What makes the ocean wave?

Summer Seminar
ISC5939
....
John Burkardt
Department of Scientific Computing
Florida State University
https://people.sc.fsu.edu/~jburkardt/presentations/...
...shallow_water_2012_fsu.pdf

17/19 July 2012

Many physical systems end up being described by variations of just a few typical families of mathematical equations. We might a term or a parameter or a boundary condition, of course. But sometimes it's not obvious why the physical behavior is modeled by those equations, or what those equations tell us about the possible physical behaviors.

Fluids transmit signals. A splash in the pond creates a wave that has a shape and a velocity that is recognizable for a great distance.

The same hyperbolic equations that describe fluids also govern sound, electricity, radio waves, and light. Our cell phones use the wave equation; we could never use the heat equation to send a directional signal!

For the classic wave equation

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

it's not hard to see that a traveling wave $u(x, t) = f(x - ct)$, for <u>any</u> function $f()$, is a solution. The fact that some mediums, such as air and water, can support waves, in which the medium does not move (very much) but a signal or disturbance does move, is the fundamental property of waves.

However, the mathematician's wave equation misses other important features, including that wave motion in water can depend on the depth of the medium, that waves tend to disperse, and that waves of different frequency may travel at different speeds.

Introduction: The Navier Stokes Equations

A more detailed model of the behavior of water can be found in the (incompressible) Navier-Stokes equations:

$$\rho(\frac{\partial v}{\partial t} + v \cdot \nabla v) = -\nabla p + \mu \nabla^2 v \quad \text{F=ma}$$

$$\frac{\partial \rho}{\partial t} + \nabla(\rho v) = 0 \qquad \text{mass conservation}$$

Here ρ and μ are the fluid density and viscosity, while v is the velocity vector and p is the fluid pressure.

However, accurate solutions of this equation require fine spatial meshes, tiny time steps.

Using the Navier-Stokes equations to model waves that cross the ocean would be a hopeless task.

Introduction: The Shallow Water Equations

However, the behavior of water in an ocean is a very special case. Compared to the horizontal distances involved (thousands of kilometers), the typical vertical distance is negligible (4 km). Thus, on a global scale, the ocean is more "shallow" than a bathtub with a millimeter of water.

When we are studying waves, the water itself doesn't move much, except for a back-and-forth motion. Relative to the speed of the wave signal, we can treat the ocean as essentially standing still.

What "moves" is a pressure disturbance, that is indicated by temporary changes in the water height - storing and transmitting energy. This is the signal, or wave, that is rapidly transmitted.

Using these insights, and some other approximations, it is possible to derive the Shallow Water Equations, which are a good, simplified model for large scale wave behavior.

Here is one form of the shallow water equations:

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

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

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

At time t and position (x, y), the fluid has a height h, a pressure p, and a (horizontal) velocity vector (u, v). The fluid is incompressible, with a density of ρ_0, so h is essentially a mass measurement.

You can see that, even though the shallow water equations are supposed to be a simplified model, it's not at all likely that we can spot exact mathematical solutions.

However, these equations can be used as a guide for construction computational programs to approximate the solution over a simplified, discretized geometry. Such a program makes it easy to investigate the effects of various parameters, and to consider the kinds of solutions that arise.

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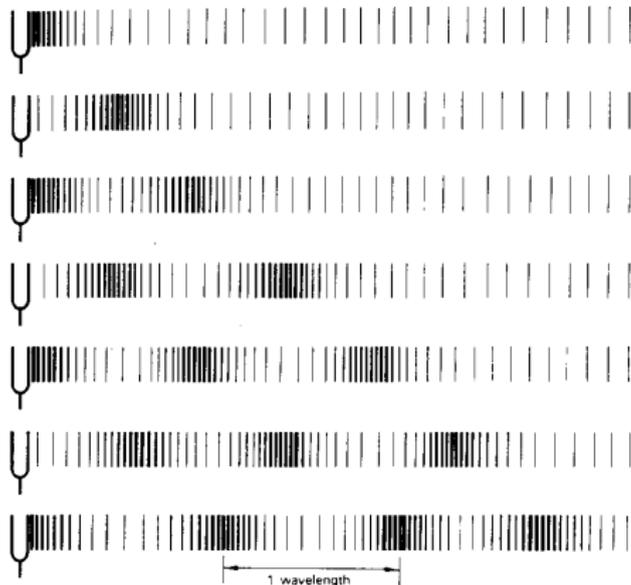
Physics: Longitudinal Waves in a Slinky
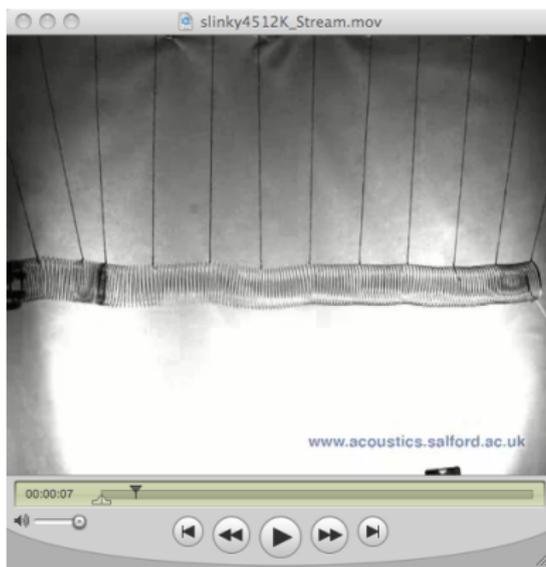
A "Slinky" is a coiled metal wire which acts like a gentle spring. Pushing it on the left, the coils compress, storing energy, and then separate, and the pressure disturbance moves to the right.

http://people.sc.fsu.edu/~jburkardt/latex/sem_2012_shallow/slinky.mov

1 wavelength

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

Physics: The Air is Just a Medium for the Wave

A sound wave is not so much the motion of air, as it is the transmission of air pressure. A packet of air at higher pressure squeezes its neighbor, which compresses, and then passes on the pressure to the next neighbor, and so on. The air is essentially transporting the pressure as though it were a signal.

