

Advection / Hyperbolic PDEs

Notes

- In addition to the slides and code examples, my notes on PDEs with the finite-volume method are up online:
 - https://github.com/Open-Astrophysics-Bookshelf/numerical_exercises

Linear Advection Equation

- The linear advection equation provides a simple problem to explore methods for hyperbolic problems

$$a_t + ua_x = 0$$

- Here, u represents the speed at which information propagates
- First order, linear PDE
 - We'll see later that many hyperbolic systems can be written in a form that looks similar to advection, so what we learn here will apply later.

Linear Advection Equation

- We need initial conditions

$$a(x, 0) = a_0(x)$$

- and boundary conditions

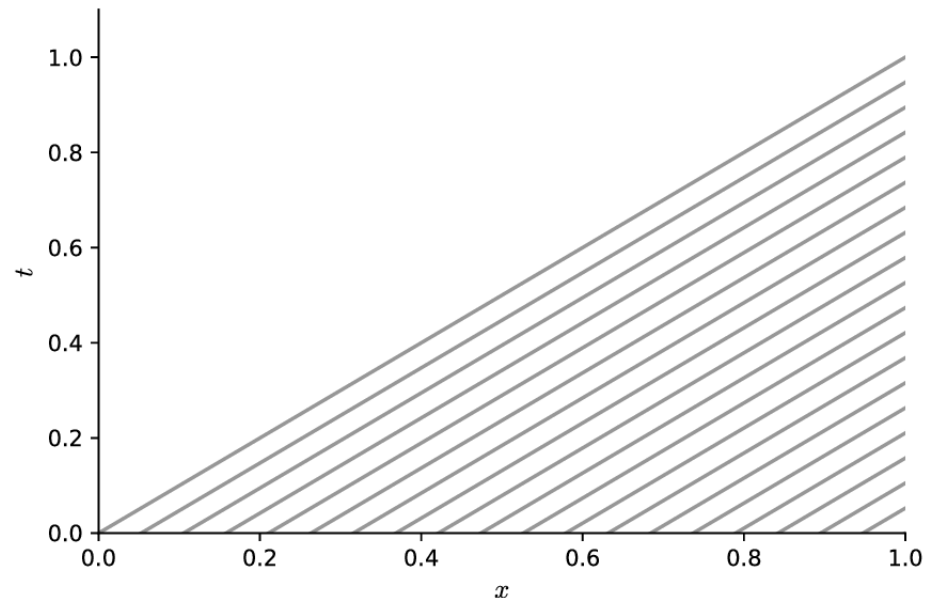
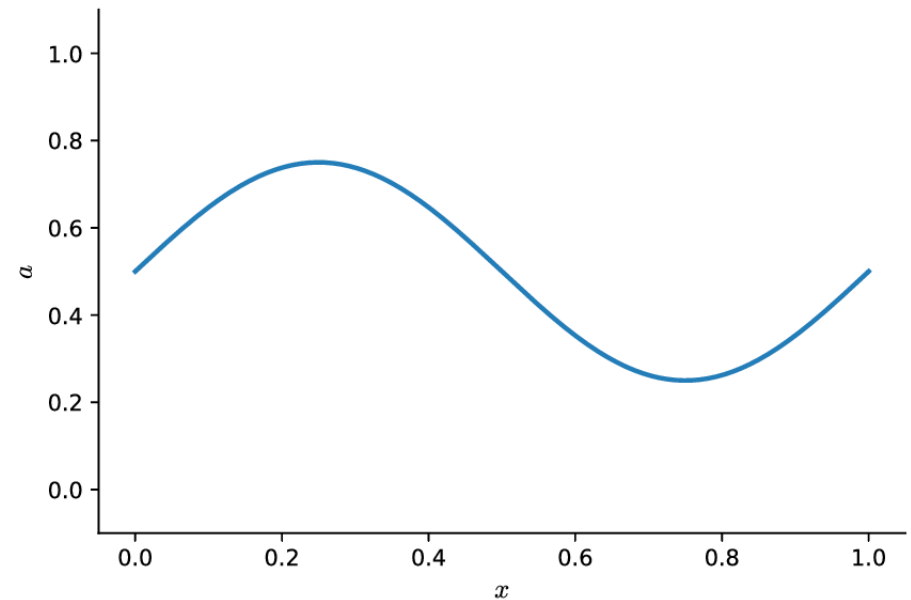
$$a(0, t) = a_l(t)$$

$$a(L, t) = a_r(t)$$

- We'll see in a moment that we only really need 1 boundary condition, since this is a first-order equation

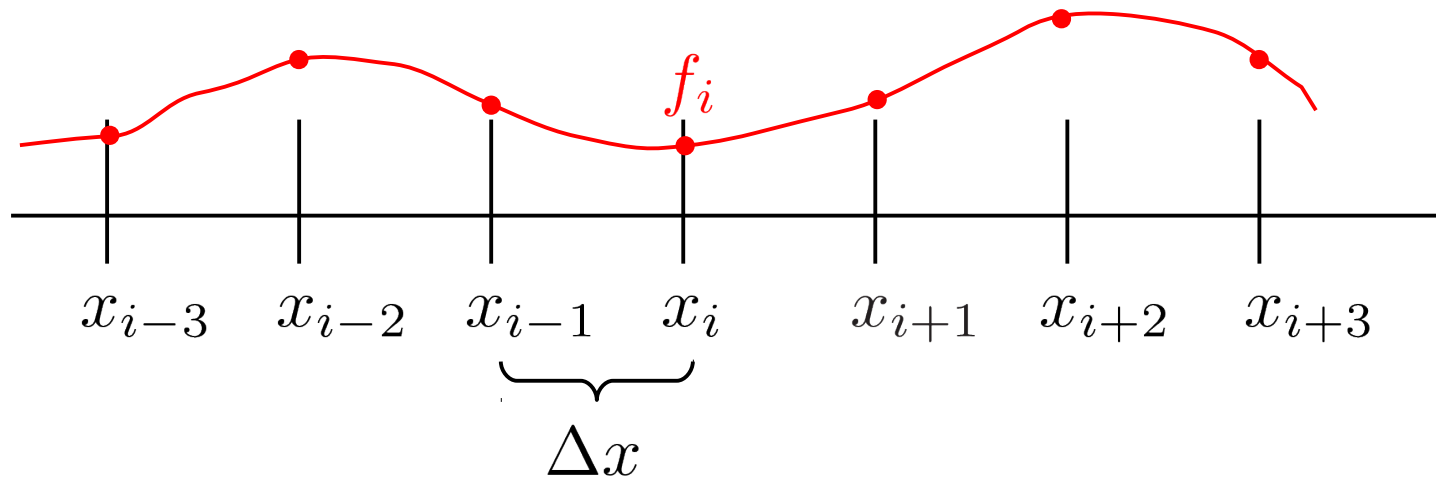
Linear Advection Equation

- Solution is trivial—**any initial configuration simply shifts to the right** (for $u > 0$)
 - e.g. $a(x - ut)$ is a solution
 - This demonstrates that the solution is constant on lines $x = ut$ –these are called the characteristics
- **This makes the advection problem an ideal test case**
 - Evolve in a periodic domain
 - Compare original profile with evolved profile after 1 period
 - Differences are your numerical error



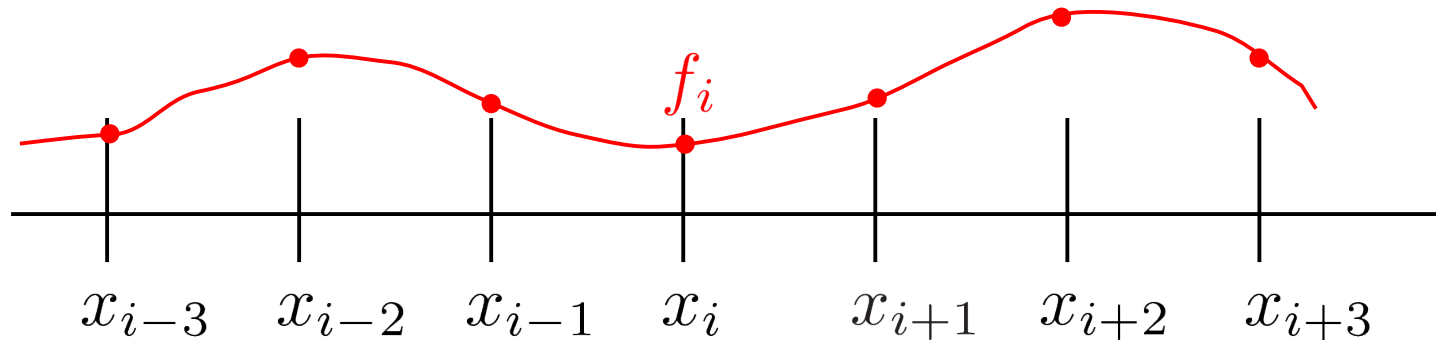
Gridded Data

- Discretized data is represented at a finite number of locations
 - Integer subscripts are used to denote the position (index) on the grid
 - Structured/regular: spacing is constant



- Data is known only at the grid points: $f_i = f(x_i)$

First Derivative / Order of Accuracy



- Taylor expansion:

$$f_{i+1} = f(x_i + \Delta x) = f_i + \left. \frac{df}{dx} \right|_{x_i} \Delta x + \frac{1}{2} \left. \frac{d^2 f}{dx^2} \right|_{x_i} \Delta x^2 + \dots$$

- Solving for the first derivative:

$$\underbrace{\left. \frac{df}{dx} \right|_{x_i}}_{\text{Discrete approximation to } f'} = \underbrace{\frac{f_{i+1} - f_i}{\Delta x}}_{\text{Discrete approximation to } f'} - \underbrace{\frac{1}{2} \left. \frac{d^2 f}{dx^2} \right|_{x_i} \Delta x}_{\text{Leading term in the truncation error}} + \dots$$

First Derivative / Order of Accuracy

- This is a first-order accurate expression for the derivative at point i
 - Alternately, we can use the point to the left (blackboard)
 - These are called difference or finite-difference formulae
- Shorthand: $\mathcal{O}(\Delta x)$
 - “big-O notation”
- How can we get higher order?

First Derivative / Order of Accuracy

- First derivative approximations:

- First-order (one-sided):

$$f' = \frac{f_i - f_{i-1}}{\Delta x} \quad f' = \frac{f_{i+1} - f_i}{\Delta x}$$

2-point stencil

- Second-order (centered):

$$f' = \frac{f_{i+1} - f_{i-1}}{2\Delta x}$$

3-point stencil

- Fourth-order:

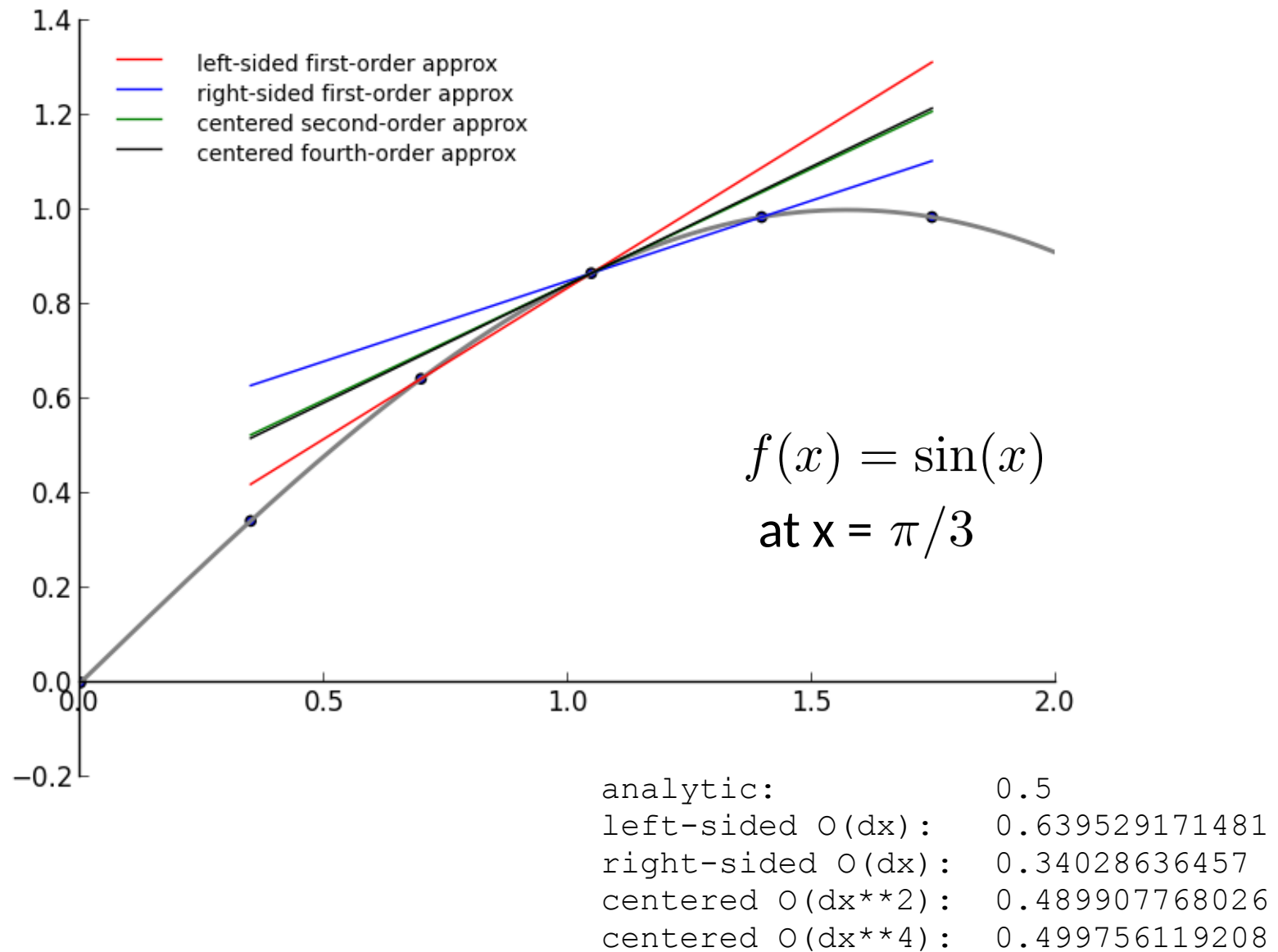
$$f' = \frac{-f_{i+2} + 8f_{i+1} - 8f_{i-1} + f_{i-2}}{12\Delta x}$$

5-point stencil

- Range of points involved is called the stencil

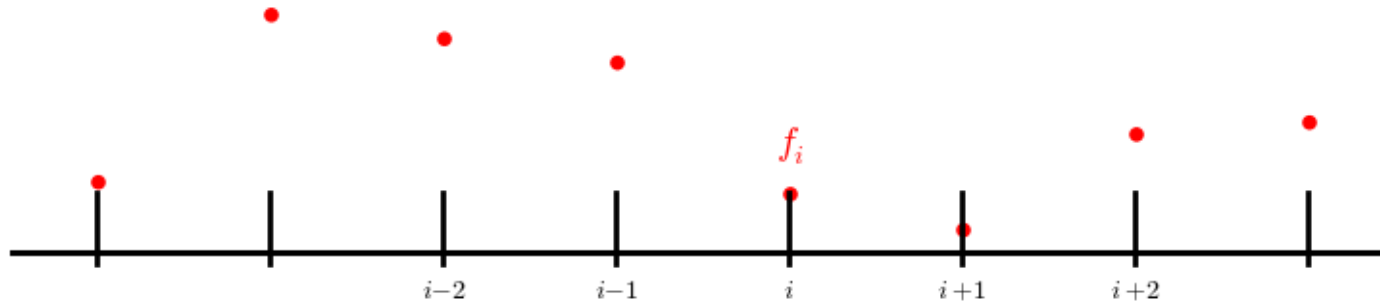
- Some points may have a '0' coefficient

First Derivative Comparison



Finite-Difference Approximation

- We store the function value at each point in our grid:



- We will use the notation: a_i^n
 - Superscripts = time discretization; subscripts = spatial discretization
 - We'll use 0-based indexing
- Simple discretization:

$$\frac{a_i^{n+1} - a_i^n}{\Delta t} = -u \frac{a_{i+1}^n - a_{i-1}^n}{2\Delta x}$$

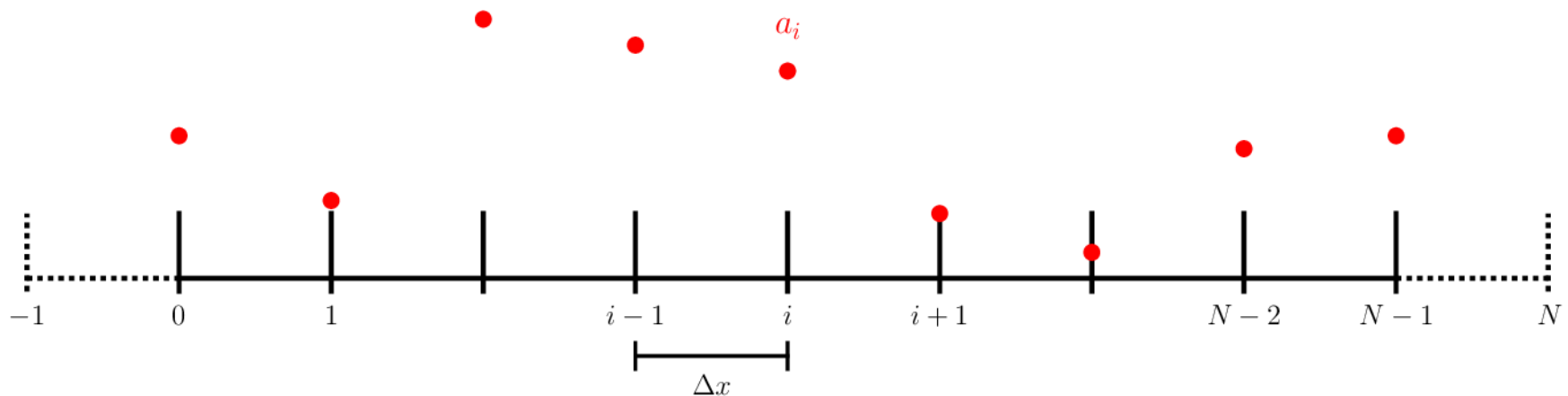
- Explicit
- 2nd order in space, 1st order in time (FTCS method)

