u
                        v[i,j,t] = 0
                    elif i == N-1 and 0< j <N-1:
                        v_y = (v[i,j+1,t-1]-v[i,j-1,t-1])/(2*del_y)
                        u_y = (u[i,j+1,t-1]-u[i,j-1,t-1])/(2*del_y)
                        h_y = (h[i,j+1,t-1]-h[i,j-1,t-1])/(2*del_y)
                        h[i,j,t] = (-u[i,j,t-1]*(h[i,j,t-1]-h[i-1,j,t-1])/(del_x) - h[i,j,t-1]*(u[i,j,t-1
                        u[i,j,t] = 0
                        v[i,j,t] = (-f*u[i,j,t-1]-g*h_y-b*v[i,j,t-1]-u[i,j,t-1]*(v[i,j,t-1]-v[i-1,j,t-1]
                    elif j ==0 and 0< i < N-1:
                        v_x = (v[i+1,j,t-1]-v[i-1,j,t-1])/(2*del_x)
                        u_x = (u[i+1,j,t-1]-u[i-1,j,t-1])/(2*del_x)
                        h_x = (h[i+1,j,t-1]-h[i-1,j,t-1])/(2*del_x)
                        h[i,j,t] = (-u[i,j,t-1]*h_x - h[i,j,t-1]*u_x - v[i,j,t-1]*(h[i,j+1,t-1]-h[i,j,t-1
                        u[i,j,t] = (f*v[i,j,t-1] - g*h_x - b*u[i,j,t-1] - u[i,j,t-1]*u_x - v[i,j,t-1]*(u
                        v[i,j,t] = 0
                    elif (i==0 and j==0):
                        u[i,j,t] = 0
```

```
                    v[i,j,t] = 0
                    h[i,j,t] = (-u[i,j,t-1]*(h[i+1,j,t-1]-h[i,j,t-1])/(del_x) - h[i,j,t-1]*(u[i+1,j,t
                elif (i==0 and j == N-1):
                    u[i,j,t] = 0
                    v[i,j,t] = 0
                    h[i,j,t] = (-u[i,j,t-1]*(h[i+1,j,t-1]-h[i,j,t-1])/(del_x) - h[i,j,t-1]*(u[i+1,j,t
                elif (i==N-1 and j==0):
                    u[i,j,t] = 0
                    v[i,j,t] = 0
                    h[i,j,t] = (-u[i,j,t-1]*(h[i,j,t-1]-h[i-1,j,t-1])/(del_x) - h[i,j,t-1]*(u[i,j,t-1
                elif (i ==N-1 and j==N-1):
                    u[i,j,t] = 0
                    v[i,j,t] = 0
                    h[i,j,t] = (-u[i,j,t-1]*(h[i,j,t-1]-h[i-1,j,t-1])/(del_x) - h[i,j,t-1]*(u[i,j,t-1
                else:
                    h_x = (h[i+1,j,t-1]-h[i-1,j,t-1])/(2*del_x)
                    h_y = (h[i,j+1,t-1]-h[i,j-1,t-1])/(2*del_y)
                    u_x = (u[i+1,j,t-1]-u[i-1,j,t-1])/(2*del_x)
                    u_y = (u[i,j+1,t-1]-u[i,j-1,t-1])/(2*del_y)
                    v_x = (v[i+1,j,t-1]-v[i-1,j,t-1])/(2*del_x)
                    v_y = (v[i,j+1,t-1]-v[i,j-1,t-1])/(2*del_y)
                    h[i,j,t] = (-u[i,j,t-1]*h_x - h[i,j,t-1]*u_x - v[i,j,t-1]*h_y - h[i,j,t-1]*v_y)*d
                    u[i,j,t] = (f*v[i,j,t-1] - g*h_x - b*u[i,j,t-1] - u[i,j,t-1]*u_x - v[i,j,t-1]*u_y
                    v[i,j,t] = (-f*u[i,j,t-1]-g*h_y-b*v[i,j,t-1]-u[i,j,t-1]*v_x-v[i,j,t-1]*v_y)*del_t

        else:
            for i in range(N):
                for j in range(N):
                    if i == 0 and 0 < j < N-1: # boudary condition
                        v_y = (v[i,j+1,t-1]-v[i,j-1,t-1])/(2*del_y)
                        u_y = (u[i,j+1,t-1]-u[i,j-1,t-1])/(2*del_y)
                        h_y = (h[i,j+1,t-1]-h[i,j-1,t-1])/(2*del_y)
                        h[i,j,t] = (- u[i,j,t-1]*(h[i+1,j,t-1]-h[i,j,t-1])/(del_x) \
                                    - h[i,j,t-1]*(u[i+1,j,t-1]-u[i,j,t-1])/(del_x) \
                                    - v[i,j,t-1]*h_y \
                                    - h[i,j,t-1]*v_y)*2*del_t \
                                    + h[i,j,t-2]
                        u[i,j,t] = 0
                        v[i,j,t] = ( - f*u[i,j,t-1] - g*h_y - b*v[i,j,t-1] - u[i,j,t-1]*(v[i+1,j,t-1] - v
                    elif j == N-1 and 0< i <N-1:
                        v_x = (v[i+1,j,t-1]-v[i-1,j,t-1])/(2*del_x)
                        u_x = (u[i+1,j,t-1]-u[i-1,j,t-1])/(2*del_x)
                        h_x = (h[i+1,j,t-1]-h[i-1,j,t-1])/(2*del_x)
```

```
            h[i,j,t] = (- u[i,j,t-1]*h_x \
                       - h[i,j,t-1]*u_x \
                       - v[i,j,t-1]*(h[i,j,t-1]-h[i,j-1,t-1])/(del_y) \
                       - h[i,j,t-1]*(v[i,j,t-1]-v[i,j-1,t-1])/(del_y))*2*del_t \
                       + h[i,j,t-2]
            u[i,j,t] = (   f*v[i,j,t-1] - g*h_x - b*u[i,j,t-1] - u[i,j,t-1]*u_x - v[i,j,t-1]
            v[i,j,t] = 0
        elif i == N-1 and 0< j <N-1:
            v_y = (v[i,j+1,t-1]-v[i,j-1,t-1])/(2*del_y)
            u_y = (u[i,j+1,t-1]-u[i,j-1,t-1])/(2*del_y)
            h_y = (h[i,j+1,t-1]-h[i,j-1,t-1])/(2*del_y)
            h[i,j,t] = (-u[i,j,t-1]*(h[i,j,t-1]-h[i-1,j,t-1])/(del_x) - h[i,j,t-1]*(u[i,j,t-1
            u[i,j,t] = 0
            v[i,j,t] = (-f*u[i,j,t-1]-g*h_y-b*v[i,j,t-1]-u[i,j,t-1]*(v[i,j,t-1]-v[i-1,j,t-1]
        elif j ==0 and 0< i < N-1:
            v_x = (v[i+1,j,t-1]-v[i-1,j,t-1])/(2*del_x)
            u_x = (u[i+1,j,t-1]-u[i-1,j,t-1])/(2*del_x)
            h_x = (h[i+1,j,t-1]-h[i-1,j,t-1])/(2*del_x)
            h[i,j,t] = (-u[i,j,t-1]*h_x - h[i,j,t-1]*u_x - v[i,j,t-1]*(h[i,j+1,t-1]-h[i,j,t-1
            u[i,j,t] = (f*v[i,j,t-1] - g*h_x - b*u[i,j,t-1] - u[i,j,t-1]*u_x - v[i,j,t-1]*(u
            v[i,j,t] = 0
        elif (i==0 and j==0):
            u[i,j,t] = 0
            v[i,j,t] = 0
            h[i,j,t] = (-u[i,j,t-1]*(h[i+1,j,t-1]-h[i,j,t-1])/(del_x) - h[i,j,t-1]*(u[i+1,j,t
        elif (i==0 and j == N-1):
            u[i,j,t] = 0
            v[i,j,t] = 0
            h[i,j,t] = (-u[i,j,t-1]*(h[i+1,j,t-1]-h[i,j,t-1])/(del_x) - h[i,j,t-1]*(u[i+1,j,t
        elif (i==N-1 and j==0):
            u[i,j,t] = 0
            v[i,j,t] = 0
            h[i,j,t] = (-u[i,j,t-1]*(h[i,j,t-1]-h[i-1,j,t-1])/(del_x) - h[i,j,t-1]*(u[i,j,t-1
        elif (i ==N-1 and j==N-1):
            u[i,j,t] = 0
            v[i,j,t] = 0
            h[i,j,t] = (-u[i,j,t-1]*(h[i,j,t-1]-h[i-1,j,t-1])/(del_x) - h[i,j,t-1]*(u[i,j,t-1
        else:
            h_x = (h[i+1,j,t-1]-h[i-1,j,t-1])/(2*del_x)
            h_y = (h[i,j+1,t-1]-h[i,j-1,t-1])/(2*del_y)
            u_x = (u[i+1,j,t-1]-u[i-1,j,t-1])/(2*del_x)
            u_y = (u[i,j+1,t-1]-u[i,j-1,t-1])/(2*del_y)
            v_x = (v[i+1,j,t-1]-v[i-1,j,t-1])/(2*del_x)
```

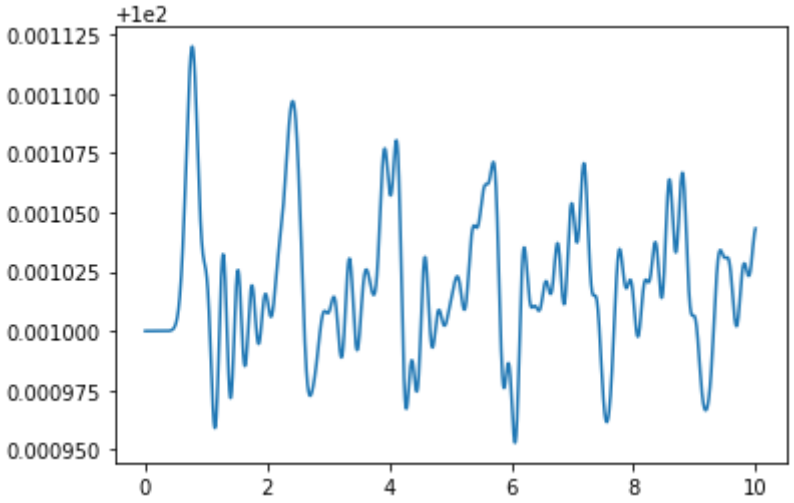
```
                    v_y = (v[i,j+1,t-1]-v[i,j-1,t-1])/(2*del_y)
                    h[i,j,t] = (- u[i,j,t-1]*h_x - h[i,j,t-1]*u_x - v[i,j,t-1]*h_y - h[i,j,t-1]*v_y)
                    u[i,j,t] = (  f*v[i,j,t-1] - g*h_x - b*u[i,j,t-1] - u[i,j,t-1]*u_x - v[i,j,t-1]*u
                    v[i,j,t] = (- f*u[i,j,t-1] - g*h_y - b*v[i,j,t-1] - u[i,j,t-1]*v_x - v[i,j,t-1]*v

