NUMERICAL SOLUTION OF THE SHALLOW WATER EQUATIONS

John Burkardt

ICAM/Information Technology Department
Virginia Tech

March 22-24, 2010
Lectures 23 and 24
http://people.sc.fsu.edu/~jburkardt/presentations/...
shallow_water_2010_vt.pdf

Introduction to Part 1:

The equations we have been studying this semester are used to describe fluid behavior, but the equations themselves don't actually tell you what they're modeling!

Perhaps the most interesting behavior that fluids have is that they can be used to transmit signals. If you make a splash in the pond, a disturbance is formed, and that disturbance moves away from its origin at a regular rate, keeping its shape (though perhaps weakening in magnitude).

The same hyperbolic equations that describe fluids also govern sound, electricity, and radio waves, which is why our cell phones are taking advantage of the wave equation; we could never use the heat equation to send a directional signal!

Introduction

For the classic wave equation

$$u_{tt} = c^2 u_{xx}$$

it's not hard to see that a traveling wave is a solution.

But for complicated systems like the shallow water equations, it is extremely valuable to have computational simulations available, to study the effects of various parameters, and to consider the kinds of solutions that arise.

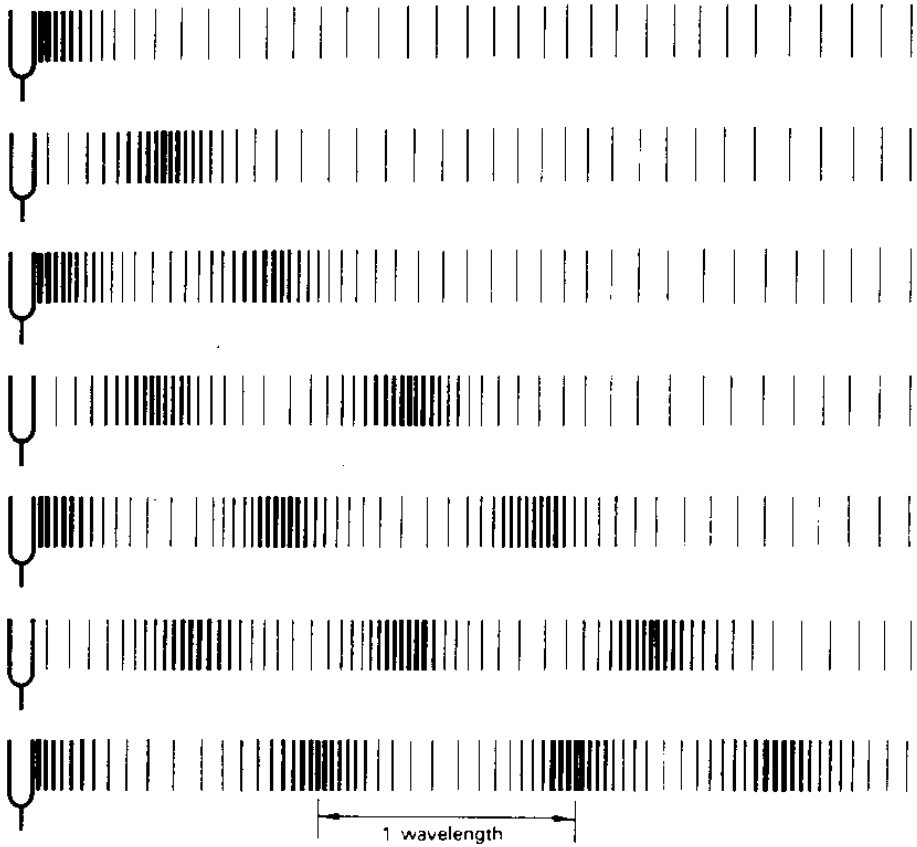
The lectures based on this set of notes will introduce some of the ideas and techniques involved in turning a set of equations into a computer program. Our goal is that you will to be able to understand, describe, explain, use and modify the computations involved in Cleve Moler's 2D shallow water simulation program.

Physics: Longitudinal Waves in a Slinky

A "Slinky" is a coiled metal wire which acts like a gentle spring. Pushing it on the left, the coils compress, storing energy, and then separate, and the pressure disturbance moves to the right.

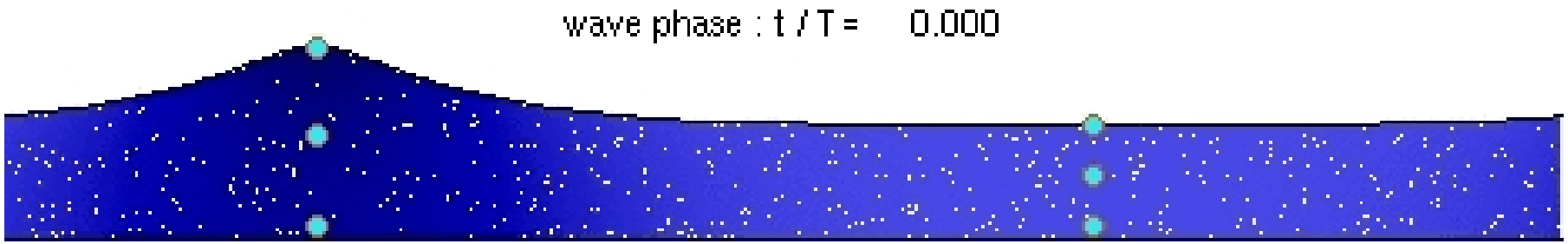
http://people.sc.fsu.edu/~jburkardt/latex/shallow_water_2010/slinky.mov

Physics: Longitudinal Waves in the Air



We hear sound because our ear drum detects pressure changes transmitted by the rarefaction and compression of air.

Physics: Longitudinal Waves in Water



Water is not compressible, but shallow water can store and release energy by locally varying its height "slightly".

The equations relating height h, horizontal velocities u and v, and the pressure p are called the *shallow water equations*.

---

http://people.sc.fsu.edu/~jburkardt/latex/shallow_water_2010/shallow_water_wave.mov

Equations: Conservative Form (Iliescu)

Here is the form of the equations as discussed at the end of Lecture 12:

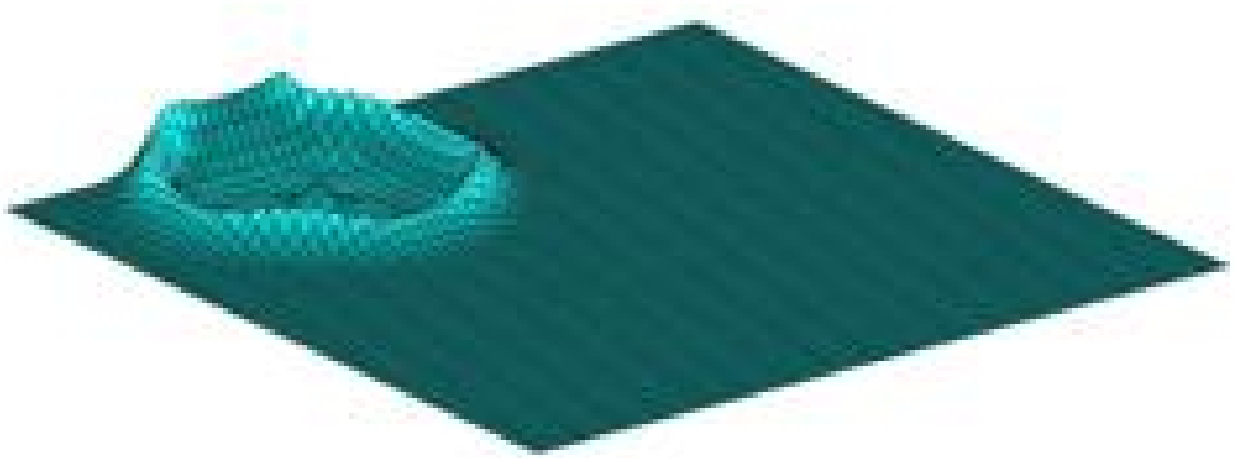
$$\frac{\partial h\,u}{\partial t} = -\frac{h}{\rho_0}\frac{\partial p}{\partial x} - \frac{\partial h\,u^2}{\partial x} - \frac{\partial h\,u\,v}{\partial y} + f\,h\,v$$

$$\frac{\partial h\,v}{\partial t} = -\frac{h}{\rho_0}\frac{\partial p}{\partial y} - \frac{\partial h\,u\,v}{\partial x} - \frac{\partial h\,v^2}{\partial y} - f\,h\,u$$

$$\frac{\partial h}{\partial t} + \frac{\partial h\,u}{\partial x} + \frac{\partial h\,v}{\partial y} = 0$$

Moler's 2D Shallow Water Program

Moler lets a giant droplet of water fall onto a tranquil ocean:



http://people.sc.fsu.edu/∼jburkardt/m_src/shallow_water_2d/shallow_water_2d.m

Moler's Program

Cleve Moler has implemented a simulation of the 2D shallow water equations in a simple MATLAB code, as part of his electronic book *"Experiments in MATLAB"*.

The source code is available at

http://www.mathworks.com/moler/exm/exm/waterwave.m.

Moler's 4 page discussion of the shallow water equations is available at

http://www.mathworks.com/moler/exm/chapters/water.pdf.

We are going to try to understand how the physics and math of the shallow water equations was turned into the lines of MATLAB code that form the program.

Moler's Program: Modeling Issues

We will consider modeling issues:

- what terms do you neglect, or regroup, or rewrite?
- how do we model the geometry: the shape of the region
- how do we model the geometry: replacing space by discrete points.
- how do we model a function $u(x, y, t)$ which is now defined on points?
- how do we model the derivative of such a function?
- how do we handle conditions at the boundary?
- how do we handle conditions at the initial time?

Moler's Program: Algorithmic/Computational Issues

We will consider algorithmic issues:

- how do we approximate the solution of a partial differential equation?
- what error can we expect in spatial and time approximations?
- when we use an explicit time integration scheme, are there limits to our time step?

We will consider computational design issues:

- Are there conserved quantities which we can use as a check?
- How can we report or display the results?

Equations: Moler's Variations

Moler takes as his state variables the set h, hu and hv, that is, he works with *momentum* or *mass-velocity* rather than velocity.

Moler neglects the Coriolis force terms $f * h * v$ and $f * h * u$.

The gravitational constant g relates pressure p and height h:

$$p = (\rho_0 * h) * g$$

so Moler rewrites the pressure terms, such as:

$$\frac{h}{\rho_0} \frac{\partial p}{\partial x} = \frac{\partial \frac{1}{2} g \, h^2}{\partial x}$$

so that pressure disappears from the equations.

Equations: Conservative Form (Moler)

After reordering equations, dropping the Coriolis force, rewriting the pressure relation, and bringing all terms to the left hand side, we have:

$$\frac{\partial h}{\partial t} + \frac{\partial h\,u}{\partial x} + \frac{\partial h\,v}{\partial y} = 0$$

$$\frac{\partial h\,u}{\partial t} + \frac{\partial (h\,u^2 + \frac{1}{2} g\,h^2)}{\partial x} + \frac{\partial h\,u\,v}{\partial y} = 0$$

$$\frac{\partial h\,v}{\partial t} + \frac{\partial h\,u\,v}{\partial x} + \frac{\partial (h\,v^2 + \frac{1}{2} g\,h^2)}{\partial y} = 0$$

Equations: Vector Definitions

Using Moler's formulation, we define the following quantities:

$$U = \begin{pmatrix} h \\ h\,u \\ h\,v \end{pmatrix}$$

$$F(U) = \begin{pmatrix} h\,u \\ h\,u^2 + \frac{1}{2} g\,h^2 \\ h\,u\,v \end{pmatrix}$$

$$G(U) = \begin{pmatrix} h\,v \\ h\,u\,v \\ h\,v^2 + \frac{1}{2} g\,h^2 \end{pmatrix}$$

Equations: Vector Version of Conservative Form

We can now write our equations in the very abstract but very concise form:

$$\frac{\partial U}{\partial t} + \frac{\partial F(U)}{\partial x} + \frac{\partial G(U)}{\partial y} = 0$$

This form is very common in situations involving flows, and is known as a *hyperbolic conservation law*.

Equations: Hyperbolic Conservation Law

A *conservation law* identifies a quantity whose behavior is strictly limited. It can never simply disappear. It can only change by entering or exiting through the boundaries of the region.

Consider a scalar law of the form

$$\frac{\partial u}{\partial t} + \frac{\partial f(u)}{\partial x} + \frac{\partial g(u)}{\partial y} = 0$$

Integrating over a region Ω, we have:

$$\int_\Omega \frac{\partial u}{\partial t}\, d\Omega + \int_\Omega \nabla \cdot (f(u), g(u))\, d\Omega = 0$$

Equations: Hyperbolic Conservation Law

Assuming appropriate smoothness for the boundary of Ω and the various derivatives, we can use the divergence theorem to arrive at:

$$\frac{d}{dt}\int_{\Omega} u\, d\Omega + \int_{\partial\Omega} (f(u), g(u)) \cdot \hat{n}\, \partial\Omega = 0$$

So $(f(u), g(u))$ represents the flux of u at the boundaries of Ω.

If u decreases in Ω, then it must increase correspondingly in the neighboring regions. It never "disappears" (except by entering or exiting the region over which the law holds).

Geometry: Model 3D Region by a Rectangular Array

We are modeling a flow in a geometric region. Our model of the geometry will be influenced by how much detail we want, and what will make our algorithm easy to program.

Because we will be using the finite difference method, the geometry will be represented by a rectangular array of points. In order to estimate derivatives easily, we need the axes of the rectangle to be aligned with the coordinate axes. For convenience in forming the differences, we will prefer to use equally spaced nodes in both x and y directions.

Because these are "shallow water" equations, the z direction is not broken down into individual points.

Geometry: Rectangular Array Simplifies Geometry

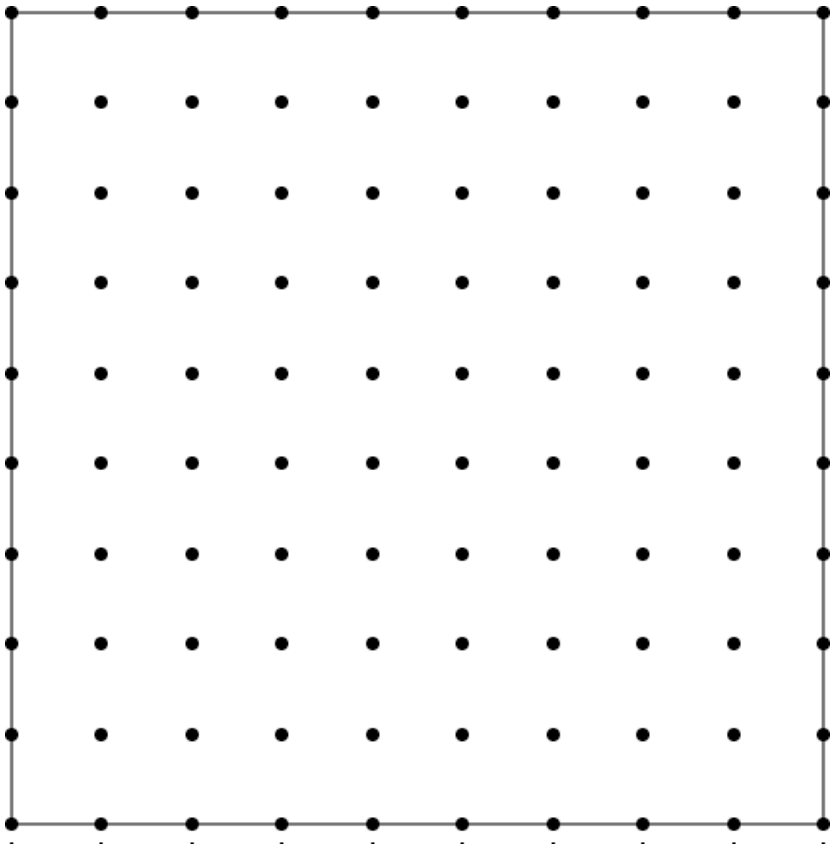
The values of a physical quantity $u(x, y)$ will be stored in an M by N array called U, with a typical entry being $U(I, J)$.