Waves occur in water as well. A wave can start in San Francisco and travel across the ocean to Shanghai. But the water in San Francisco doesn't actually travel to Shanghai. Instead, it's the disturbance or energy or some kind of signal that travels.

Unlike air, water is not easily compressible. That means that a wave signal doesn't move by compressing the water. Instead, the energy is stored by increasing the height of the water, which involves very small left-to-right motions of the water itself.

Physics: Longitudinal Waves in Water

wave phase : t / T = 0.000

Especially when the wave has a long way to travel over a relatively shallow basin, the energy can be transmitted in this way, by having the water store and release energy by local height variations.

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/sem_2012_shallow/shallow_water_wave.mov

Moler lets a giant droplet of water fall onto a tranquil ocean.
A coarse 64x64 grid is used, unsuitable for Navier Stokes calculation.
Even the shallow water calculation eventually breaks down here.
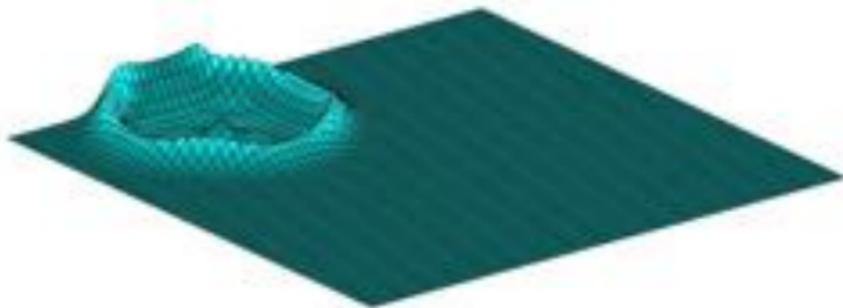
http://people.sc.fsu.edu/~jburkardt/m_src/shallow_water_2d/shallow_water_2d.m

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

What terms do you neglect, or regroup, or rewrite?

How do you model:

- the region: approximating shape;
- the geometry: replacing areas by grid points;
- a function $u(x, y, t)$ now defined on grid points;
- spatial derivatives u_x, u_y only defined on grid points;
- conditions at the boundary;
- time discretization, and time steps;
- time derivatives u_t;
- initial conditions.

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 $f * h * u$, important for long range waves on the surface of the earth.

He assumes that the ocean bottom is of constant depth (otherwise, the varying depth significantly affects the equations!)

He assumes the pressure simply reflects the mass of the water column, so 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 p is eliminated.

Recall our statement of the shallow water equations:

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

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

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

At time t and position (x, y), the fluid has a height h, a pressure p, and a (horizontal) velocity vector (u, v). The fluid is incompressible, with a density of ρ_0, so h is essentially a mass measurement.

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

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

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

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

Unless we add some source terms, the only things that will now make this system do anything are boundary and initial conditions.

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

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

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

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

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

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

This form is known as a *hyperbolic conservation law*. It identifies a quantity which can never simply disappear. It can only change by fluxes that enter or exit 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:

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

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

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

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.

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!

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.

Our geometry can be defined by:

- M and N, the number of rows and columns of points;
- DX and DY, the constant X and Y spacings;

State variables are computed at grid points, and stored in MxN arrays, such as $U(I, J)$.

Although the flow is 3D, we are really using a 2D geometry, so the quantity $H(I, J)$, the water height at each grid point, is not really a geometric quantity, but another state variable, to be computed and updated.

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;

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**: $U(1) = U(N+1)$;

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.

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.

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 $U(R)$.

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

$$\frac{\partial U_{ij}}{\partial t} \approx \frac{U_{ij}^{k+1} - U_{ij}^k}{\Delta T}$$
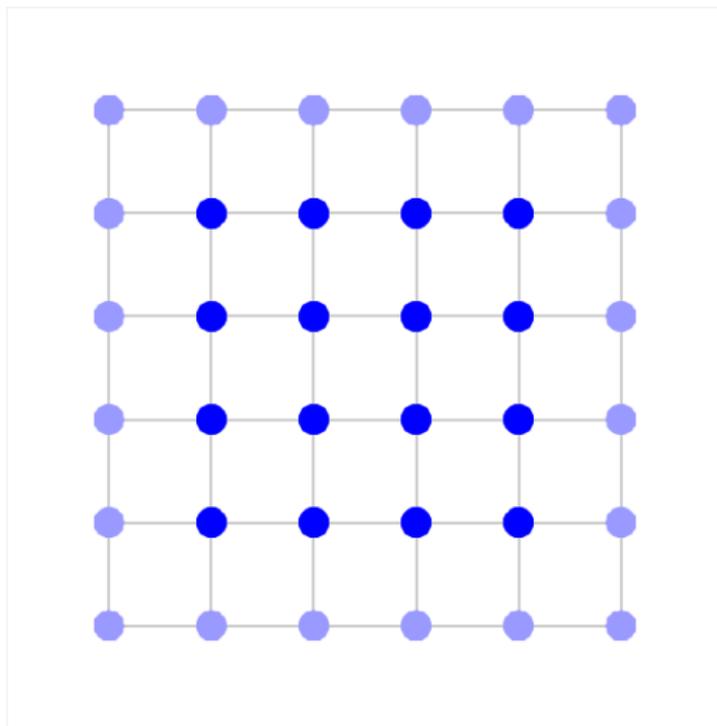
The light blue points will handle boundary conditions.
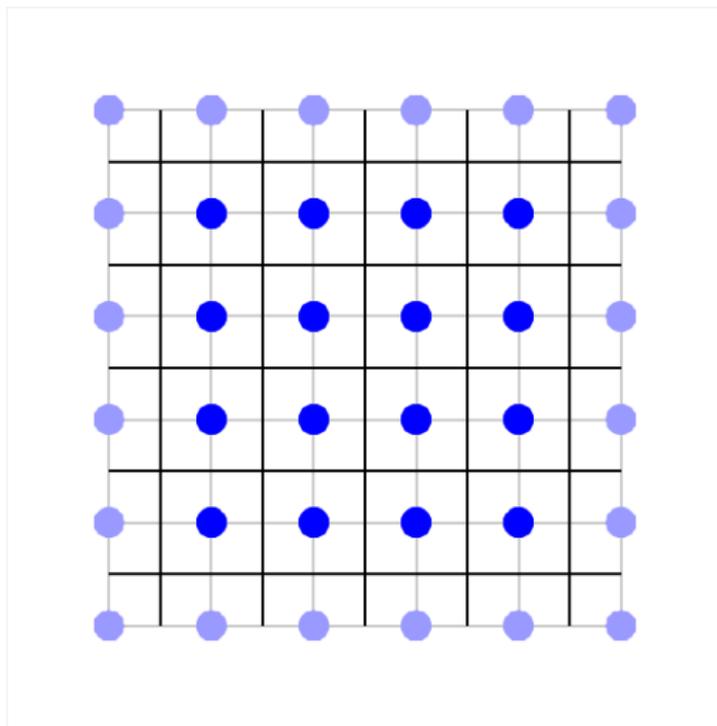
Physical properties are assumed constant in each region.
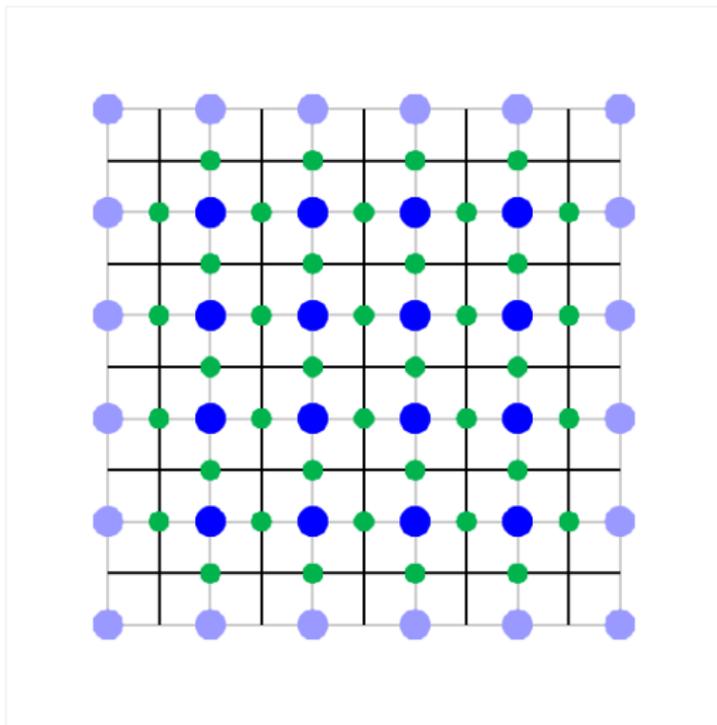
Fluxes will be estimated at the green nodes.

Geometry: Solve One Time Level at a Time

Marching Forward in Time

T = 5

T = 4

T = 3

T = 2

T = 1

T = 0

Initial Conditions
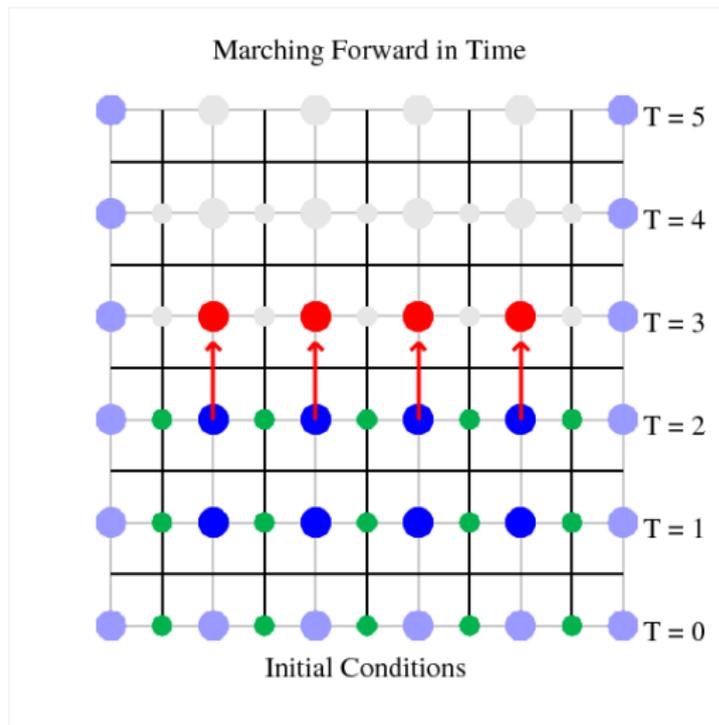
Here, the vertical axis represents time.

In part 2, we write out the discrete equations, look at two ways to approximate the time derivative, and then examine a 1D version of Moler's algorithm.
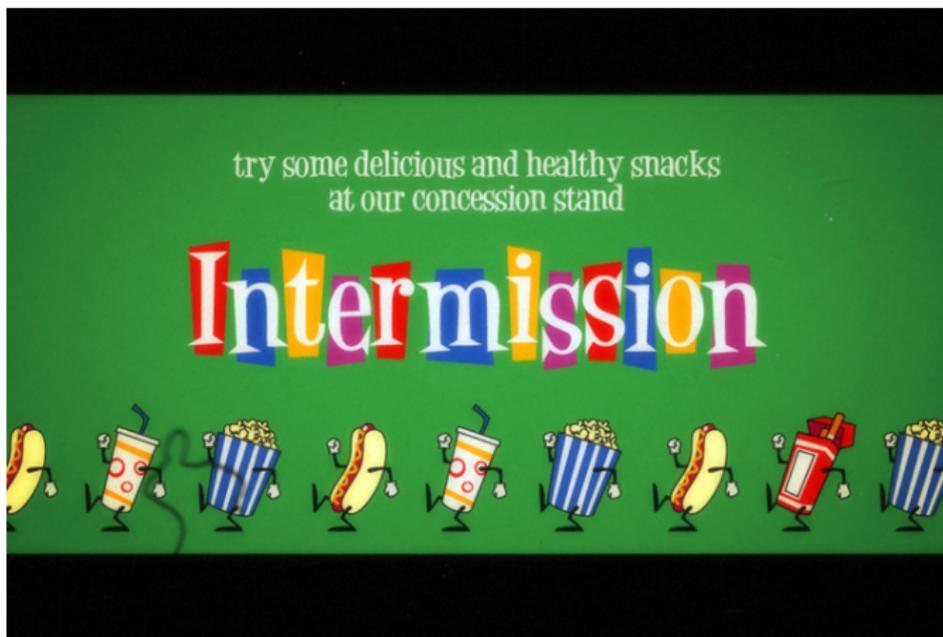
We start with the solution at time level 0. In order to advance to the next time level, we simply need to write a discrete version of the shallow water equations, using the time derivative term to tell us what the values should be at the next time level.

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

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:

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

First order in time stencil
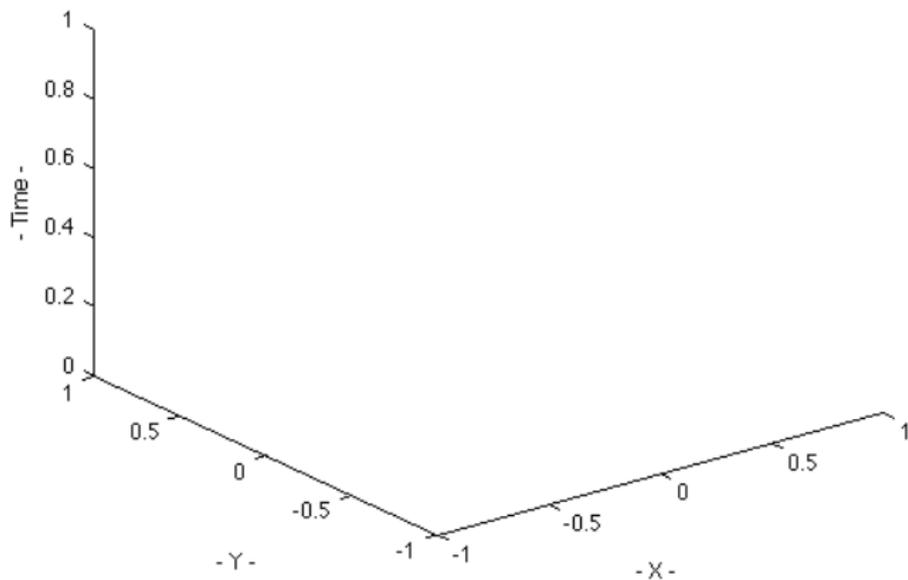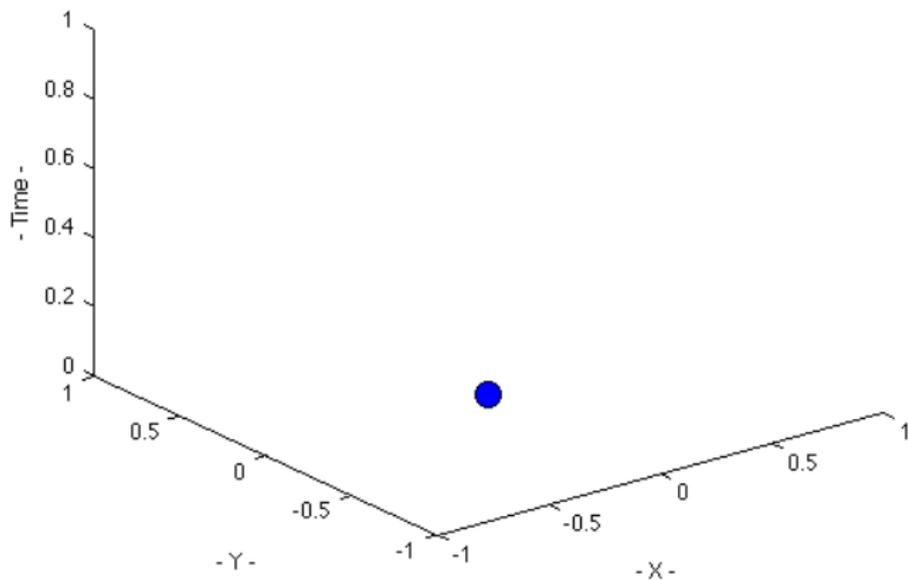
A node where we have the solution.
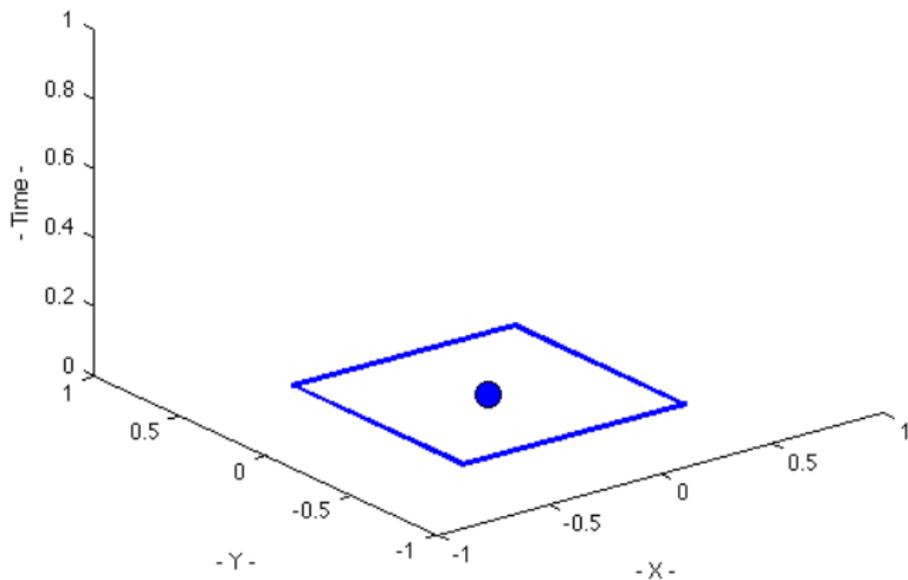
The region associated with the node.