            if t%5 == 0: # smooth
                h[:,:,t-1] = h[:,:,t-1] + 0.01*(h[:,:,t]-2*h[:,:,t-1]+h[:,:,t-2])
                u[:,:,t-1] = u[:,:,t-1] + 0.01*(u[:,:,t]-2*u[:,:,t-1]+u[:,:,t-2])
                v[:,:,t-1] = v[:,:,t-1] + 0.01*(v[:,:,t]-2*v[:,:,t-1]+v[:,:,t-2])

    return u,v,h
```

In [57]:
```
H, N, del_t, del_x, del_y, itr, f, g, h, b = initial_2d_v()
U,V,H = leap_frog_central_scheme_3d_zeroboundary_advection(H, N, del_t, del_x, del_y, itr, f, g, h, b)
```

In [46]:
```
H, N, del_t, del_x, del_y, itr, f, g, h, b = initial_2d_v()
U_,V_,H_ = leap_frog_central_scheme_3d_zeroboundary_advection(H, N, del_t, del_x, del_y, itr, f, g, h, b
```

In [61]:
```
plt.plot(np.linspace(0,10,2000), np.mean(H,axis=(0,1)))
```

Out[61]:
```
[<matplotlib.lines.Line2D at 0x1134a6908>]
```



In [67]:
```
## calculate total energy
def total_energy(H,U,V):
    H2 = H**2
    K = np.sum(H*(U**2 + V**2)/2, axis = (0,1))
    P = np.sum(g*H2/2, axis = (0,1))- np.mean(H,axis=(0,1))**2*g/2*N*N
    return K,P
```

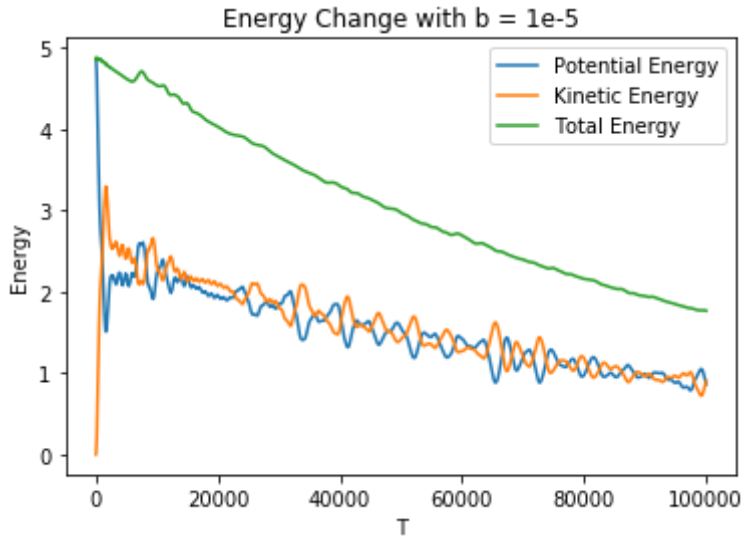
In [80]:
```
k_with_vis, p_with_vis = total_energy(H,U,V)
```

In [ ]:
```
```

In [71]:
```
H.shape
```

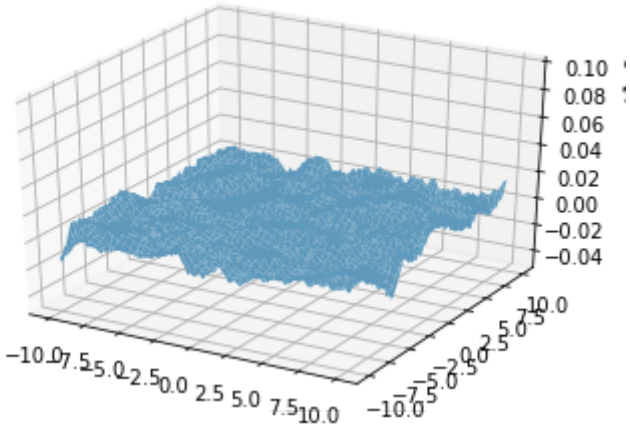
Out[71]:
```
(100, 100, 2000)
```

In [81]:
```
fig = plt.figure()
plt.plot(np.linspace(0,100000,2000),p_with_vis, label = 'Potential Energy')
plt.plot(np.linspace(0,100000,2000),k_with_vis, label='Kinetic Energy')
plt.plot(np.linspace(0,100000,2000),k_with_vis + p_with_vis, label='Total Energy')
plt.legend()
plt.xlabel('T')
plt.ylabel('Energy')
plt.title('Energy Change with b = 1e-5')
plt.savefig('./pictures/Energy_with_b=1e-5.png')
```

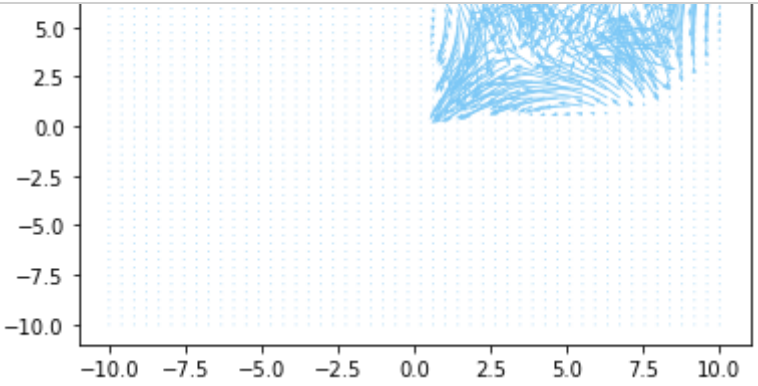


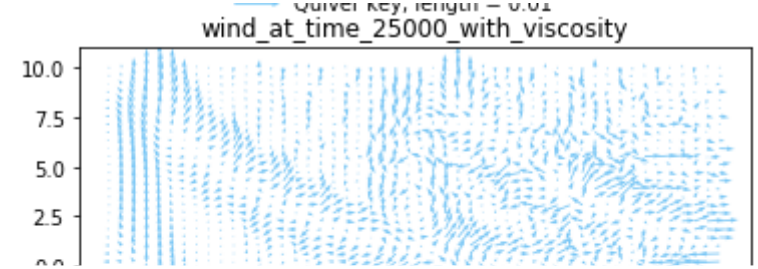
In [8]:

```
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.pyplot as plt
from matplotlib import cm


fig = plt.figure()
ax = fig.gca(projection='3d')