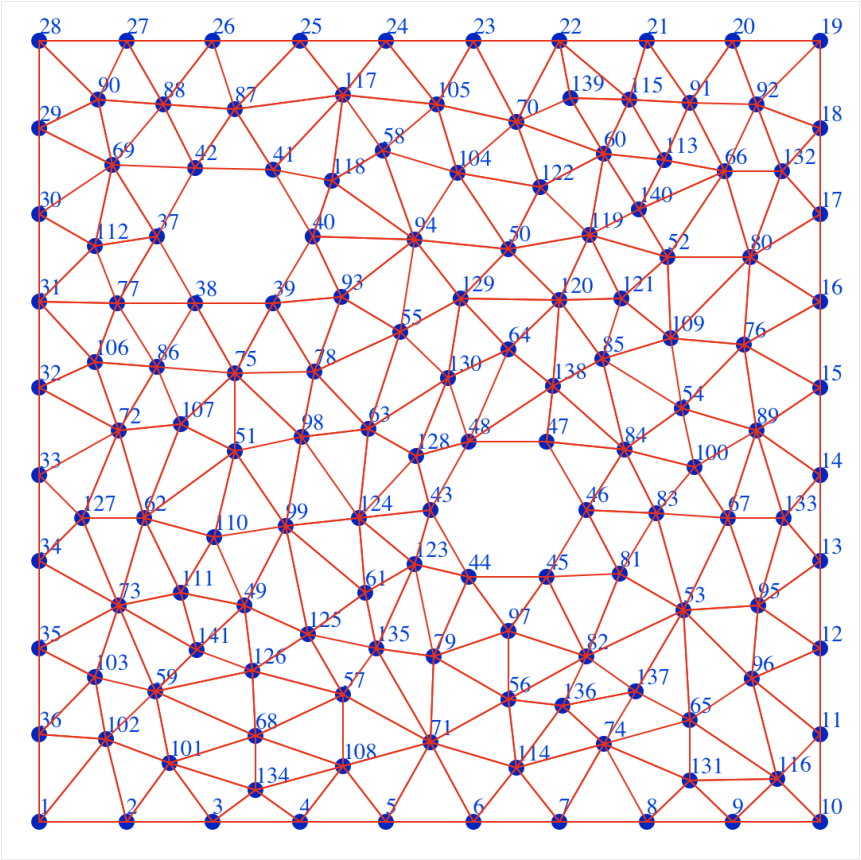
In finite differences, it is important to be able to located values associated with neighboring points. Using an array makes this trivial.

It is also crucial to be able to determine whether a point is on the boundary of the region or in the interior. Because we are using an array, this is also trivial.

By contrast, consider that these questions are not trivial for a finite element code!

Geometry: Rectangular Array Simplifies Geometry





Questions about directions, neighbors, and boundaries are much simpler for finite differences than for finite elements.

Geometry: Rectangular Array Simplifies Geometry

Because we have chosen a rectangular array, some things have become easy. Suppose we are not happy with this simple geometry? Are there some modifications that aren't too hard?

- Vary the spacing in the X or Y directions (easy)
- Use a region that is only a subset of the rectangle (some work)
- Include internal "holes" in the geometry (a little work)
- Use a "logically rectangular" array that is curved (much work)

Once we start using curved geometry, the approximation of derivatives becomes more difficult.

Geometry: Computational Description of Geometry

Our simple geometry model can therefore be defined by the following information:

- M and N, the number of rows and columns of points;
- DX and DY, the constant horizontal and "vertical" spacings;
- various arrays such as $U(I, J)$, which record values at all the points.

For our purposes, the information about the *height* of the fluid is not really part of the geometry. It's regarded simply as a measure of the local mass concentration, is stored as $H(I, J)$, and does not affect our geometric reasoning.

Geometry: Boundary Conditions

Our problem will include *boundary conditions* that describe what happens at points on the boundary of the region.

We can add an exterior layer of nodes to the problem, and manipulate the solution values at these nodes to enforce the boundary conditions.

One way to do this is consider M and N the number of rows and columns of interior nodes. Then our arrays will actually be of size $M + 2$ by $N + 2$, with the first and last rows and columns containing boundary data.

To work on just the interior nodes, we set up loops that run $I = 2 : M + 1$ and $J = 2 : N + 1$, for instance.

Geometry: Boundary Conditions

You might expect to specify boundary conditions that are

- **Dirichlet**: specify solution values;
- **Neumann**: specify solution derivatives;
- **Robin**: specify a relation between values and derivatives;

but who really expects to "control" the boundary of the ocean? Unless we have good intuition, specifying these kinds of boundary conditions will just send signals across the interior. We'd much rather investigate the natural behaviors that might arise in unforced conditions. Here are three boundary conditions more typical of this kind of study:

- **reflective**: the boundary behaves like a mirror;
- **free**: the boundary exerts no stress;
- **periodic**: the left and right boundaries are joined;

Geometry: Boundary Conditions

In the 1D case, here is one way to enforce these conditions for the variable U at node 1:

- **reflective**: U(1) = - U(2)
- **free**: U(1) = U(2);
- **periodic**: U(1) = U(N+1);

Geometry: Boundary Conditions

For our problem, it probably makes sense to use the free boundary condition for the height H, and either the reflective or periodic condition for the momentum HU.

The choice of boundary condition has an effect on the solution of the problem, and we should be aware of what we are trying to model.

The boundary conditions can also show up in the conservation laws! Remember, the conservation laws hold within the region, but at external boundaries we may find inflow or outflow.

Geometry: Estimating the Solution at Interfaces

We may need to estimate solution values at the midpoint M of the interface between the regions around nodes L and R.

Assuming both cells are the same size, then we might estimate

$$U(M) \approx \frac{U(L) + U(R)}{2}$$

This means we are modeling U as a piecewise linear function between any pair of neighboring nodes...but we are **not** saying that U is modeled by a piecewise bilinear function for arbitrary points in the domain. That would actually be tricky to work out.

Geometry: Estimating the Flux at Interfaces

We may need to estimate the flux $\frac{\partial F(U)}{\partial x}$ at the midpoint M of the interface between the regions around nodes L and R.

We can estimate this quantity as:

$$\frac{\partial F(U)}{\partial x} \approx \frac{F(U(R)) - F(U(L))}{\Delta X}$$

We can regard this as simply the slope of the linear interpolant between the values of F at $U(L)$ and $U(R)$.

Geometry: Estimating the Time Derivative

We wish to approximate the solution over the entire region for $NT + 1$ equally spaced times from T_0 to T_{NT}.

The initial condition gives us data for T_0.

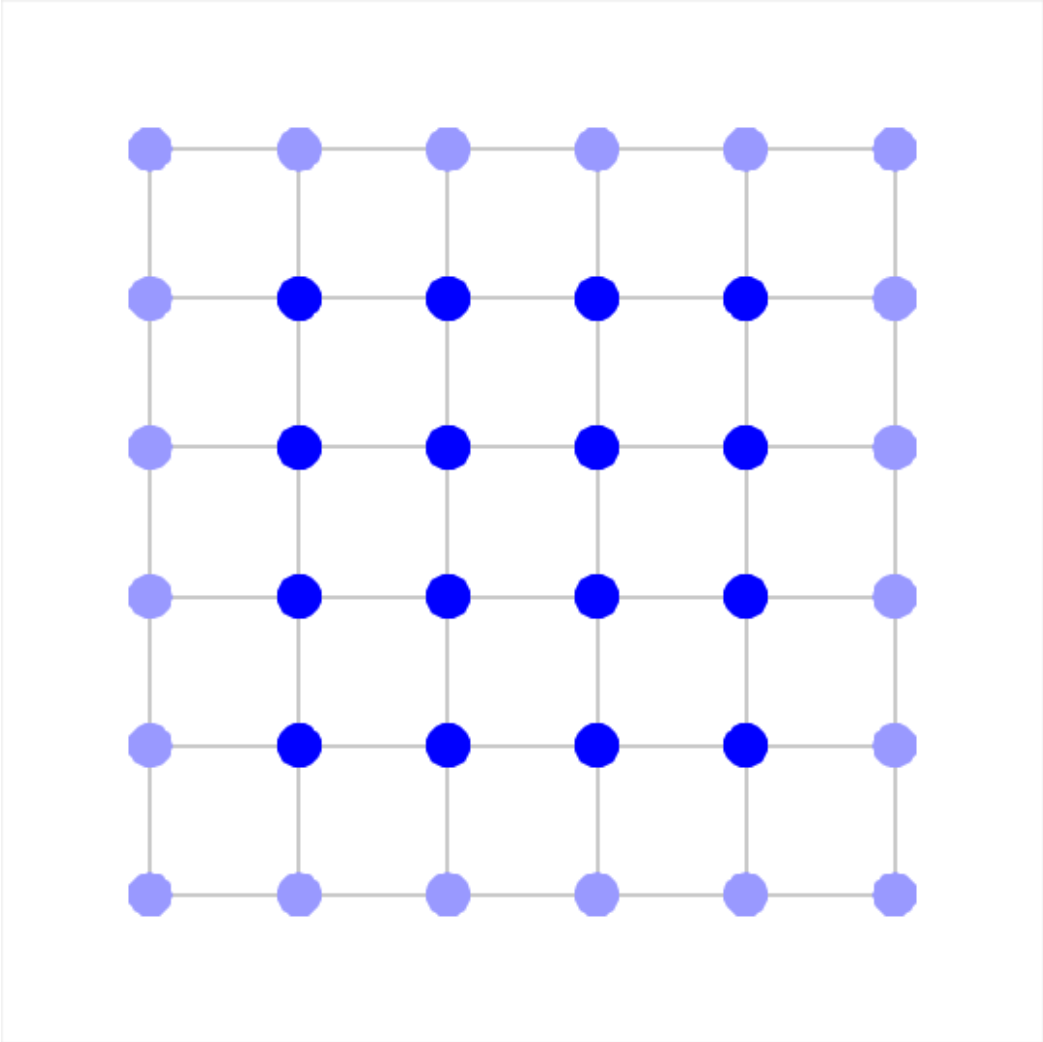
We will be using an explicit marching algorithm.

Assuming we have the approximation for time T_k, we use that information to estimate the time derivatives of the data, and march forward one further time step.

We can estimate a time derivative by

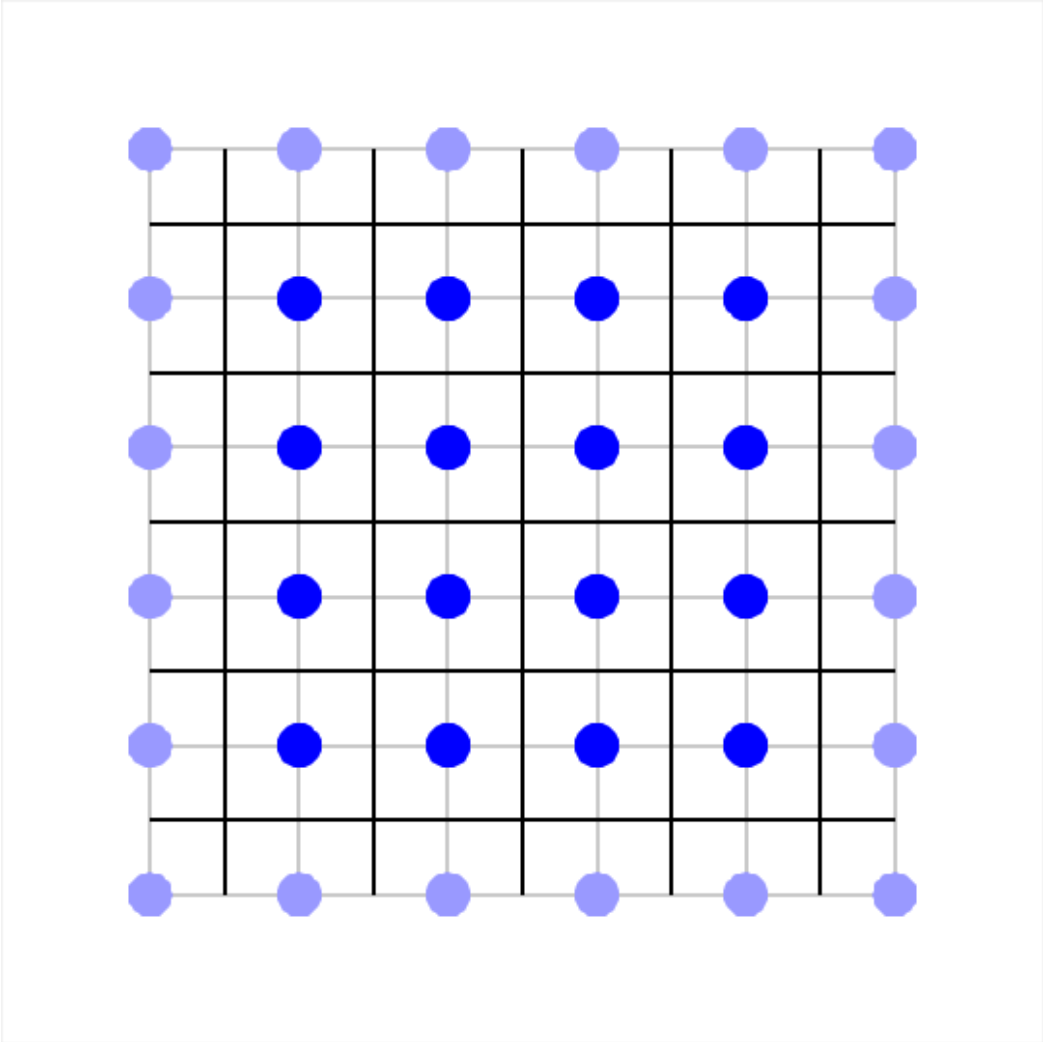
$$\frac{\partial U_{ij}}{\partial t} \approx \frac{U_{ij}^{k+1} - U_{ij}^{k}}{\Delta T}$$

Geometry: Seek Solution Values at Nodes



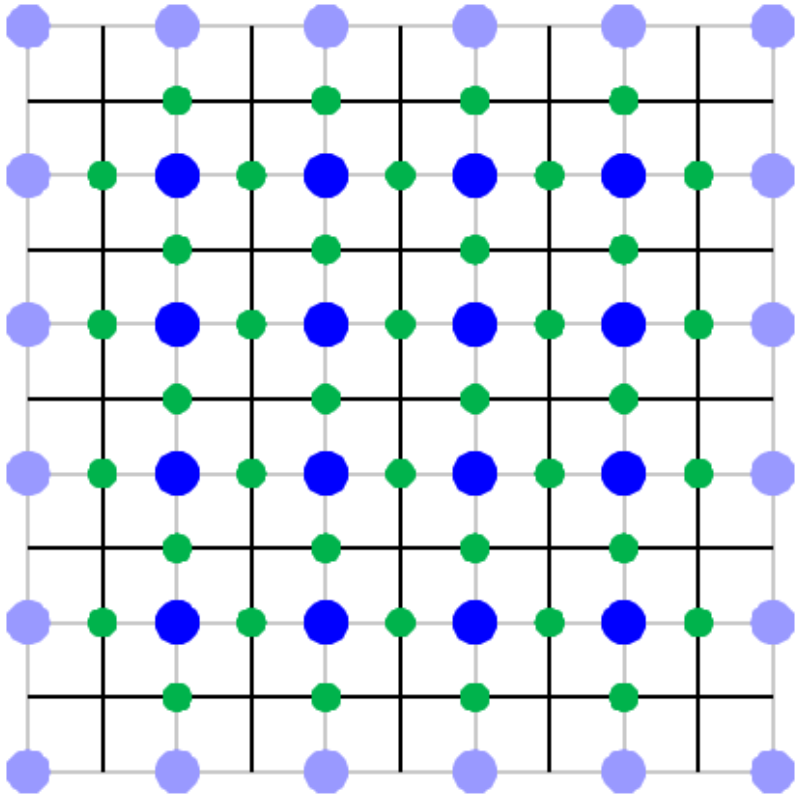
The light blue points will handle boundary conditions.

Geometry: Interior Nodes Define Square Regions



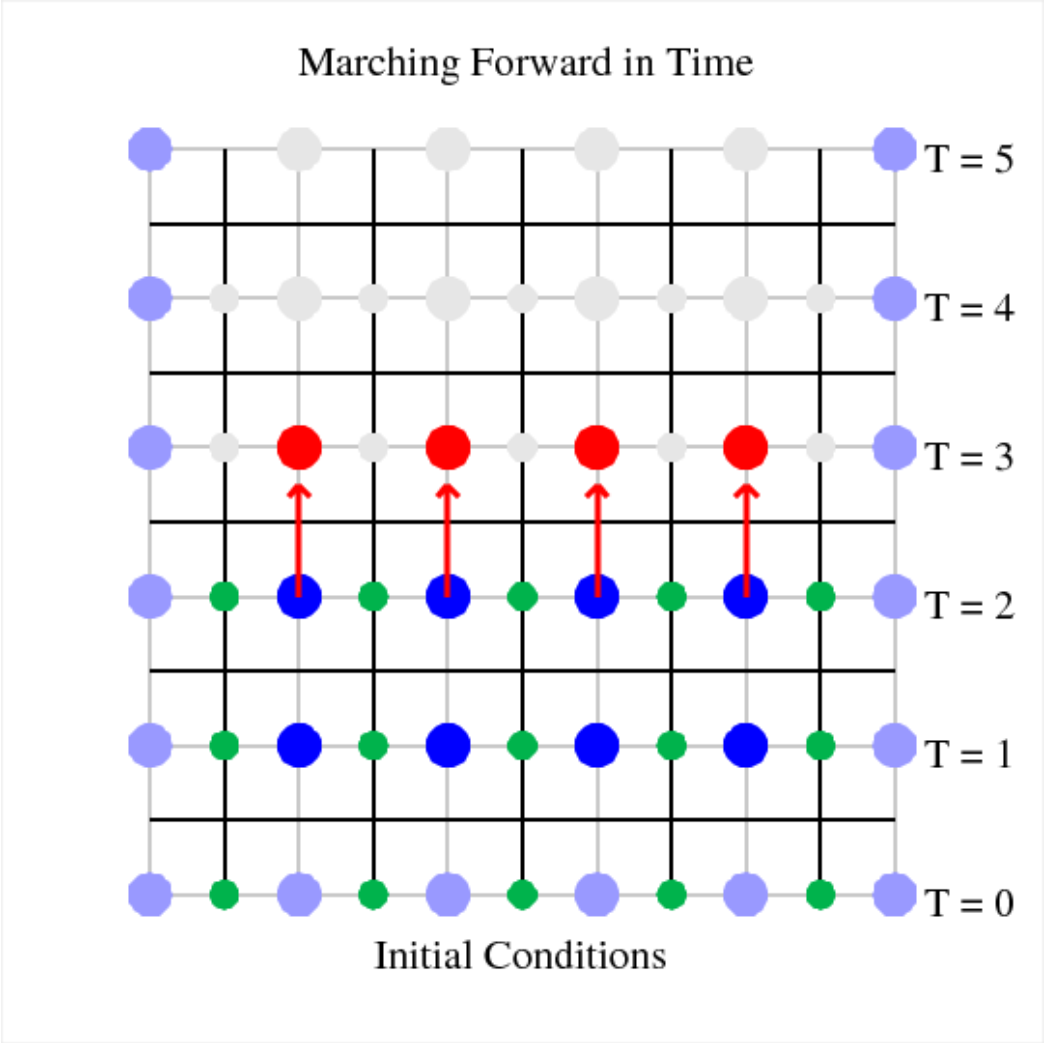
Physical properties are assumed constant in each region.