Boundary Conditions

- We want to be able to apply the same update equation to all the grid points:

$$a_i^{n+1} = a_i^n - \frac{C}{2}(a_{i+1}^n - a_{i-1}^n)$$

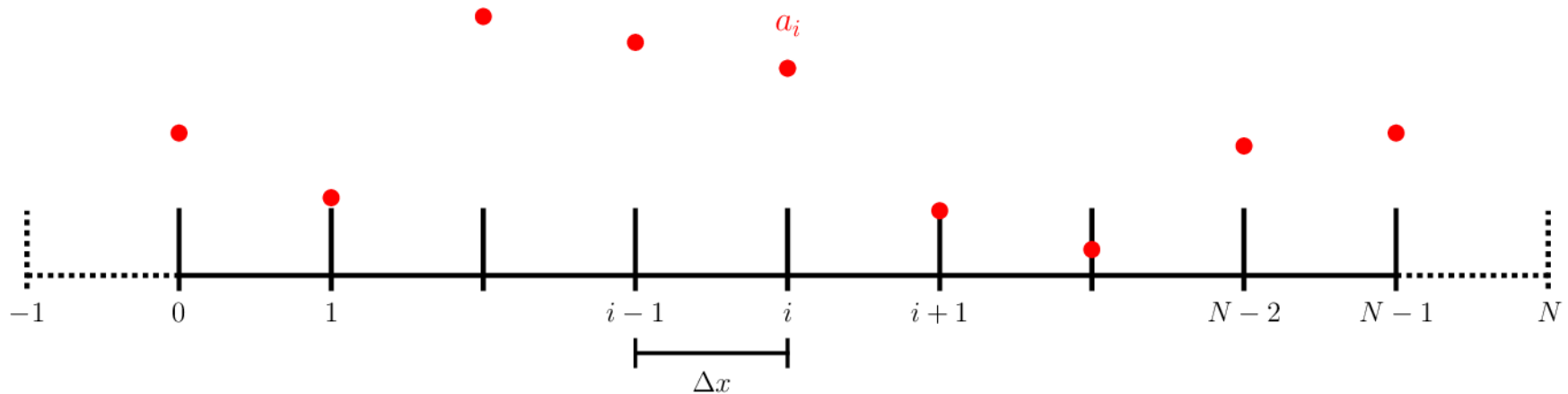
- Here, $C = u\Delta t / \Delta x$ is the fraction of a zone we cross per timestep—this is called the **Courant-Friedrichs-Lewy number** (or **CFL number**)
- Notice that if we attempt to update zone $i = 0$ we “fall off” the grid
- Solution: ghost points (for finite-volume, we'll call them ghost cells)



Grid for N points

Boundary Conditions

- Before each timestep, we fill the ghost points with data that represents the boundary conditions

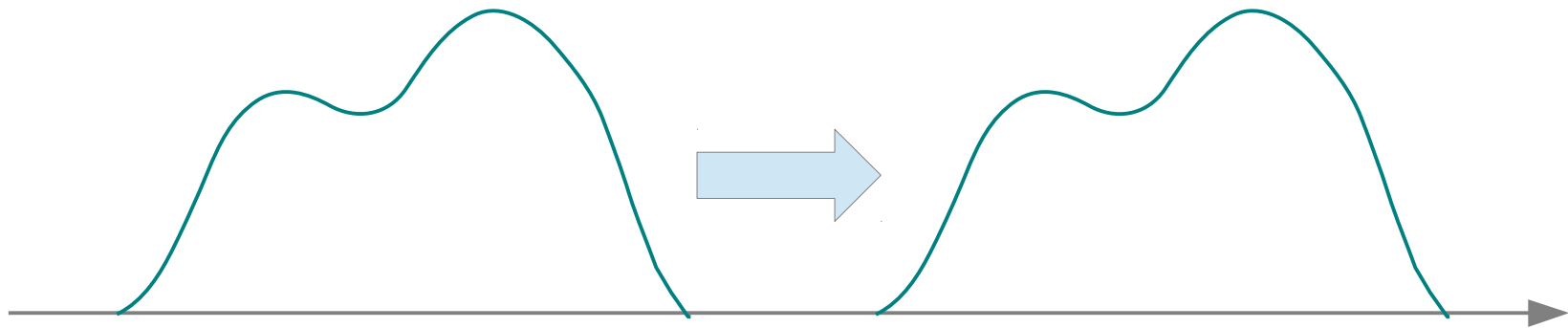


- Note that with this discretization, we have a point exactly on each boundary (we only really need to update one of them)
- Periodic BCs would mean: $a_0^n = a_{N-1}^n$
 - $a_N^n = a_1^n$; $a_{-1}^n = a_{N-2}^n$
- Other common BCs are outflow (zero derivative at boundary)

General grid
of N points

Implementation and Testing

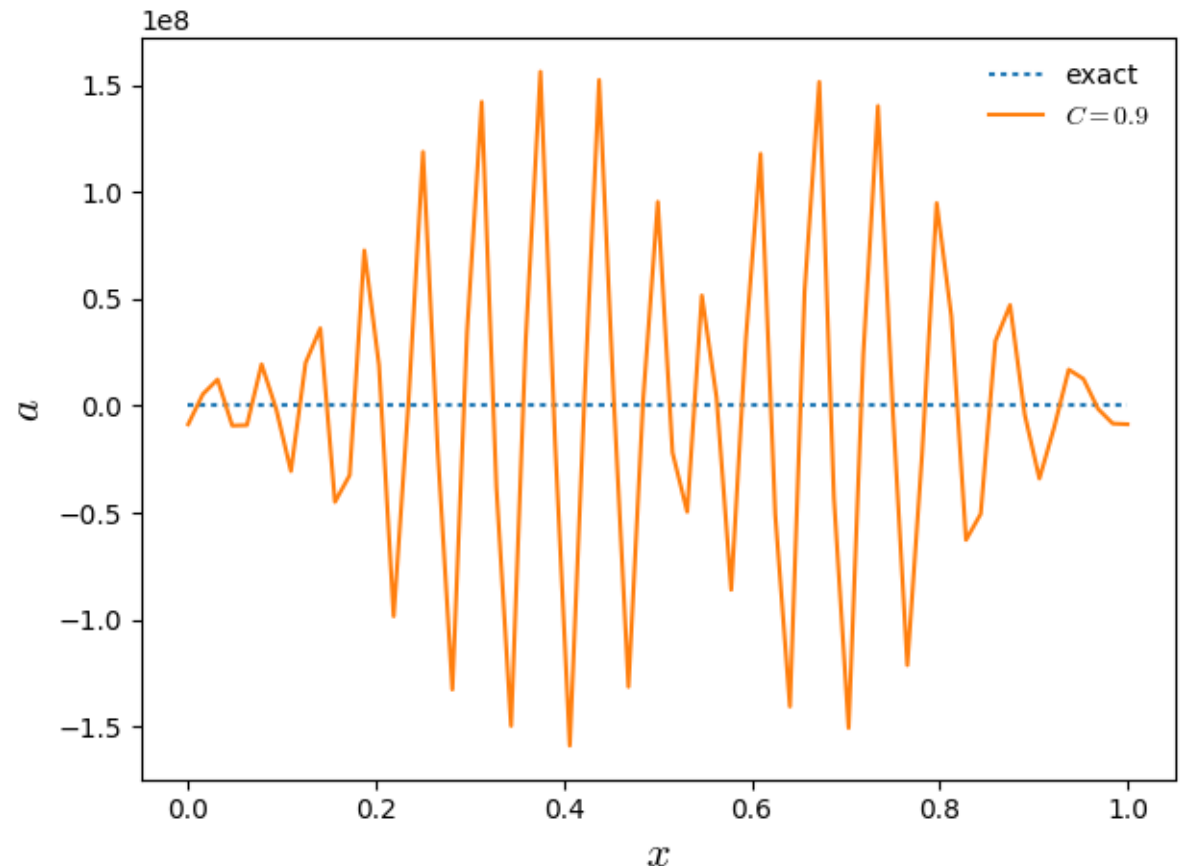
- Recall that the solution is to just propagate any initial shape to the right:



- We'll code this up with periodic BCs and compare after 1 period
 - On a domain $[0,1]$, one period is simply: $1/u$
- We'll use a tophat initial profile
- Let's look at the code...

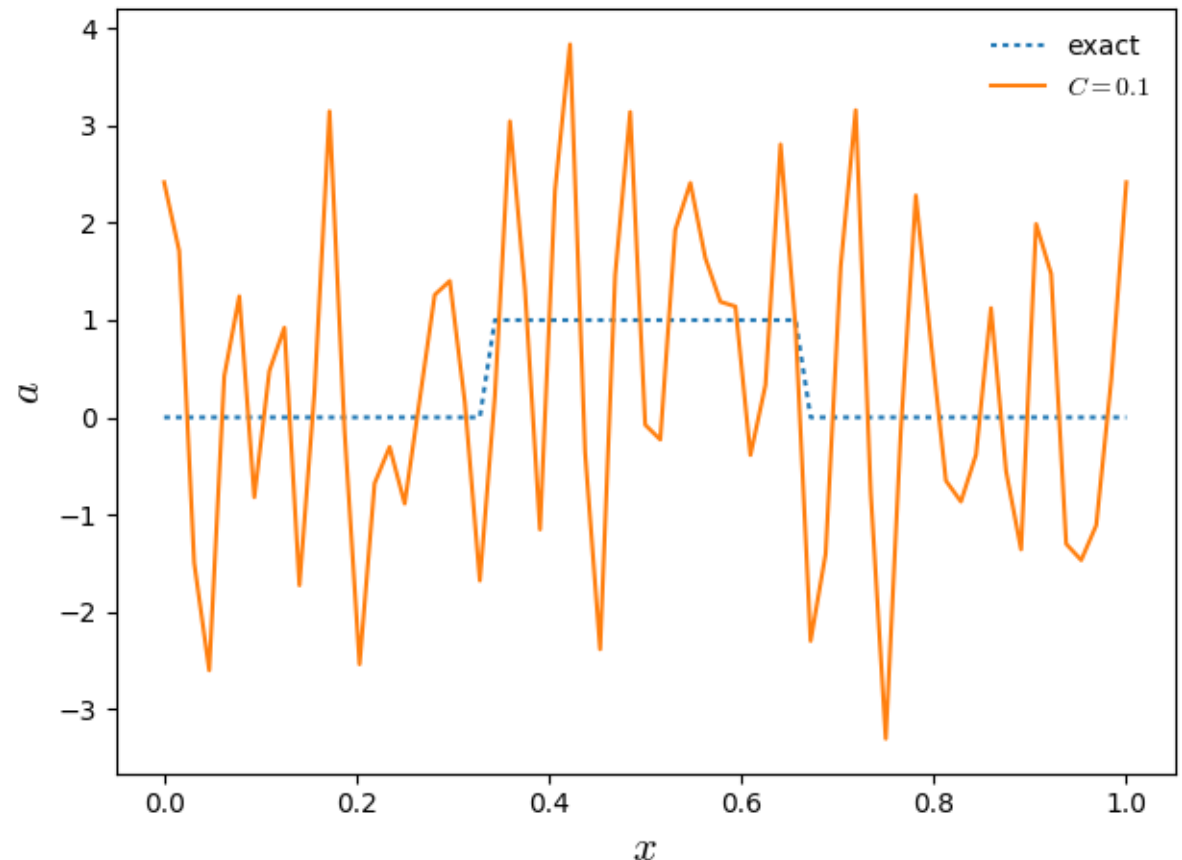
FTCS Linear Advection

- Evolution with $N=65$, $C=0.9$, after 1 period
 - Note the vertical scale!!!!
 - We see nothing that looks like a tophat
 - Any ideas?



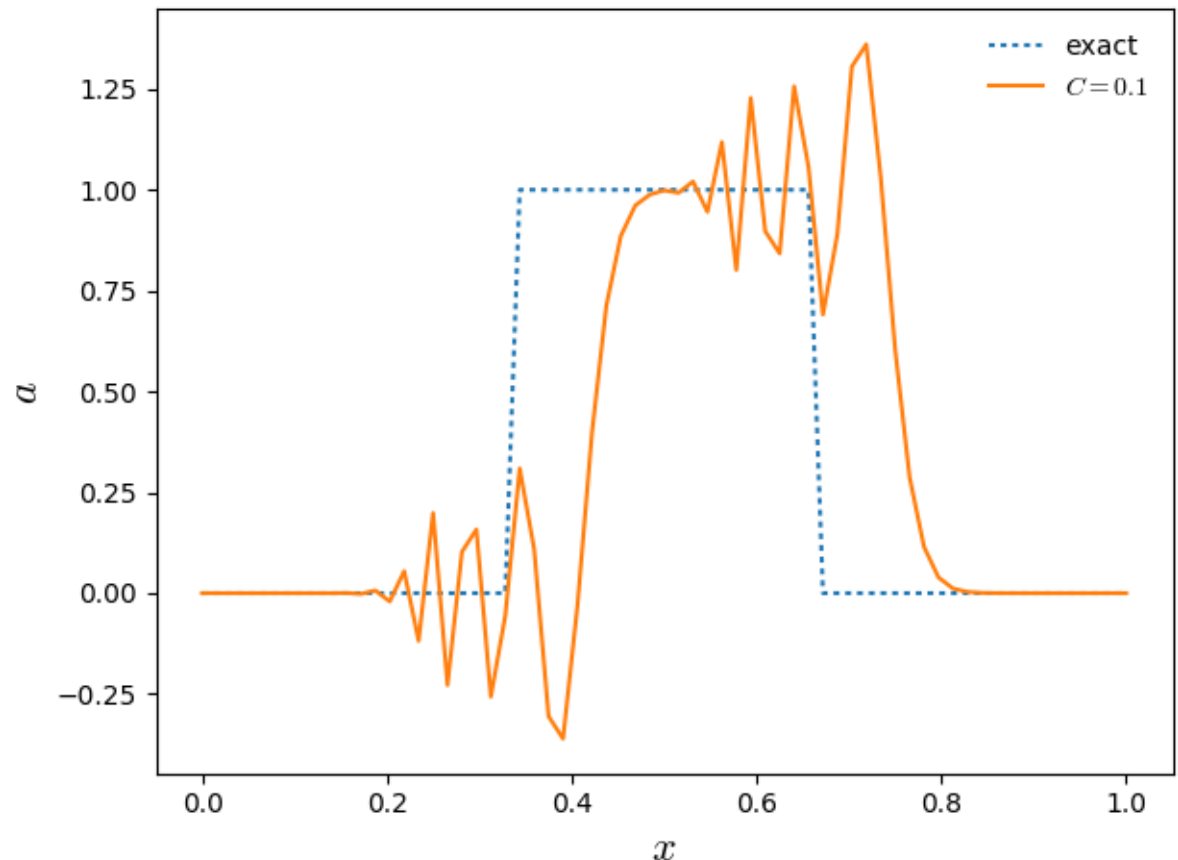
FTCS Linear Advection

- Reducing the timestep is equivalent to reducing the CFL number
 - CFL = 0.1, still 1 full period
 - The scale is much reduced, but it is still horrible
 - Let's look how these features develop



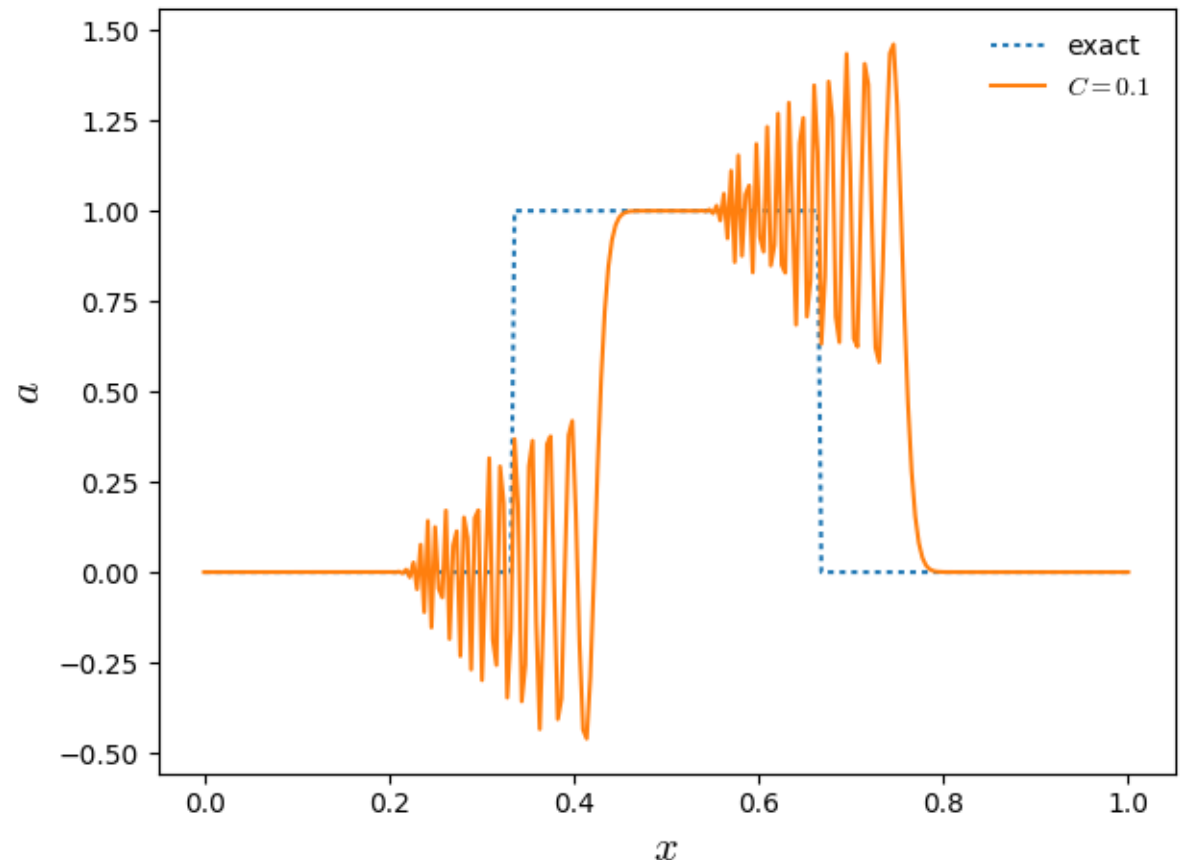
FTCS Linear Advection

- Here we evolve for only 1/10th of a period
 - CFL = 0.1
 - Notice that the oscillations are developing right near the discontinuities
 - So far we've looked at timestep, what about resolution?



FTCS Linear Advection

- This is with $N = 257$ points
 - There is something more fundamental than resolution or CFL number going on here...



code: `fdadvect.py`

Stability

- The problem is that FTCS is not **stable**
- There are a lot of ways we can investigate the stability
 - It is instructive to just work through an update using pencil + paper to see how things grow
 - Growth of a single Fourier mode
 - Truncation analysis
 - Graphically: we can trace back the characteristics to see what information the solution depends on.

Truncation Analysis

- Observe that:

Finite-difference methods solve linear advection equations approximately, but they solve modified linear advection equations exactly

—Laney (p. 265)

- Let's figure out what physical equation this difference approximation better represents
- Substitute in

$$a_i^{n+1} = a_i^n + \dot{a}\Delta t + \frac{1}{2}\ddot{a}\Delta t^2 + \mathcal{O}(\Delta t^3)$$

$$a_{i\pm 1}^n = a_i^n \pm a_x\Delta x + \frac{1}{2}a_{xx}\Delta x^2 + \mathcal{O}(\Delta x^3)$$

- And lot's of algebra...

Truncation Analysis


- We find:

$$\underbrace{a_t + ua_x}_{\text{This is our original equation}} = \underbrace{-\frac{1}{2}\Delta t u^2 a_{xx}}_{\text{This looks like diffusion, but look at the sign!}}$$


- We are more accurately solving an advection/diffusion equation
- But the diffusion is negative!
 - This means that it acts to take smooth features and make them strongly peaked—this is unphysical!
- The presence of a numerical diffusion (or numerical viscosity) is quite common in difference schemes, but it should behave physically!