# Make data.
X = np.linspace(-10, 10, N)
Y = np.linspace(-10, 10, N)
X, Y = np.meshgrid(X, Y)
# Plot the surface.

surf = ax.plot_surface(X, Y, H[:,:,-1],color = 'xkcd:lightblue')
ax.set_zlim(99.95,100.1)
plt.show()
```



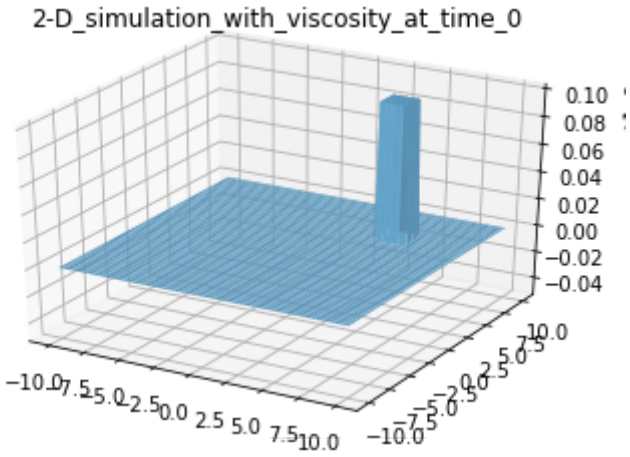
In [9]:
```
X = np.linspace(-10, 10, int(N/2))
Y = np.linspace(-10, 10, int(N/2))
X, Y = np.meshgrid(X, Y)

temp = [0, 100, 500, 1000, 1500, 1999]
for i in temp:
    fig, ax = plt.subplots()
    q = ax.quiver(X, Y, U[::2,::2,i], V[::2,::2,i],color = 'xkcd:lightblue')
    ax.quiverkey(q, X=0.3, Y=1.1, U=0.01,
                 label='Quiver key, length = 0.01', labelpos='E')
    ax.set_title('wind_at_time_{}_with_viscosity'.format(i*50))
    plt.savefig('./pictures/wind_at_time_{}_with_viscosity.png'.format(i*50))
```

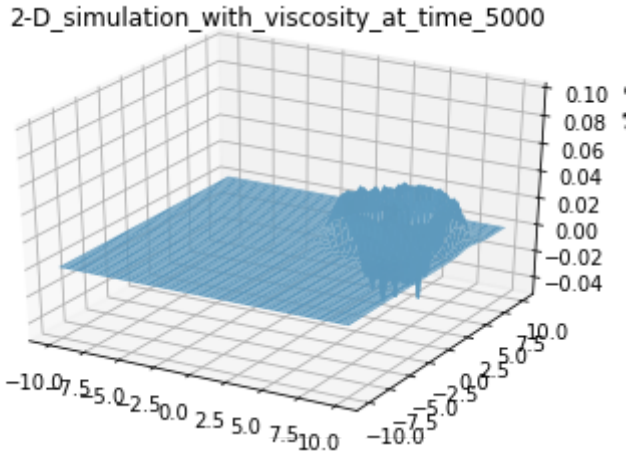


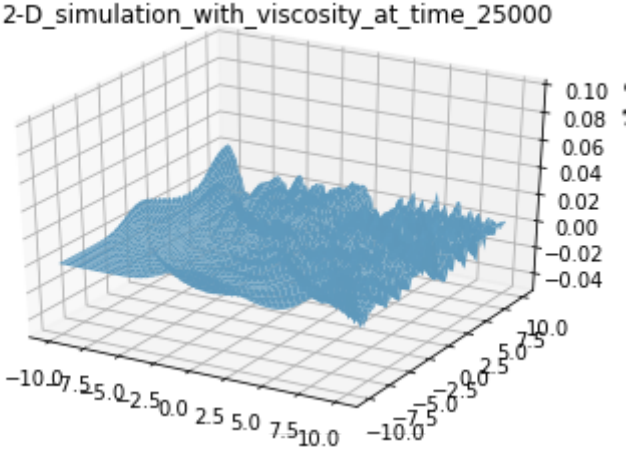


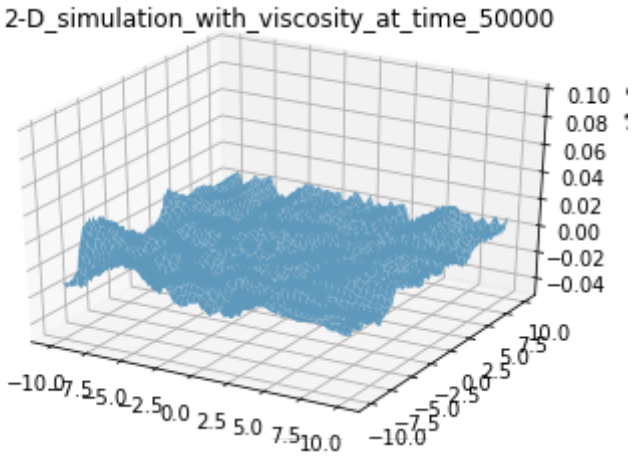
In [10]:

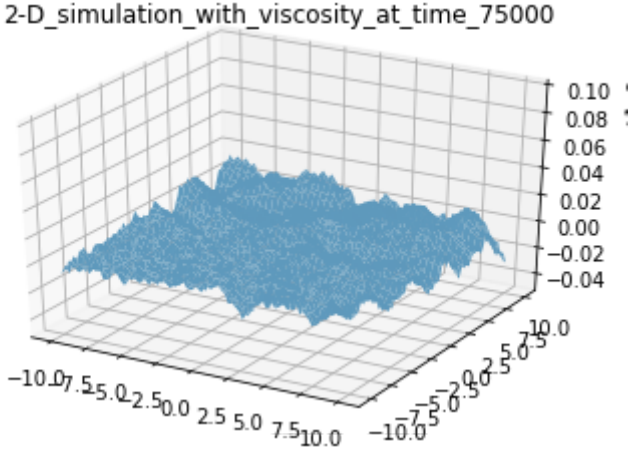
```
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.pyplot as plt
from matplotlib import cm

temp = [0, 100, 500, 1000, 1500, 1999]
for i in temp:
    fig = plt.figure()
    ax = fig.gca(projection='3d')

    # Make data.
    X = np.linspace(-10, 10, N)
    Y = np.linspace(-10, 10, N)
    X, Y = np.meshgrid(X, Y)
    # Plot the surface.
    ax.set_title('2-D_simulation_with_viscosity_at_time_{}'.format(i*50))
    ax.set_zlim(99.95,100.1)
    surf = ax.plot_surface(X, Y, H[:,:,i],color = 'xkcd:lightblue')
    plt.savefig('./pictures/2-D_simulation_with_viscosity_at_time_{}.png'.format(i*50))
```

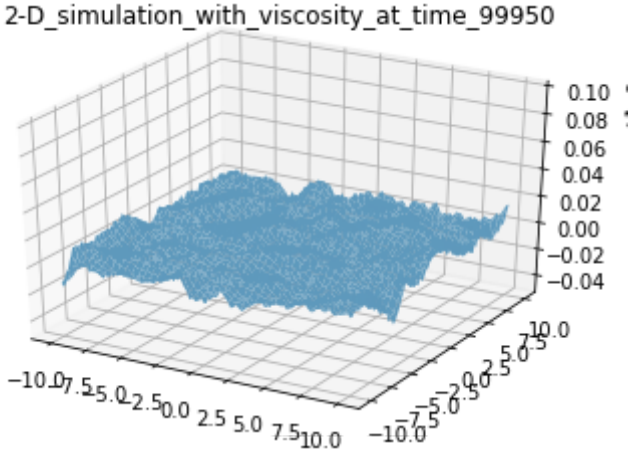



2-D_simulation_with_viscosity_at_time_5000
0.10
0.08
0.06
0.04
0.02
0.00
−0.02
−0.04
−10.0
−7.5
−5.0
−2.5
0.0
2.5
5.0
7.5
10.0


2-D_simulation_with_viscosity_at_time_25000
0.10
0.08
0.06
0.04
0.02
0.00
−0.02
−0.04
−10.0
−7.5
−5.0
−2.5
0.0
2.5
5.0
7.5
10.0


2-D_simulation_with_viscosity_at_time_50000
0.10
0.08
0.06
0.04
0.02
0.00
−0.02
−0.04
−10.0
−7.5
−5.0
−2.5
0.0
2.5
5.0
7.5
10.0


2-D_simulation_with_viscosity_at_time_75000
0.10
0.08
0.06
0.04
0.02
0.00
−0.02
−0.04
−10.0
−7.5
−5.0
−2.5
0.0
2.5
5.0
7.5
10.0

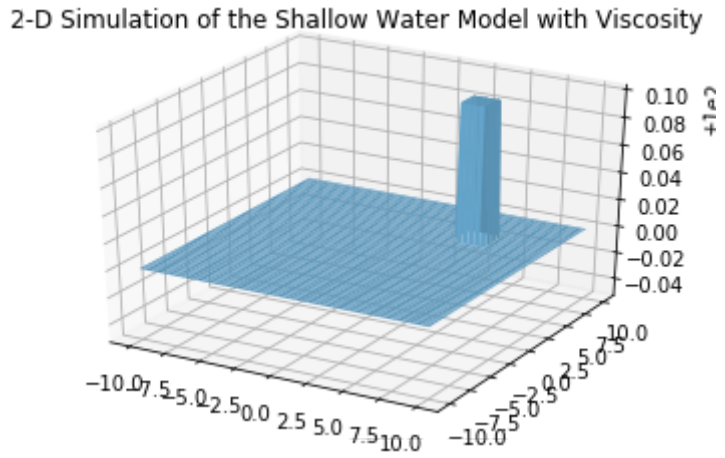

2-D_simulation_with_viscosity_at_time_99950
0.10
0.08
0.06
0.04
0.02
0.00
−0.02
−0.04
−10.0
−7.5
−5.0
−2.5
0.0
2.5
5.0
7.5
10.0

In [11]:

```
fig = plt.figure()
ax = fig.gca(projection='3d')
ax.set_title('2-D Simulation of the Shallow Water Model with Viscosity')
# Make data.
X = np.linspace(-10, 10, N)
Y = np.linspace(-10, 10, N)
X, Y = np.meshgrid(X, Y)
# Plot the surface.

surf = None
def animate(i):
    global surf
    # If a line collection is already remove it before drawing.
    if surf:
        ax.collections.remove(surf)
    surf = ax.plot_surface(X, Y, H[:,:,i*5],color = 'xkcd:lightblue')
    ax.set_zlim(99.95,100.1)

    return surf,


ani = animation.FuncAnimation(
    fig, animate, frames = 399,interval=30, blit=False)

HTML(ani.to_html5_video())
```

Out[11]:

0:00


2-D Simulation of the Shallow Water Model with Viscosity
0.10
0.08
0.06
0.04
0.02
0.00
−0.02
−0.04
−10.0
−7.5
−5.0
−2.5
0.0
2.5
5.0
7.5
10.0

Adams-Bashforth Method

In [8]:

```
# initialization
H = 100
L = 10**6  # L is the total length of x
N = 500  # grid number in x axis

del_t = 10
del_s = L/N
T = 100000
itr = int(T/del_t)  # itr is the number of iteration times w.r.t time

f = 0.0001 # Coriolis force
g = 9.8

u = np.zeros((N,itr),dtype='float')  # u is the velocity along x axis
v = np.zeros((N,itr),dtype='float')  # v is the velocity along y axis
h = np.zeros((N,itr),dtype='float')  # h is the height

for i in range(N):
    if 200 <i <= 300:
        h[i,0] = 200.1
    else:
        h[i,0] = 200

#guassian
def gaussian(x, mu, sig):
    return np.exp(-np.power(x - mu, 2.) / (2 * np.power(sig, 2.)))

def initial_gaussian_1d():
    L = 10**6
    N = 200  # grid number in x axis

    del_t = 2
    del_s = L/N
    T = 50000
    itr = int(T/del_t)

    f = 0.0001 # Coriolis force
    g = 9.8

    u = np.zeros((N,itr),dtype='float')
    v = np.zeros((N,itr),dtype='float')
    h = np.zeros((N,itr),dtype='float')
```

```
#     for i in range(N):
#         if 100 <i <= 150:
#             h[i,0] = 100.1
#         else:
#             h[i,0] = 100
    x_values = np.linspace(-3, 3, N)
    y_values = gaussian(x_values, 0, 10)*0.1 + 99.9
    for i in range(N):
        h[i,0] = 100*y_values[i]
    H = np.mean(h[:,0],axis=0)
    return H, N, del_t, del_s, itr, f, g, u, v, h
```

Adams-Bashforth Method

If $\frac{dy}{dt} = F(y)$**, then** $y^{n+1} = y^n + \frac{\Delta t}{2}(3F(y^n) - F(y^{n-1}))$

In [9]:
```
################ Adams-Bashforth Method #############

def fu(f, v, g, hup, hdown, deltx):
    return f*v - g*(hup-hdown)/(2*deltx)

def fh(h, uup, udown, deltx):
    return -h*(uup - udown)/(2*deltx)

def fv(f, u):
    return -f*u

def k1_func(func, deltt):
    #return deltt * func


def AB_central(u_, v_, h_, fu, fh, fv, g, itr, N, del_t, del_s, f, H):  # AB_central is the function for A
    h = copy.deepcopy(h_)
    u = copy.deepcopy(u_)
    v = copy.deepcopy(v_)
    #print(h[:,0])
    for i in range(1,itr):  #i: time
        if i == 1 : # forward difference
            # I use RK2 to determine the values at time = 1,
            #then use Adams-Bashforth to determine the rest values.
            for j in range(N):  #j: x grid
                if j == 0:
                    h[j,i] = del_t*(-H*(u[j+1,i-1]-u[-1,i-1])/(2*del_s)) + h[j,i-1]
                    u[j,i] = del_t*(-g*(h[j+1,i-1]-h[-1,i-1])/(2*del_s) + f*v[j,i-1]) + u[j,i-1]
                    v[j,i] = -f*u[j,i-1]*del_t + v[j,i-1]
                elif j == N-1:
                    h[j,i] = del_t*(-H*(u[0,i-1]-u[j-1,i-1])/(2*del_s)) + h[j,i-1]
                    u[j,i] = del_t*(-g*(h[0,i-1]-h[j-1,i-1])/(2*del_s) + f*v[j,i-1]) + u[j,i-1]
                    v[j,i] = -f*u[j,i-1]*del_t + v[j,i-1]
                else:
                    h[j,i] = del_t*(-H*(u[j+1,i-1]-u[j-1,i-1])/(2*del_s)) + h[j,i-1]
                    u[j,i] = del_t*(-g*(h[j+1,i-1]-h[j-1,i-1])/(2*del_s) + f*v[j,i-1]) + u[j,i-1]
                    v[j,i] = -f*u[j,i-1]*del_t + v[j,i-1]
            #print (h[:,i])
        else:
            for j in range(N):
                if j == 0:
                    h[j,i] = h[j,i-1] + (del_t/2)*(3*fh(h[j, i-1], u[j+1, i-1], u[-1, i-1], del_s)
```

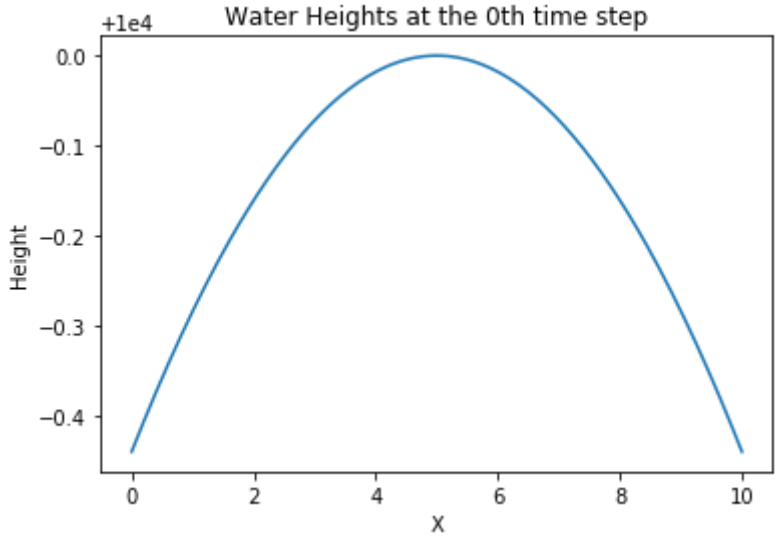
```
                                                - fh(h[j, i-2], u[j+1, i-2], u[-1, i-2], del_s))
                    #print('i', i,'h[j,i]', h[j,i])
                    u[j,i] = u[j,i-1] + (del_t/2)* (3*fu(f, v[j, i-1], g, h[j+1, i-1], h[-1, i-1], del_s)
                                                - fu(f, v[j, i-2], g, h[j+1, i-2], h[-1, i-2], del_s))
                    v[j,i] = v[j, i-1] + (del_t/2)*(3*fv(f, u[j, i-1]) - fv(f, u[j, i-2]))
                    if math.isnan(h[j,i]):
                        print('i', i, 'j', j, 'h[j,i]', h[j,i], 'u[j,i]', u[j,i], 'v[j,i]', v[j,i])
                elif j == N-1:
                    h[j,i] = h[j,i-1] + (del_t/2)*(3*fh(h[j, i-1], u[0, i-1], u[j-1, i-1], del_s)
                                                - fh(h[j, i-2], u[0, i-2], u[j-1, i-2], del_s))
                    u[j,i] = u[j,i-1] + (del_t/2)* (3*fu(f, v[j, i-1], g, h[0, i-1], h[j-1, i-1], del_s)
                                                - fu(f, v[j, i-2], g, h[0, i-2], h[j-1, i-2], del_s))
                    v[j,i] = v[j, i-1] + (del_t/2)*(3*fv(f, u[j, i-1]) - fv(f, u[j, i-2]))
                    if math.isnan(h[j,i]):
                        print('i', i, 'j', j, 'h[j,i]', h[j,i], 'u[j,i]', u[j,i], 'v[j,i]', v[j,i])
                else:
                    h[j,i] = h[j,i-1] + (del_t/2)*(3*fh(h[j, i-1], u[j+1, i-1], u[j-1, i-1], del_s)
                                                - fh(h[j, i-2], u[j+1, i-2], u[j-1, i-2], del_s))
                    u[j,i] = u[j,i-1] + (del_t/2)* (3*fu(f, v[j, i-1], g, h[j+1, i-1], h[j-1, i-1], del_s)
                                                - fu(f, v[j, i-2], g, h[j+1, i-2], h[j-1, i-2], del_s)
                    v[j,i] = v[j, i-1] +(del_t/2)*(3*fv(f, u[j, i-1]) - fv(f, u[j, i-2]))
                    if math.isnan(h[j,i]):
                        print('i', i, 'j', j, 'h[j,i]', h[j,i], 'u[j,i]', u[j,i], 'v[j,i]', v[j,i])


    return u, v, h
```

In [10]:

```
H, N, del_t, del_s, itr, f, g, u, v, h = initial_gaussian_1d()
UAB,VAB,HAB = AB_central(u, v, h, fu, fh, fv, g, itr, N, del_t, del_s, f, H)
```

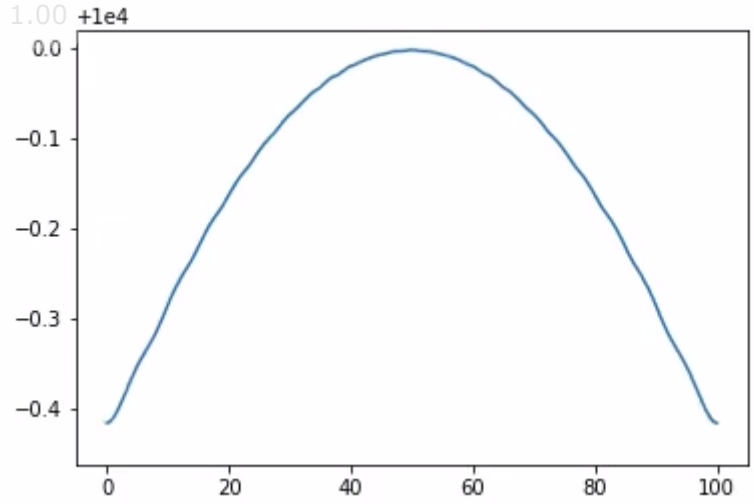
In [11]:

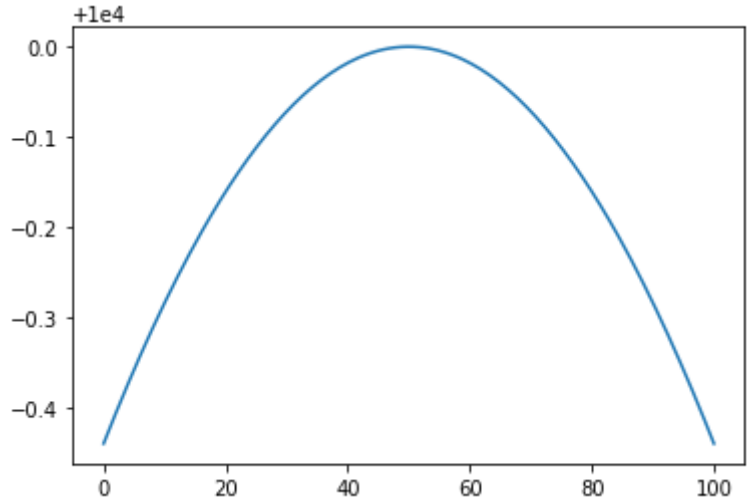
```
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.pyplot as plt
from matplotlib import cm

temp = [0, 100, 500, 1000, 5000, 10000, 12000,15000, 20000]
for i in temp:
    plt.figure()
    plt.plot(np.linspace(0,10,N), HAB[:,i])
    plt.title('Water Heights at the {}th time step'.format(i*10))
    #plt.ylim((99.88,100.0))
    plt.xlabel('X')
    plt.ylabel('Height')
    plt.savefig('./ab_pictures/ab_h_at_time_{}.png'.format(i*10))
```



In [12]:
```
from matplotlib import animation
from IPython.display import HTML
fig, ax = plt.subplots()

x = np.linspace(0,100,N)
line, = ax.plot(x, HAB[:,0])


def init():  # only required for blitting to give a clean slate.
    line.set_ydata(HAB[:,0])
    return line,



def animate(i):
    line.set_ydata(HAB[:,i*5])  #  the data.
    return line,


ani = animation.FuncAnimation(
    fig, animate, init_func=init, frames = 500, interval=30, blit=False)

HTML(ani.to_html5_video())
```

Out[12]:





Runge-Kutta 4 Method

In [13]:

```
################## RK4 Method ##############

def fu(f, v, g, hup, hm, deltx, half = False):
    return f*v - g*(hup-hm)/deltx   #hup = h[j+1, i-1], hm = h[j,i-1]

def fh(H, uup, um, deltx, half = False):
    #print('fh', -h*(u[j+1,i-1] - u[j-1,i-1])/(2*deltx))
    return -H *(uup - um)/deltx


def fv(f, u, half = False):
    return -f*u       #u = u[j,i-1]


def whichf(func, v, u, g, up, med, deltx,f,H, half = False):

    if func == fu:
        res = fu(f, v, g, up, med, deltx, deltx)
    elif func == fh:
        res = fh(H, up, med, deltx)
    else:
        res = fv(f, u)
    return res


def k1_func(func, v, u, g, up, med, deltx, deltt, f,H):
    res = whichf(func, v, u, g, up, med, deltx,f,H)
    #print('res', res)
    return deltt * res

def k2_func(func, v, u, g, up, med, deltx,deltt, f,H):
    res = whichf(func, v, u, g, up, med, deltx,f,H)
    return deltt * res

def k3_func(func, v, u, g, up, med, deltx,deltt, f,H):
    res = whichf(func, v, u, g, up, med, deltx,f,H)
    return deltt * res

def k4_func(func, v, u, g, up, med, deltx,deltt, f,H,):
    res = whichf(func, v, u, g, up, med, deltx,f,H,)
    return deltt*res
```

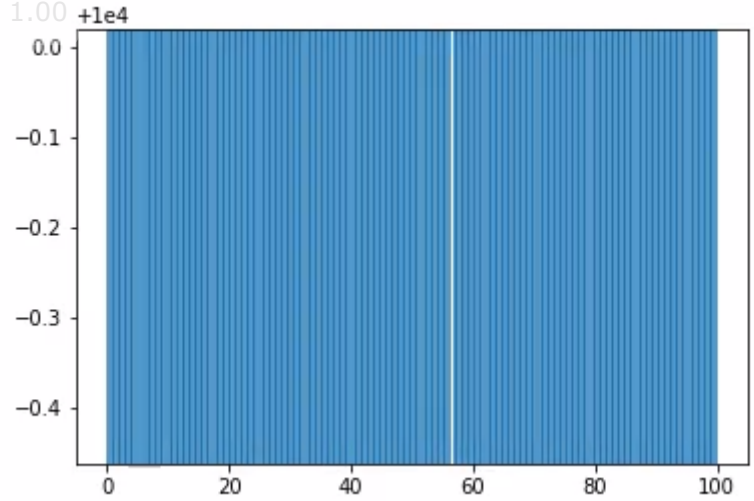
```
def RK4(u_, v_, h_, fu, fh, fv, g, itr, N, del_t, del_s, f, H):
    h = copy.deepcopy(h_)
    u = copy.deepcopy(u_)
    v = copy.deepcopy(v_)
    for i in range(1,itr):
        for j in range(N):
            #if j == 0:
                #h[j,i] = del_t*(-H*(u[j+1,i-1]-u[j,i-1])/(2*del_s)) + h[j,i-1]
                #u[j,i] = del_t*(-g*(h[j+1,i-1]-h[-1,i-1])/(2*del_s) + f*v[j,i-1]) + u[j,i-1]
                #v[j,i] = -f*u[j,i-1]*del_t + v[j,i-1]
            if j == N-1:
                h[j,i] = del_t*(-H*(u[0,i-1]-u[j,i-1])/(del_s)) + h[j,i-1]
                u[j,i] = del_t*(-g*(h[0,i-1]-h[j,i-1])/(del_s) + f*v[j,i-1]) + u[j,i-1]
                v[j,i] = -f*u[j,i-1]*del_t + v[j,i-1]
            else:
                k1h = k1_func(fh, v[j, i-1], u[j, i-1], g, u[j+1, i-1], u[j, i-1], del_s, del_t, f,H)
                k1u = k1_func(fu, v[j, i-1], u[j, i-1], g, h[j+1, i-1], h[j, i-1], del_s, del_t, f,H)
                k1v = k1_func(fv, v[j, i-1], u[j, i-1], g, h[j+1, i-1], h[j, i-1], del_s, del_t, f,H)