Geometry: Neighboring Nodes Define Fluxes



Fluxes will be estimated at the green nodes.

Geometry: Solve One Time Level at a Time



Here, the vertical axis represents time.

Discretization

We start with the solution at time level 0. In order to advance to the next time level, we simply need to write a discrete version of the shallow water equations, using the time derivative term to tell us what the values should be at the next time level.

So now it's time to put together our approximations for the fluxes and time derivatives, so we can prescribe our marching algorithm.

Discretization: A Simple Finite Difference Approach

It's easy to write an explicit finite difference stencil for the equations.

The fluxes can be estimated by centered differences:

$$\frac{\partial F(U)}{\partial x} \approx \frac{F(U^{right}) - F(U^{left})}{\Delta X}$$

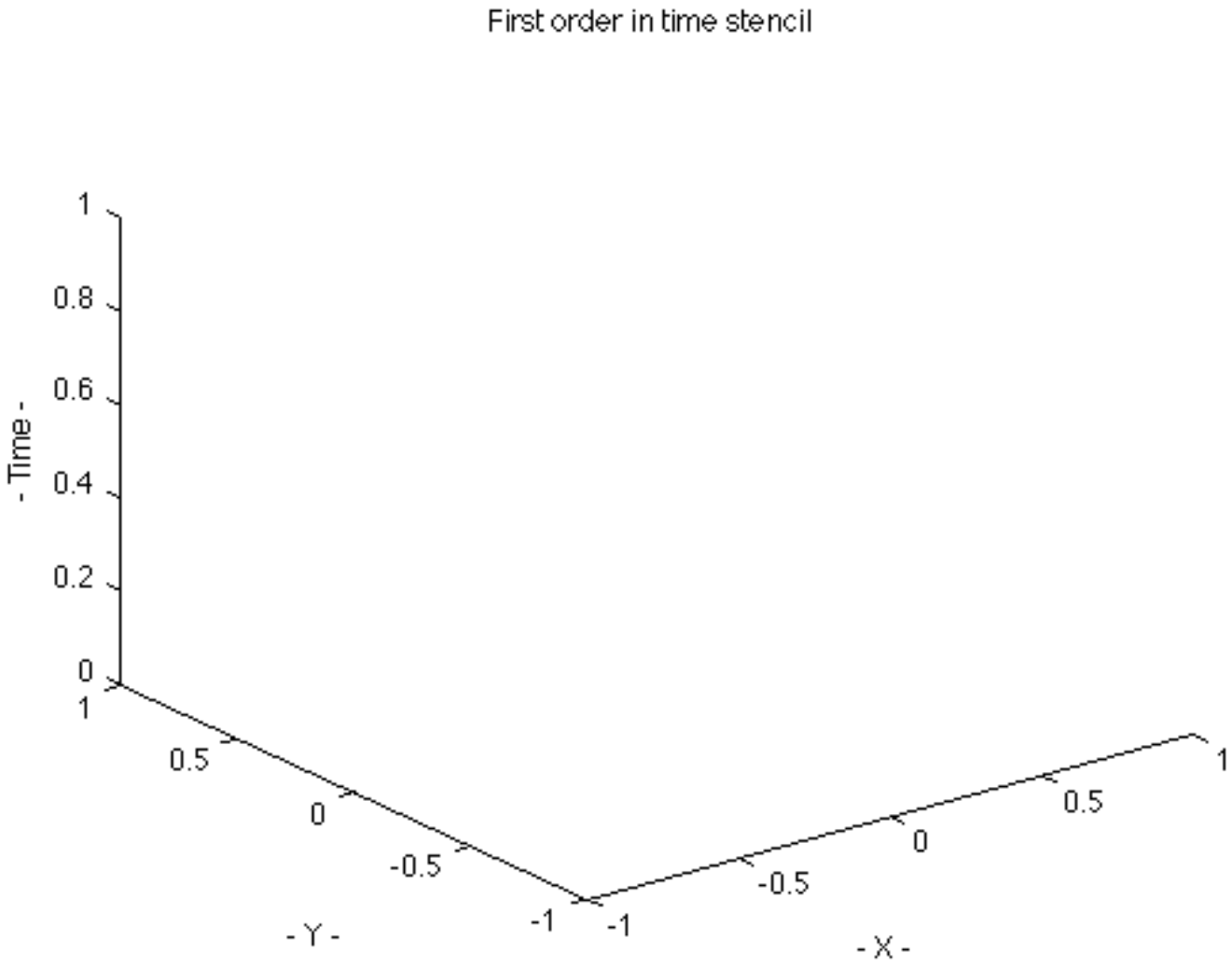
and the time step by a forward difference:

$$\frac{\partial U}{\partial t} \approx \frac{U^{new} - U^{current}}{\Delta T}$$

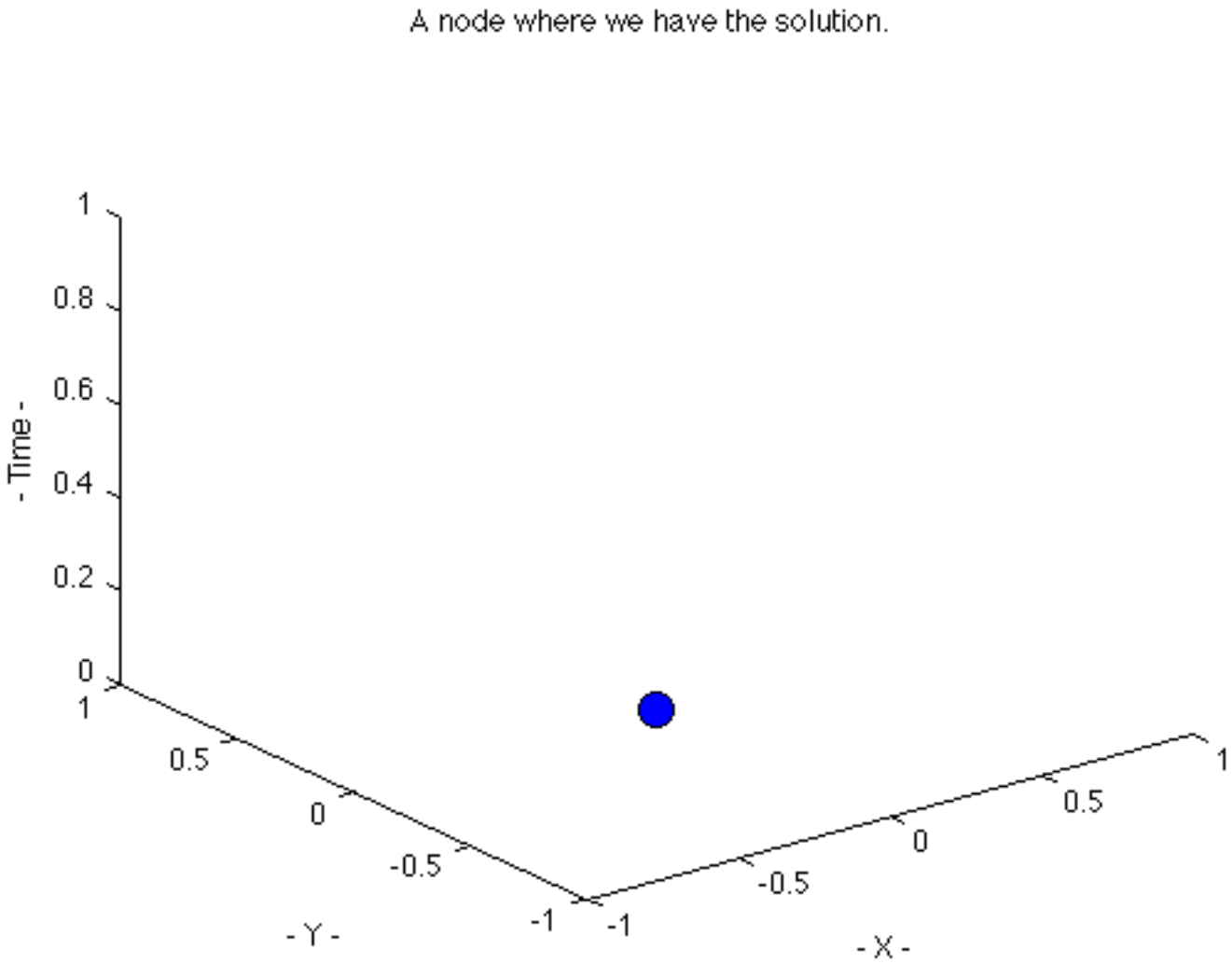
resulting in:

$$\frac{U^{new} - U^{center}}{\Delta T} + \frac{F(U^{right}) - F(U^{left})}{\Delta X} + \frac{G(U^{up}) - G(U^{down}))}{\Delta Y} = 0$$