The node and its four neighbors.
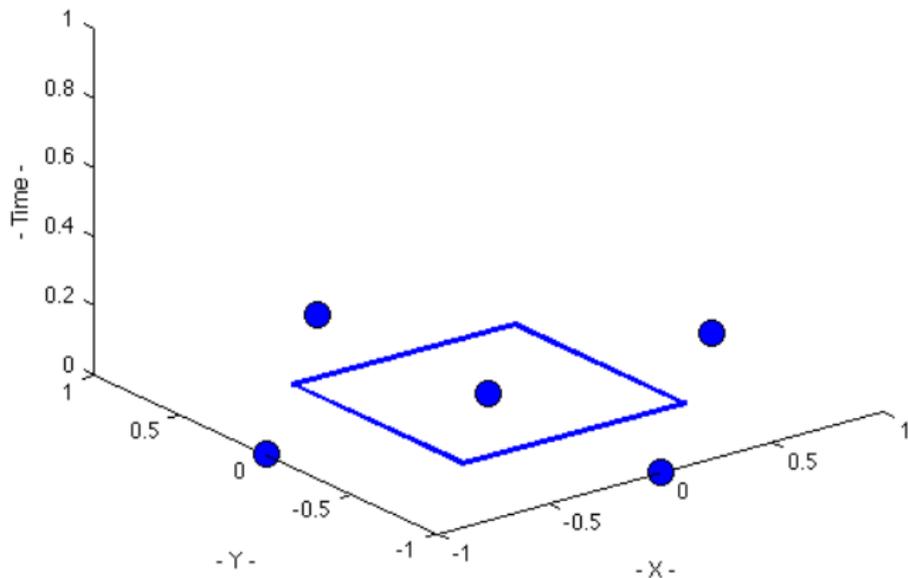
Fluxes are differences across the neighbors.
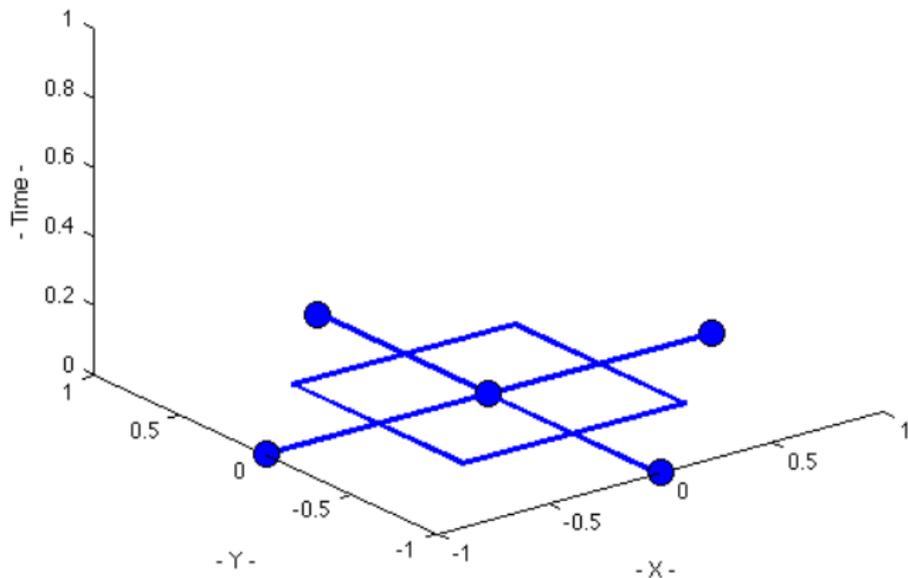
A full time step is now taken.
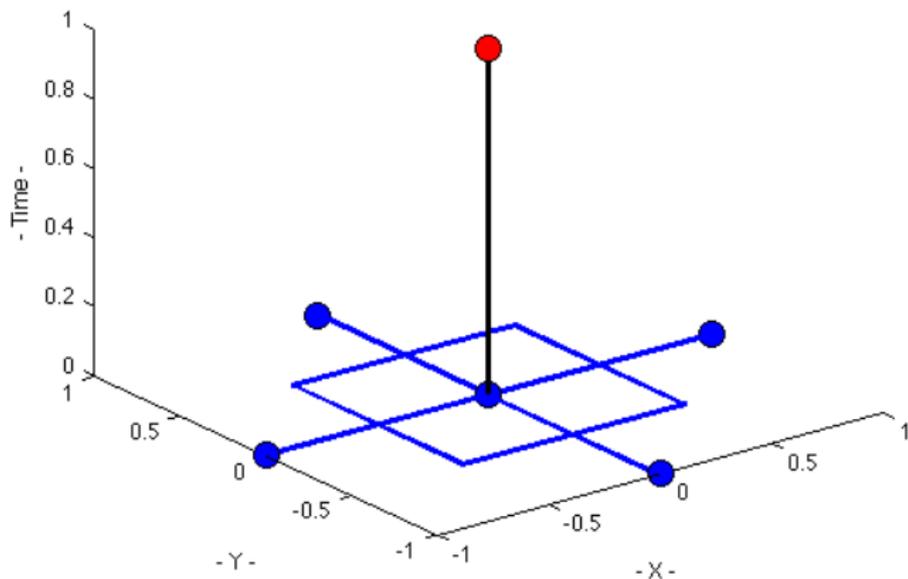
We have an estimate for the solution in this region.
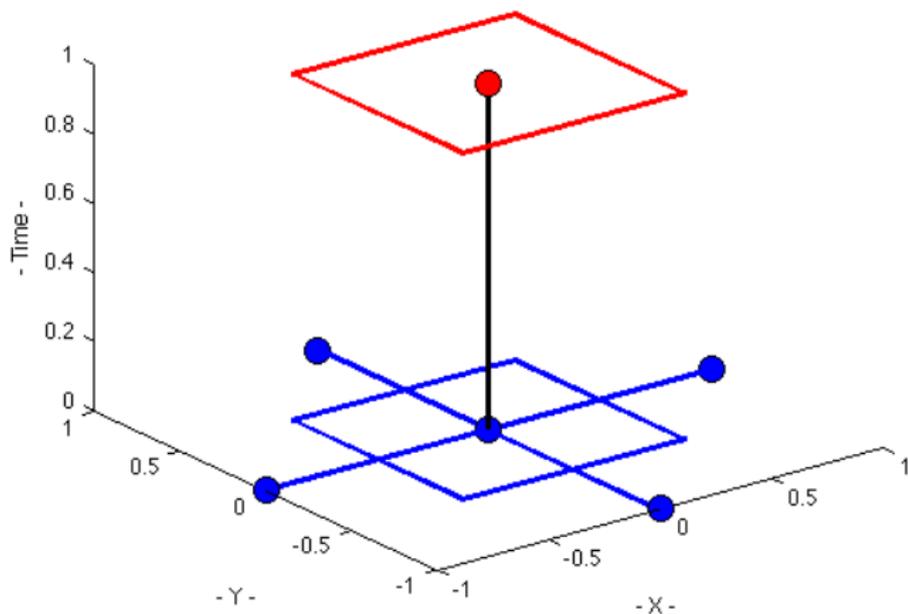
All the nodes can be advanced in this way.
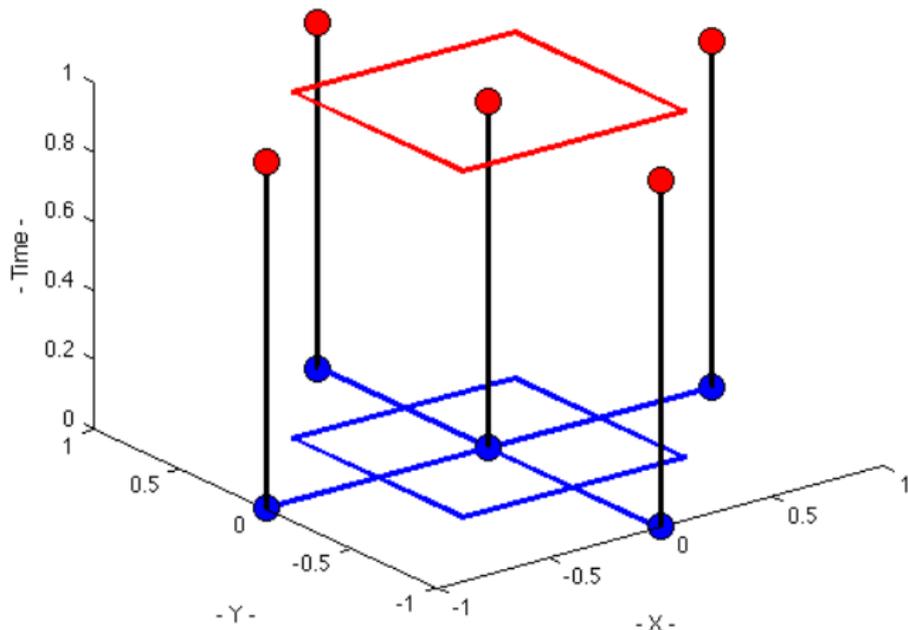
Our approximation to the fluxes is second order accurate, but our time integration is a simple Euler method, which is only first order accurate.

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.

A method that is second order accurate in time and space can take bigger time steps, and use a coarse mesh.

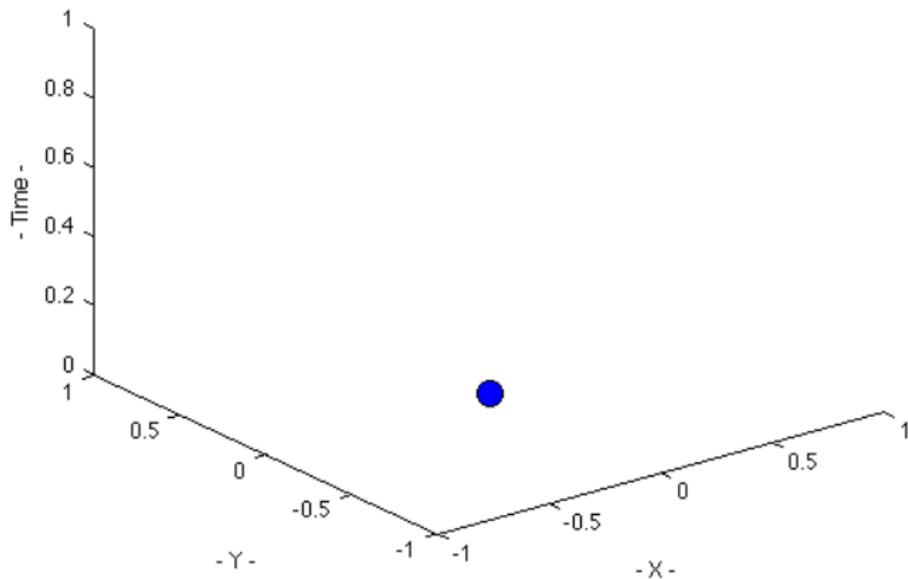This could be the difference between waves in a pond or the ocean.
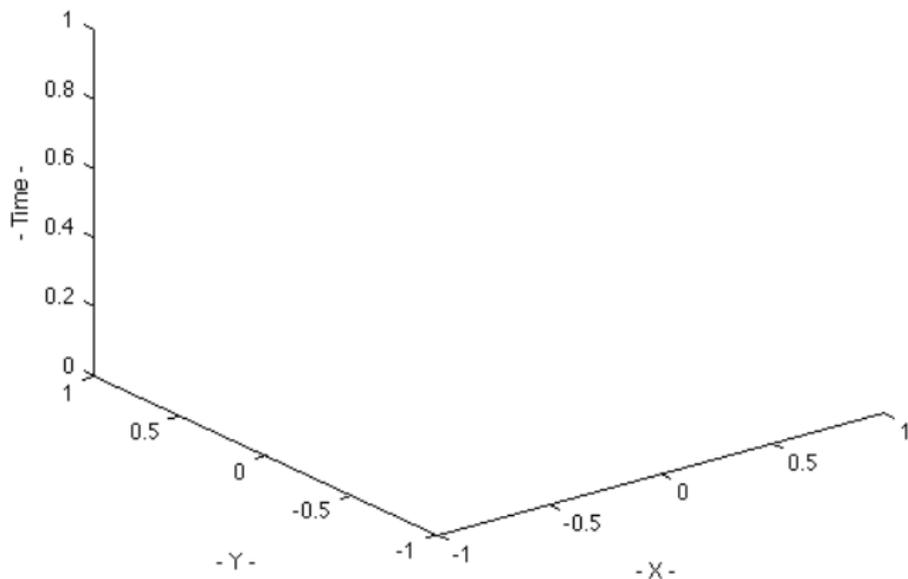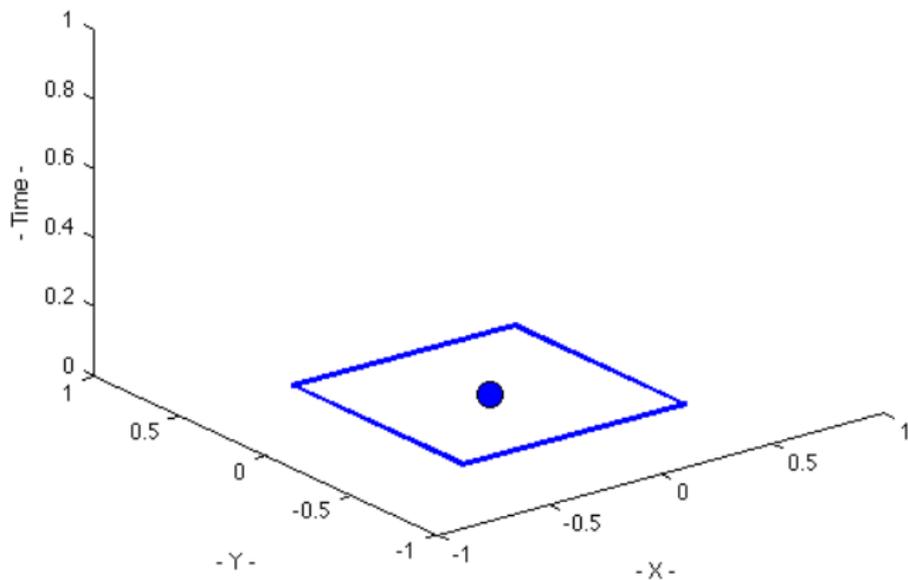
Lax-Wendroff scheme

A node where we have the solution.

The region associated with the node.

The neighboring nodes.

Values at midsides by averaging.

Values at midside, half time step time.

Estimate fluxes

Estimate derivative at node, half time step.

Take full step to new time.

Advance all data to new time.

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

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!

In a 1D world, we have two state functions, h and u, which depend on x and t according to the following 1D shallow water equations:

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

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

If we define the quantities:

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

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

we can write our equations as:

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

which gives us our 1D *hyperbolic conservation law*.

Lax-Wendroff 1D scheme

A node where we have the solution.

The region associated with the node.

The neighboring nodes.

Values at midsides by averaging.

Values at midside, half time step.

Estimate fluxes

Estimate derivative at node, half time step.

Take full step to new time.

Advance all data to new time.

Ready to compute next time step
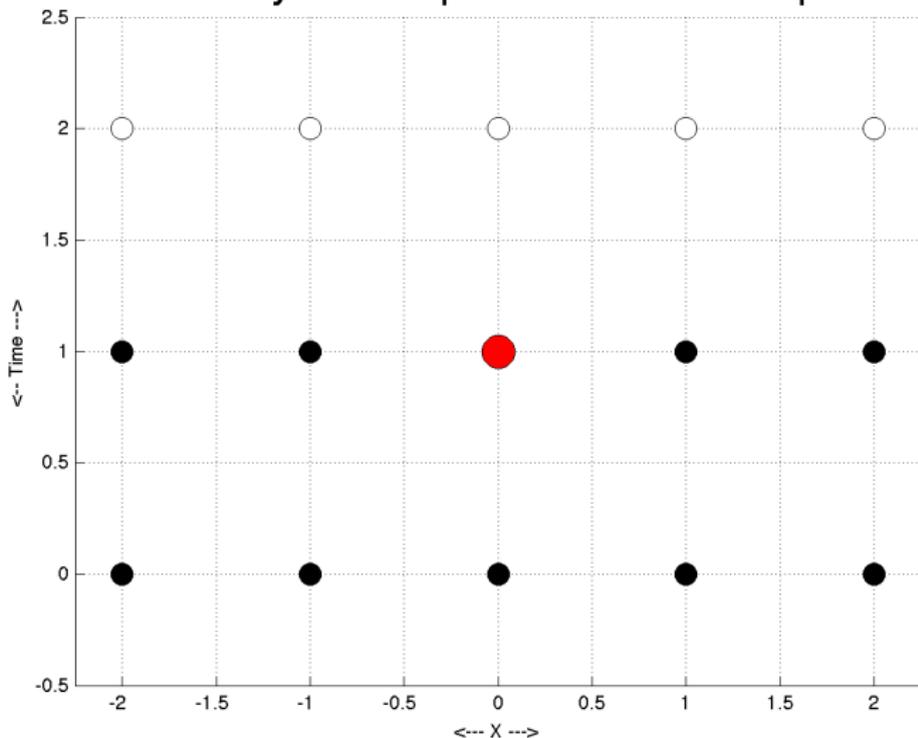
The 1D program is invoked by the command:

[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

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:

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

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;
```

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;
```

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.

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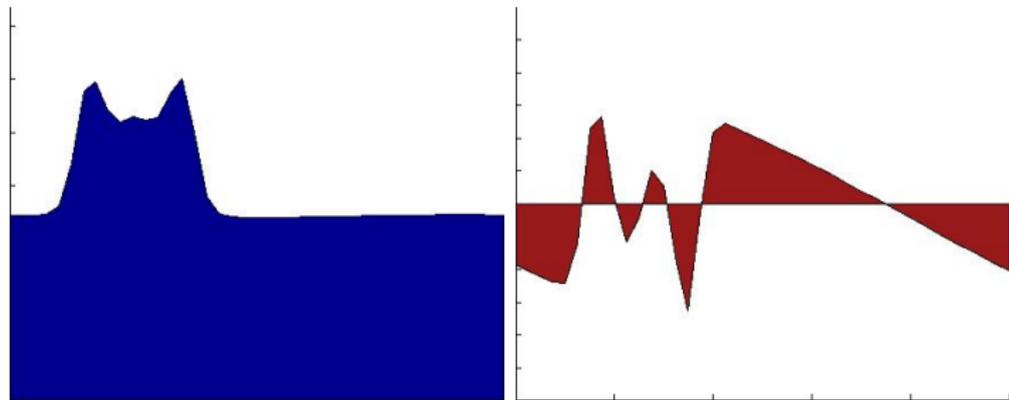
The height is the blue "Batman" shape on the left. The mass velocity is shown in red on the right.

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.

```
   +------------------+-------------------+
   |                  |                   |
 Hx(i-1,j)  H(i,j)  Hx(i,j)  H(i+1,j)  Hx(i+1,j)
   |                  |                   |
   *         O        *         O         *
   |                  |                   |
   |                  |                   |
   +------------------+-------------------+
```
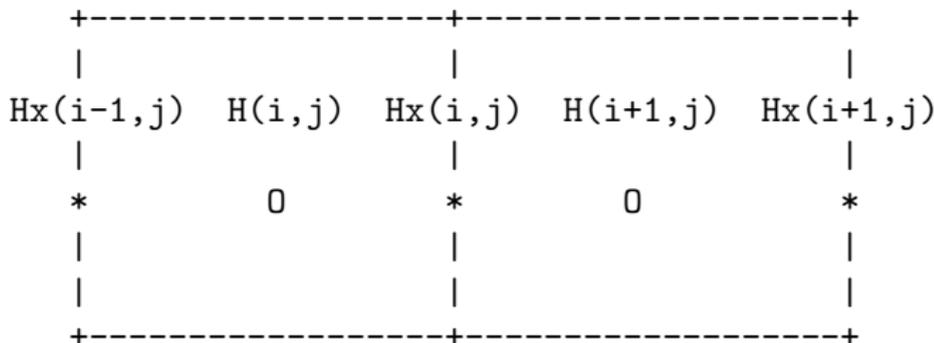
In the 2D code, our discretized PDE should look something like this:

```
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?

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:

```
+------G flux------+-------G flux------+
|                  |                   |
|                  |                   |
F flux       O     F flux       O      F flux
|                  |                   |
|                  |                   |
+------G flux------+-------G flux------+
```

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

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:

```
    +------G flux-----+
    |         A       |
    |         :       |
 F flux --> O --> F flux
    |         A       |
    |         :       |
    +------G flux-----+
```

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.

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;
```

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!

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

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.

Here's what happens if I use 4 times as many nodes as in the suggested input, after just 6 time steps:
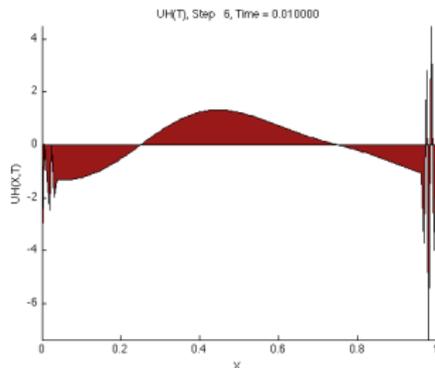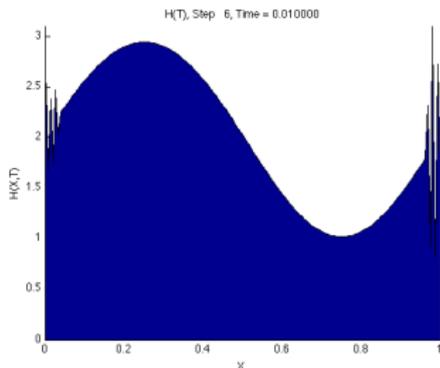
The jagged shapes at the ends are the beginning of a fatal and violent loss of convergence!

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

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

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

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, can be the starting point for your own investigations.

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?
- can you make a parallel version of the program?
- can you make a C or Fortran or Python version of the program?

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

Some references and demonstration software are available:

- en.wikipedia.org/wiki/Shallow_water_equations
- www.amath.washington.edu/∼rjl/research/tsunamis
- www.amath.washington.edu/∼dgeorge/tsunamimodeling.html
- www.amath.washington.edu/∼claw/applications/shallow/www