                k2h = k2_func(fh, v[j, i-1], u[j, i-1], g, u[j+1, i-1], u[j, i-1], del_s, del_t, f,H)
                k2u = k2_func(fu, v[j, i-1]+0.5*k1v*del_t, u[j, i-1], g, h[j+1, i-1], h[j, i-1], del_s,
                k2v = k2_func(fv, v[j, i-1], u[j, i-1] + 0.5*k1u*del_t, g, u[j+1, i-1], u[j, i-1], del_s
                h[j,i] = h[j,i-1] + (k1h + k2h)/2
                u[j,i] = u[j,i-1] + (k1u + k2u)/2
                v[j,i] = v[j,i-1] + (k1v + k2v)/2

        if i%5 == 0: # smooth
            # this is the diffusion term in the
            h[:,i-1] = h[:,i-1] + 0.01*(h[:,i]-2*h[:,i-1]+h[:,i-2])
            u[:,i-1] = u[:,i-1] + 0.01*(u[:,i]-2*u[:,i-1]+u[:,i-2])
            v[:,i-1] = v[:,i-1] + 0.01*(v[:,i]-2*v[:,i-1]+v[:,i-2])
                        #print(k1h)



    return u, v, h
```

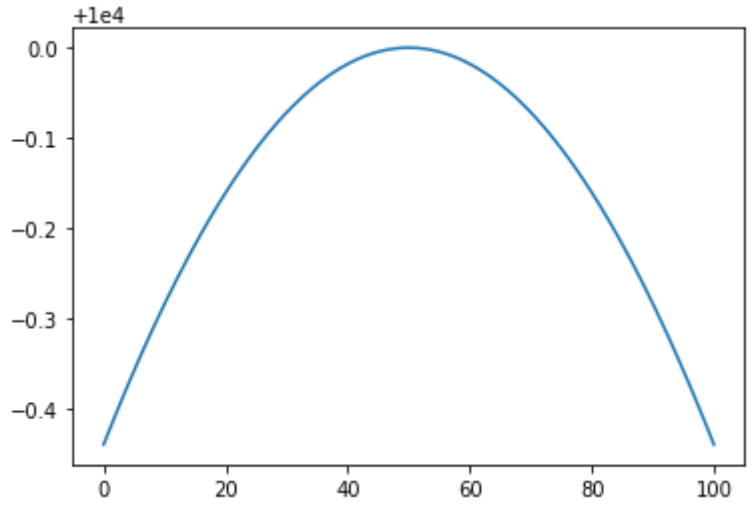
In [14]:
```
H, N, del_t, del_s, itr, f, g, u, v, h = initial_gaussian_1d()

URK,VRK,HRK = RK4(u, v, h, fu, fh, fv, g, itr, N, del_t, del_s, f, H)
```

```
/anaconda3/lib/python3.6/site-packages/ipykernel_launcher.py:8: RuntimeWarning: overflow encountered i
n double_scalars

/anaconda3/lib/python3.6/site-packages/ipykernel_launcher.py:4: RuntimeWarning: invalid value encounte
red in double_scalars
  after removing the cwd from sys.path.
/anaconda3/lib/python3.6/site-packages/ipykernel_launcher.py:71: RuntimeWarning: invalid value encount
ered in subtract
```

In [15]:
```
from matplotlib import animation
from IPython.display import HTML
fig, ax = plt.subplots()

x = np.linspace(0,100,N)
line, = ax.plot(x, HRK[:,0])


def init():  # only required for blitting to give a clean slate.
    line.set_ydata(HRK[:,0])
    return line,



def animate(i):
    line.set_ydata(HRK[:,i*5])  #  the data.
    return line,


ani = animation.FuncAnimation(
    fig, animate, init_func=init, frames = 500, interval=30, blit=False)

HTML(ani.to_html5_video())
```

Out[15]:





Waterfall Application with Horizontal Advection and Viscosity

In [16]:
```
# initialization
def initial_2d():
    X = 10**6
    Y = 10**6
    N = 100  # grid number in

    del_t = 50
    del_x = X/N
    del_y = Y/N
    T = 100000

    itr = int(T/del_t)

    f = 0.0001 # Coriolis force
    g = 9.8

    h = np.zeros((N,N,itr),dtype='float')

    h[:,:,0] = 100
    h[70:80,70:80,0] = 100.1
    H = np.mean(h[:,:,0])
    return H, N, del_t, del_x, del_y, itr, f, g, h
```

In [17]:

```
def leap_frog_central_scheme_3d_zeroboundary(H, N, del_t, del_x, del_y, itr, f, g, h_):
    h = copy.deepcopy(h_)
    v = np.zeros((N,N,itr),dtype='float')
    u = np.zeros((N,N,itr),dtype='float')

    for t in range(1,itr):
        if t == 1 : # forward difference
            for i in range(N):
                for j in range(N):
                    if i == 0 and 0 < j < N-1: # boudary condition
                        h[i,j,t] = del_t*(-H*((u[i+1,j,t-1]-u[i,j,t-1])/(del_x) + (v[i,j+1,t-1]-v[i,j-1,
                        u[i,j,t] = 0
                        v[i,j,t] = (-f*u[i,j,t-1]-g*(h[i,j+1,t-1] - h[i,j-1,t-1])/(2*del_y))*del_t + v[i
                    elif j == N-1 and 0< i <N-1:
                        h[i,j,t] = del_t*(-H*((u[i+1,j,t-1]-u[i-1,j,t-1])/(2*del_x) + (v[i,j,t-1]-v[i,j-
                        u[i,j,t] = del_t*(-g*(h[i+1,j,t-1]-h[i-1,j,t-1])/(2*del_x) + f*v[i,j,t-1]) + u[i
                        v[i,j,t] = 0
                    elif i == N-1 and 0< j <N-1:
                        h[i,j,t] = del_t*(-H*((u[i,j,t-1]-u[i-1,j,t-1])/(del_x) + (v[i,j+1,t-1]-v[i,j-1,
                        u[i,j,t] = 0
                        v[i,j,t] = (-f*u[i,j,t-1]-g*(h[i,j+1,t-1] - h[i,j-1,t-1])/(2*del_y))*del_t + v[i
                    elif j ==0 and 0< i < N-1:
                        h[i,j,t] = del_t*(-H*((u[i+1,j,t-1]-u[i-1,j,t-1])/(2*del_x) + (v[i,j+1,t-1]-v[i,
                        u[i,j,t] = del_t*(-g*(h[i+1,j,t-1]-h[i-1,j,t-1])/(2*del_x) + f*v[i,j,t-1]) + u[i
                        v[i,j,t] = 0
                    elif (i==0 and j==0):
                        u[i,j,t] = 0
                        v[i,j,t] = 0
                        h[i,j,t] = del_t*(-H*((u[i+1,j,t-1]-u[i,j,t-1])/(del_x) + (v[i,j+1,t-1]-v[i,j,t-
                    elif (i==0 and j == N-1):
                        u[i,j,t] = 0
                        v[i,j,t] = 0
                        h[i,j,t] = del_t*(-H*((u[i+1,j,t-1]-u[i,j,t-1])/(del_x) + (v[i,j,t-1]-v[i,j-1,t-
                    elif (i==N-1 and j==0):
                        u[i,j,t] = 0
                        v[i,j,t] = 0
                        h[i,j,t] = del_t*(-H*((u[i,j,t-1]-u[i-1,j,t-1])/(del_x) + (v[i,j+1,t-1]-v[i,j,t-
                    elif (i ==N-1 and j==N-1):
                        u[i,j,t] = 0
                        v[i,j,t] = 0
                        h[i,j,t] = del_t*(-H*((u[i,j,t-1]-u[i-1,j,t-1])/(del_x) + (v[i,j,t-1]-v[i,j-1,t-
                    else:
```

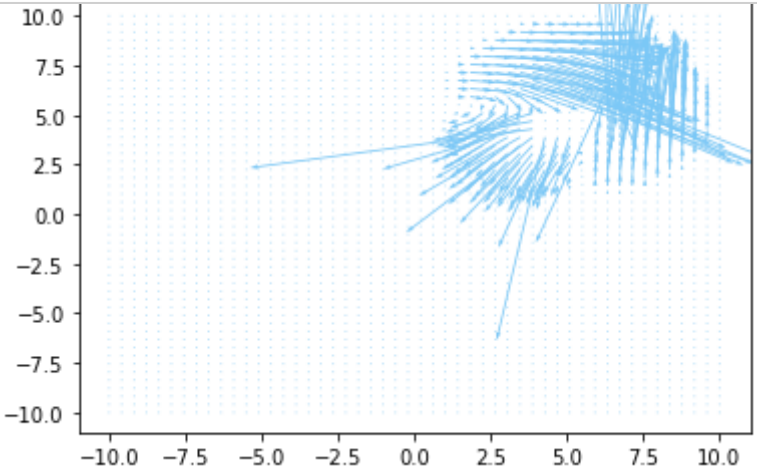
```
                    h[i,j,t] = del_t*(-H*((u[i+1,j,t-1]-u[i-1,j,t-1])/(2*del_x) + (v[i,j+1,t-1]-v[i,
                    u[i,j,t] = del_t*(-g*(h[i+1,j,t-1]-h[i-1,j,t-1])/(2*del_x) + f*v[i,j,t-1]) + u[i
                    v[i,j,t] = (-f*u[i,j,t-1]-g*(h[i,j+1,t-1] - h[i,j-1,t-1])/(2*del_y))*del_t + v[i