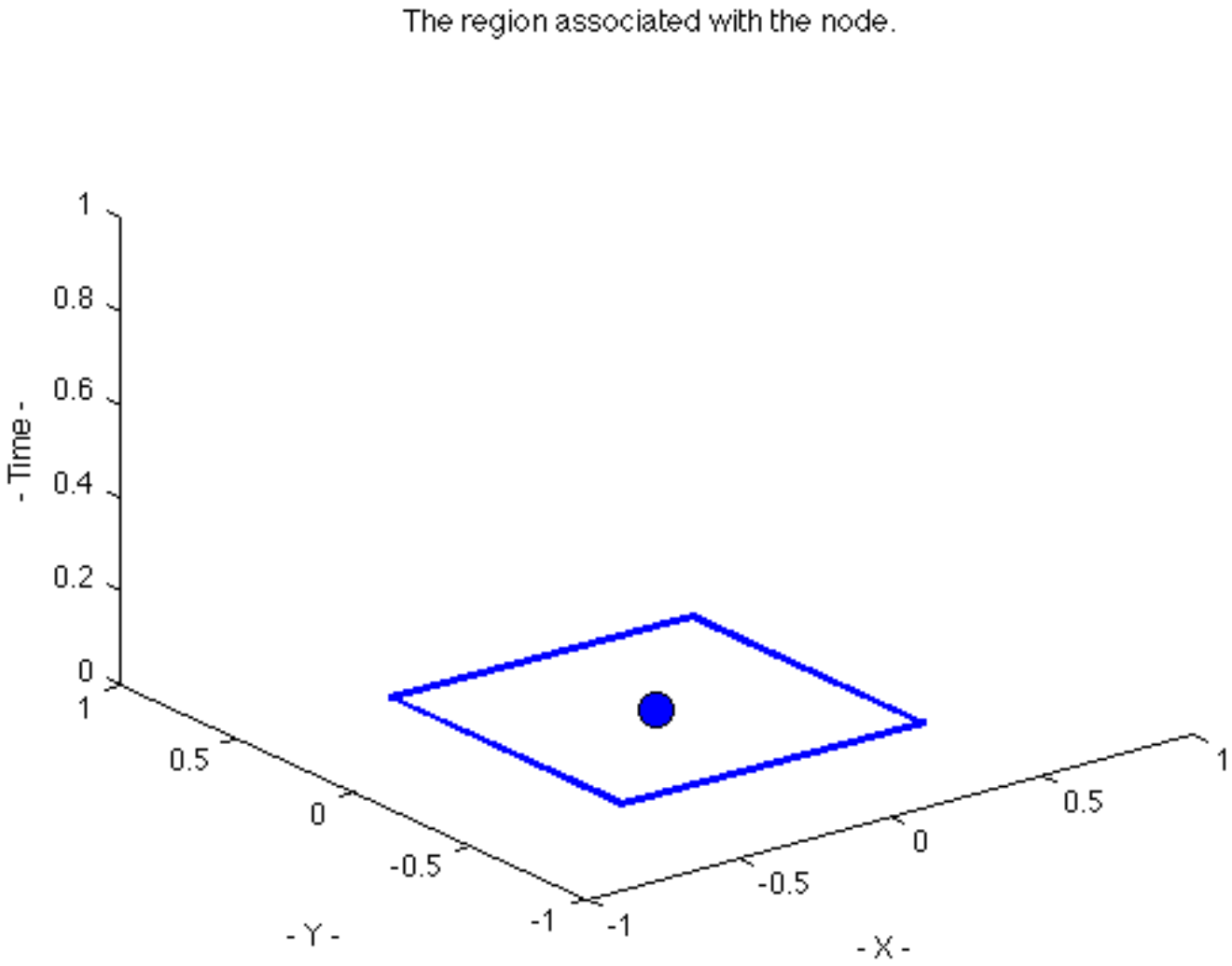
Discretization: First Order Scheme



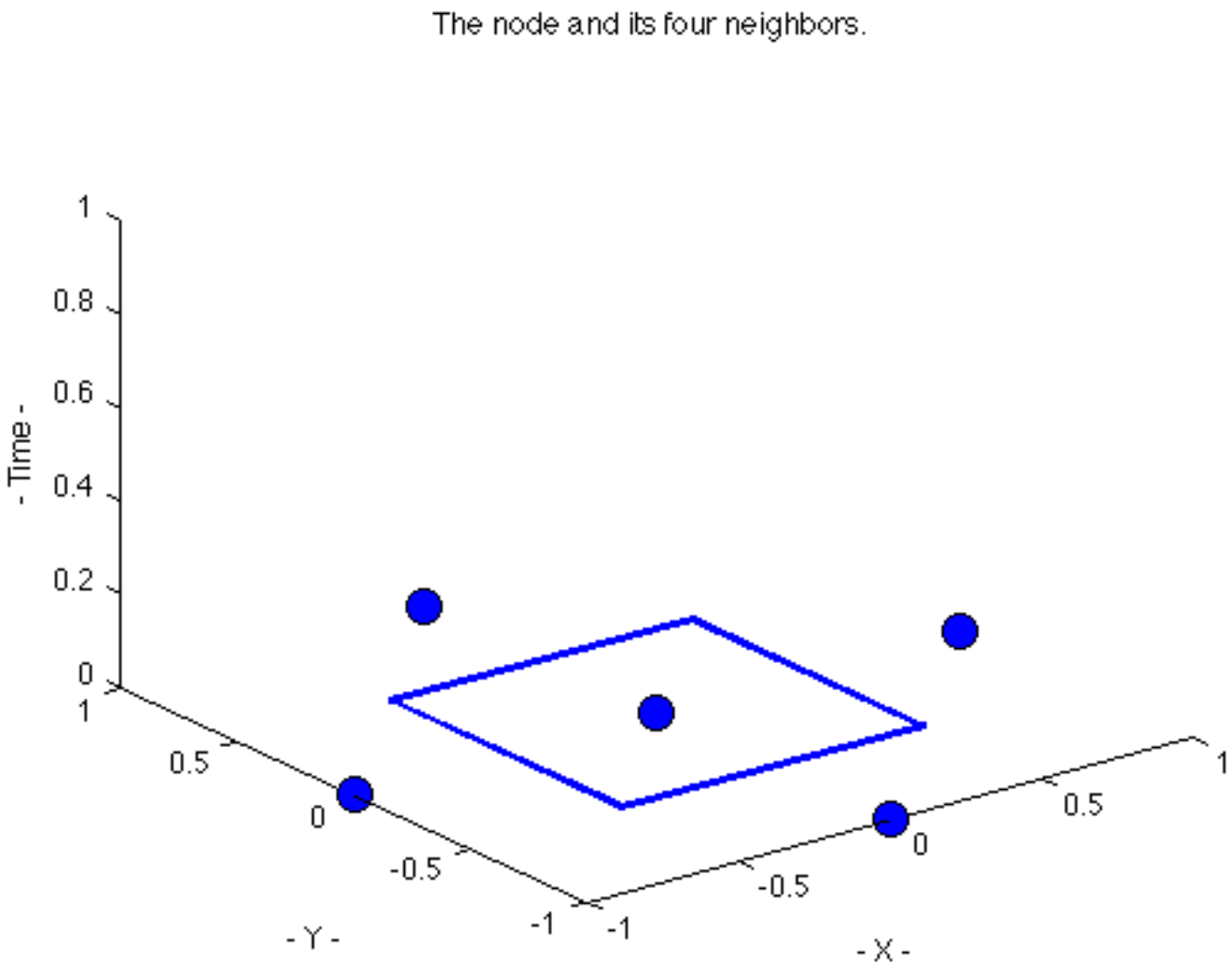
Discretization: First Order Scheme



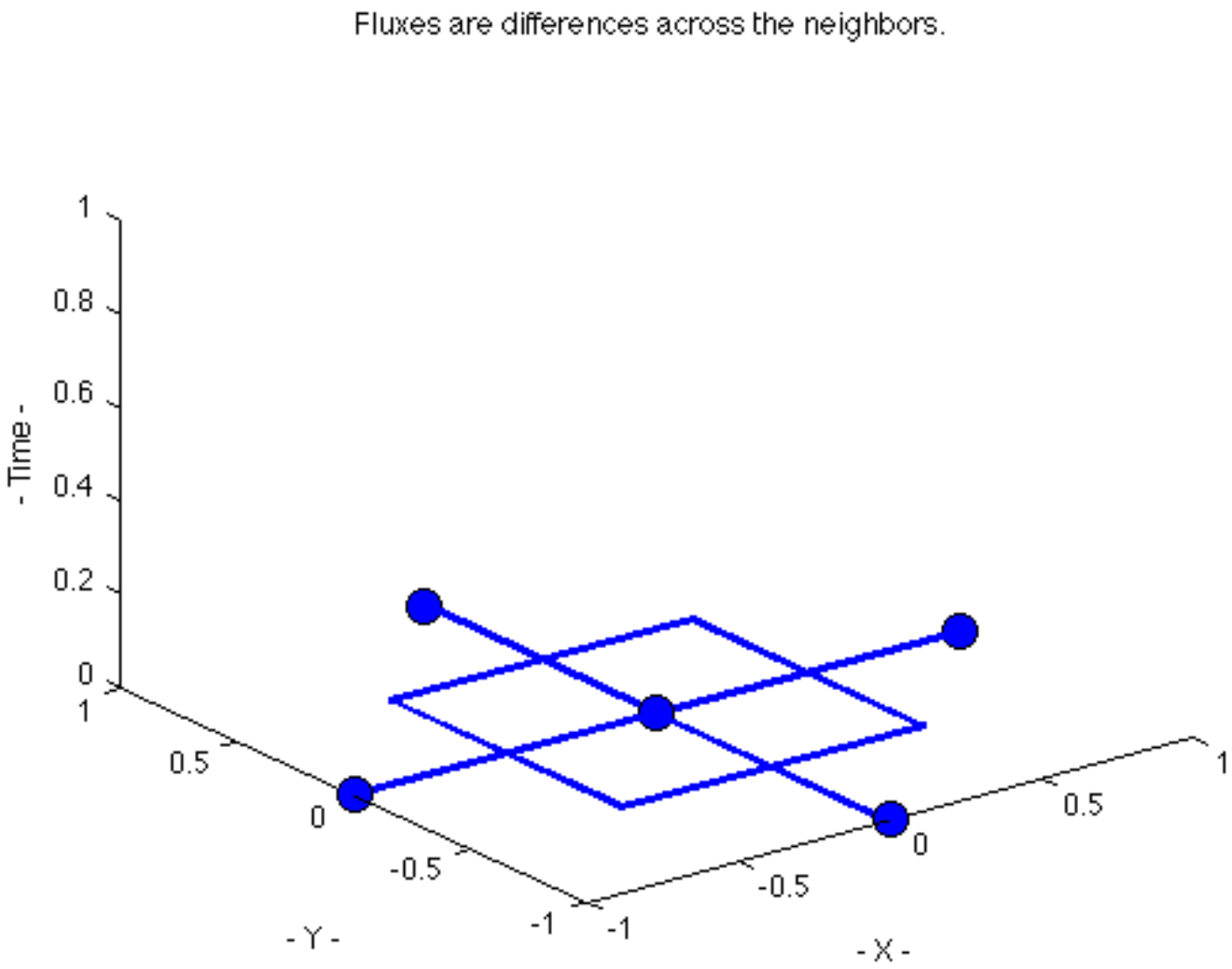
Discretization: First Order Scheme



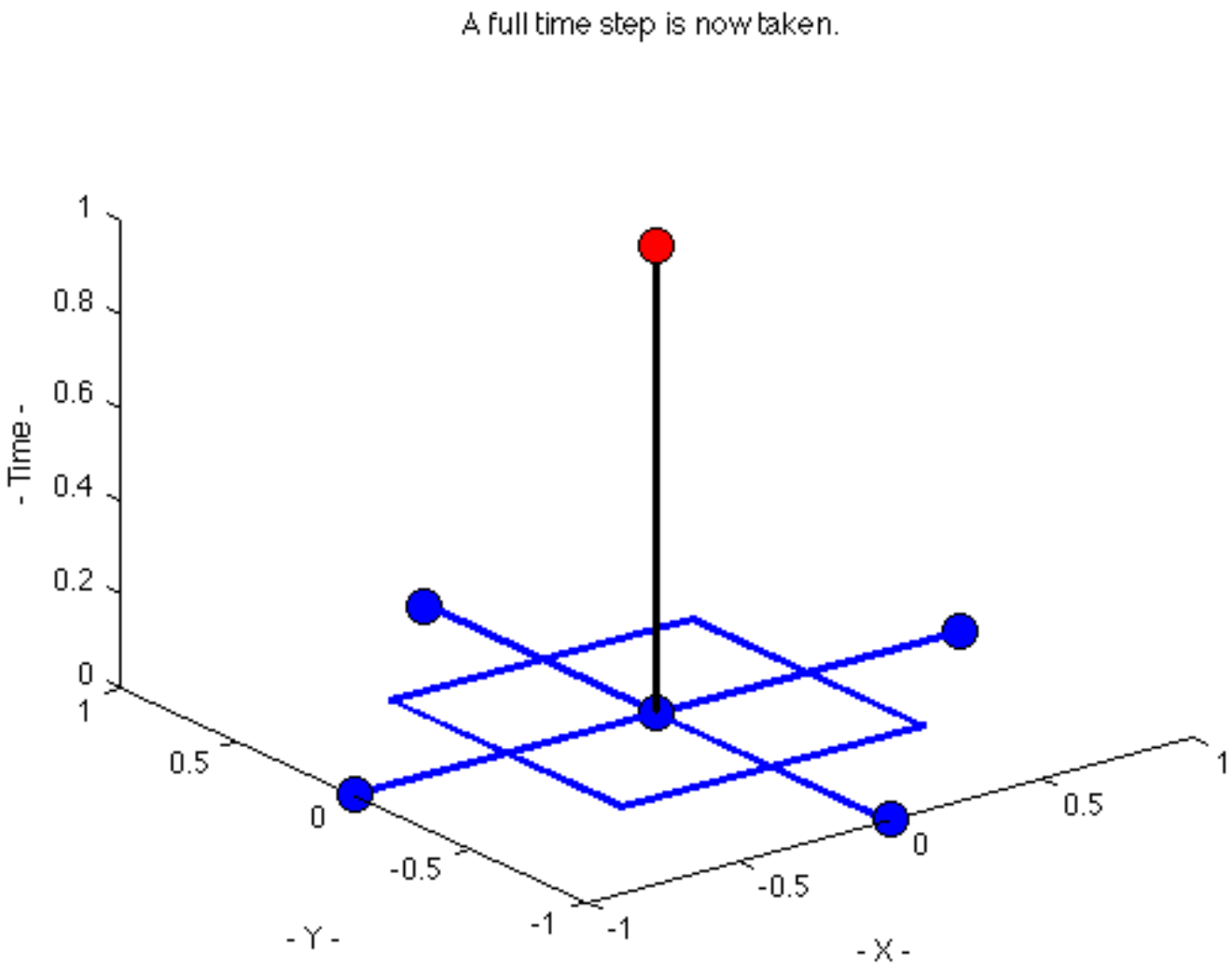
Discretization: First Order Scheme



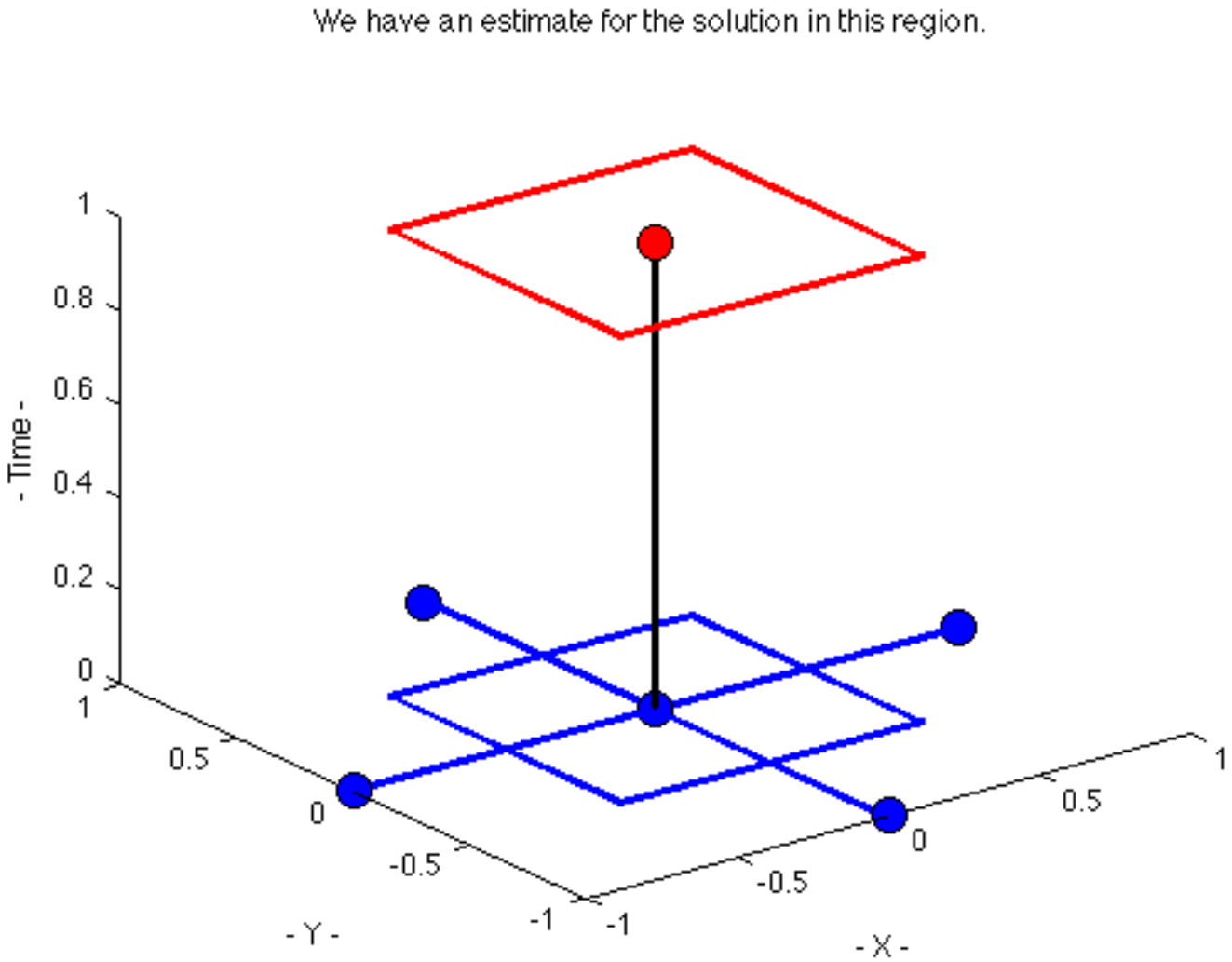
Discretization: First Order Scheme



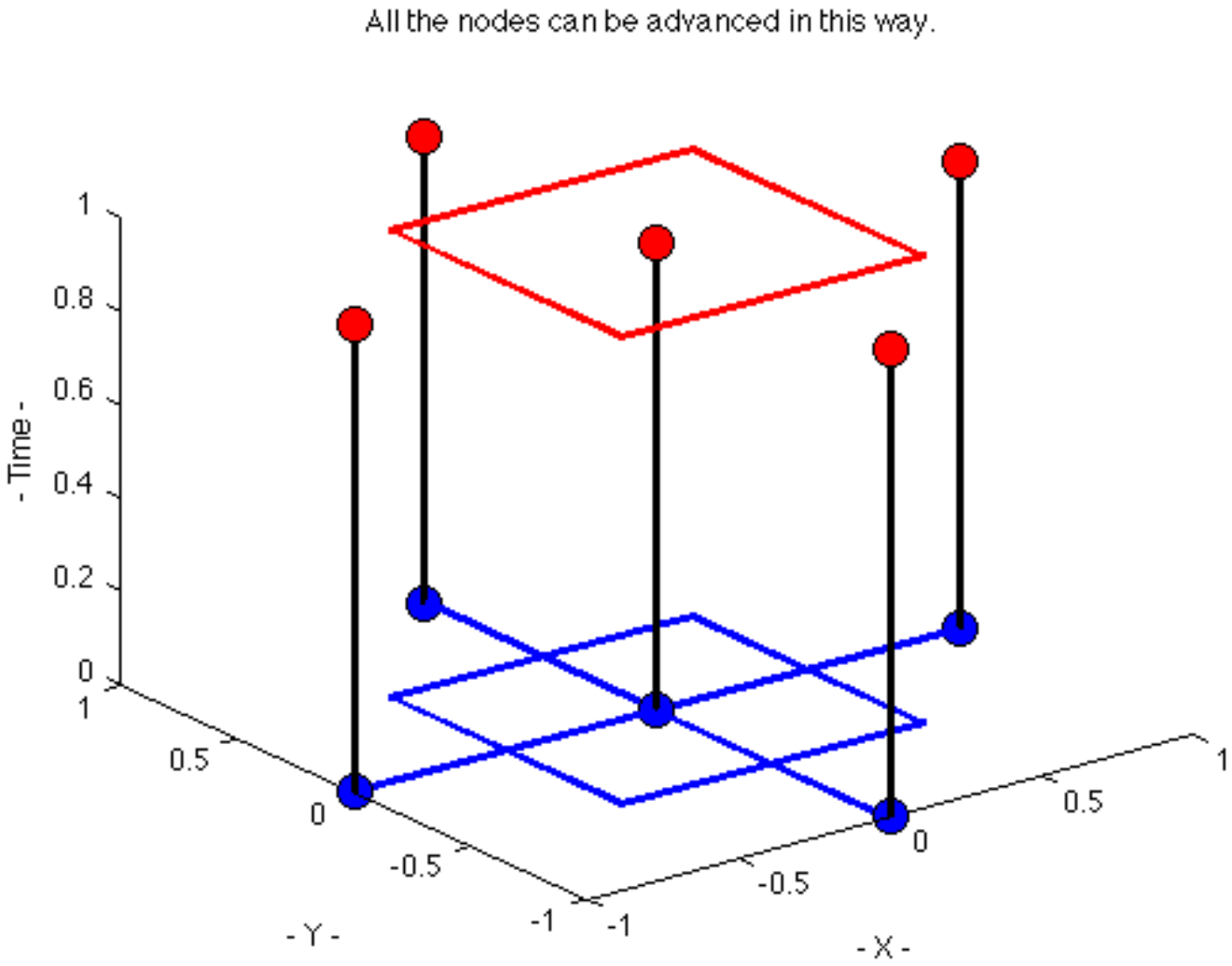
Discretization: First Order Scheme



Discretization: First Order Scheme



Discretization: First Order Scheme



Discretization: The Lax-Wendroff Stencil

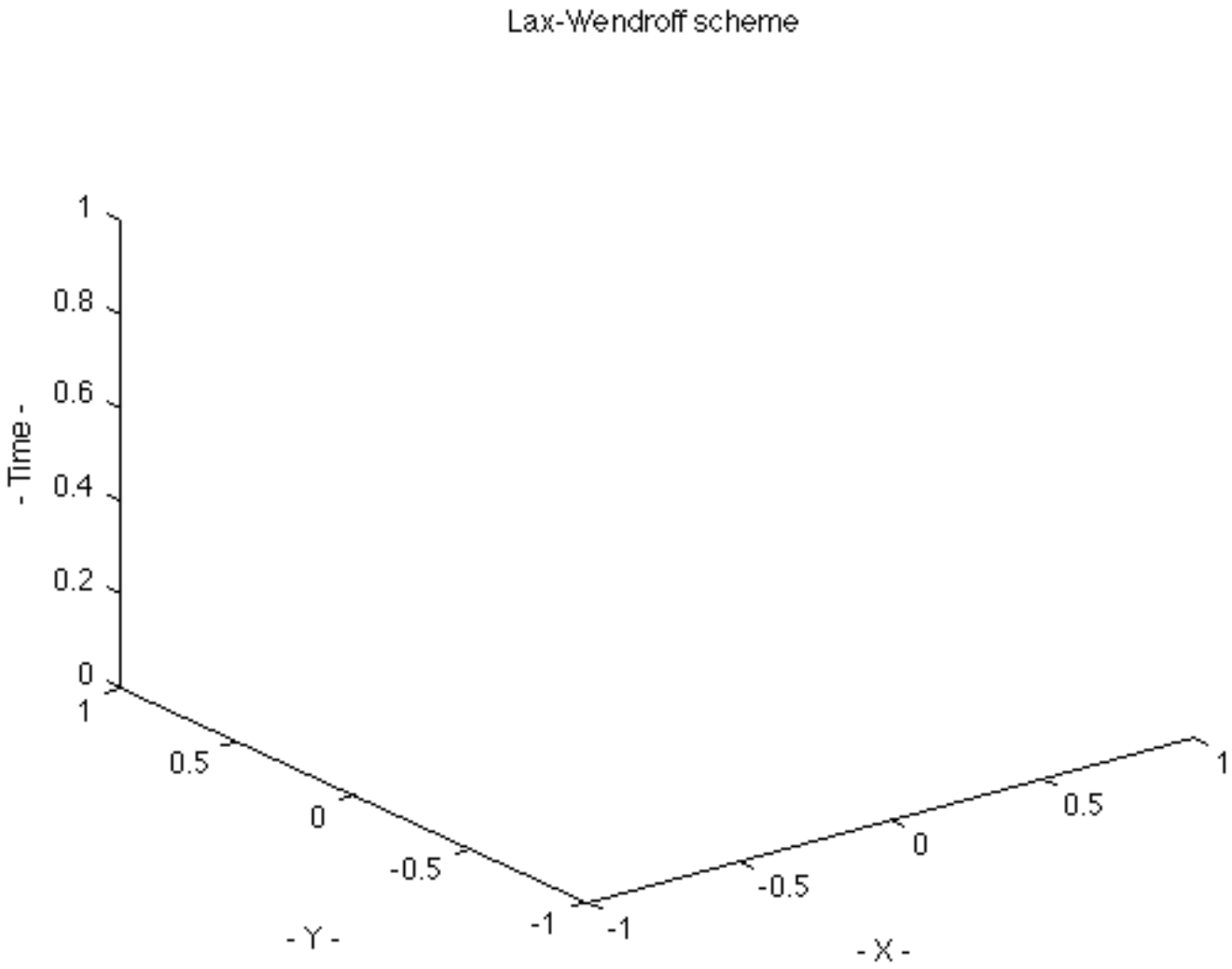