Upwinding

- Let's go back to the original equation and try a different discretization
- Instead of a second-order (centered-difference) spatial difference, let's try first order.
 - We have two choices:

$$a_x = \frac{a_i^n - a_{i-1}^n}{\Delta x}$$


upwind

$$a_x = \frac{a_{i+1}^n - a_i^n}{\Delta x}$$


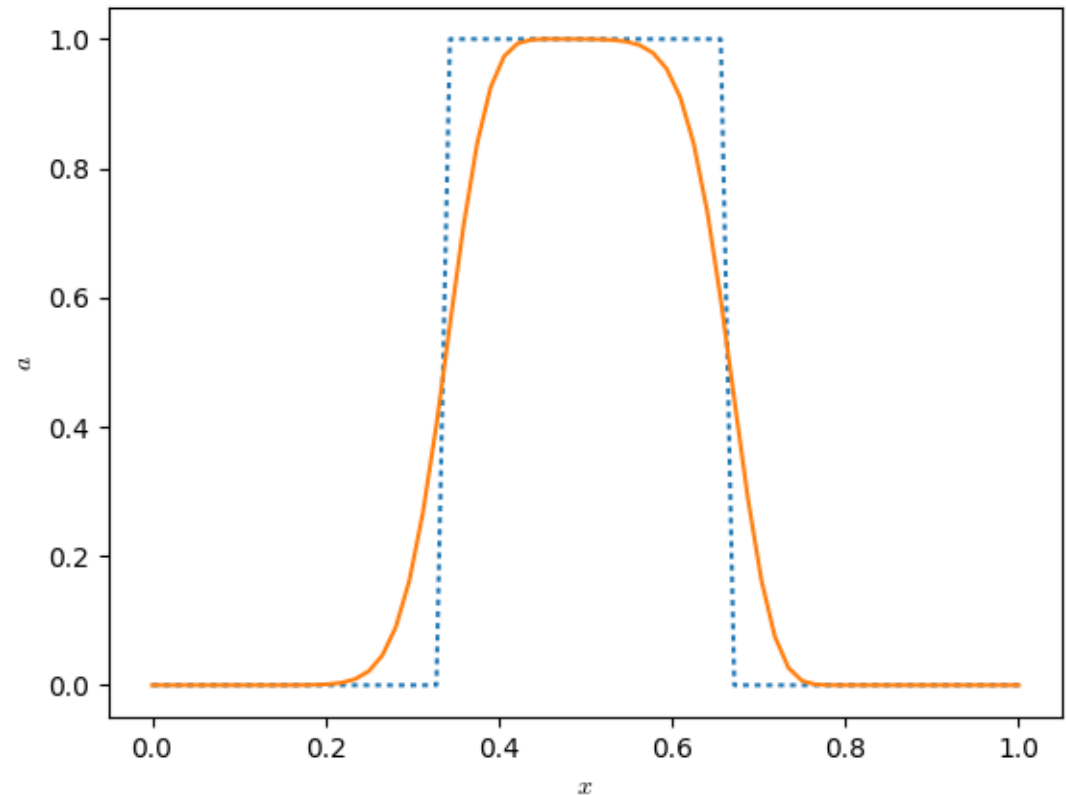
downwind

- Let's try the upwinded difference—that gives:

$$a_i^{n+1} = a_i^n - C(a_i^n - a_{i-1}^n)$$

Upwinding

- Upwinding solution with $N = 65$, $C = 0.9$, after 1 period
 - Much better
 - There still is some error, but it is not unstable
 - Numerical diffusion is evident



code: `fdadvect.py`

Upwinding

- Notice that if we do $C = 1$, we get an exact translation of the data from one zone to the next:

$$a_i^{n+1} = a_i^n - C(a_i^n - a_{i-1}^n)$$

- However, as we will see, for nonlinear eqs. and systems of linear advection eqs., we cannot in general do $C = 1$, because there is not a single speed over all the grid.
- Let's play with the code and explore resolution, number of periods, etc...

Stability Analysis of Upwinded Eq.

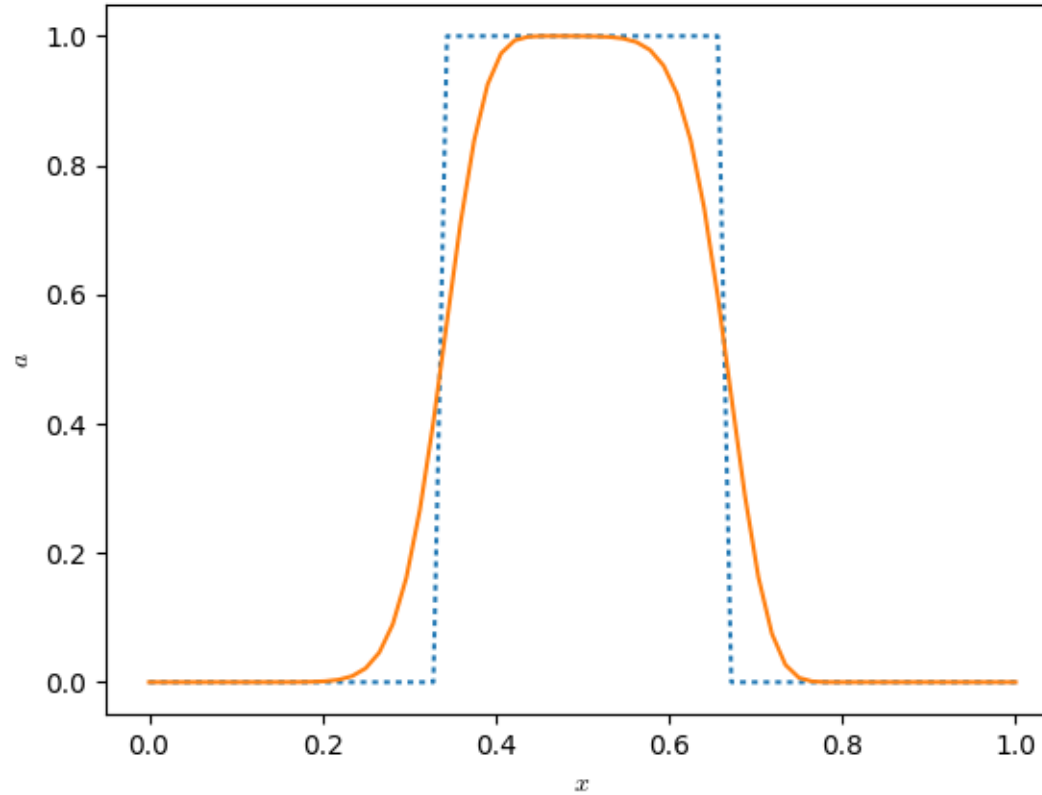
- Truncation analysis would show this method is equivalent to

$$a_t + ua_x = \frac{1}{2}u\Delta x(1 - C)a_{xx}$$

- Represents physical diffusion so long as $1 - C > 0$
 - This also shows that we get the exact solution for $C = 1$
- Note that if we used the downwind difference, our method would be unconditionally unstable
 - Direction of difference based on sign of velocity
- Physically, the choice of upwinding means that we make use of the information from the direction the wind is blowing
 - This term originated in the weather forecasting community

Upwind Results

- Finite-difference (node-centered) grid, with $N=65$, 1 period:
 - Tophat initial conditions, $C = 0.8$

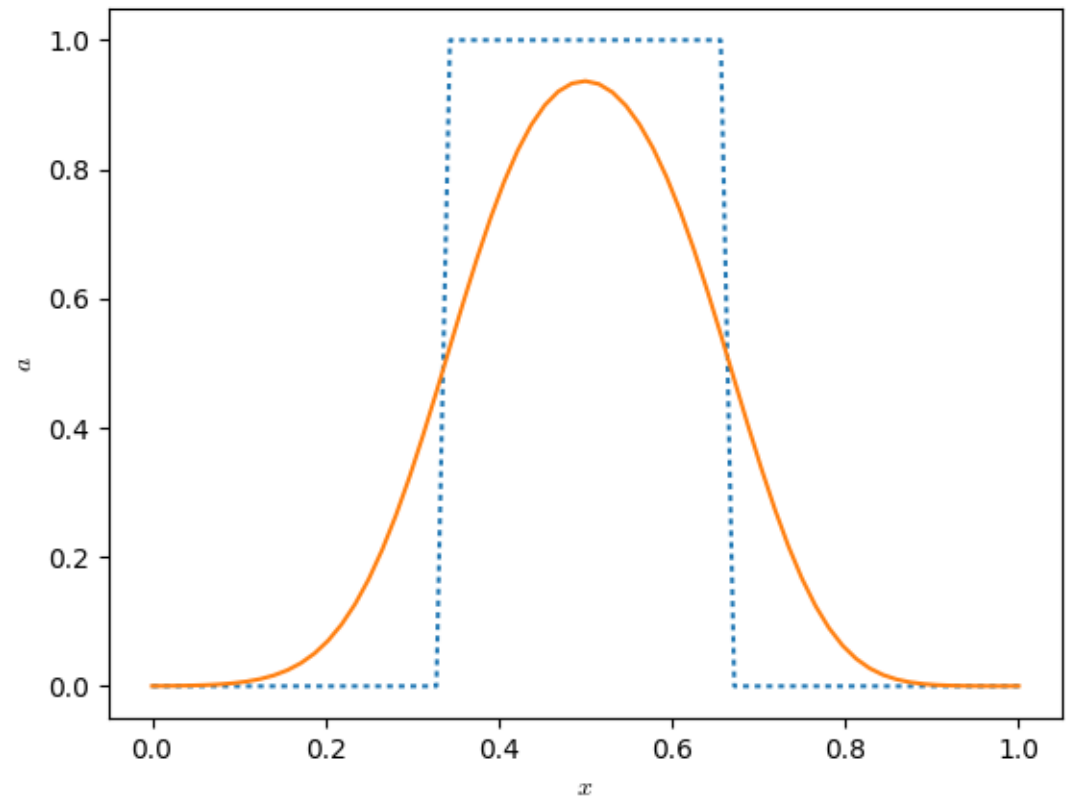


Upwind Results

- Finite-difference (node-centered) grid, with $N=65$, 5 periods:
 - Tophat initial conditions, $C = 0.9$

Notice that the numerical diffusion is strongly apparent here

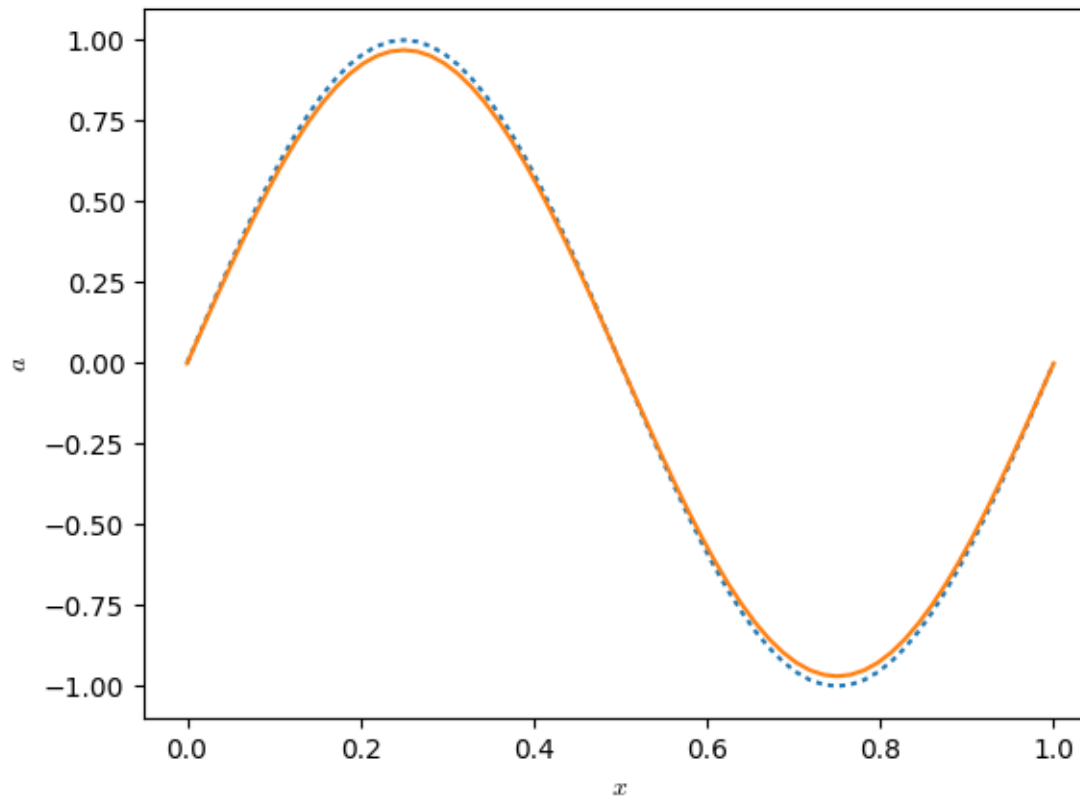
What about smooth initial conditions?



code: fdupwind.py

Upwind Results

- Finite-difference (node-centered) grid, with $N=65$, 1 period:
 - sine wave, $C = 0.9$



code: fdupwind.py

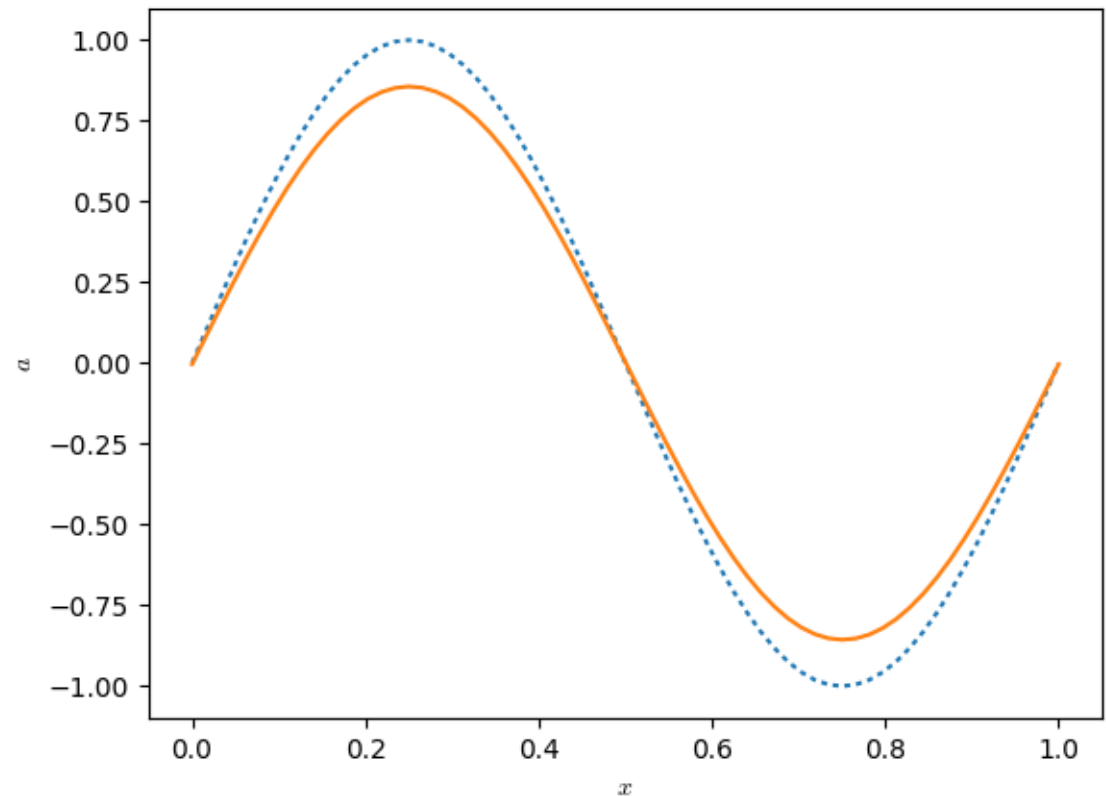
Upwind Results

- Finite-difference (node-centered) grid, with $N=65$, 5 periods
 - sine wave, $C = 0.9$

Note that the sine wave stays in phase (that's a good thing)

Diffusion still apparent.

Just for fun, let's try a downwinded method...

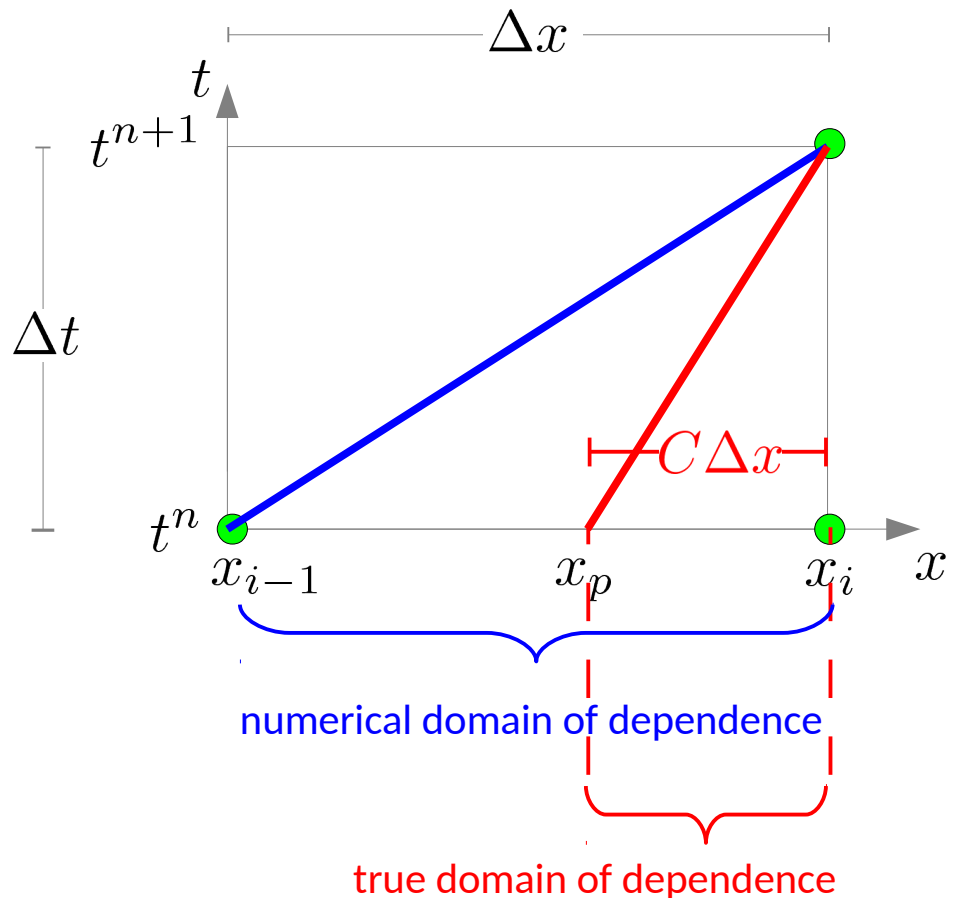


code: fdupwind.py

A Little More Insight on Stability

- A necessary (**but not sufficient**) condition for stability is that the **numerical domain of dependence** contain the **true domain of dependence** (see Lahey, Ch. 12 for a nice discussion)
 - True domain of dependence is found by tracing the characteristics backwards in time

Upwind method ►
For $u > 0$



A Little More Insight on Stability

- The downwind method clearly violates this
- Ex from Lahey:

- Consider initially discontinuous data

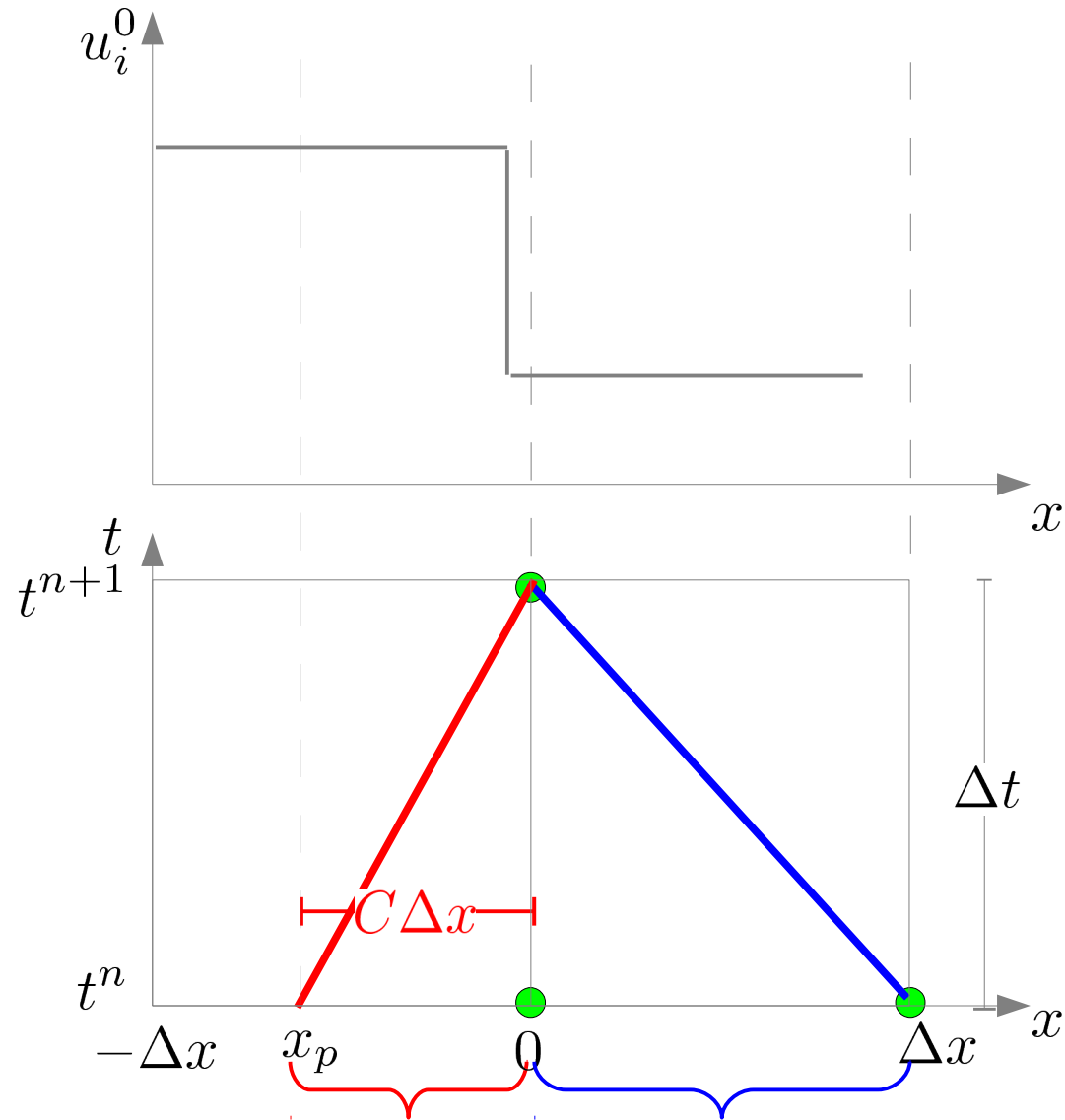
$$a_i^0 = a(i\Delta x, 0) = \begin{cases} 1 & i < 0 \\ 0 & i \geq 0 \end{cases}$$

- True solution at $x = 0$ is $a_0^1 = 1$
- Downwind method:

$$a_i^{n+1} = a_i^n - C(a_{i+1}^n - a_i^n)$$

gives:

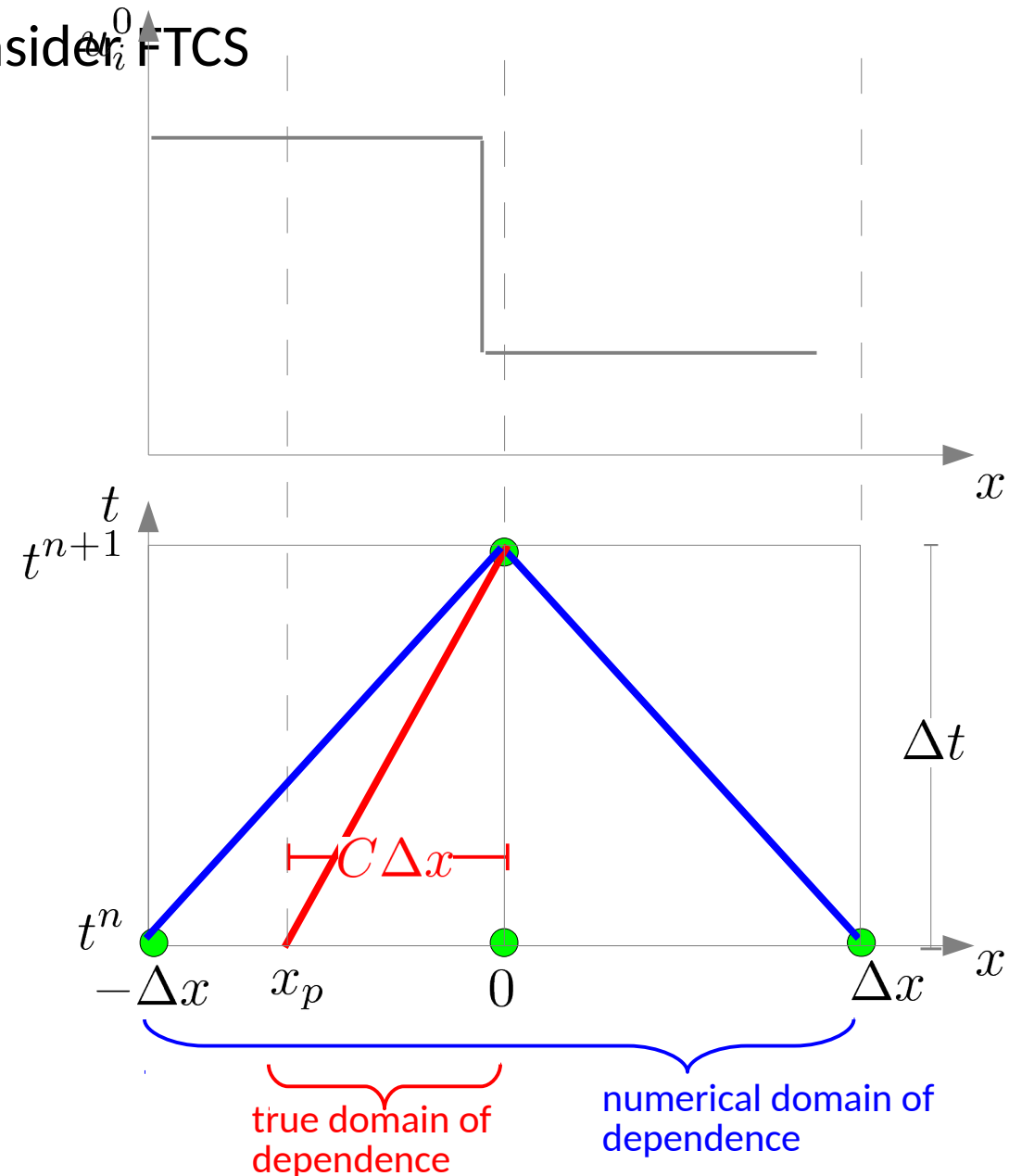
$$a_0^1 = a_0^0 - C(a_1^0 - a_0^0) = 0$$



The numerical domain of dependence doesn't include the true domain of dependence, and we get the wrong answer.

A Little More Insight on Stability

- Necessary, but not sufficient—consider FTCS



Other Finite-Difference Methods

- There are many other finite-difference methods that build off of what we already saw
- **Lax-Friedrichs:**
 - Start with FTCS: $a_i^{n+1} = a_i^n - \frac{C}{2}(a_{i+1}^n - a_{i-1}^n)$
 - Simple change, replace: $a_i^n \rightarrow \frac{1}{2}(a_{i-1}^n + a_{i+1}^n)$
 - We get:
$$a_i^{n+1} = \frac{1}{2}(1 + C)a_{i-1}^n + \frac{1}{2}(1 - C)a_{i+1}^n$$
 - This is stable for $C < 1$, but notice that it doesn't contain the point we are updating! (**Let's explore this...**)
 - Can suffer from odd-even decoupling
 - Can be more diffusive than upwinding (and gets worse for smaller C)
- Other methods build on these same ideas, e.g., **Lax-Wendroff**

Measuring Convergence

- As we move to higher-order methods, we will want to measure the convergence of our methods
 - We have data at N points for each time level
 - We need a norm that operates on the gridded data. A popular choice is the L2 norm:

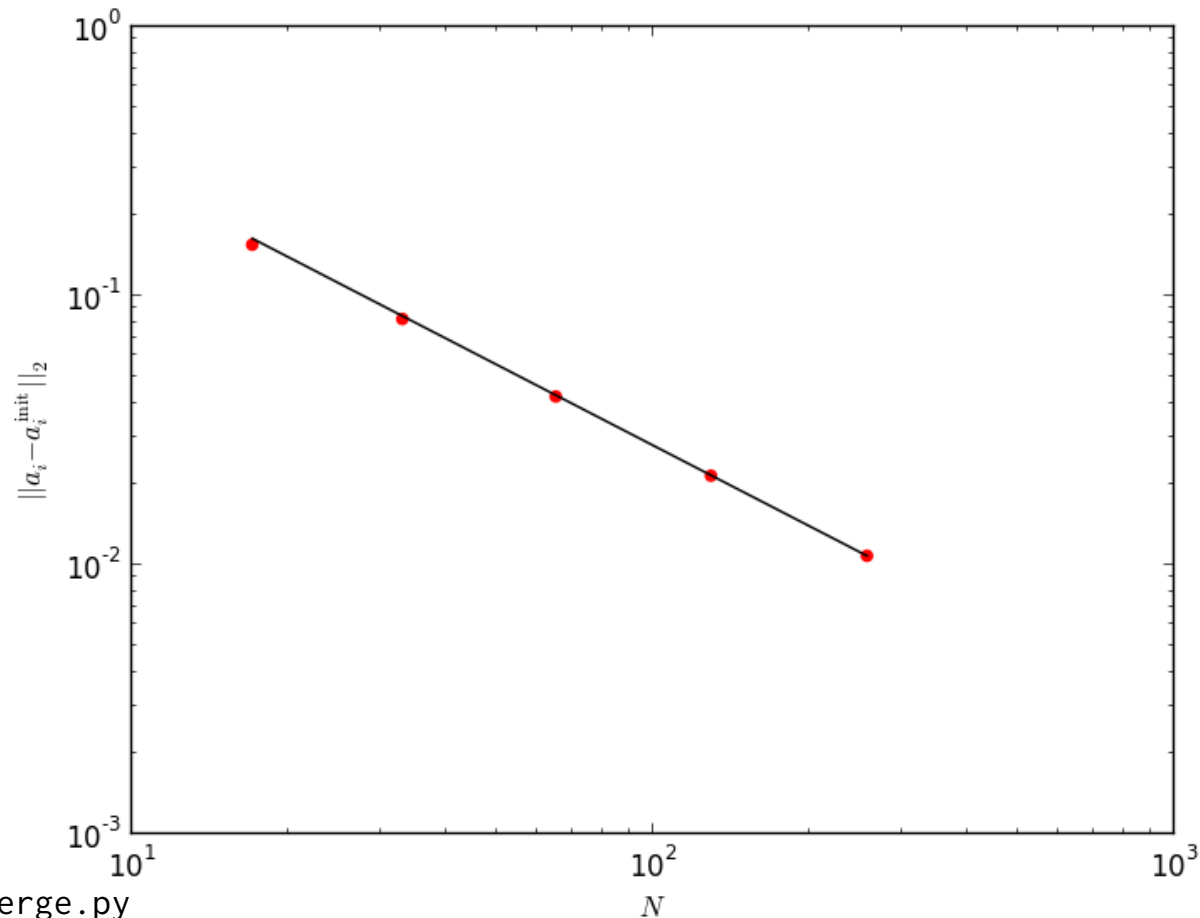
$$\|\phi\|_2 = \left(\Delta x \sum_{i=0}^{N-1} \phi_i^2 \right)^{1/2}$$

- Using the grid width ensure that this measure is resolution independent
- We can take the error of our method to be:

$$\epsilon = \|a_i - a^{\text{exact}}(x_i)\|_2$$

Measuring Convergence

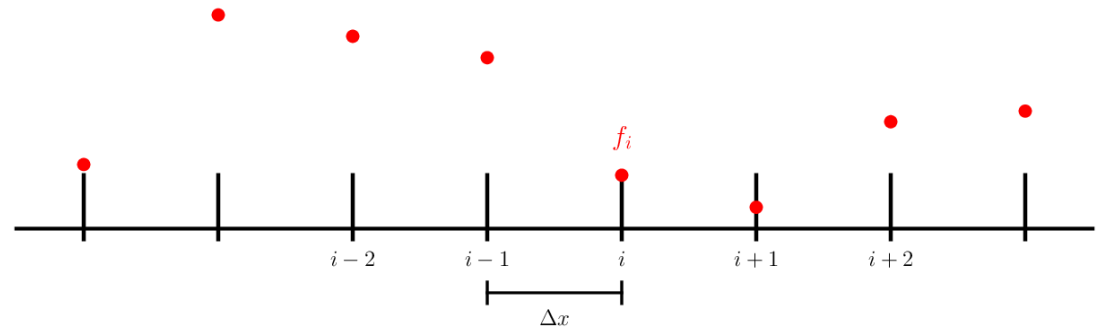
- For advection with periodic BCs, compare to the initial conditions
 - Note: you need to be careful to end always at the same time (e.g. exactly on a period)



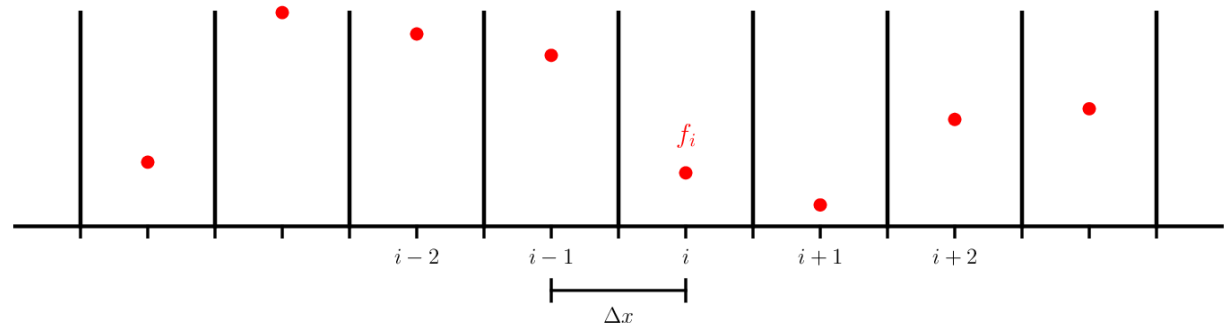
code: fdupwind_converge.py

Grid Types

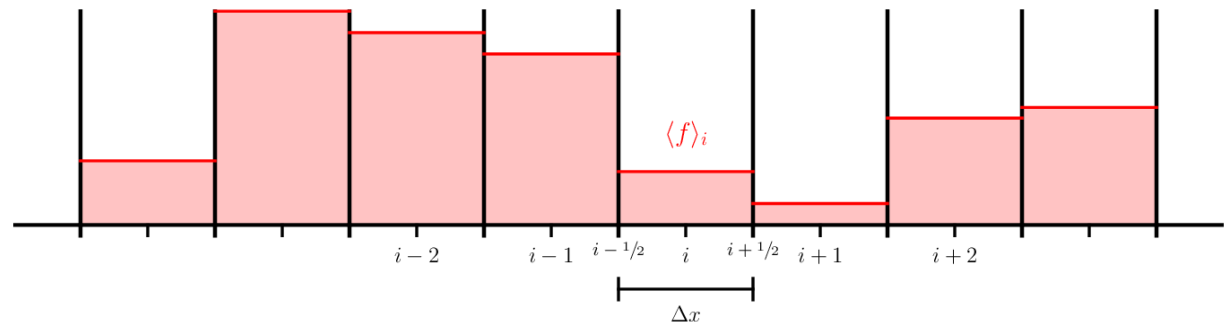
“Regular” **finite-difference grid**. Data is associated with nodes spaced Δx apart. Note that here we can have a point exactly on the boundary



Cell-centered finite-difference grid. Here we consider cells of width Δx and associate the data with a point at the center of the cell. Note that data will be $\Delta x/2$ inside the boundary



Finite-volume grid—similar to the cell-centered grid, we divide the domain into cells/zones. Now we store the average value of the function in each zone.



Grid Types

- Finite-difference

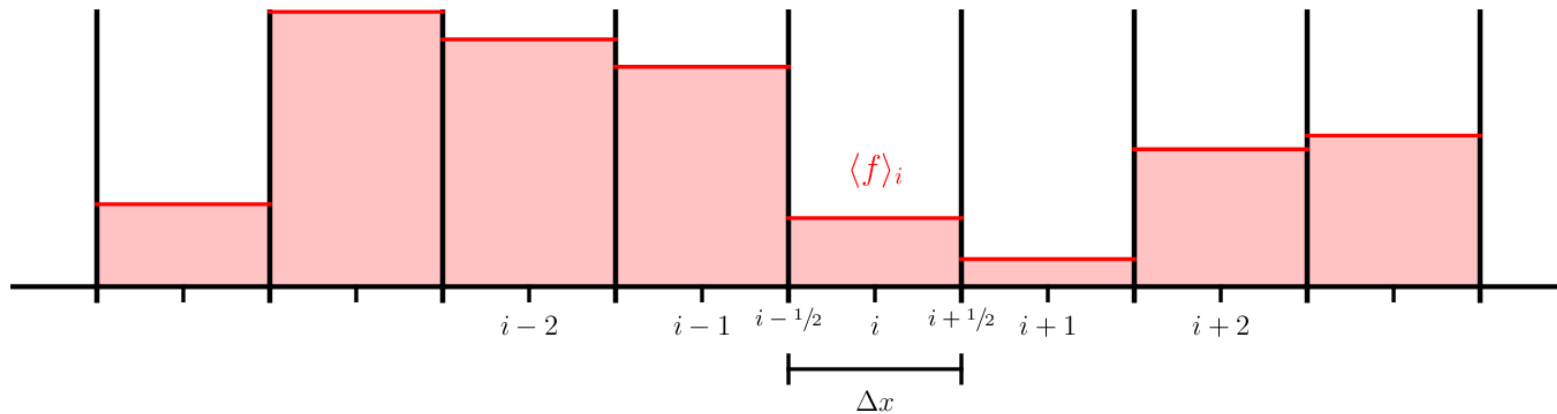
- Some methods use staggered grids: some variables are on the nodes (cell-boundaries) some are at the cell centers
- Boundary condition implementation will differ depending on the centering of the data
- For cell-centered f-d grids: ghost cells implement the boundary conditions

- Finite-volume

- This is what we'll focus on going forward
- Very similar in structure to cell-centered f-d, but the interpretation of the data is different.