        else:
            for i in range(N):
                for j in range(N):
                    if i >=70 and i <80 and j>= 70 and j < 80:
                        h[i,j,t] = 100.1
                        u[i,j,t] = 0
                        v[i,j,t] = 0
                    elif i == 0 and 0 < j < N-1: # boudary condition
                        h[i,j,t] = 2*del_t*(-H*((u[i+1,j,t-1]-u[i,j,t-1])/(del_x) + (v[i,j+1,t-1]-v[i,j-
                        u[i,j,t] = 0
                        v[i,j,t] = (-f*u[i,j,t-1]-g*(h[i,j+1,t-1] - h[i,j-1,t-1])/(2*del_y))*del_t*2 + v
                    elif j == N-1 and 0< i <N-1:
                        h[i,j,t] = 2*del_t*(-H*((u[i+1,j,t-1]-u[i-1,j,t-1])/(2*del_x) + (v[i,j,t-1]-v[i,
                        u[i,j,t] = 2*del_t*(-g*(h[i+1,j,t-1]-h[i-1,j,t-1])/(2*del_x) + f*v[i,j,t-1]) + u
                        v[i,j,t] = 0
                    elif i == N-1 and 0< j <N-1:
                        h[i,j,t] = 2*del_t*(-H*((u[i,j,t-1]-u[i-1,j,t-1])/(del_x) + (v[i,j+1,t-1]-v[i,j-
                        u[i,j,t] = 0
                        v[i,j,t] = (-f*u[i,j,t-1]-g*(h[i,j+1,t-1] - h[i,j-1,t-1])/(2*del_y))*del_t*2 + v
                    elif j ==0 and 0< i < N-1:
                        h[i,j,t] = 2*del_t*(-H*((u[i+1,j,t-1]-u[i-1,j,t-1])/(2*del_x) + (v[i,j+1,t-1]-v[
                        u[i,j,t] = 2*del_t*(-g*(h[i+1,j,t-1]-h[i-1,j,t-1])/(2*del_x) + f*v[i,j,t-1]) + u
                        v[i,j,t] = 0
                    elif (i==0 and j==0):
                        u[i,j,t] = 0
                        v[i,j,t] = 0
                        h[i,j,t] = 2*del_t*(-H*((u[i+1,j,t-1]-u[i,j,t-1])/(del_x) + (v[i,j+1,t-1]-v[i,j,
                    elif (i==0 and j == N-1):
                        u[i,j,t] = 0
                        v[i,j,t] = 0
                        h[i,j,t] = 2*del_t*(-H*((u[i+1,j,t-1]-u[i,j,t-1])/(del_x) + (v[i,j,t-1]-v[i,j-1,
                    elif (i==N-1 and j==0):
                        u[i,j,t] = 0
                        v[i,j,t] = 0
                        h[i,j,t] = 2*del_t*(-H*((u[i,j,t-1]-u[i-1,j,t-1])/(del_x) + (v[i,j+1,t-1]-v[i,j,
                    elif (i ==N-1 and j==N-1):
                        u[i,j,t] = 0
                        v[i,j,t] = 0
                        h[i,j,t] = 2*del_t*(-H*((u[i,j,t-1]-u[i-1,j,t-1])/(del_x) + (v[i,j,t-1]-v[i,j-1,
```

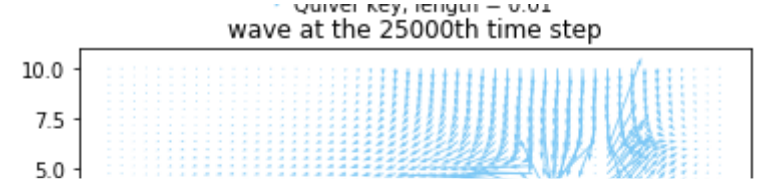
```
                else:
                    h[i,j,t] = 2*del_t*(-H*((u[i+1,j,t-1]-u[i-1,j,t-1])/(2*del_x) + (v[i,j+1,t-1]-v[
                    u[i,j,t] = 2*del_t*(-g*(h[i+1,j,t-1]-h[i-1,j,t-1])/(2*del_x) + f*v[i,j,t-1]) + u
                    v[i,j,t] = (-f*u[i,j,t-1]-g*(h[i,j+1,t-1] - h[i,j-1,t-1])/(2*del_y))*del_t*2 + v

            if t%5 == 0: # smooth
                h[:,:,t-1] = h[:,:,t-1] + 0.01*(h[:,:,t]-2*h[:,:,t-1]+h[:,:,t-2])
                u[:,:,t-1] = u[:,:,t-1] + 0.01*(u[:,:,t]-2*u[:,:,t-1]+u[:,:,t-2])
                v[:,:,t-1] = v[:,:,t-1] + 0.01*(v[:,:,t]-2*v[:,:,t-1]+v[:,:,t-2])

    return u,v,h
```

In [18]:

```
H, N, del_t, del_x, del_y, itr, f, g, h = initial_2d()
U,V,H = leap_frog_central_scheme_3d_zeroboundary(H, N, del_t, del_x, del_y, itr, f, g, h)
```

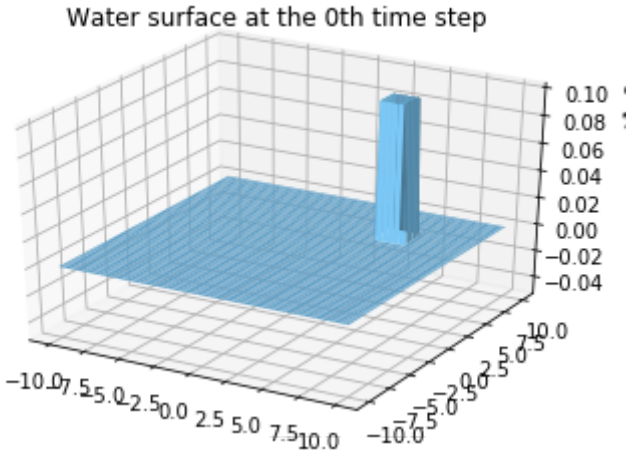
In [19]:
```
X = np.linspace(-10, 10, int(N/2))
Y = np.linspace(-10, 10, int(N/2))
X, Y = np.meshgrid(X, Y)