Our approximation to the fluxes is second order accurate, but our time integration method is a simple Euler method, which is only first order accurate.

As a better approach, we consider the **Lax-Wendroff scheme**. This is based on an integration scheme known as the *modified Euler method*, or the *Euler half-step method*, or the *Runge-Kutta 2 method* .

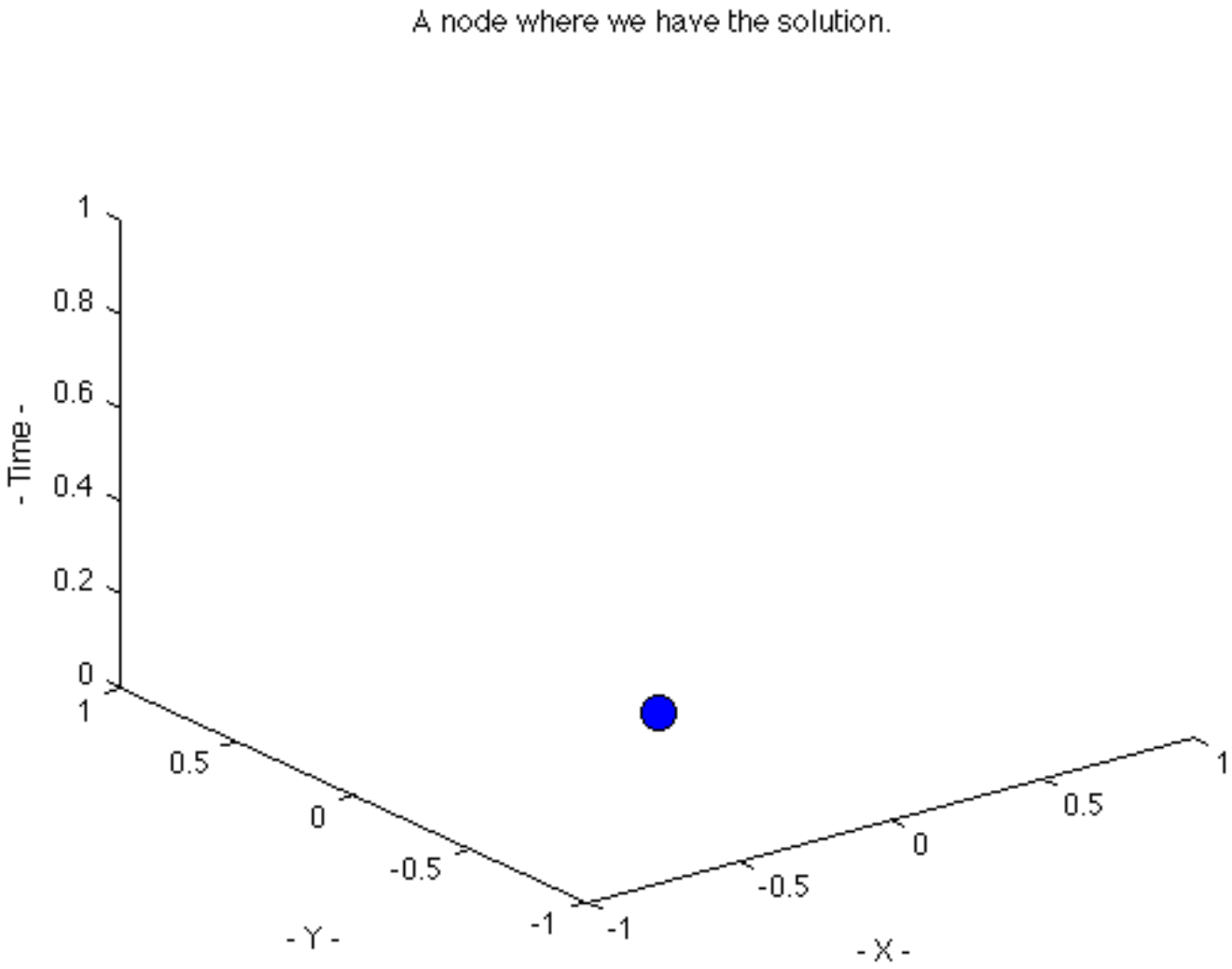
$$y_m = y_0 + \frac{1}{2}\Delta T\, f(y_0)$$
$$y_1 = y_0 + \Delta T\, f(y_m)$$

By taking a half-step and evaluating the derivative there, this method produces results which are second order accurate in time.

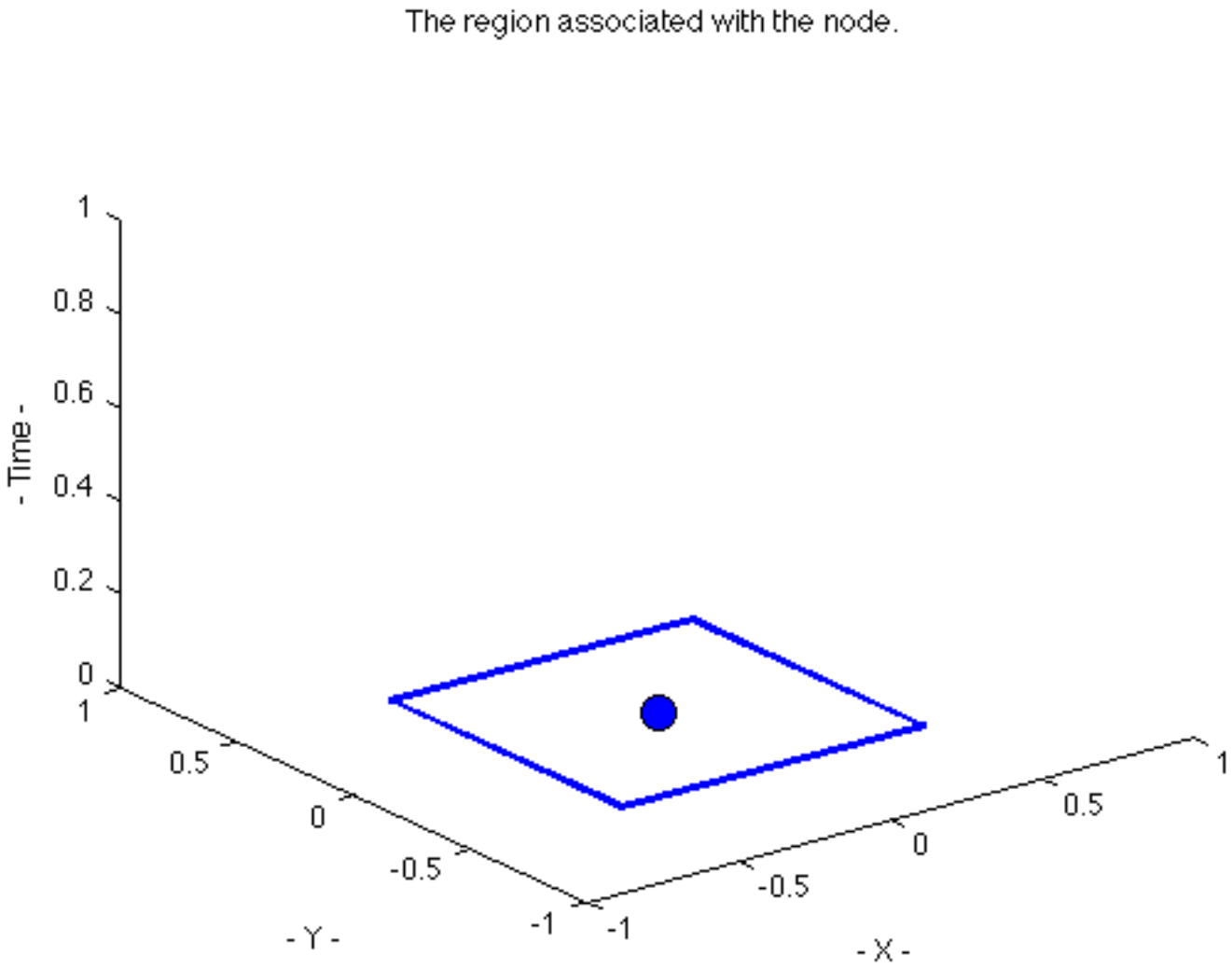
Discretization: Lax Wendroff Scheme



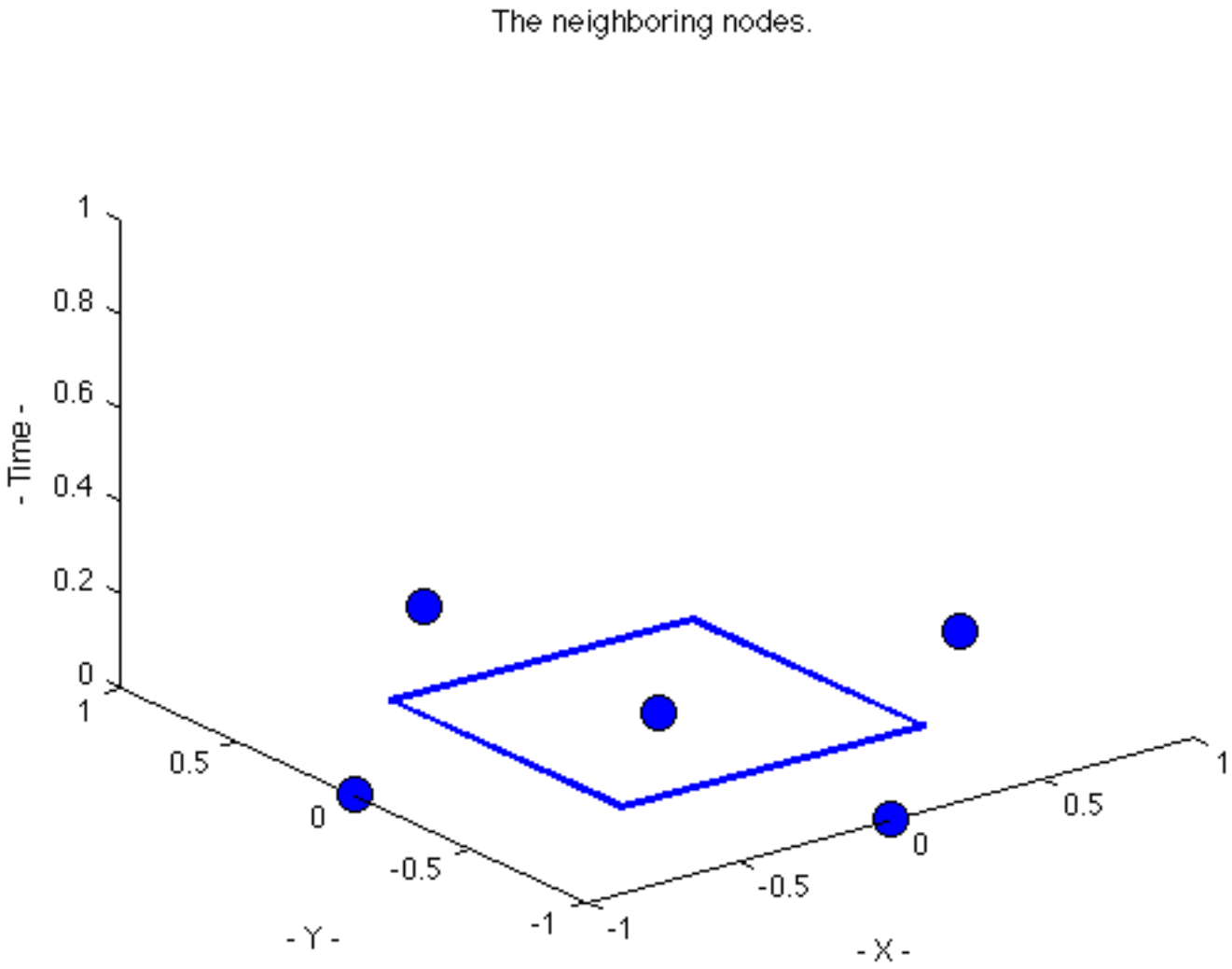
Discretization: Lax Wendroff Scheme



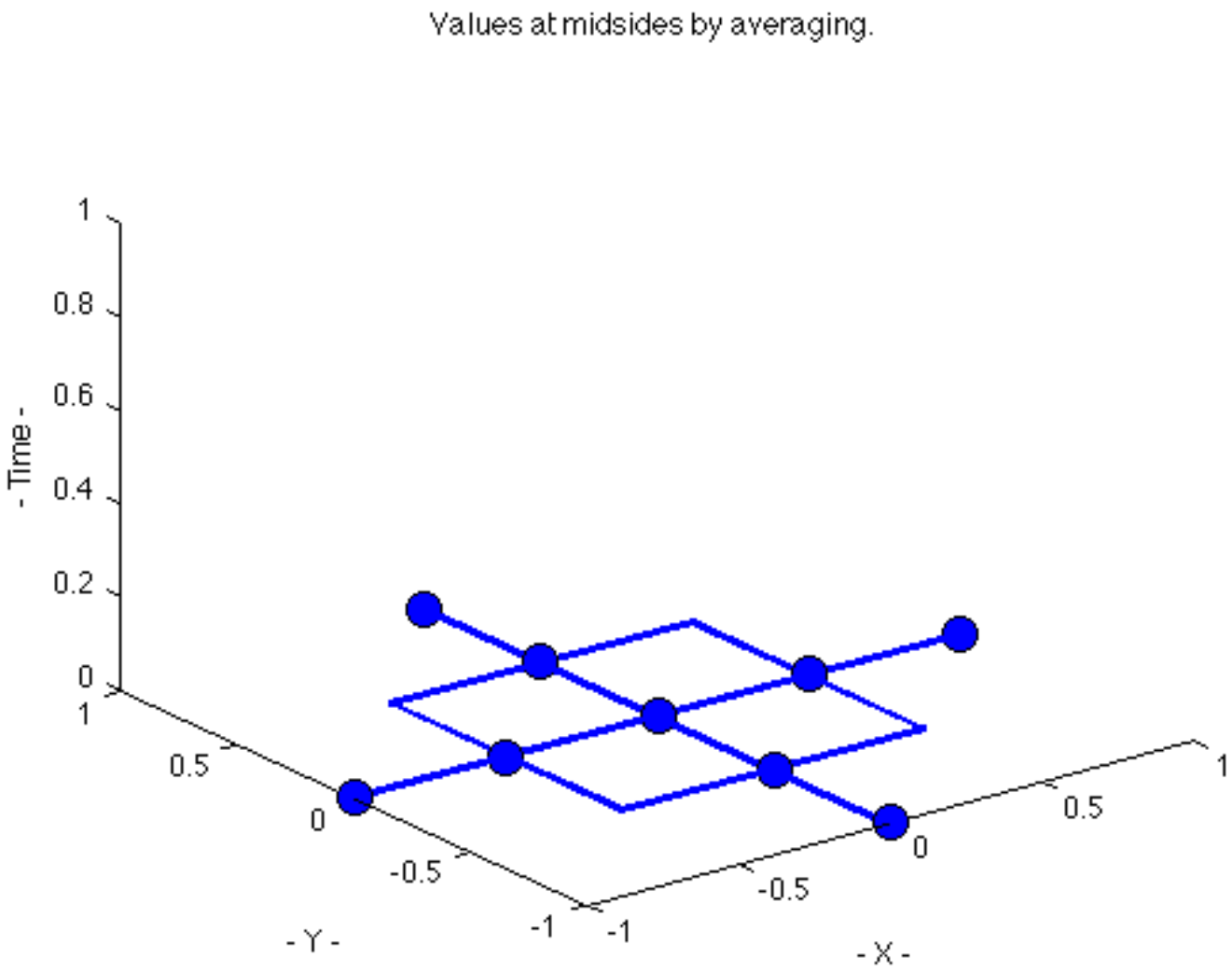
Discretization: Lax Wendroff Scheme



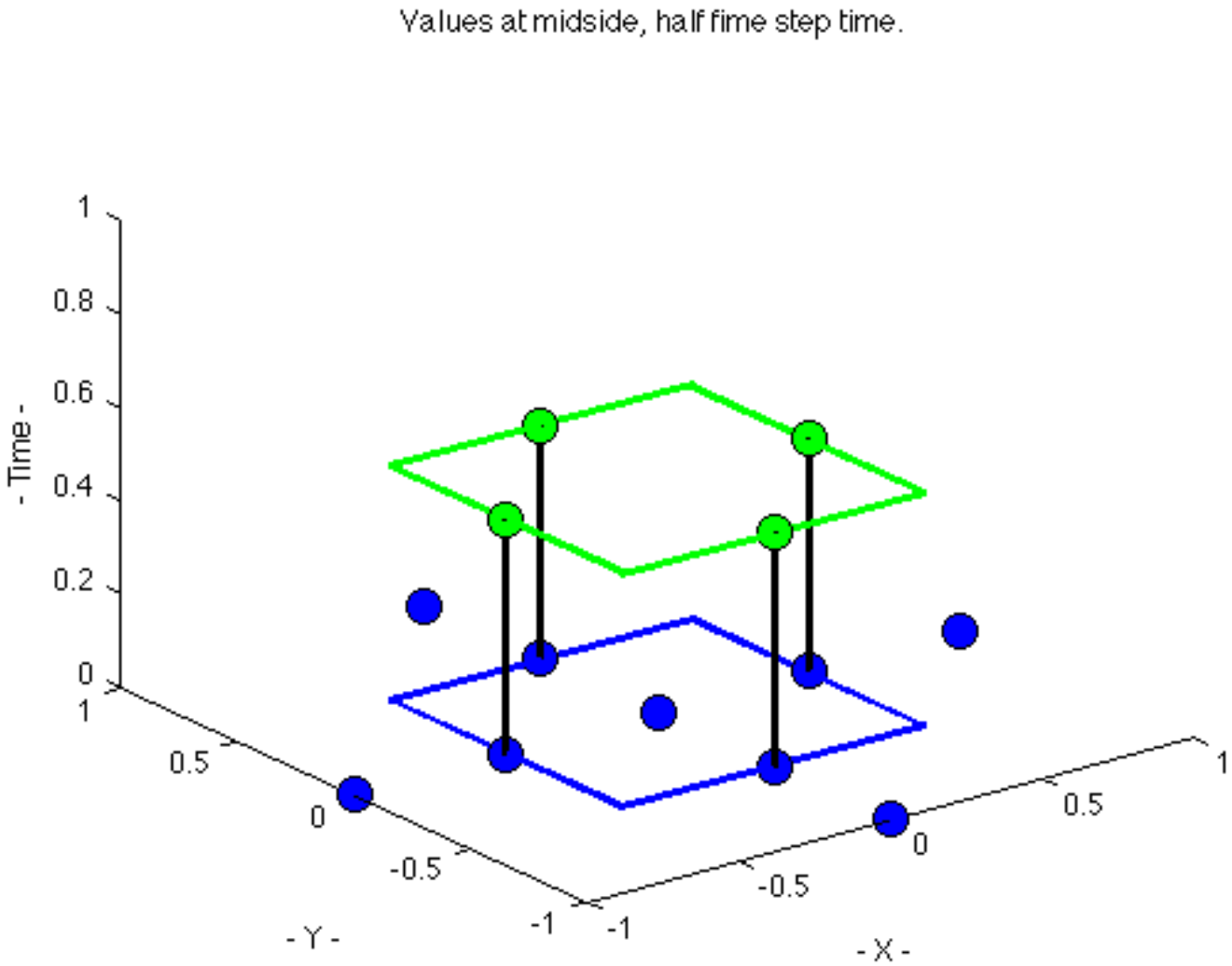
Discretization: Lax Wendroff Scheme



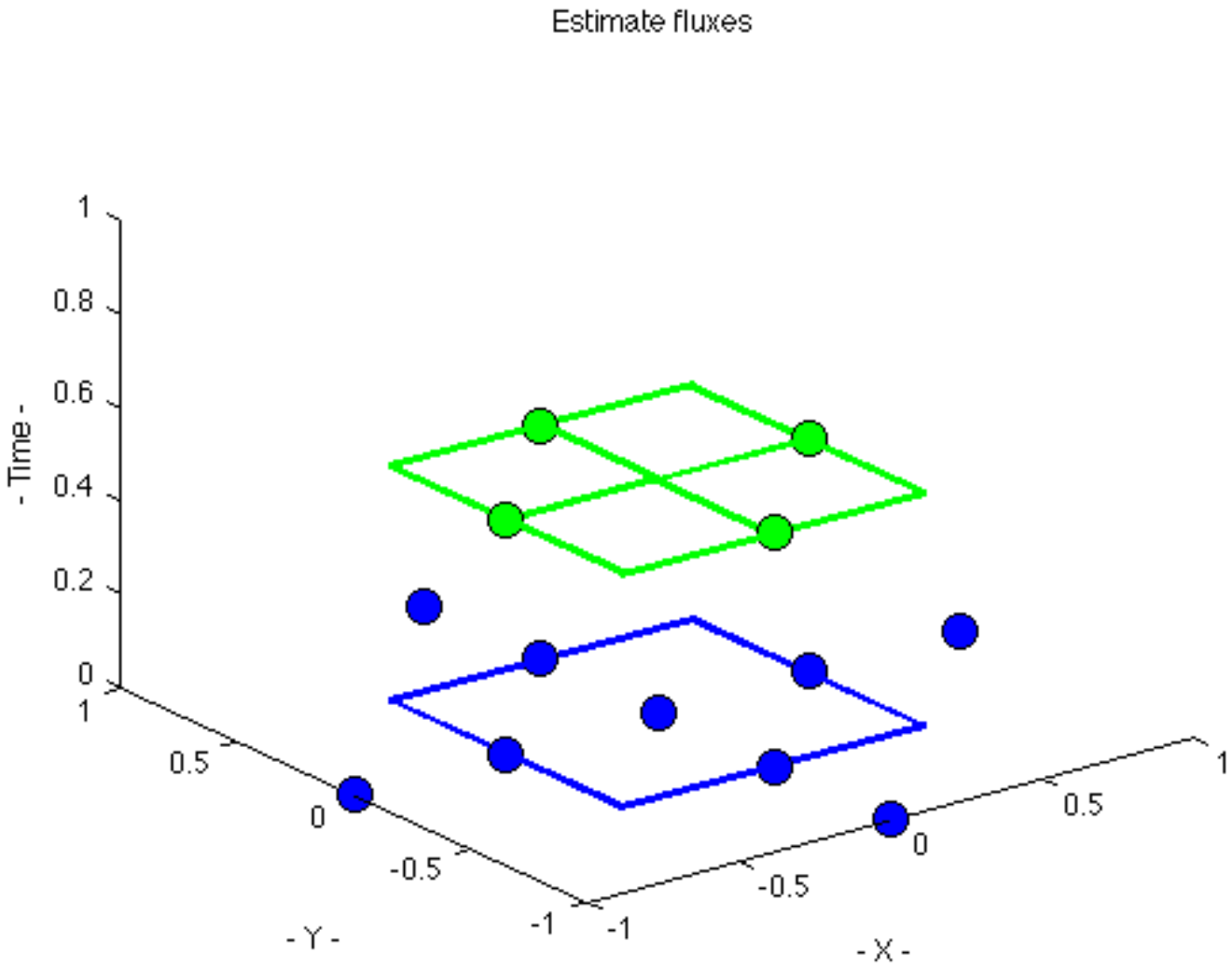
Discretization: Lax Wendroff Scheme



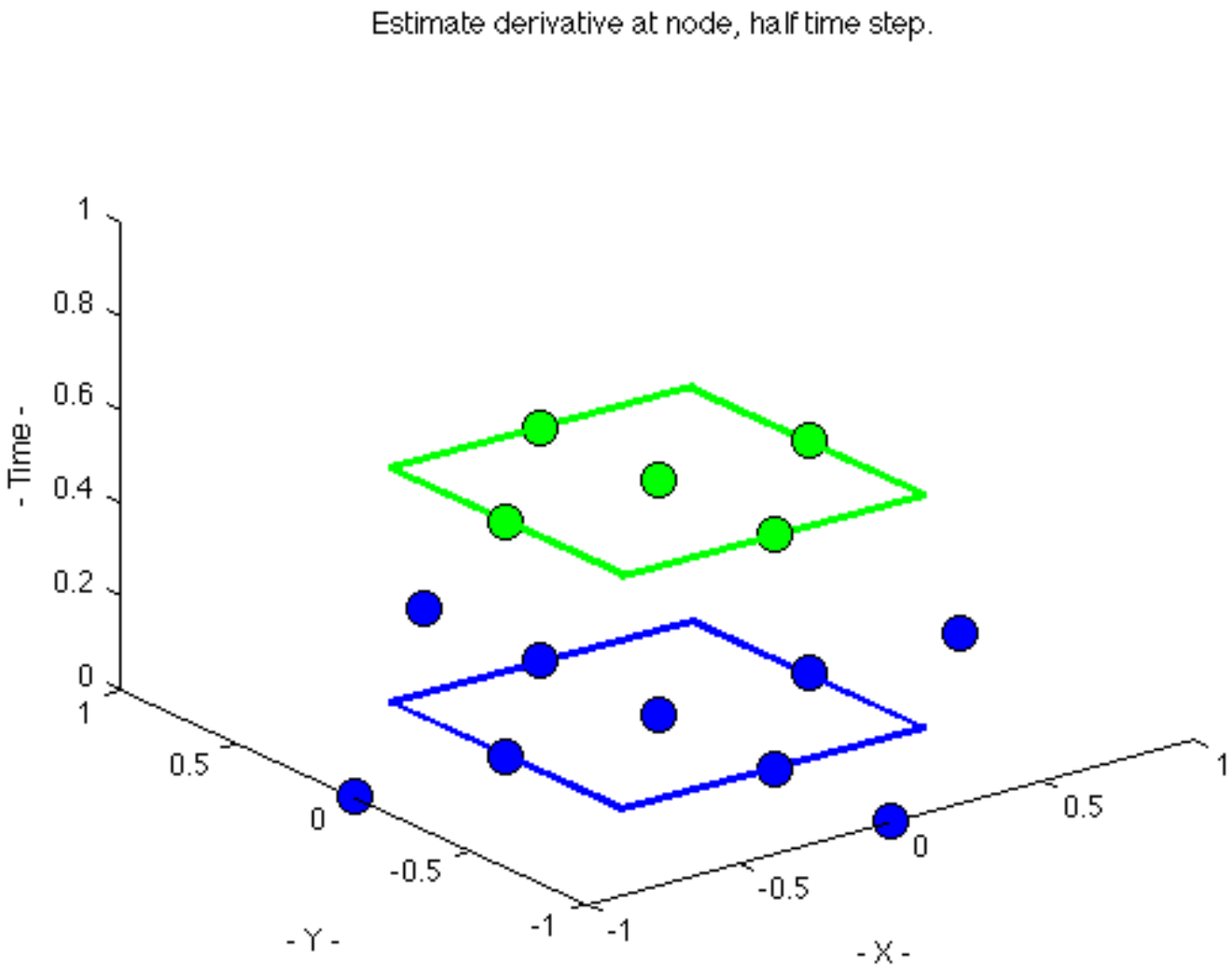
Discretization: Lax Wendroff Scheme



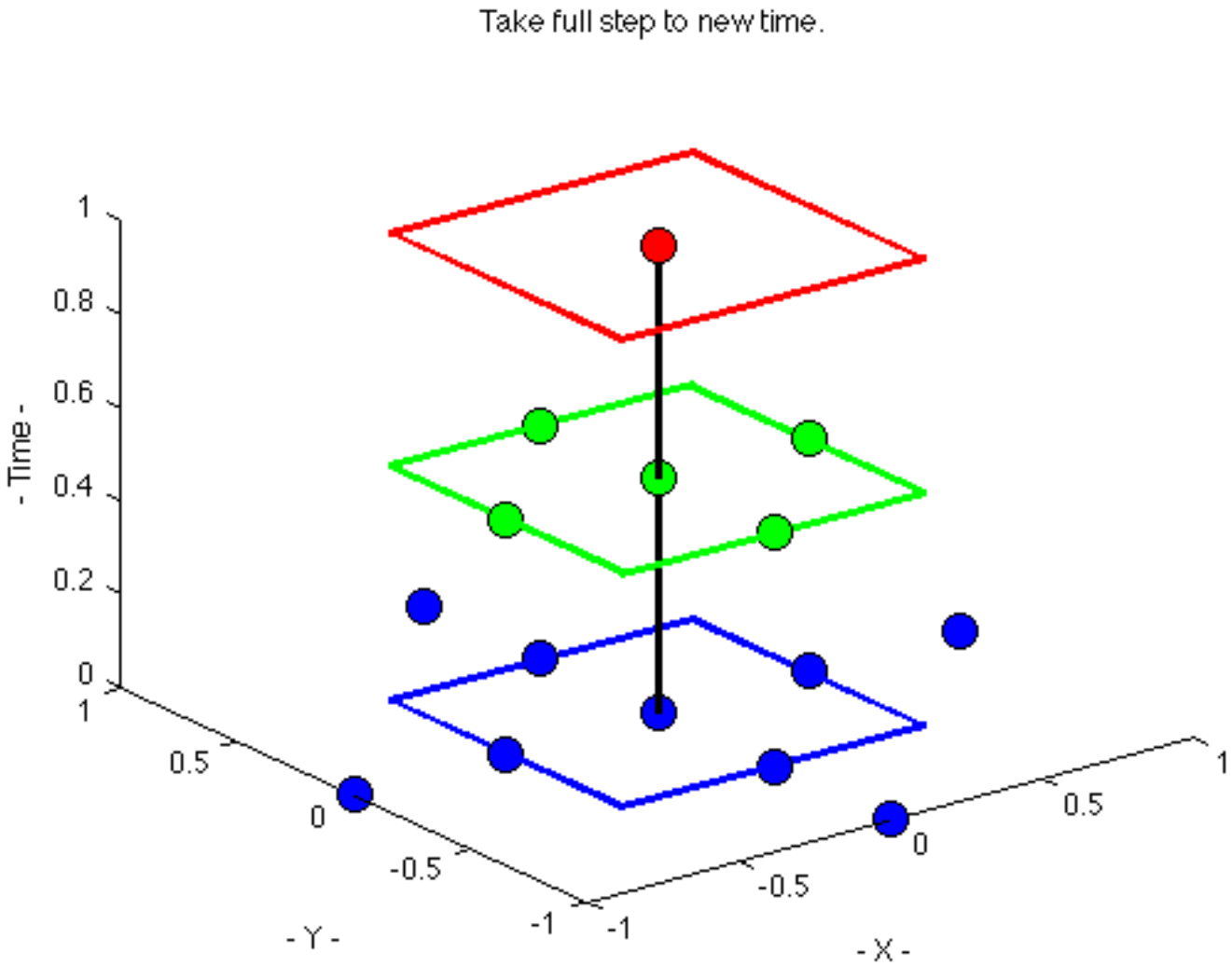
Discretization: Lax Wendroff Scheme



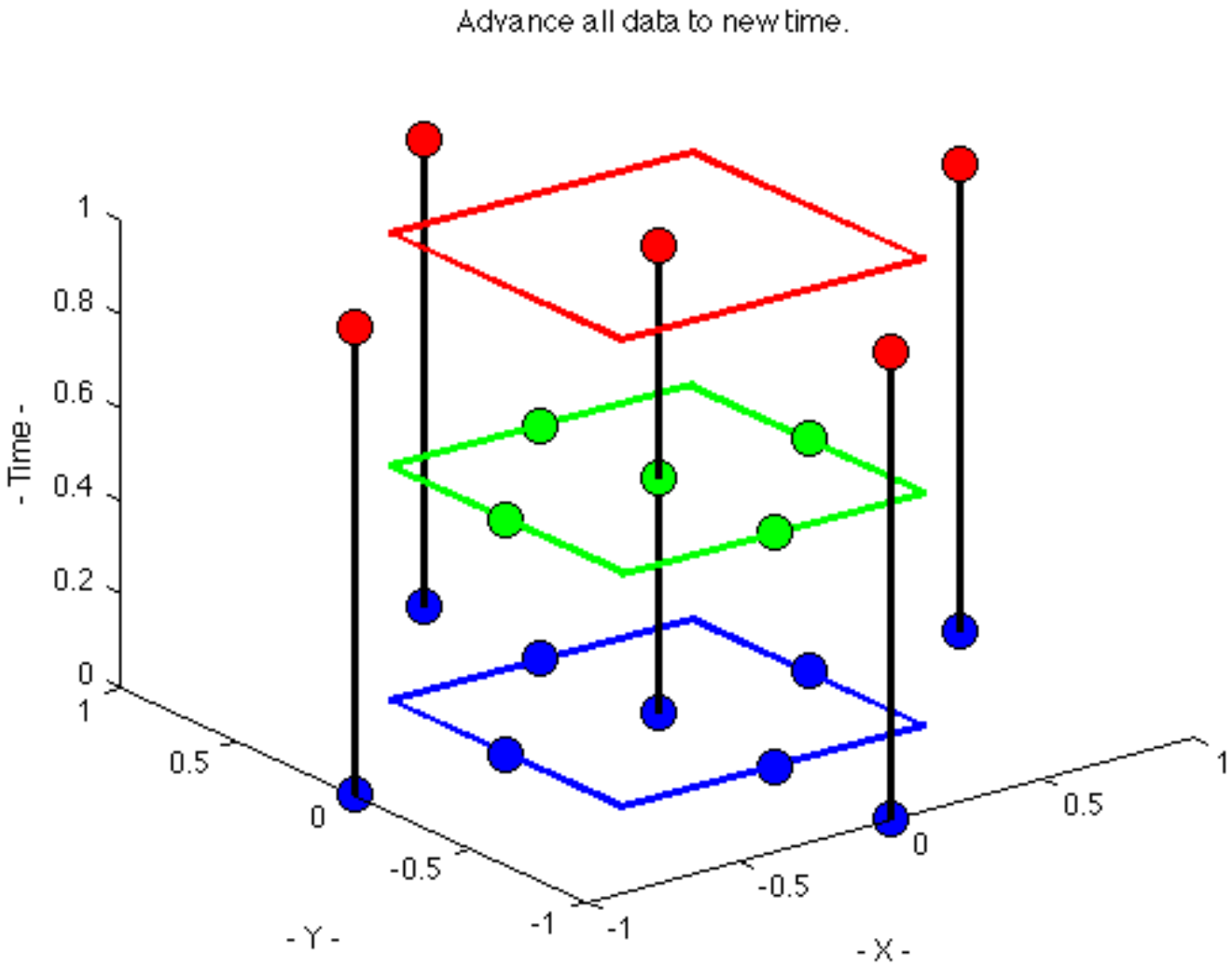
Discretization: Lax Wendroff Scheme



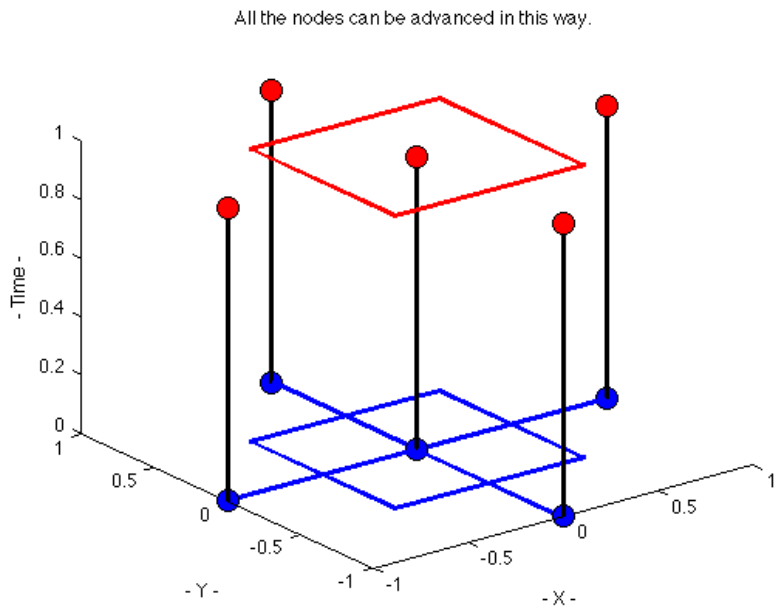
Discretization: Lax Wendroff Scheme

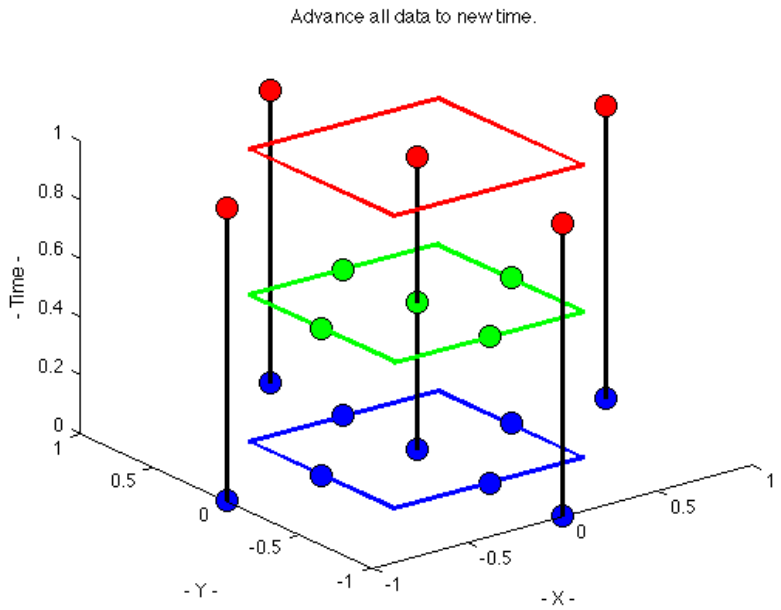


Discretization: Lax Wendroff Scheme



Discretization: Comparison





The first order scheme estimates the derivative at the current time.
Lax-Wendroff estimates it a half timestep into the future.

Introduction to Part 2:

We have talked about the physical problem, the mathematical formulation of the differential equations, and some discretizations that lead to a system of finite difference equations.

We also have suggested how we might handle the location of interior and boundary points, how to approximate spatial and time derivatives.

Now it is time to see how these ideas are assembled into a unified program for solving the shallow water equations.

1D Program: The "Training Wheels" Version

I will not concentrate on Moler's 2D code. Instead, I present a code for the **1D version** of the problem.

This simplifies many things.

It makes less impressive pictures; on the other hand, it will be easier to pose some experiments, to observe the simulated solutions, and to try to understand what they mean!

If you understand the 1D code, you will still find the 2D code an interesting but not impossible puzzle to work out!

1D Program: Command Line Input

The 1D program is invoked by the command:

[H, UH, x, t] = shallow_water_1d (nx, nt, x_length, t_length, g);

so that we can easily specify input:

- **nx** and **nt** are the number of space and time steps;
- **x_length** and **t_length** are the space and time step sizes;
- **g** is the magnitude of the gravitational force.

and retrieve useful output:

- **H** and **UH** contain the **nx** by **nt**+1 solution arrays;
- **x** and **t** contain the **nx** space and **nt**+1 time vectors;

---

http://people.sc.fsu.edu/~jburkardt/m_src/shallow_water_1d/shallow_water_1d.m

1D Program: Typical Input Values

A reasonable set of values to start with is:

[H, UH, x, t] = shallow_water_1d (41, 100, 1.0, 0.2, 9.8);

This means that **H** and **UH** will be arrays of dimension 41 x 101, while **x** will be a vector of length 41, and **t** a vector of length 101.

The spacing **dx** between nodes will be 1.0 / 40.

The spacing **dt** between time steps will be 0.2 / 100.

Gravity is set to 9.8 m/s/s.

1D Program: Initial Conditions

The user must modify the function which defines the initial conditions:

```
function [ h, uh ] = initial_conditions ( nx, nt, h, uh, x )

h(1:nx) = 2.0 + sin ( 2 * pi * x(1:nx) );
uh(1:nx) = 0.0;

return
end
```

This example starts with zero mass velocity, and an average height of 2 with a sinusoidal variation.

Another interesting flow would have h = 1 and uh = x. What might happen then?

1D Program: Boundary Conditions

The user must modify the function which defines the boundary conditions:

```
function [ h, uh ] = boundary_conditions ( nx, nt, h, uh, t )

h(1) = h(2);
h(nx) = h(nx-1);
uh(1) = - uh(2);
uh(nx) = - uh(nx-1);

return
end
```

This example sets **free** conditions for **h**, **reflective** conditions for **uh**.

1D Program: Structure

To fill up the H and UH arrays, we start with the initial condition, and then take repeated time steps. The program outline is:

1. initialize arrays, define constants like **dx** and **dt**
2. set solution at time step 0 by initial conditions
3. begin time loop on **nt** steps
4. estimate **h** and **uh** at cell midpoints, one half time step
5. evaluate derivative at half time step, take full step
6. apply boundary conditions
7. end of time loop

1D Program: The Half Step for H

The half step for **h** is carried out at the midway point **m** between two nodes **lm** and **rm**. We will estimate the time change in **h** here, so first we have to estimate the current value of **h** as the average of the neighbors.

$$h(m) \approx \frac{h(lm) + h(rm)}{2}$$

Now we also need the flux of **uh** at **m**, but that's just the difference of the values at the nodes to right and left of **m**:

$$\frac{\partial uh(m)}{\partial x} \approx \frac{uh(rm) - uh(lm)}{dx}$$

The DE then says that the time derivative of **h** plus the flux of **uh** is zero:

$$\frac{h^{\frac{1}{2}}(m) - h(m)}{dt/2} + \frac{uh(rm) - uh(lm)}{dx} = 0$$

1D Program: The Half Step for H

Rearranging, we have

$$h^{\frac{1}{2}}(m) = \frac{h(lm) + h(rm)}{2} - (dt/2) * \frac{uh(rm) - uh(lm)}{dx}$$

and if we use **hm** as the name of our temporary value for **h** at the half timestep and midway point, this corresponds exactly to the line in the program which computes this value at *all* the midpoints:

```
hm(1:nx-1) = ( h(1:nx-1) + h(2:nx) ) / 2.0 ...
  - ( dt / 2.0 ) * ( uh(2:nx) - uh(1:nx-1) ) / dx;
```

1D Program: The Full Step for H

In the same way, we have also estimated **uh** at each midpoint **m**, advanced a half step in time, and called this value $uh^{\frac{1}{2}}(m)$ or **UHM**.

Now we are ready to take a full step, to advance the value of **h** at each node **c**, using a derivative that is evaluated at the half time step, and using midside nodes **lc** and **rc** to the left and right of **c**:

$$\frac{h^1(c) - h^0(c)}{dt} + \frac{uh^{\frac{1}{2}}(rc) - uh^{\frac{1}{2}}(lc)}{dx} = 0$$

which in the code is written:

```
h(2:nx-1) = h(2:nx-1)
  - dt * ( uhm(2:nx-1) - uhm(1:nx-2) ) / dx;
```

1D Program: The Half Step and Full Step for UH

Similar calculations are used for **uh**. Of course, we must take the half step in both **h** and **uh** before taking the full step.

Also, the right hand side for **uh** includes terms like $u^2 h$. To express this in terms of our variables, we must write expressions like **(uh * uh)/ h**!

Overall, the treatment of **uh** is the same as **h**, so we won't discuss it further.

1D Program: Storing All the Data

The program really only needs two vectors of length NX to store H and its half-step update, and two more for UH.

However, if we want to plot the data at the end we need to save all the values at every time step. (This would not be practical for large problems.)

Inside the program, the vectors are called *h* and *uh*, using lowercase. The corresponding arrays *H* and *UH* are used to save a copy of these quantities for the initial condition and for every time step (which is why there are NT+1 sets of values!).

These arrays, plus the x and t grids, are the output of the program.

1D Program: Output

We said that the program outputs some large arrays: H, UH, x and t.

This output can be used as input to a program that will display successive snapshots of the solution:

shallow_water_1d_display (H, UH, x, t)

will display the height and mass velocity at each time step, advancing as you hit RETURN.

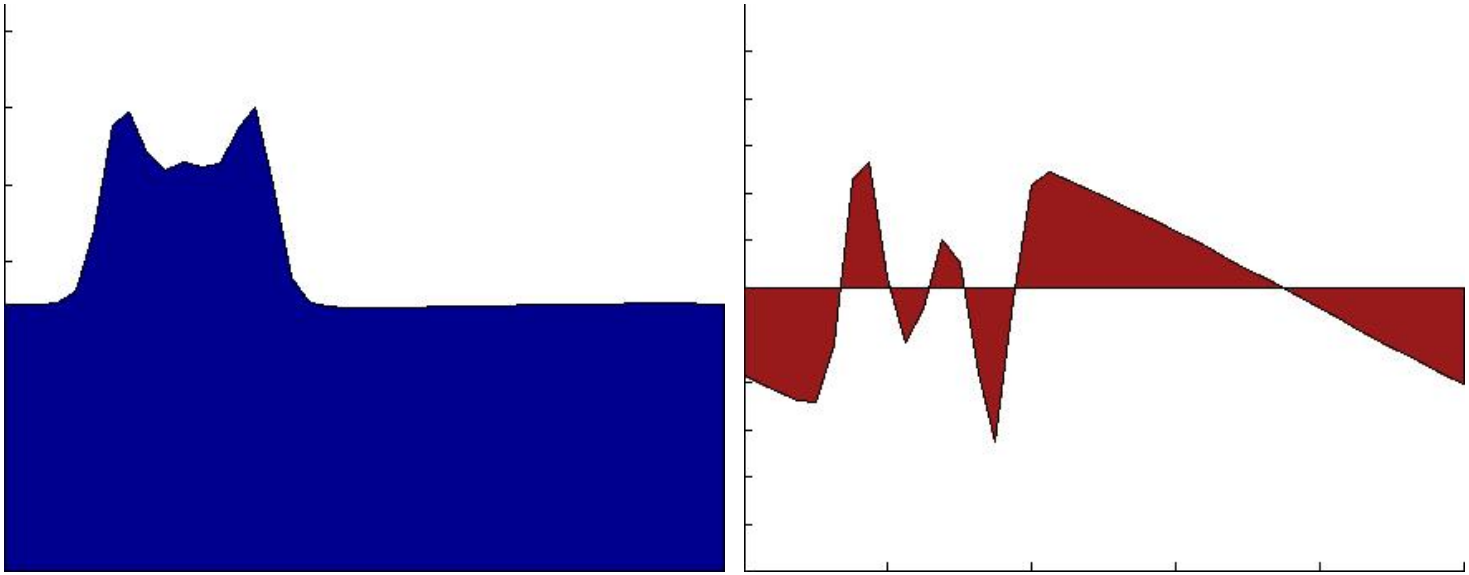
Another program, **shallow_water_1d_animation** can make a series of JPEG files that can be turned into an animation.

---

http://people.sc.fsu.edu/~jburkardt/m_src/shallow_water_1d/shallow_water_1d_display.m

http://people.sc.fsu.edu/~jburkardt/m_src/shallow_water_1d/shallow_water_1d_animation.m

1D Program: Graphical Display



The height is the blue "Batman" shape on the left. The mass velocity is shown in red on the right.

2D Program: Remarks

The 2D program does not have commandline input. However, at the beginning of the program there are some variables whose values can be modified, in particular: **n**, **g**, **dt**, **dx**, **dy**.

This is a 2D program, so saving all the data can be expensive. Instead, the graphical display is made simultaneously as the time steps are taken.

The height, X momentum and Y momentum are stored in **n+2** by **n+2** arrays **H**, **U** and **V** (but remember that **U** and **V** are "really" **HU** and **HV**, that is, mass velocities, not velocities!)

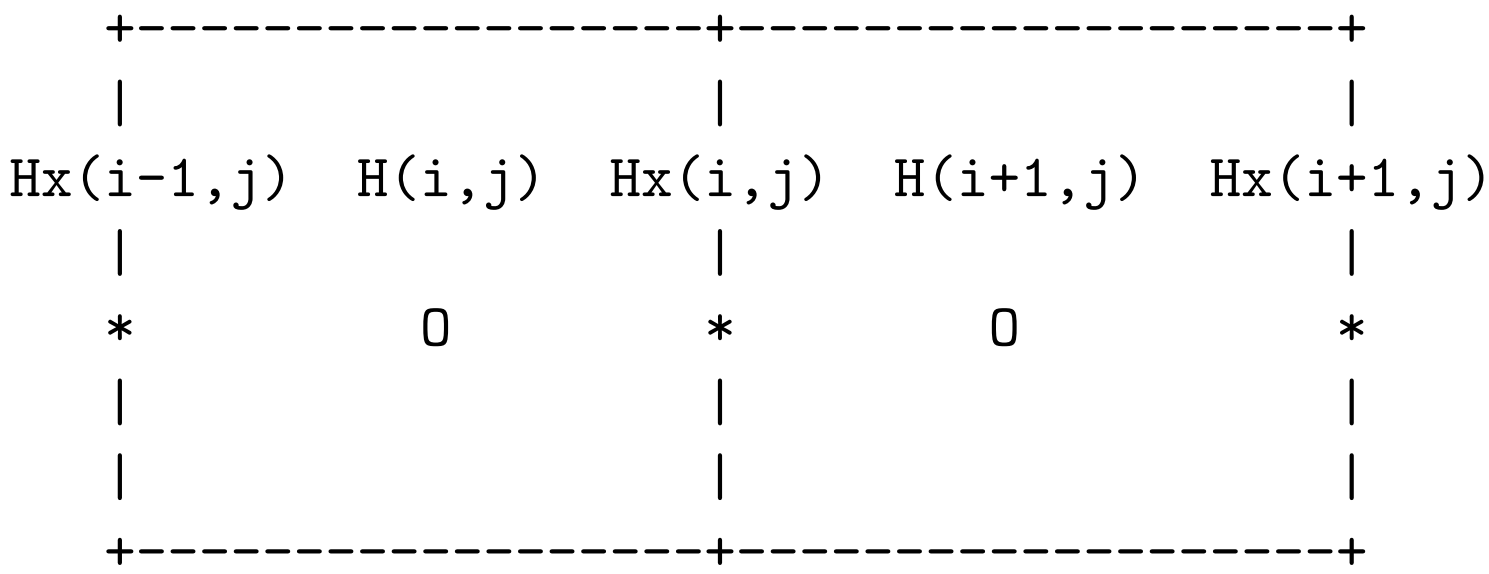
The boundary data is stored in the first and last rows and columns.

2D Program: How are Fluxes Stored?

When the half step time integration is carried out, the midside flux data is stored as **Hx**, **Hy**, **Ux**, **Uy**, **Vx** and **Vy**.

The x direction increases with array index i, y with array index j.

There are **n**+2 nodes in each row and column, so there are **n**+1 midside nodes. Thus the flux arrays are evaluated *between* the regular arrays.



2D Program: The Half Step Flux Terms

In the 2D code, our discretized PDE should look something like this:

```
U^{new} = U - dt * ( ( F(UR) - F(UL) ) / dx )
                   + ( G(UU) - G(UD) ) / dy ) )
```

whether we are approximating the half step or the full step.

But when we compute half-step midside fluxes on the left and right, we use

```
U^{new} = U - dt * ( F(UR) - F(UL) ) / dx )
```

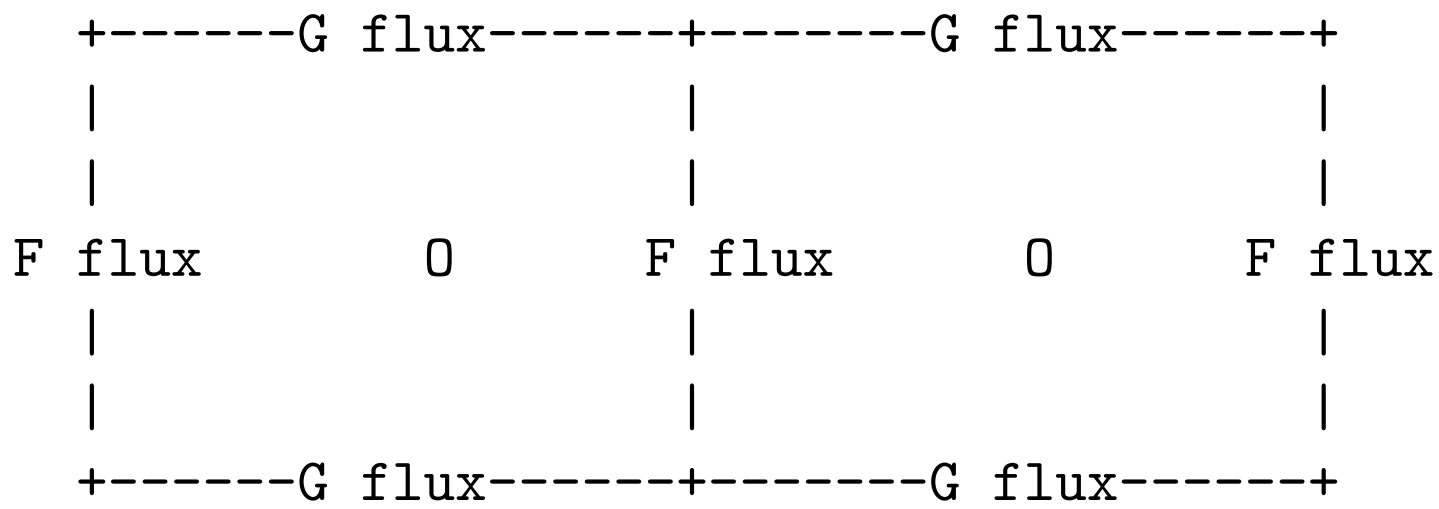
and for the midside fluxes "up" and "down", we use

```
U^{new} = U - dt * ( G(UU) - G(UD) ) / dy )
```

Why could we drop one term?

2D Program: The Half Step Flux Terms

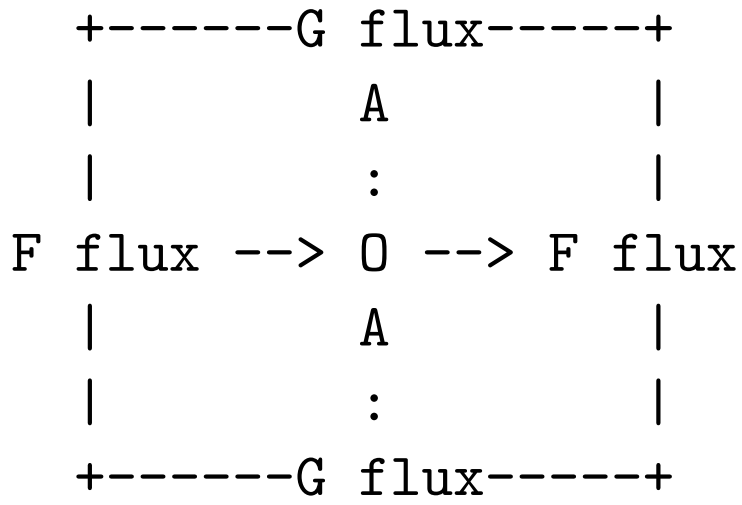
The reasoning here is that the fluxes only occur at cell boundaries and are normal to them. Therefore, when we estimate the flux at on the left or right side of a region, there is no G flux:



and similarly, when we estimate fluxes on the "up" or "down" sides of a region, the F flux vanishes.

2D Program: The Full Step Flux Terms

But when it comes time to generate the full step, we are estimating the flux at the center node, and in that case, we consider both F and G terms:



In the 2D code, compare the right hand sides for **Hx** (horizontal midside, half step), **Hy** (vertical midside, half step) and **H** (central node, full step).

2D Program: Boundary Conditions

The 2D says it uses "reflective" boundary conditions. It actually uses free boundary conditions for H and reflective for UH and VH.

In 2D, the implementation of reflective boundary conditions is a little more complicated than in 1D.

Look carefully at how UH is handled. On the left and right boundaries, the boundary value is the negative of the neighbor. But on the top and bottom, it is set *equal* to the neighbor.

The reflection or reversal of the horizontal velocity only happens at the horizontal boundaries. At the vertical boundaries, we essentially use a free condition as well.

Similar remarks hold for the vertical velocity.

2D Program: MATLAB Remarks - Indexing

You may be used to using loops to set a variable:

```
for i = 1 : nx - 1
  flux(i) = ( value(i+1) - value(i) ) / 2;
end
```

In MATLAB, we can write the equivalent vector statement:

```
flux(1:nx-1) = ( value(2:nx) - value(1:nx-1) ) / 2;
```

It is also possible to assign a name to a frequently used set of array indices:

```
left = 2:nx;
right = 1:nx-1;
flux(right) = ( value(left) - value(right) ) / 2;
```

2D Program: MATLAB Remarks - Dot Operations

When you replace a loop and a scalar assignment by a vector assignment, MATLAB will generally apply the operations to each element.

But certain operators are ambiguous, including *, /, and ^. If A, B and C are vectors or arrays, MATLAB interprets the statement

```
A = B * C;
```

as a request for a dot product, matrix-vector multiplication, or matrix-matrix multiplication. If, instead, you simply want an element by element multiplication, you must signal this by writing

```
A = B .* C;
```

Similarly, use the ./ and .^ operators if you want the non-vector operators. The dot operator is used extensively in the 2D program!

Conservation:

We expect the following quantities to be conserved:

- **mass**, H;
- **momentum** or **mass velocity**, HU and HV;
- **energy**, $\frac{1}{2}(HU^2 + HV^2 + gH^2)$;
- **potential vorticity**;

Of course, conservation laws don't mean things can't change in time, just that time changes correspond to fluxes.

Conservation:

Suppose we start a problem with zero mass velocity (but perhaps with a perturbed value of H). As the fluid adjusts itself, nonzero mass velocities will arise, but we'd expect that they still sum to zero.

However, consider the animation **uh_reflective.mov** from the 1D code. The plots clearly suggest that, after a short time, mass velocity is no longer summing up to zero.

- Has conservation been violated?
- What other explanation is there?
- Is there a way of running the simulation so that mass velocity is constant over time?

---

http://people.sc.fsu.edu/~jburkardt/m_src/shallow_water_1d/uh_reflective.mov

CFL Condition:

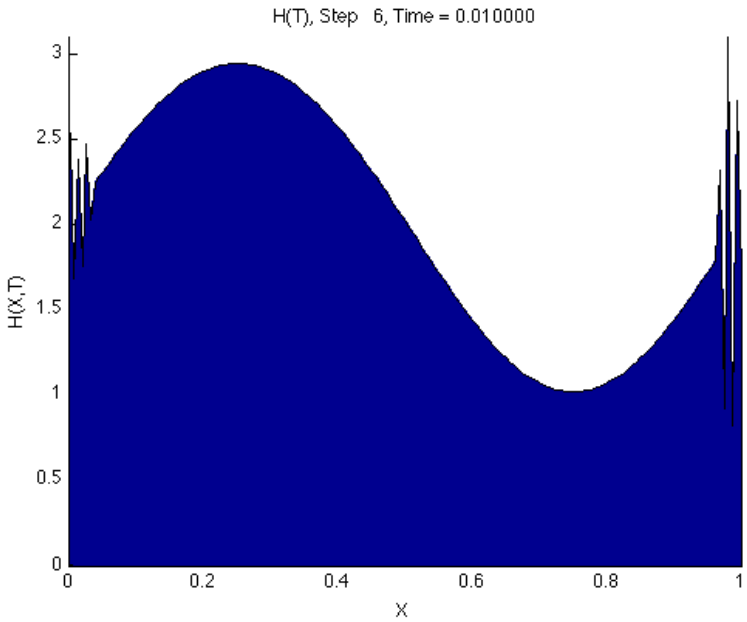
When we discretize the shallow water equations, we must specify step sizes in both space **dx** and time **dt**.

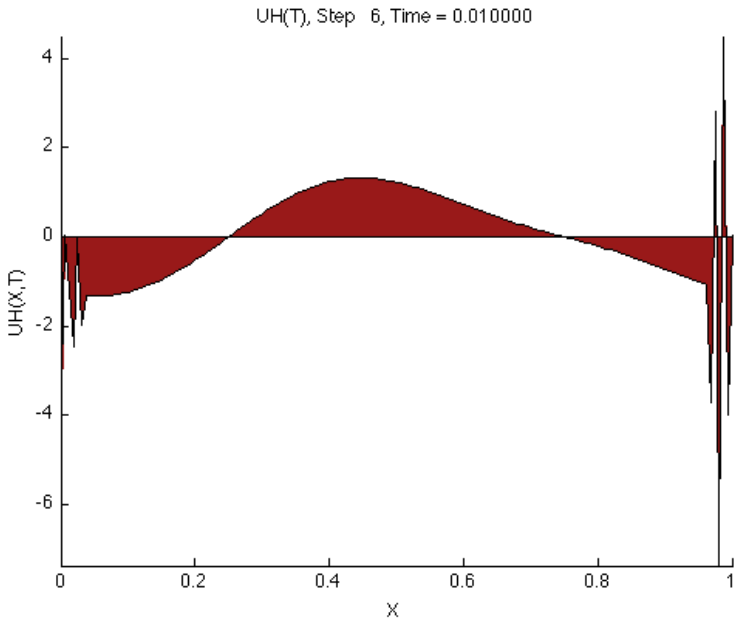
We know that choosing these values too large will give us inaccurate results. We assume, though, that making either quantity smaller is guaranteed to improve the accuracy.

For an explicit code, we don't have a linear system to worry about, so the work is easy to compute. Halving **dt** doubles our timestep work; halving **dx** doubles or quadruples our spatial work, depending on whether we are in 1D or 2D.

CFL Condition:

Here's what happens if I use 4 times as many nodes as in the suggested input, after just 6 time steps:





The jagged shapes at the ends are the beginning of a fatal and violent loss of convergence!

CFL Condition: Time Step Must be Small Enough

It turns out that the issue that arises here is that we have violated the *Courant-Friedrichs-Lewy condition* or "CFL".

Essentially, this rule says that, in an explicit time scheme for a hyperbolic PDE, the timestep must be small enough that no particle has time to cross an entire cell, or else the computation is likely to diverge.

For 2D shallow water equations, this takes the form of

$$\mathbf{dt} \leq \frac{\mathbf{dx}}{|u| + |v| + \sqrt{gh}}$$

Here $\sqrt{gh}$ measures motion due to gravity waves. The time step has to be small enough that this inequality holds at each node.

CFL Condition: Reduced DX May Require Reduced DT

From the initial conditions, we can sometimes make a guess as to a time step that will satisfy the CFL, but as the solution proceeds, we may encounter larger velocities and heights that would require us to stop, or to reduce our time step before proceeding.

And for a given problem, this means that if we reduce **dx**, we may have to reduce **dt** as well, if we are near the CFL limit.

So, in other words, you can always reduce **dt** to get more accuracy in time, but if you reduce **dx**, you may have to reduce your timestep **dt** as well.

Conclusion

You should now have a reasonable background mathematically, algorithmically, and computationally, for understanding the shallow water equations as implemented in Cleve Moler's code.

You can see that when a single line of MATLAB appears in the code, it reflects many decisions and choices.

If you are interested in numerical computations, you should study this program and try to get a feeling for what is going on in every computational line!

Moler's 2D code, or the simpler 1D version, may also be a starting point for your final project.

Conclusion

The 1D version of Moler's code is available at
http://people.sc.fsu.edu/~jburkardt/...
m_src/shallow_water_1d/shallow_water_1d.html

- **shallow_water_1d.m**, computes solutions;
- **shallow_water_1d_display.m**, displays the solution
- **shallow_water_1d_animation.m**, makes animations
- **h_periodic.mov**, movie of H with periodic BC
- **uh_periodic.mov**, movie of UH with periodic BC
- **h_reflective.mov**, movie of H with reflective BC
- **uh_reflective.mov**, movie of UH with reflective BC

Conclusion: Suggested Projects

- add a variable bottom profile to the 1D simulation and make an animation of a wave going over a shallow rise;
- investigate the conservation of mass, momentum, energy, potential vorticity; calculate these quantities for the 1D or 2D simulation;
- compute and monitor the CFL condition on each time step;
- investigate more accurate time and space schemes;
- investigate implicit time schemes;
- add the Coriolis force to the 2D simulation;
- remove the random drops from the 2D simulation, and investigate "interesting" initial conditions;
- if the 2D code uses data with no y dependence, do the 1D and 2D codes get the same results?