Finite-Volume Grid

- We store the average of a quantity in a zone



$$\langle f \rangle_i = \frac{1}{\Delta x} \int_{x_{i-1/2}}^{x_{i+1/2}} f(x) dx$$

Conservation Laws

- Many systems appear as conservation laws:

$$U_t + [F(U)]_x = 0$$

- Linear advection: $a_t + [ua]_x = 0$

- Burgers' equation: $u_t + \left[\frac{1}{2} u^2 \right]_x = 0$

- Hydrodynamics

$$U = \begin{pmatrix} \rho \\ \rho u \\ \rho E \end{pmatrix} \quad F(U) = \begin{pmatrix} \rho u \\ \rho u u + p \\ \rho u E + u p \end{pmatrix}$$

Understanding advection is key to each of these systems

Finite-Volume Approximation

- Finite-volume methods are designed for conservation laws
 - Integrate our hyperbolic equation over a control volume

$$\frac{1}{\Delta x} \int_{x_{i-1/2}}^{x_{i+1/2}} U_t = - \frac{1}{\Delta x} \int_{x_{i-1/2}}^{x_{i+1/2}} \frac{\partial}{\partial x} F(U) dx$$
$$\frac{\partial}{\partial t} U_i = - \frac{1}{\Delta x} \{ [F(U)]_{i+1/2} - [F(U)]_{i-1/2} \}$$

- Here we write the cell-averages without the $\langle \rangle$
 - **Telescoping property**
 - Note that the flux of U that leaves one volume enters the adjacent one—guaranteeing conservation
- We still need to discretize in time

Finite-Volume Approximation

- Two different approaches for handling time discretization
 - Method of lines
 - Now that we've discretized in space, we are left with an ODE in time

$$\frac{\partial}{\partial t} U_i = -\frac{1}{\Delta x} \{ [F(U)]_{i+1/2} - [F(U)]_{i-1/2} \}$$

- Use our ODE methods (like Runge-Kutta) on this
 - Explicitly discretize in time using a difference approximation
 - Second order accuracy would require that the RHS is evaluated at the midpoint in time

Finite-Volume Approximation

- A second-order approximation for our advection equation is

$$\frac{a_i^{n+1} - a_i^n}{\Delta t} = - \frac{[f(a)]_{i+1/2}^{n+1/2} - [f(a)]_{i-1/2}^{n+1/2}}{\Delta x}$$

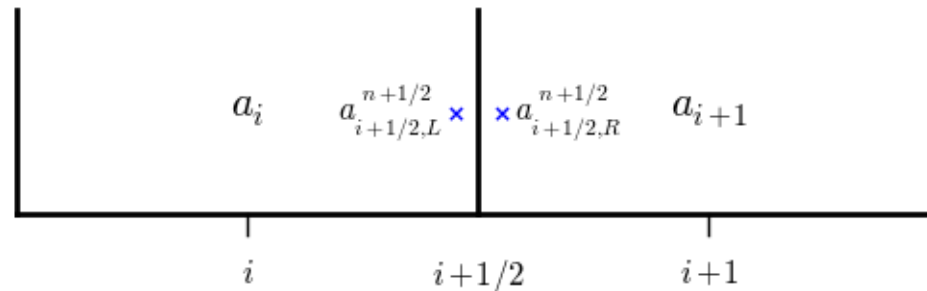
- Notice that the RHS is centered in time
- To complete this method, we need to compute the fluxes on the edges
 - We can do this in terms of the interface state

$$[f(a)]_{i+1/2}^{n+1/2} = f(a_{i+1/2}^{n+1/2})$$

- Now we need to find the interface state. There are two possibilities: coming from the left or right of the interface

First-Order Advection

- Our interface states



- Simplest choice:

$$a_{i+1/2,L}^{n+1/2} = a_i^n; \quad a_{i+1/2,R}^{n+1/2} = a_{i+1}^n$$

- Now we need to resolve the degeneracy of the states. This requires knowledge of the equation
 - **Riemann problem**: two states separated by an interface
 - **For advection, we do upwinding**

First-Order Advection

- Riemann problem (upwinding):

$$\mathcal{R}(a_{i+1/2,L}^{n+1/2}, a_{i+1/2,R}^{n+1/2}) = \begin{cases} a_{i+1/2,L}^{n+1/2} & u > 0 \\ a_{i+1/2,R}^{n+1/2} & u < 0 \end{cases}$$

- This is simple enough that we can write out the resulting update (for $u > 0$):

$$\frac{a_i^{n+1} - a_i^n}{\Delta t} = - \frac{ua_i^n - ua_{i-1}^n}{\Delta x}$$

- This is identical to the first-order upwind finite-difference discretization we already studied

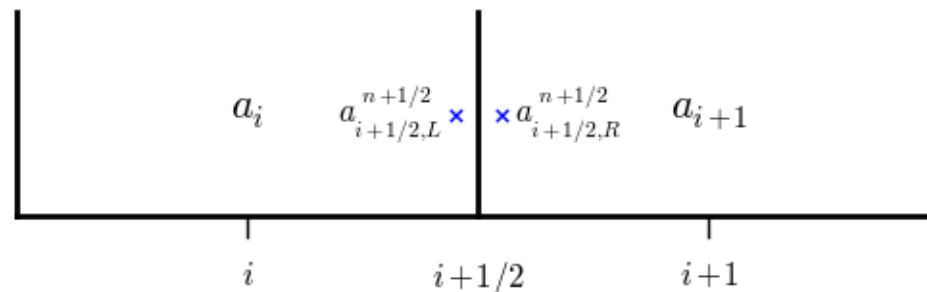
Finite-Volume Approximation

- Back to the second-order approximation for our advection equation is

$$\frac{a_i^{n+1} - a_i^n}{\Delta t} = - \frac{[f(a)]_{i+1/2}^{n+1/2} - [f(a)]_{i-1/2}^{n+1/2}}{\Delta x}$$

- Now we need to deal with the time-centering on the righthand side
- We will express the fluxes in terms of edge-centered, time-centered states

$$[f(a)]_{i+1/2}^{n+1/2} = f(a_{i+1/2}^{n+1/2})$$



Second-Order Advection

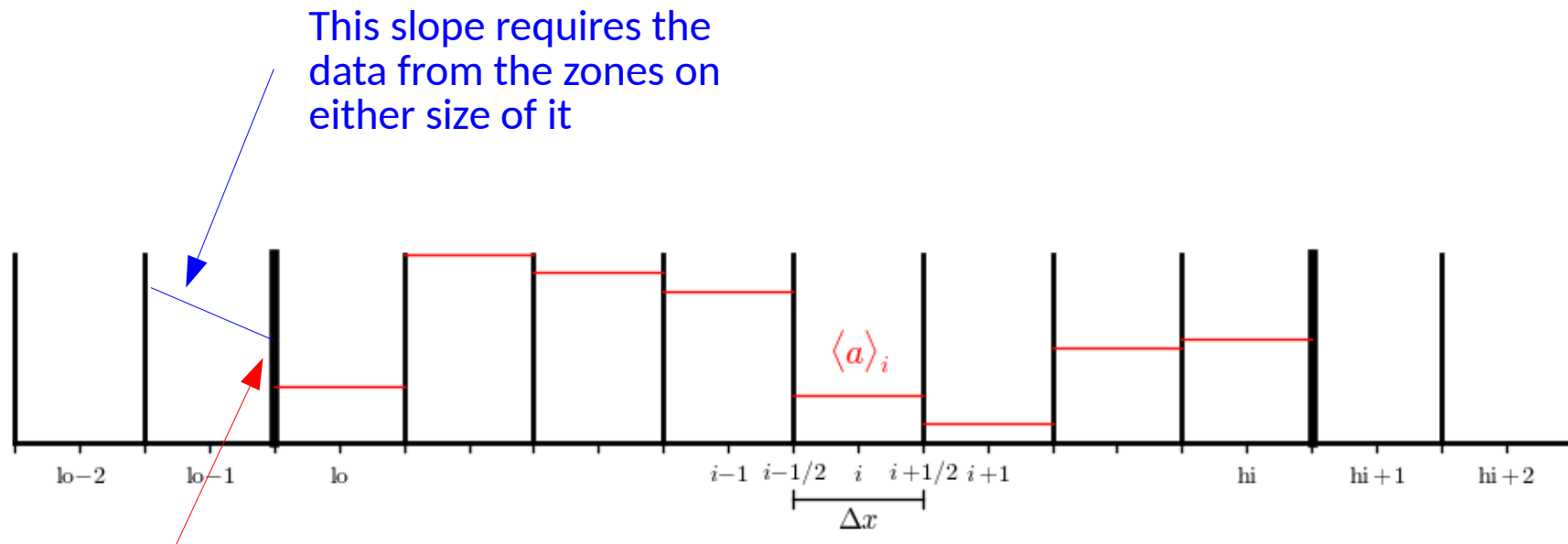
- Going to second-order requires making the interface states second-order in space and time
 - We will use a **piecewise linear reconstruction** of the cell-average data to approximate the true functional form
 - We Taylor expand in space and time to find the time-centered, interface state
- **Let's derive this on the blackboard...**

Design of a General Solver

- Simulation codes for solving conservation laws generally follow the following flow:
 - Set initial conditions
 - Main evolution loop (loop until final time reached)
 - Fill boundary conditions
 - Get the timestep
 - Compute the interface states
 - Solve the Riemann problem
 - Do conservative update
- We'll see that this same procedure can be applied to nonlinear problems and systems (like the equations of hydrodynamics)

Second-Order Advection

- We need 2 ghost cells on each end:

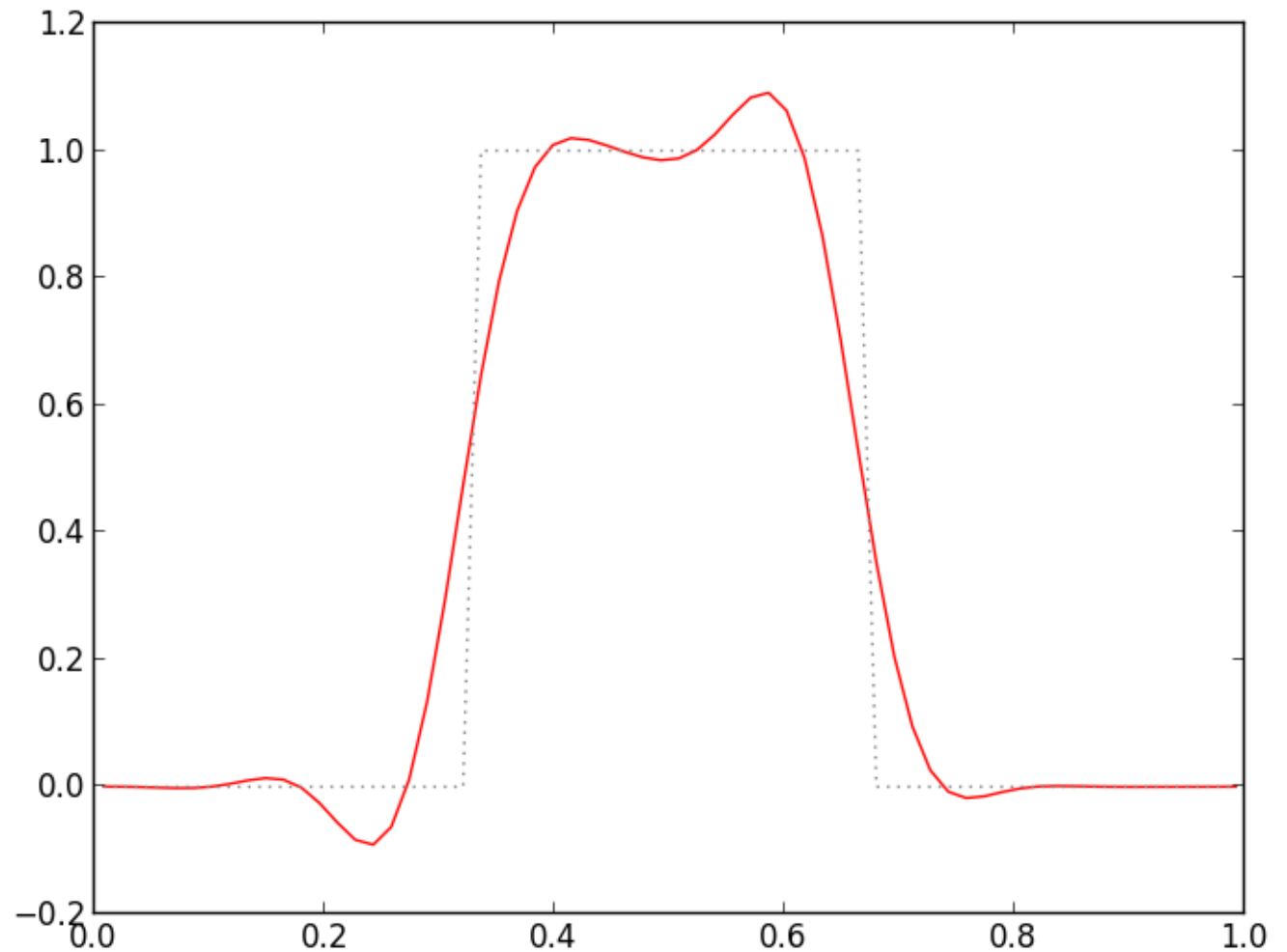


- Let's look at the code...

Second-Order Advection

- Tophat with 64 zones, $C = 0.8$

Notice the oscillations
—they seem to be
associated with the
steep jumps

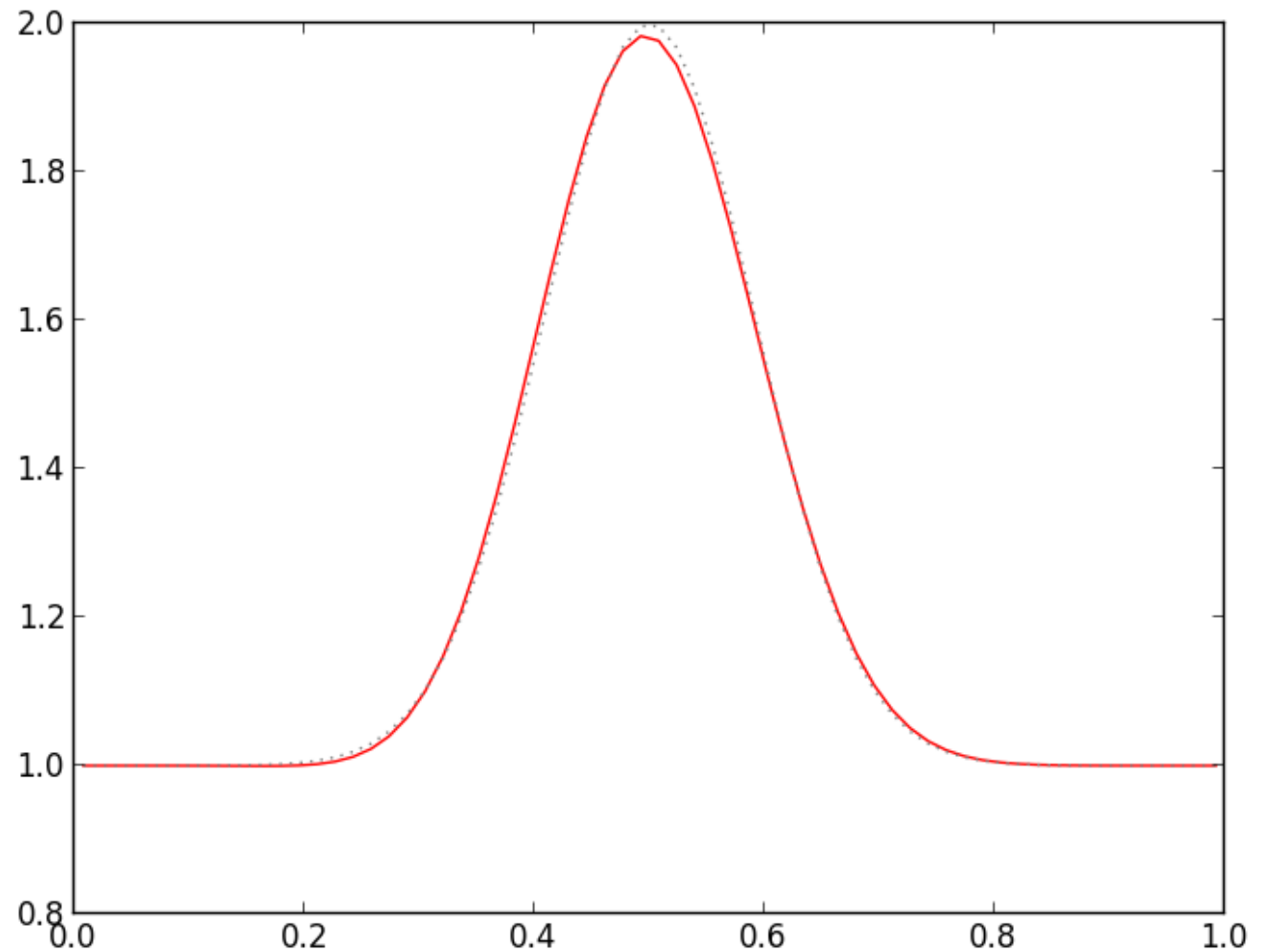


code: fv_advection.py

Second-Order Advection

- Gaussian with 64 zones, $C = 0.8$

This looks nice and smooth

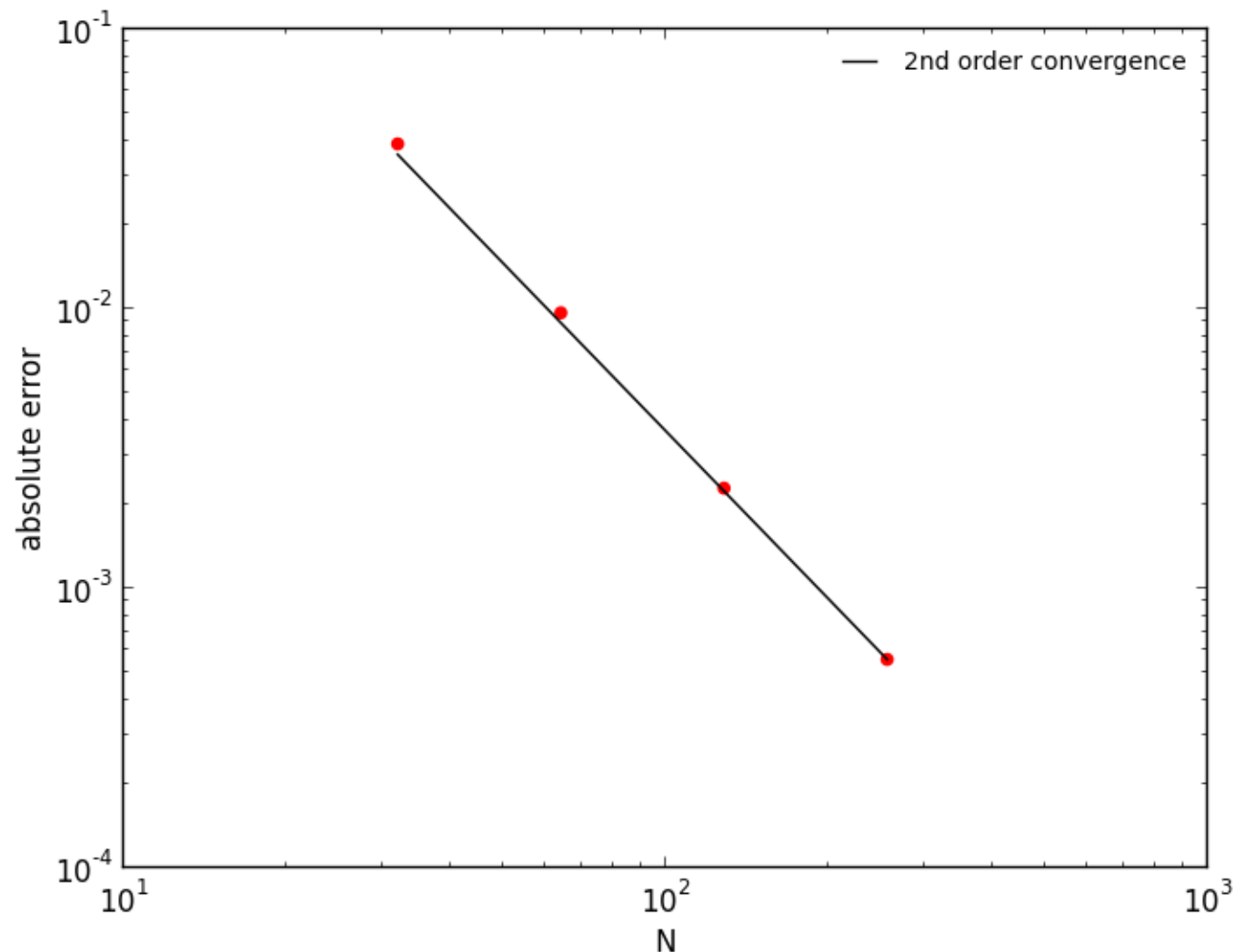


code: fv_advection.py

Second-Order Advection

- Remember—you should always look at the convergence of your code—if you do not get what you are supposed to get, you probably have a bug (or you don't understand the method well enough...)

Convergence for
the Gaussian

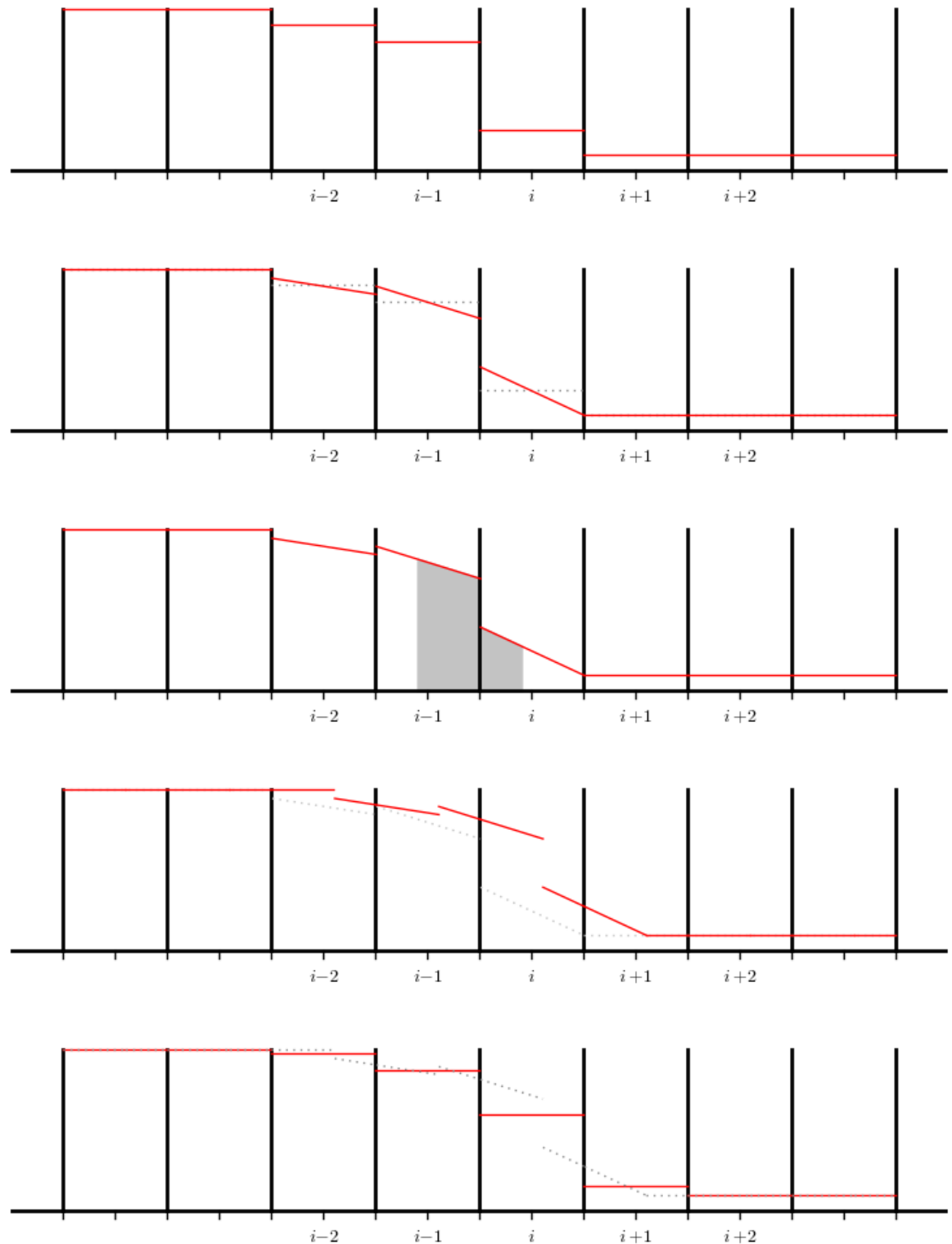


Second-Order Advection

- General notes:
 - We derived both the left and right state at each interface—for advection, we know that we are upwinding, so we only really need to do the upwinded state
 - But, for a general conservation law, the upwind direction can change from zone to zone and timestep to timestep AND there can be multiple waves (e.g. systems), so for the general problem, we need to compute both states
- If you set the slopes to 0, you reduce to the first-order method

REA

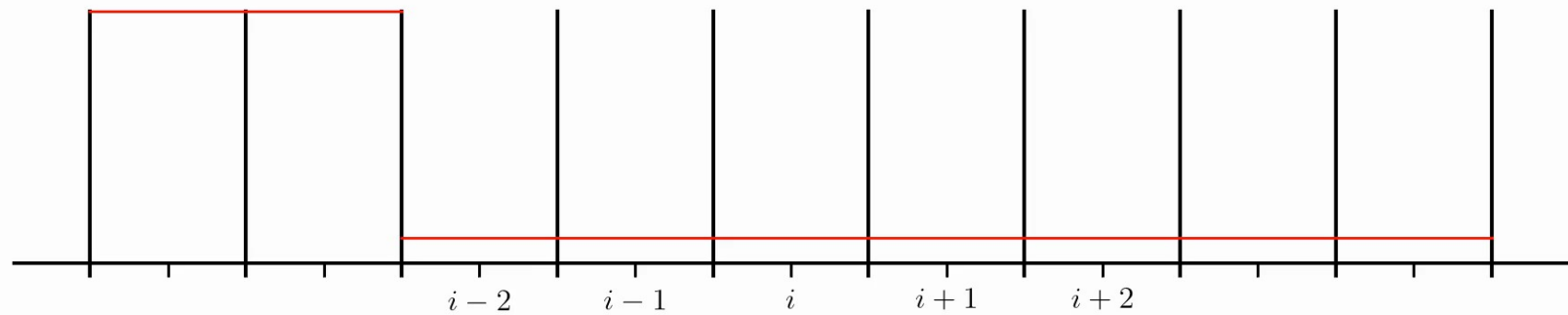
- An alternate way to think about these methods is as a Reconstruct-Evolve-Average process
 - The second-order method we derived is equivalent to using a piecewise linear reconstruction of the data in each zone
 - We can demonstrate this on the blackboard...



REA

Piecewise Linear Method for Linear Advection

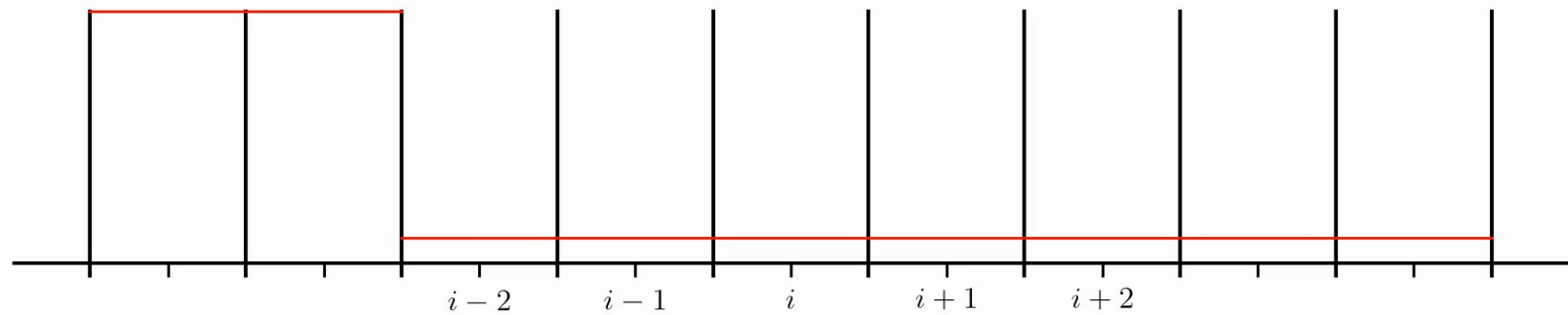
initial state (cell averages)



REA

Piecewise Linear Method for Linear Advection

initial state (cell averages)



Last Time...

- Discretizing in time gave us:

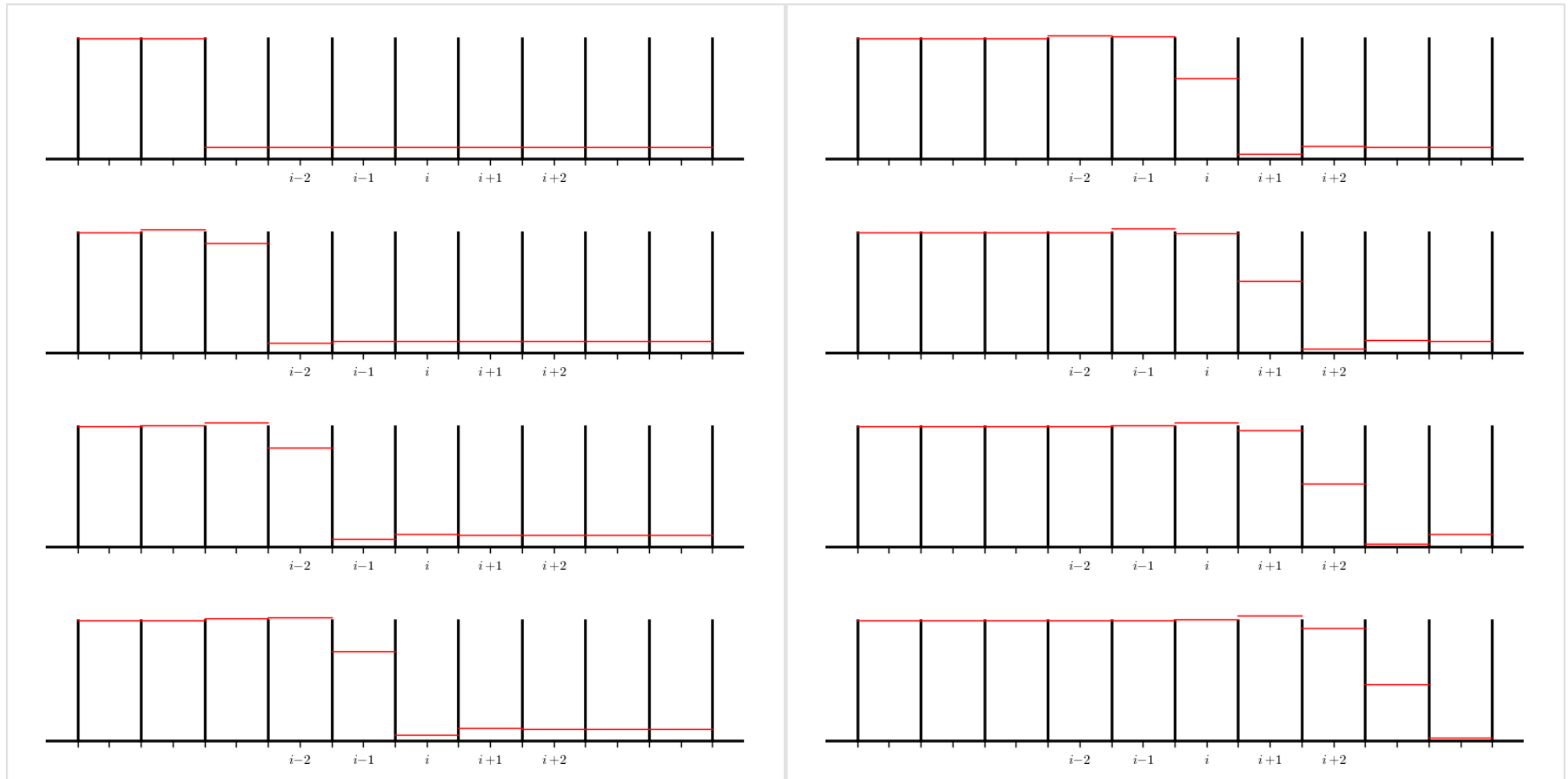
$$\frac{a_i^{n+1} - a_i^n}{\Delta t} = - \frac{[f(a)]_{i+1/2}^{n+1/2} - [f(a)]_{i-1/2}^{n+1/2}}{\Delta x}$$

$$[f(a)]_{i+1/2}^{n+1/2} = f(a_{i+1/2}^{n+1/2})$$

- To get second order in space, we Taylor expand in space and time:

$$\begin{aligned} a_{i+1/2,L}^{n+1/2} &= a_i^n + \frac{\Delta x}{2} \left. \frac{\partial a}{\partial x} \right|_i + \frac{\Delta t}{2} \left. \frac{\partial a}{\partial t} \right|_i + \mathcal{O}(\Delta x^2) + \mathcal{O}(\Delta t^2) \\ &= a_i^n + \frac{\Delta x}{2} \left. \frac{\partial a}{\partial x} \right|_i + \frac{\Delta t}{2} \left(-u \left. \frac{\partial a}{\partial x} \right|_i \right) + \dots \\ &= a_i^n + \frac{\Delta x}{2} \left(1 - \frac{\Delta t}{\Delta x} u \right) \left. \frac{\partial a}{\partial x} \right|_i + \dots \end{aligned}$$

No-limiting Example



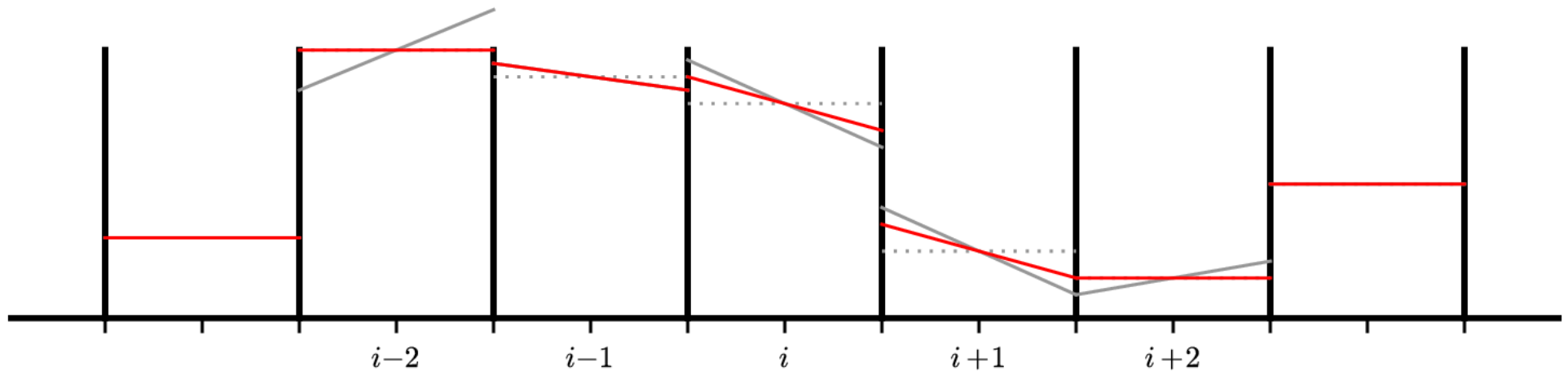
Here we see an initial discontinuity advected using a 2nd-order finite-volume method, where the slopes were taken as unlimited centered differences. Note the overshoot and undershoots

Limiting

- Limiting: modify slopes near extrema to prevent overshoots.
 - Can drop method down to 1st-order accurate near discontinuities
- Godunov's theorem:
 - Any monotonic linear method for advection is first order accurate
 - To be monotonic and 2nd-order, we need to make our method nonlinear
- There are many limiters, derived by requiring that the update not introduce any new minima or maxima.
 - Mathematically enforced by a requirement of total variation diminishing
 - Ex of a simple limiter (minmod):

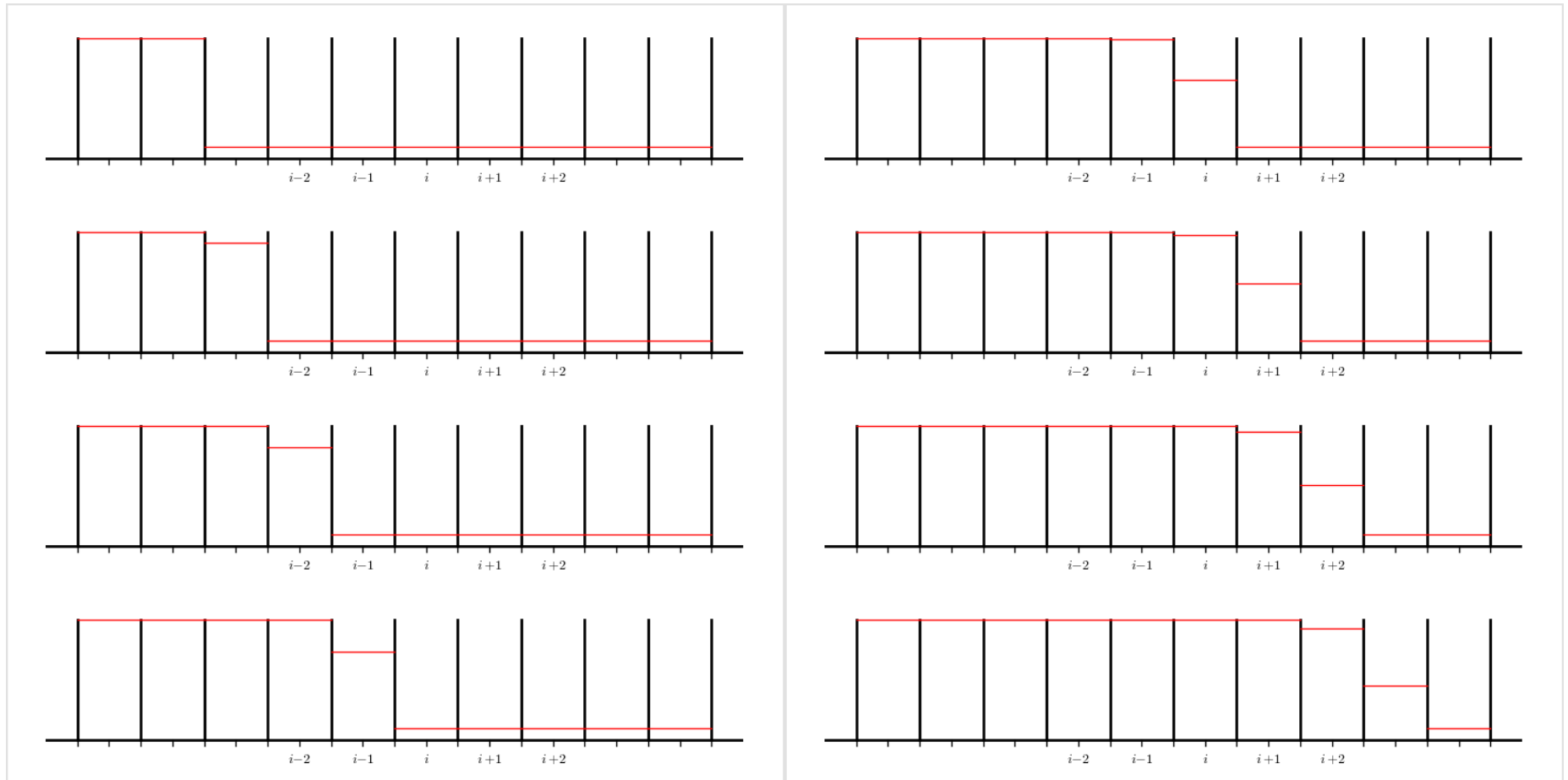
$$\left. \frac{\partial a}{\partial x} \right|_i = \text{minmod} \left(\frac{a_i - a_{i-1}}{\Delta x}, \frac{a_{i+1} - a_i}{\Delta x} \right)$$
$$\text{minmod}(a, b) = \begin{cases} a & \text{if } |a| < |b| \text{ and } a \cdot b > 0 \\ b & \text{if } |b| < |a| \text{ and } a \cdot b > 0 \\ 0 & \text{otherwise} \end{cases}$$

Limiting



A finite-volume grid showing the cell averages (gray, dotted, horizontal lines), unlimited center-difference slopes (gray, solid) and MC limited slopes (red). Note that in zones i and $i+1$, the slopes are limited slightly, so as not to overshoot or undershoot the neighboring cell value. Cell $i-1$ is not limited at all, whereas cells $i-2$, and $i+2$ are fully limited—the slope is set to 0—these are extrema.

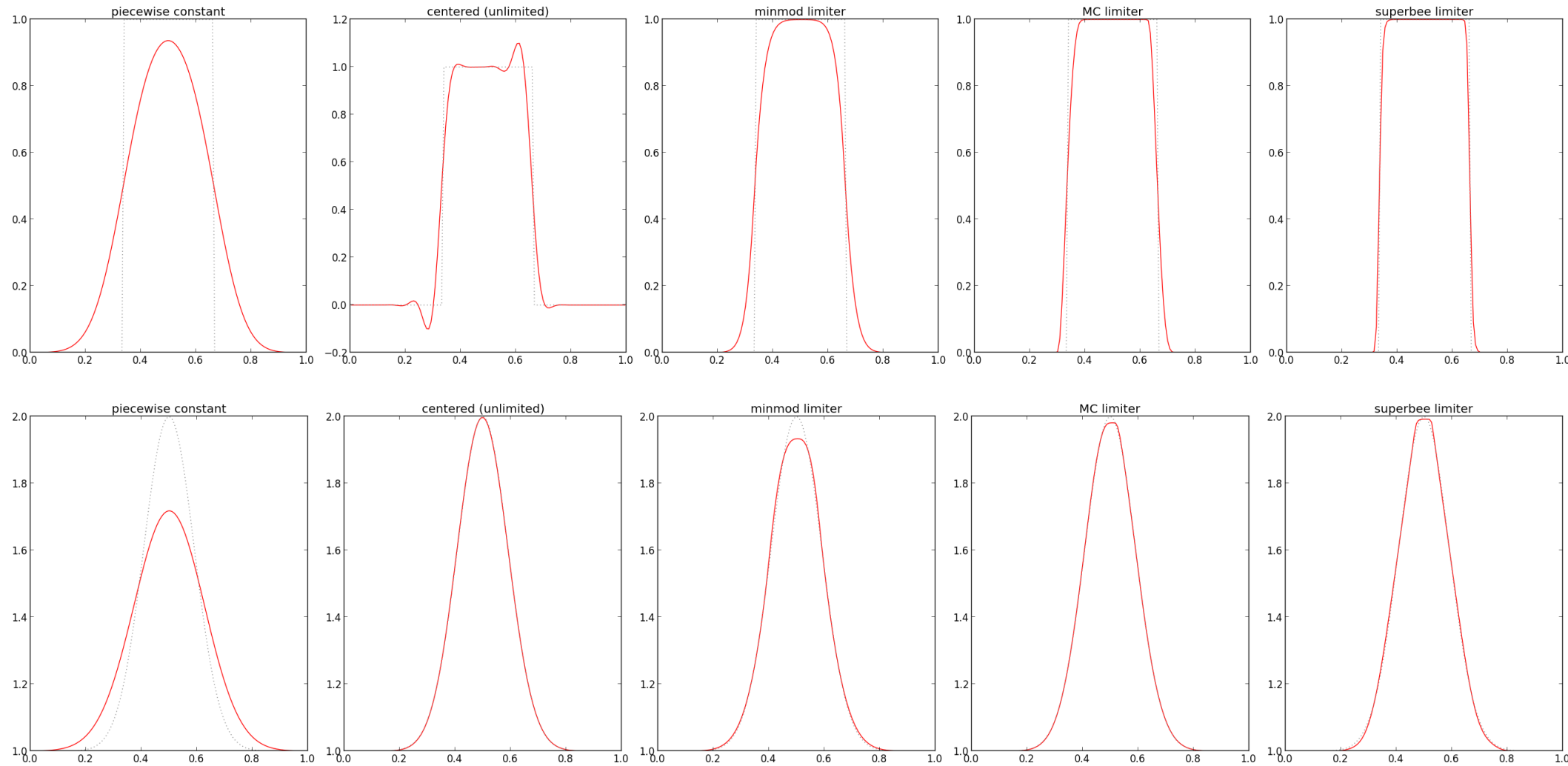
Limiting Example



This is the same initial profile advected with limited slopes. Note that the profile remains steeper and that there are no oscillations.

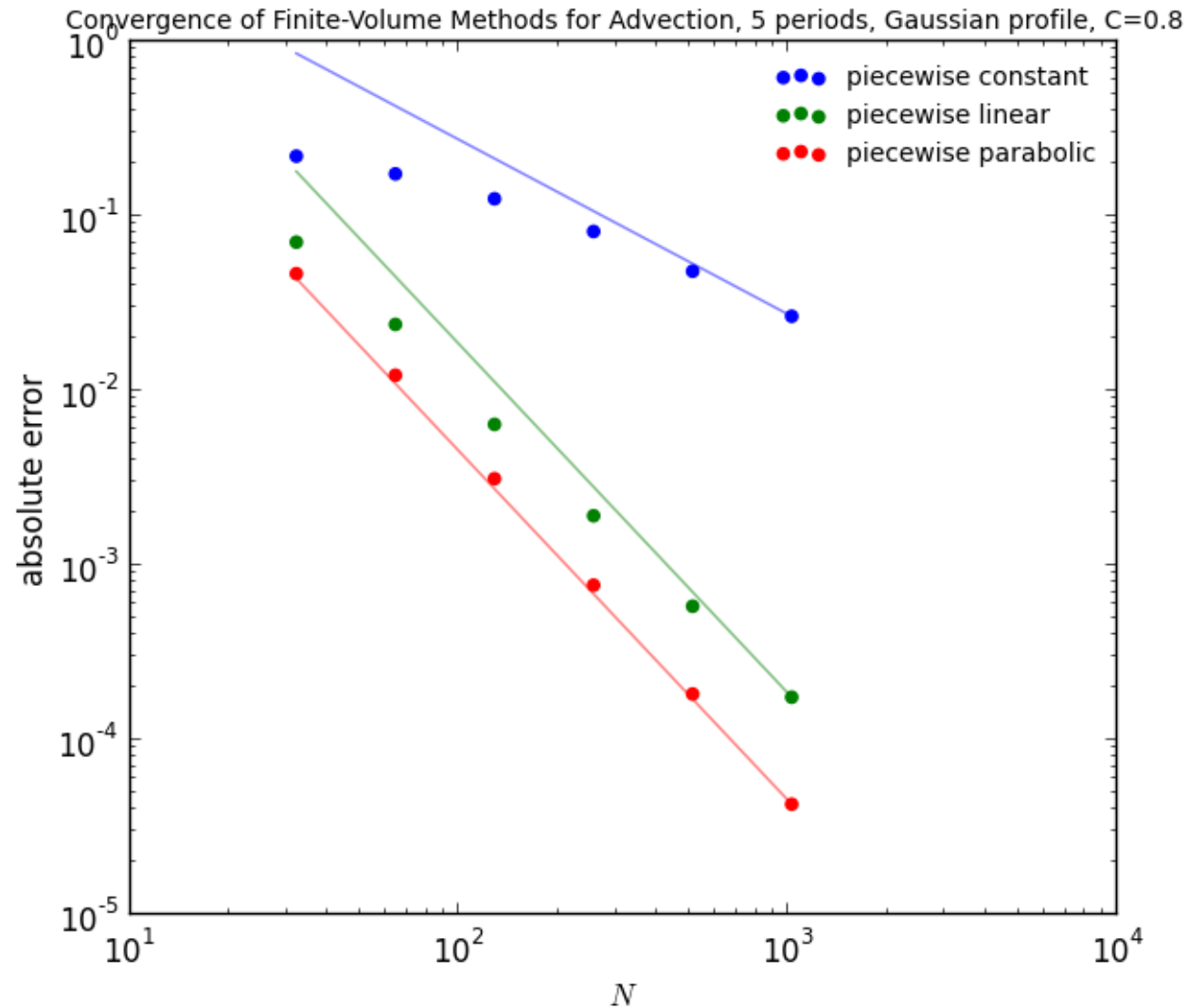
Limiting

- There are many different limiters (examples below with 128 zones, 5 periods, $C = 0.8$)



code: `fv_advection.py`

PPM



Note: this was done with limiting on, which can reduce the ideal convergence

Multi-Dimensional Advection

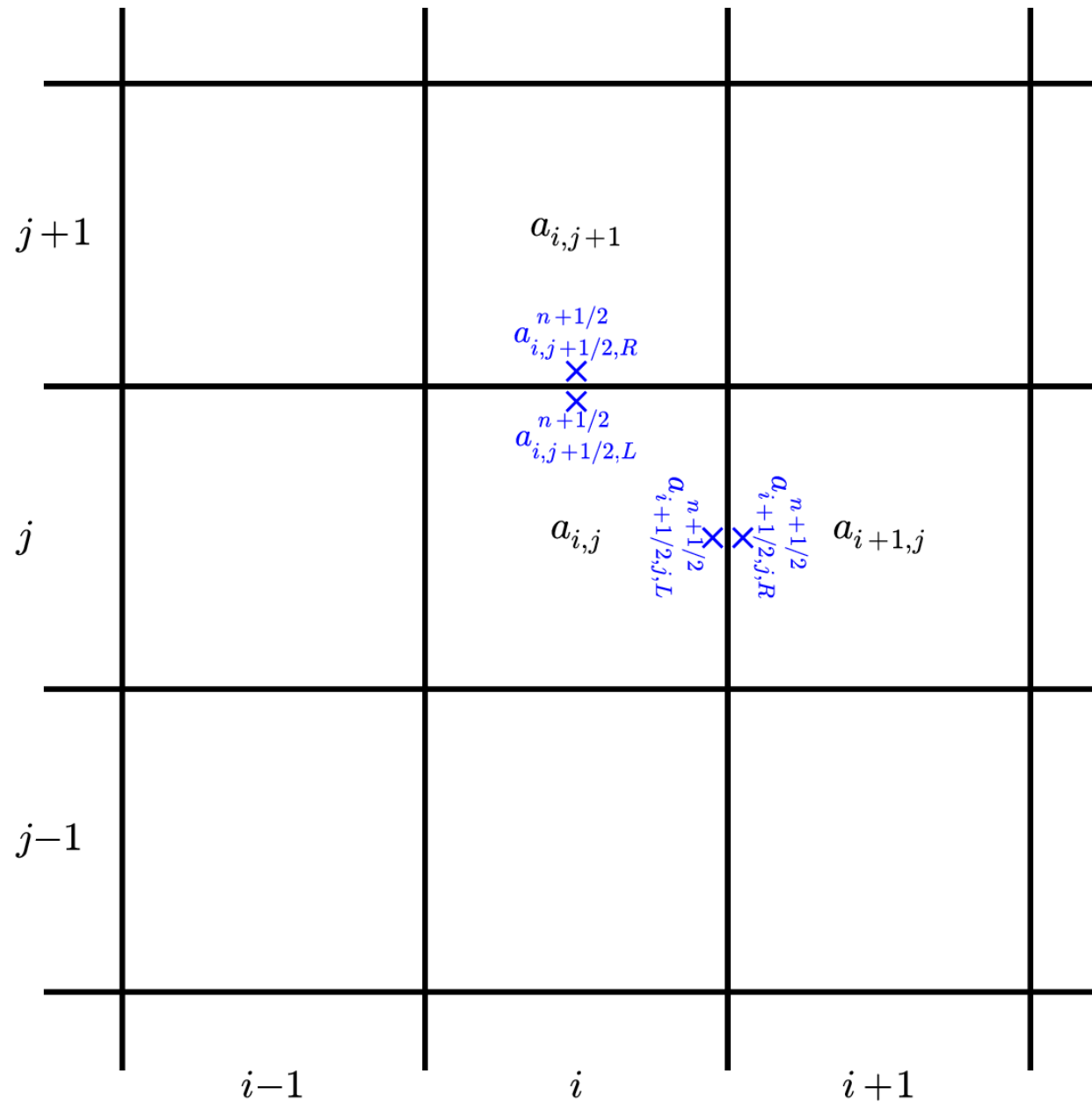
- 2-d linear advection equation:

$$a_t + ua_x + va_y = 0$$

- Average state updates due to fluxes through all interfaces

$$\frac{a_{i,j}^{n+1} - a_{i,j}^n}{\Delta t} = - \frac{(ua)_{i+1/2,j}^{n+1/2} - (ua)_{i-1/2,j}^{n+1/2}}{\Delta x} - \frac{(va)_{i,j+1/2}^{n+1/2} - (va)_{i,j-1/2}^{n+1/2}}{\Delta y}$$

Multi-Dimensional Advection



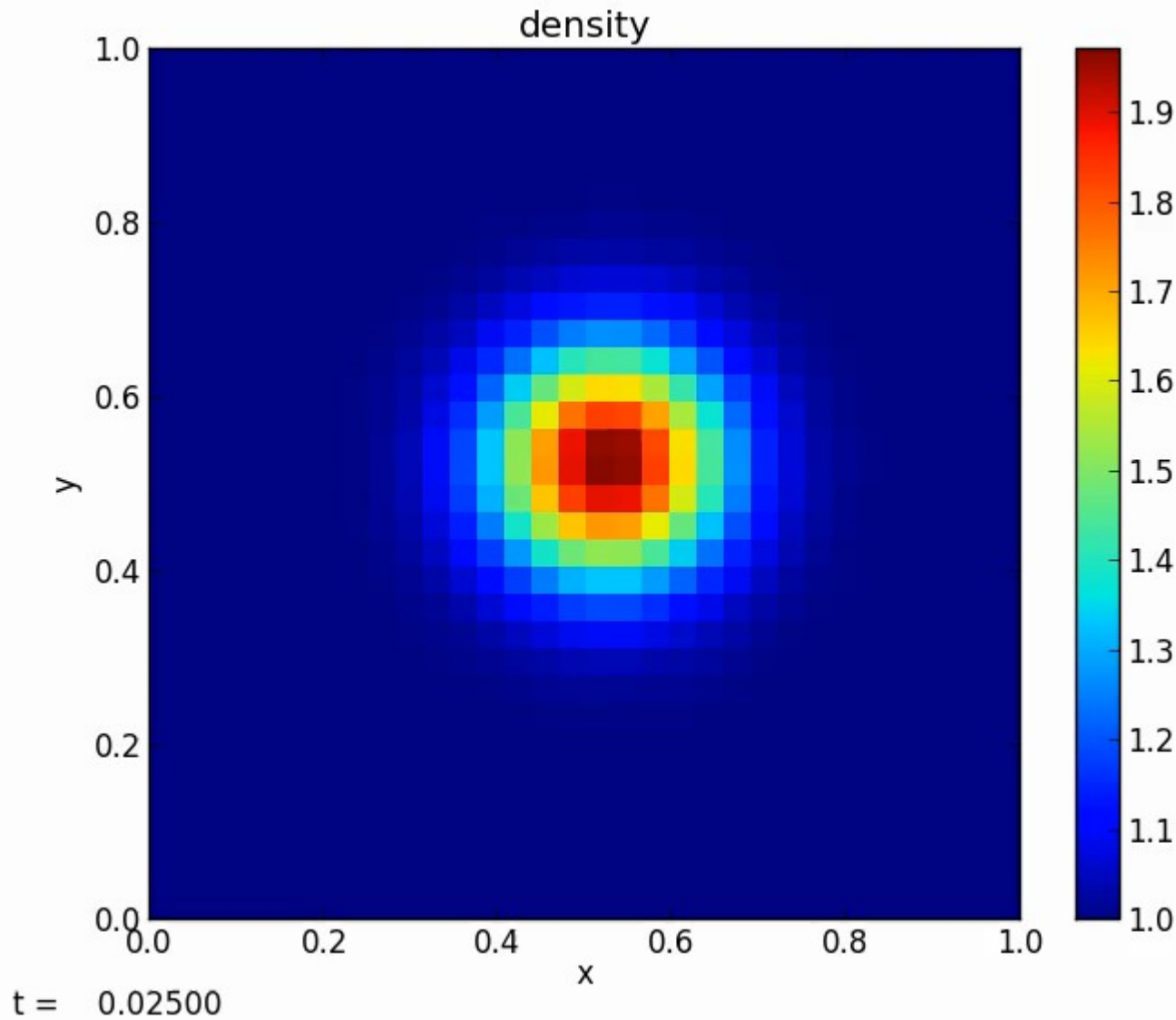
Multi-Dimensional Advection

- Simplest method: **dimensional splitting**
 - Perform the update one dimension at a time—this allows you to reuse your existing 1-d solver

$$\frac{a_{i,j}^* - a_{i,j}^n}{\Delta t} = - \frac{u a_{i+1/2,j}^{n+1/2} - u a_{i-1/2,j}^{n+1/2}}{\Delta x}$$
$$\frac{a_{i,j}^{n+1} - a_{i,j}^*}{\Delta t} = - \frac{v a_{i,j+1/2}^{*,n+1/2} - v a_{i,j-1/2}^{*,n+1/2}}{\Delta y}$$

- More complicated: **unsplit prediction** of interface states
 - In Taylor expansion, we now keep the transverse terms at each interface
 - Difference of transverse flux terms is added to normal states
 - Better preserves symmetries
 - See Colella (1990) and PDF notes on course page for overview

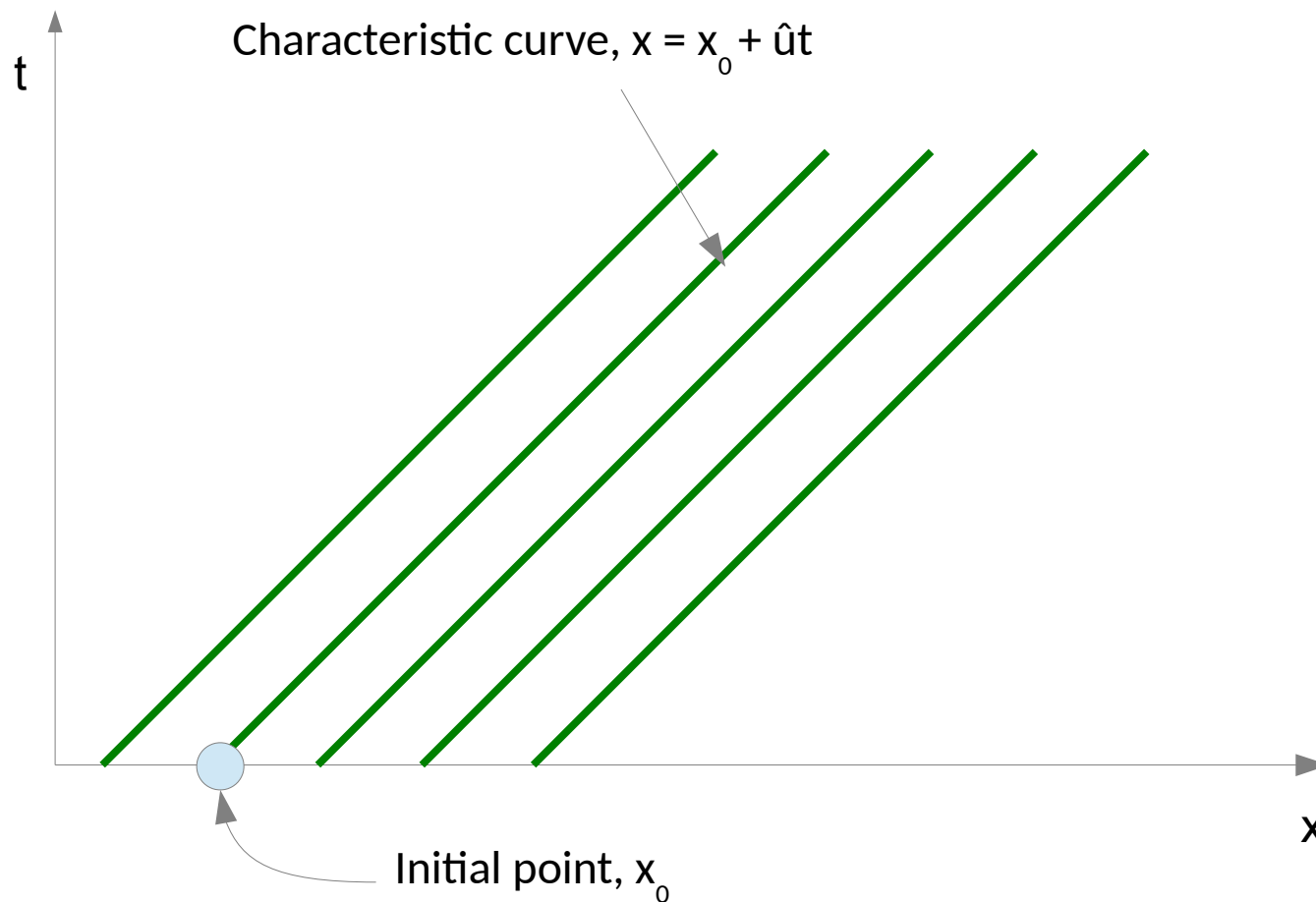
Multi-Dimensional Advection



32^3 gaussian advected with unsplit reconstruction and $C = 0.8$

Burgers' Equation

- Recall that for the advection equation, the solution is unchanged along lines $x - ut = \text{constant}$
 - These were called the characteristic curves



Burgers' Equation

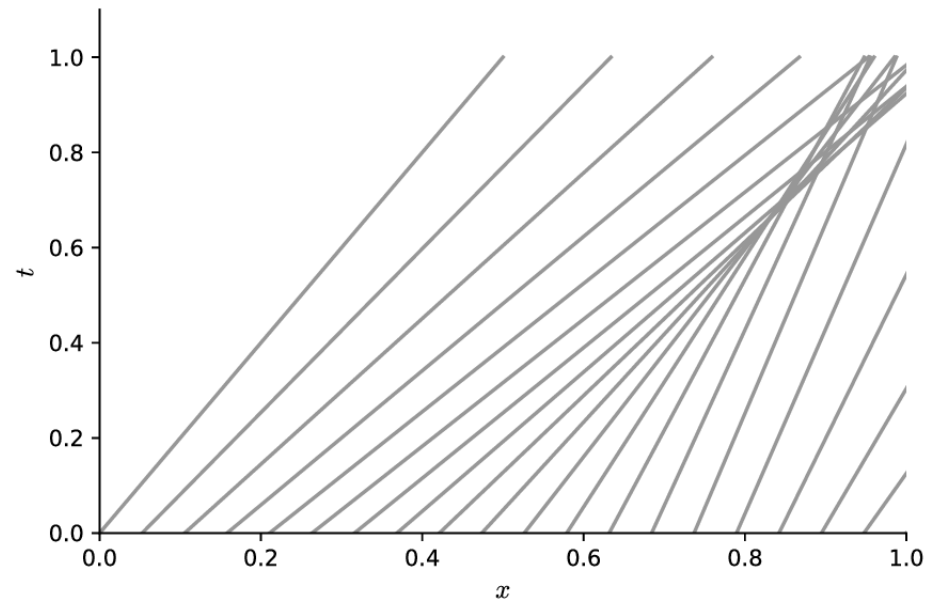
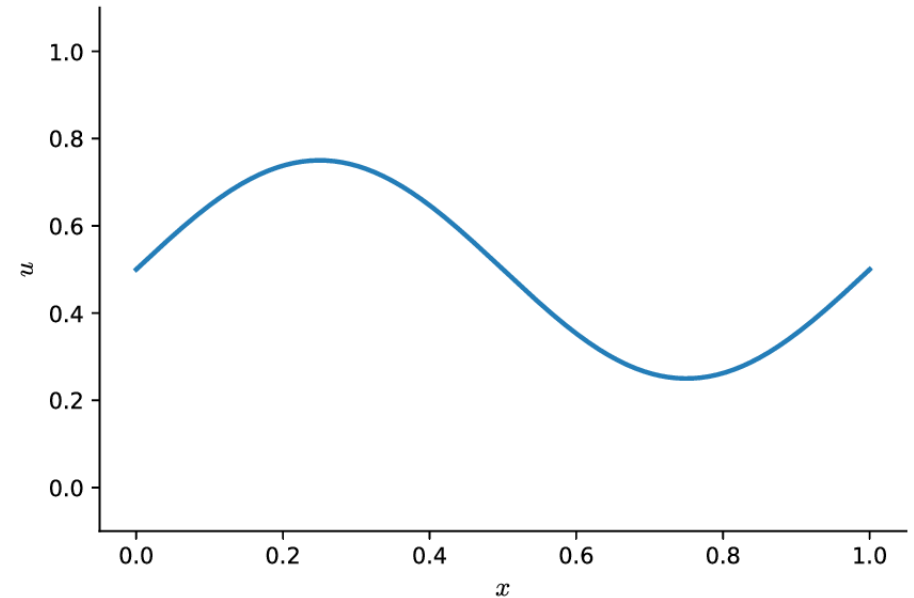
- Consider the following nonlinear hyperbolic PDE:

$$u_t + uu_x = 0$$

- This is the (inviscid) Burgers' equation
 - We can use this as a model of the nonlinear term in the momentum equation of hydrodynamics
 - The nonlinearity admits nonlinear waves like shocks and rarefactions
- Again, the characteristic curves are given as: $dx/dt = u$
 - But now u varies throughout the domain

Shocks

- Shocks form when the characteristics intersect (see Toro Ch. 2 for a nice discussion)
 - Solution is to put a shock at the intersection

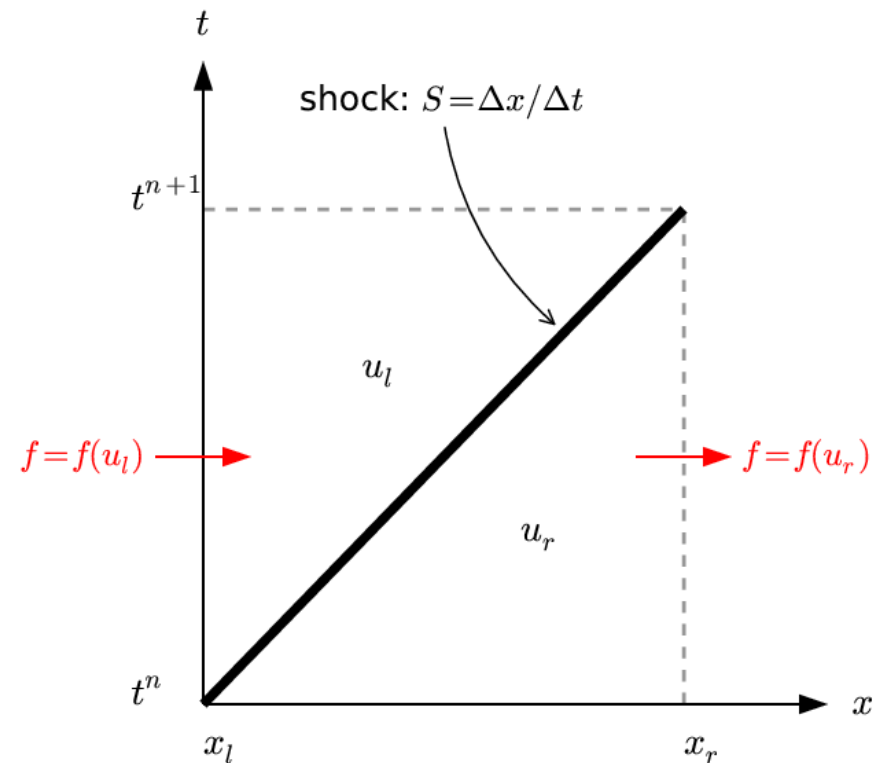


Shock Jump Conditions

(LeVeque, Ch. 11)

- Consider a (right moving) shock over a short time interval Δt
 - We can take the speed to be constant in this short interval
- Apply the integral form of the conservation law:

$$\begin{aligned} \frac{d}{dt} \int_{x_l}^{x_r} u(x, t) dx \\ = f(u(x_l, t)) - f(u(x_r, t)) \end{aligned}$$



Shock Jump Conditions

(LeVeque, Ch. 11)

- Integrating over time, we have:

$$\int_{x_l}^{x_r} [u(x, t^{n+1}) - u(x, t^n)] dx = \int_{t^n}^{t^{n+1}} [f(u)|_{x=x_l} - f(u)|_{x=x_r}] dt$$

- If the intervals are small, then u is roughly constant

$$\Delta x [u_l - u_r] \approx \Delta t [f(u_l) - f(u_r)]$$

- Taking $s \approx \Delta x / \Delta t$

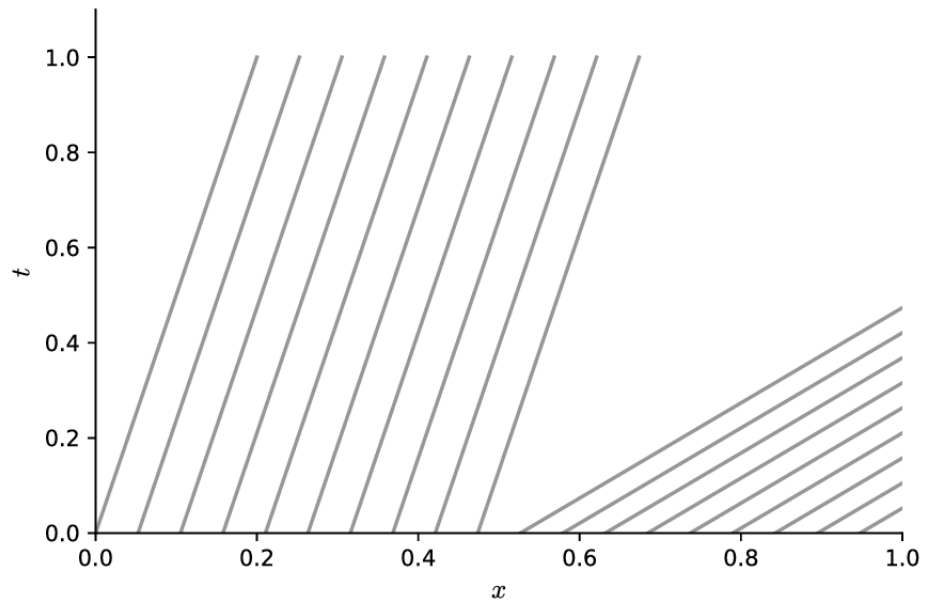
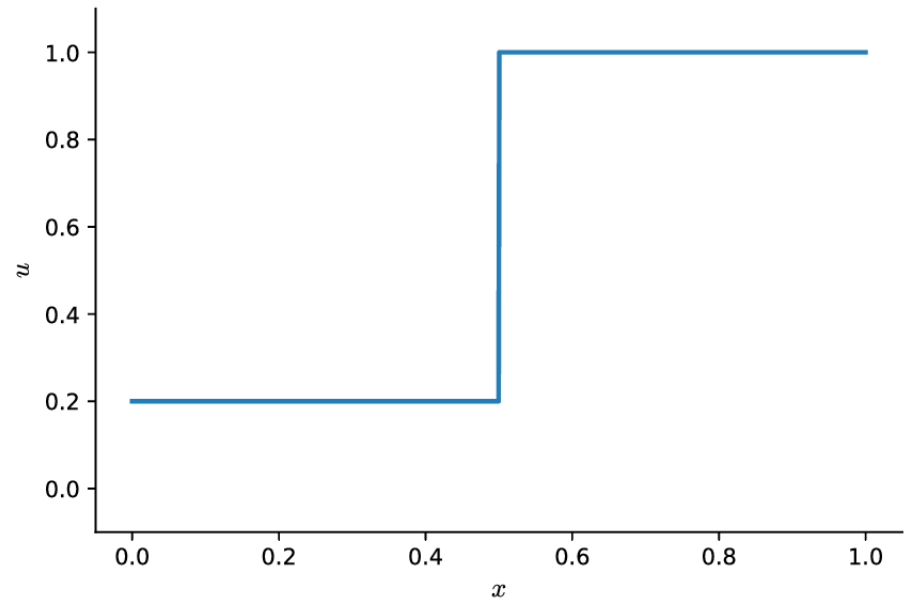
For Burger's Eq.

$$s = \frac{f(u_r) - f(u_l)}{u_r - u_l} = \frac{u_l + u_r}{2}$$

This is called the Rankine-Hugoniot condition

Rarefaction

- Consider the opposite case: characteristics diverge
 - This is a **rarefaction**. There is no compression (or shock)



Burgers' Equation Riemann Problem

- Together, these allow us to write down the Riemann solution for Burger's equation
 - We only need the solution exactly on the interface between zones
 - If $u_l > u_r$ then we are compressing (shock):

$$S = \frac{1}{2}(u_l + u_r)$$

$$u_{i+1/2} = \begin{cases} u_l & S > 0 \\ u_r & S < 0 \end{cases}$$

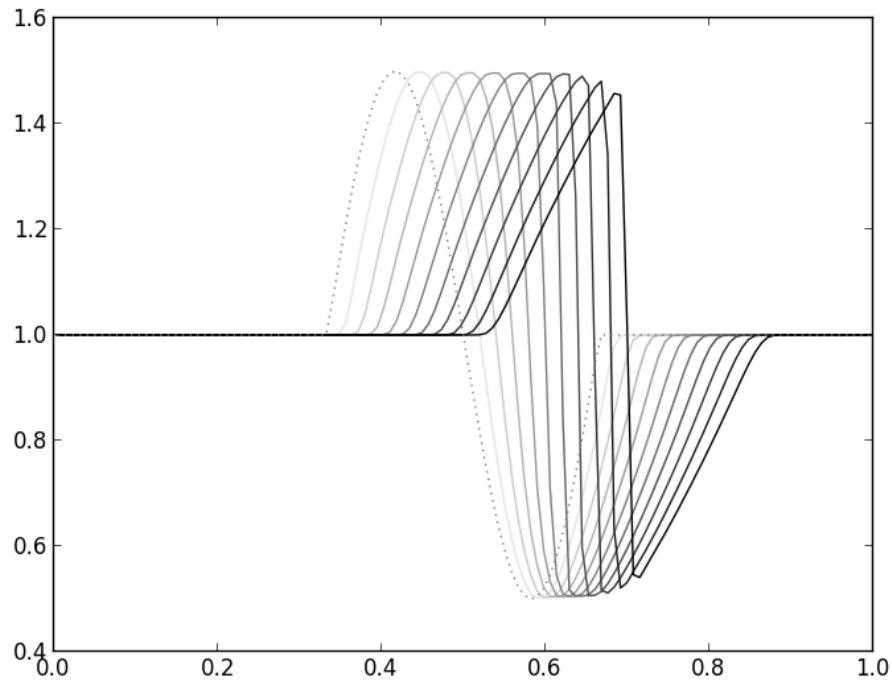
- Otherwise: rarefaction:

$$u_{i+1/2} = \begin{cases} u_l & u_l > 0 \\ u_r & u_r < 0 \\ 0 & \text{otherwise} \end{cases}$$

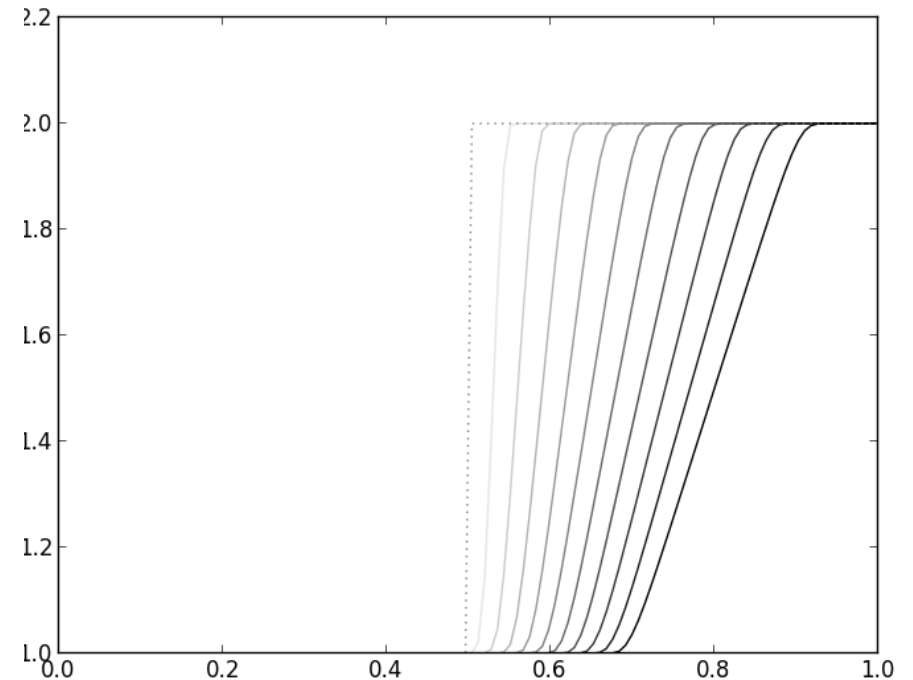
Solving Burgers' Equation

- The solution to Burgers' equation is very similar to the advection equation, with the following changes
 - We now use the Burgers' Riemann problem solution
 - The timestep is estimated by looking at the $\max \text{abs}(u)$ on the grid

Burger's Examples



Sine wave steepening into a shock



Rarefaction from an initial discontinuity

Code: https://github.com/zingale/hydro_examples/blob/master/burgers/burgers.py

Hydrodynamics

- The equations of hydrodynamics share similar ideas
 - Now there are 3 waves (and three equations). We need to consider waves moving in either direction
 - Shocks and rarefactions are still present
 - The system is nonlinear