temp = [0, 100, 500, 1000, 1500, 1999]
for i in temp:
    fig, ax = plt.subplots()
    q = ax.quiver(X, Y, U[::2,::2,i], V[::2,::2,i],color = 'xkcd:lightblue')
    ax.quiverkey(q, X=0.3, Y=1.1, U=0.01,
                 label='Quiver key, length = 0.01', labelpos='E')
    ax.set_title('wave at the {}th time step'.format(i*50))
    plt.savefig('./app_pictures/wave_at_time_{}.png'.format(i*50))
```

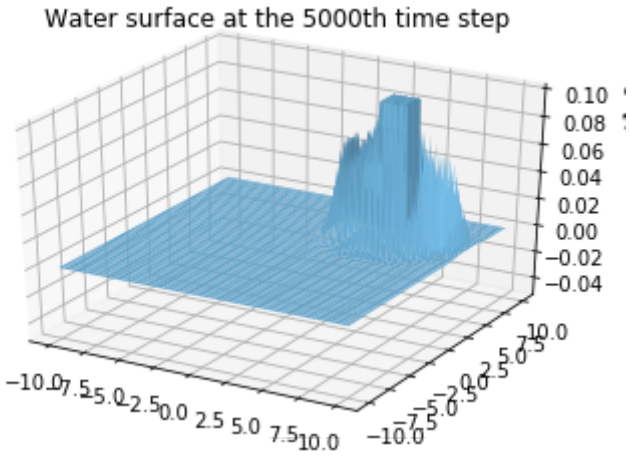


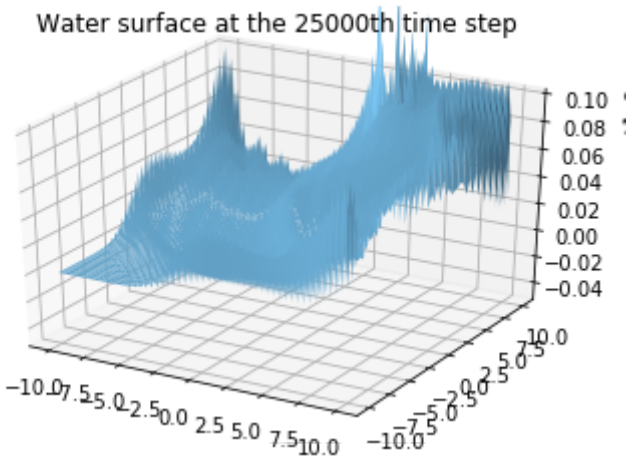


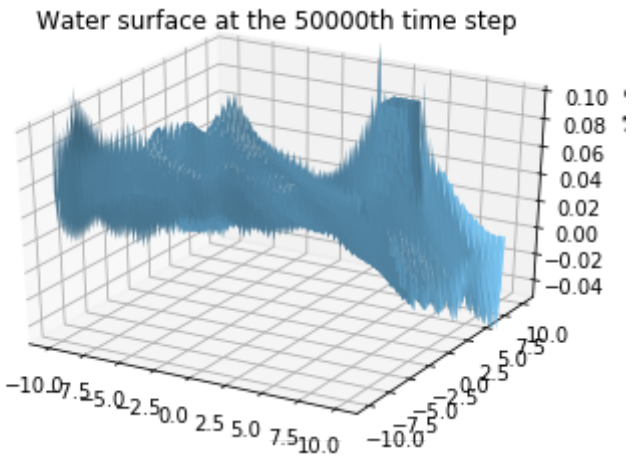
In [20]:

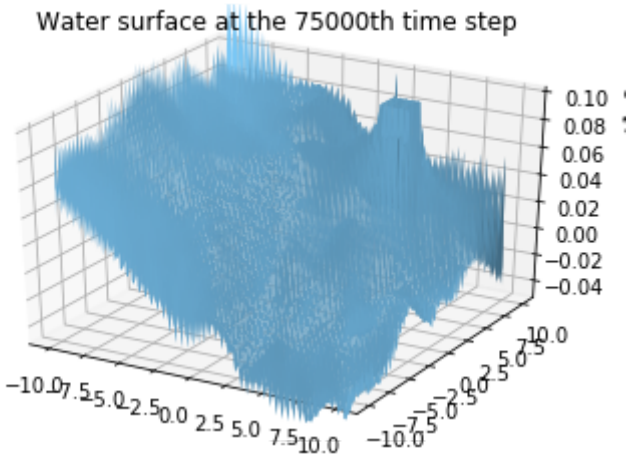
```
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.pyplot as plt
from matplotlib import cm

temp = [0, 100, 500, 1000, 1500, 1999]
for i in temp:
    fig = plt.figure()
    ax = fig.gca(projection='3d')

    # Make data.
    X = np.linspace(-10, 10, N)
    Y = np.linspace(-10, 10, N)
    X, Y = np.meshgrid(X, Y)
    # Plot the surface.
    ax.set_title('Water surface at the {}th time step'.format(i*50))
    ax.set_zlim(99.95,100.1)
    surf = ax.plot_surface(X, Y, H[:,:,i],color = 'xkcd:lightblue')
    plt.savefig('./app_pictures/app_2D_at_time_{}.png'.format(i*50))
```

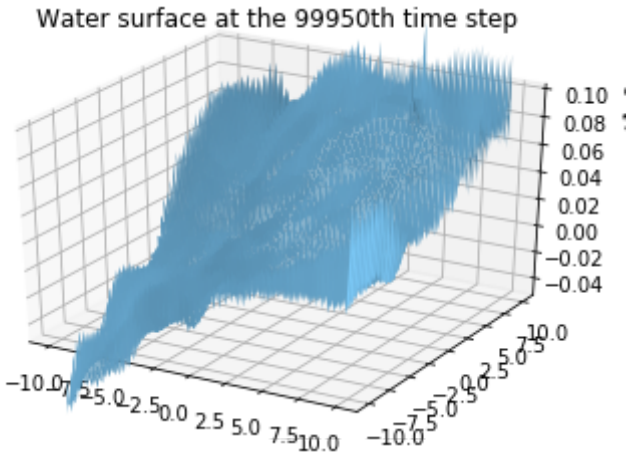



Water surface at the 5000th time step
0.10
0.08
0.06
0.04
0.02
0.00
−0.02
−0.04
−10.0
−7.5
−5.0
−2.5
0.0
2.5
5.0
7.5
10.0


Water surface at the 25000th time step
0.10
0.08
0.06
0.04
0.02
0.00
−0.02
−0.04
−10.0
−7.5
−5.0
−2.5
0.0
2.5
5.0
7.5
10.0


Water surface at the 50000th time step
0.10
0.08
0.06
0.04
0.02
0.00
−0.02
−0.04
−10.0
−7.5
−5.0
−2.5
0.0
2.5
5.0
7.5
10.0

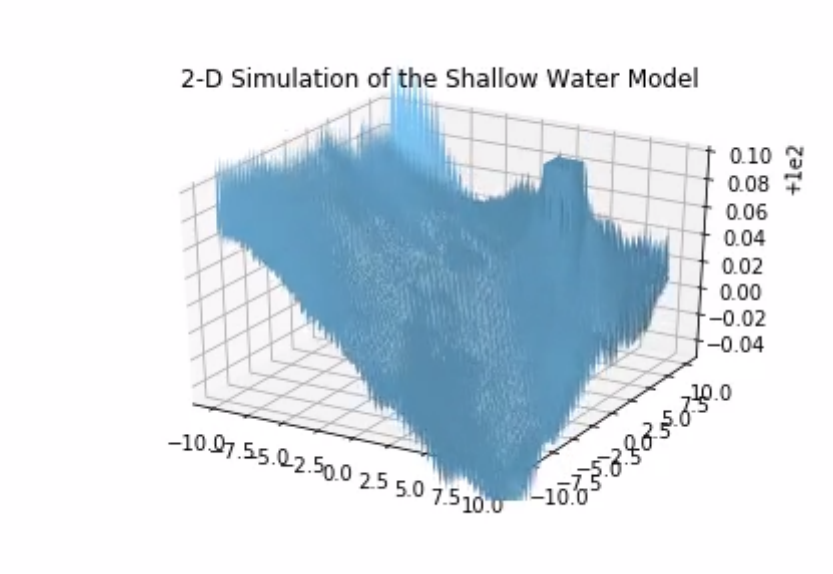

Water surface at the 75000th time step
0.10
0.08
0.06
0.04
0.02
0.00
−0.02
−0.04
−10.0
−7.5
−5.0
−2.5
0.0
2.5
5.0
7.5
10.0

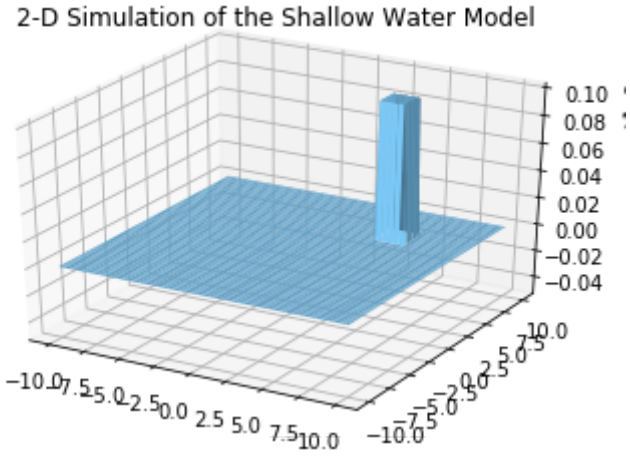

Water surface at the 99950th time step
0.10
0.08
0.06
0.04
0.02
0.00
−0.02
−0.04
−10.0
−7.5
−5.0
−2.5
0.0
2.5
5.0
7.5
10.0

In [26]:

```
fig = plt.figure()
ax = fig.gca(projection='3d')
ax.set_title('2-D Simulation of the Shallow Water Model')
# Make data.
X = np.linspace(-10, 10, N)
Y = np.linspace(-10, 10, N)
X, Y = np.meshgrid(X, Y)
# Plot the surface.

surf = None
def animate(i):
    global surf
    # If a line collection is already remove it before drawing.
    if surf:
        ax.collections.remove(surf)
    surf = ax.plot_surface(X, Y, H[:,:,i*5],color = 'xkcd:lightblue')
    ax.set_zlim(99.95,100.1)

    return surf,


ani = animation.FuncAnimation(
    fig, animate, frames = 399,interval=30, blit=False)

HTML(ani.to_html5_video())
```

Out[26]:


2-D Simulation of the Shallow Water Model
0.10
0.08
0.06
0.04
0.02
0.00
−0.02
−0.04
+1e2
−10.0 −7.5 −5.0 −2.5 0.0 2.5 5.0 7.5 10.0
−10.0 −7.5 −5.0 −2.5 0.0 2.5 5.0 7.5 10.0


2-D Simulation of the Shallow Water Model
0.10
0.08
0.06
0.04
0.02
0.00
−0.02
−0.04
−10.0 −7.5 −5.0 −2.5 0.0 2.5 5.0 7.5 10.0
−10.0 −7.5 −5.0 −2.5 0.0 2.5 5.0 7.5 10.0

In [ ]: