FYS-GEO4500 Finite volume methods for geophysical fluid dynamics

Galen Gisler, Physics of Geological Processes
University of Oslo

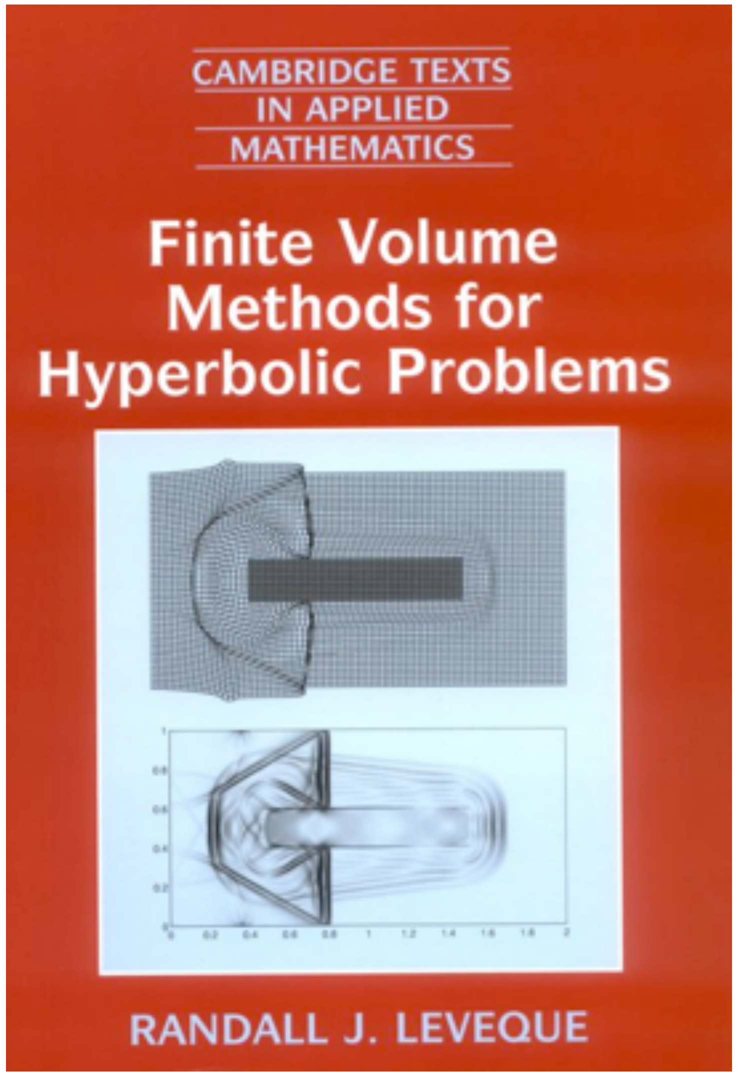
galen.gisler@fys.uio.no
Fysikk byggningen, Rm 411A Vestfløy
+47 99898013

Course Outline

The book:

Finite Volume Methods for Hyperbolic Problems, by Randall J. Leveque, ISBN 0-521-00924-3

should be available at Akademika


CAMBRIDGE TEXTS IN APPLIED MATHEMATICS
Finite Volume Methods for Hyperbolic Problems
RANDALL J. LEVEQUE

The software:

Clawpack, for Conservation *LAW PACK*age, by Leveque and his team at the University of Washington, Seattle

available at www.clawpack.org

What we will do in this course

We will explore hyperbolic equations and finite volume methods for solving them:

What are *hyperbolic equations* and why are they important?

Why are *finite volume methods* well suited to these equations?

How do we solve these equations on the computer?

How can we know whether we have solved the equations correctly?

We will apply our solution methods to problems of geophysical interest:

examples: vents, pockmarks, tsunamis, fluidised systems, explosive volcanism, atmospheric dispersion

Metrics: grades in this course will be based on:

weekly homework assignments	40%
final project	40%
classroom participation	20%

Time and Place

Normally we will meet in this room, V414 Fysikkbygnningen at the following times:

Mondays, 13.15 to 15.00

Tuesdays, 14.15 to 16.00

We’ll go through most lecture materials on Mondays, and on Tuesdays we will take care of spill-overs and address any questions or concerns.

Problem sets will be due on Mondays, and I will occasionally ask a student to illustrate how a problem is done in class the day it is due.

An approximate syllabus - subject to change

	week number	date	Topic	Chapter in LeVeque
1	35	30. aug. 2010	introduction to conservation laws, Clawpack	1 & 2
2	36	6. sep. 2010	the Riemann problem, characteristics	3 & 5
3	37	13. sep. 2010	finite volume methods for linear systems	4
4	38	20. sep. 2010	high resolution methods	6
5	*40*	*4. okt. 2010*	boundary conditions, accuracy, variable coeff.	7,8, part 9
6	*40*	*5. Oct 2010*	nonlinear conservation laws, finite volume methods	11 & 12
7	41	11. okt. 2010	nonlinear equations & systems	13 & 14
8	42	18. okt. 2010	finite volume methods for nonlinear systems	14 & 15
9	43	25. okt. 2010	source terms and multidimensions	16,17,18,19
10	44	1. nov. 2010	multidimensional systems	20 & 21
11	45	8. nov. 2010	capacity functions, source terms, project plans	
12	46	15. nov. 2010	student presentations	
13	47	22. nov. 2010	student presentations	
14	48	6. des. 2010	FINAL REPORTS DUE	

There will be problem sets assigned most weeks and due the following class time. **Keep up with the reading, and do the problems!** Things get complicated quickly, and you will flounder if you don't keep up.

Other chapters in the book may be interesting and important for some of you. Feel free to study these, and if popular will demands, we can cover some of that material in class too.

The **Code Debate**: Build and use, or Procure and use?

Which is better?

Build your own code and use it to solve problems you're interested in?

Procure an existing code and use it to solve problems you're interested in?

If you **build**:

You will know exactly what methods it uses, how it works, and its strengths and limitations.

However, you will spend months and perhaps years debugging it; tweaking its performance; adding features to it; porting it to different systems; and you may never get adequate use from it, especially if you have limited time (as a student, for example!).

The **Code Debate**: Build and use, or Procure and use?

Which is better?

Build your own code and use it to solve problems you're interested in?

Procure an existing code and use it to solve problems you're interested in?

If you **procure**:

Once you install it and prepare your input files, you will be able to start solving those problems immediately.

However, you won't necessarily know the strengths and weaknesses of its methods; you run the risk of generating mountains of meaningless output by attempting to run it on problems for which it is ill-suited; and you may not be able to defend your results.

The best (**infinite time**) solution

Spend N years developing the **world's best code** for the problems you're interested in. Debug, tweak, and add features to your heart's content.

Then freeze it.

Spend the next M years running your code on those problems, and publish lots of papers.

By this time, numerical techniques have advanced far beyond those you used in your code; the computers you wrote your code for are obsolete; you haven't kept up with changes in the compilers and operating systems; etc.

Bottom line: it's not the **world's best code** any more, and you have new problems you want to solve.

You could do it all again…

OR…

Stand on the shoulders of giants

A better solution is to find a flexible and extensible code **framework** that:

Applies the latest & best techniques to a broad class of problems
But **requires you** to write the piece specific to the problem you're solving

Has a broad and experienced user base;
and a strong development team for porting, updating, and debugging

Allows you to modify the code yourself, if you need something different or find a better technique. This condition rules out most (all?) commercial codes.

If you do this, you are **obligated** to yourself and to your science to study the framework and techniques carefully so that you clearly **understand** and **can defend** the results you get!

That, in short, is the aim of this course.

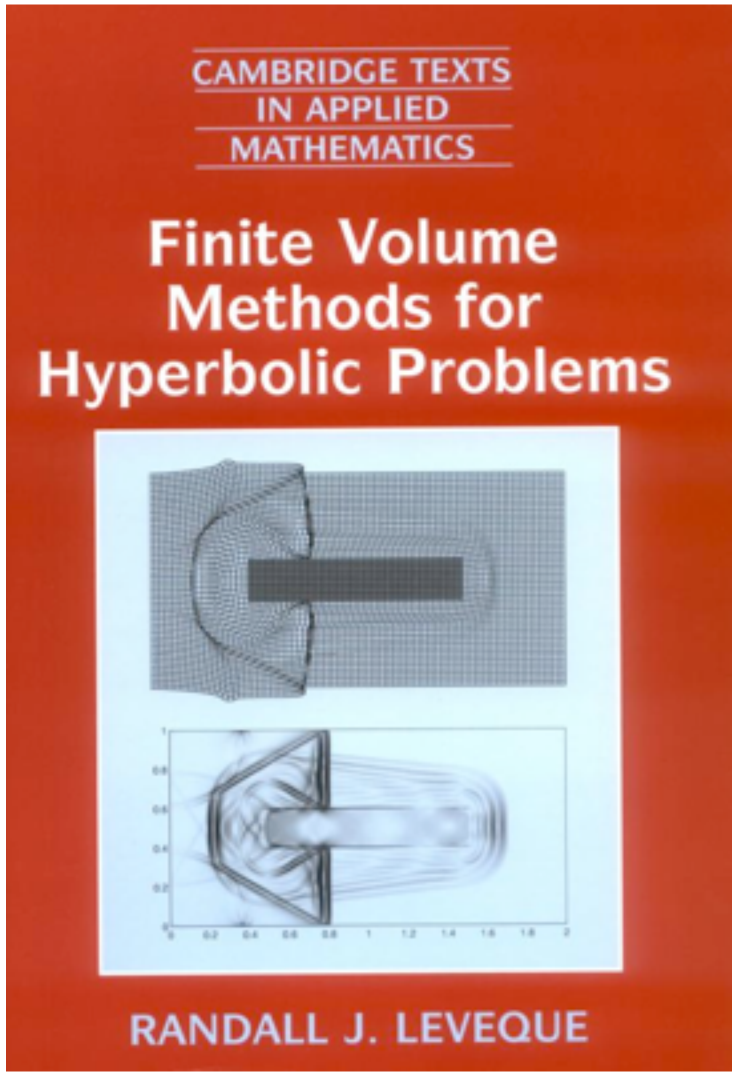
A few codes are documented with textbooks

Clawpack

www.clawpack.org

This isn't the only one, but it's the one we will use.

The techniques you learn using Clawpack and building applications with it will also help you use and build other codes.



CAMBRIDGE TEXTS IN APPLIED MATHEMATICS

Finite Volume Methods for Hyperbolic Problems

RANDALL J. LEVEQUE

ISBN 0-521-00924-3

What you’ll need to use Clawpack

A Unix/Linux/MacOS computer

A Fortran compiler (pref. gfortran, free, from http://gcc.gnu.org)

A Python interpreter (free, from www.python.org)

The SciPy extension for Python, which includes NumPy and Matplotlib
(free, from www.scipy.org or from www.macinscience.org (for Mac))
This gives you, for free, most of the capability of MatLab, which is expensive!

Clawpack itself (free, from www.clawpack.org, but requires registration)

Curiosity, motivation, patience, and cleverness.

What are *hyperbolic problems* anyway?

There are three main types of *second-order* partial differential equations. General prototypes of these are:

the *wave equation*:	$q_{tt} - \gamma^2 q_{xx} - f(x) = 0$	"hyperbolic"
the *heat equation*:	$q_t - \alpha^2 q_{xx} - f(x) = 0$	"parabolic"
and *Poisson's equation*:	$q_{xx} - f(x) = 0$	"elliptic"

Why the geometric terminology?

With suitable variable changes, all of these can be squeezed into the form:

$$a\Phi_{\chi\chi} + b\Phi_{\chi\zeta} + c\Phi_{\zeta\zeta} + f(\Phi, \Phi_\chi, \Phi_\zeta, \chi, \zeta) = 0$$

Reminiscent of the equation for conic sections:

$$ax^2 + bxy + cy^2 + d = 0$$

Conic Sections and Partial Differential Equations

The general equation for conic sections $ax^2 + bxy + cy^2 + d = 0$

has the discriminant $b^2 - 4ac$.

$$\text{If } \begin{cases} b^2 - 4ac > 0 & \text{the equation yields a hyperbola,} \\ b^2 - 4ac = 0 & \text{the equation yields a parabola,} \\ b^2 - 4ac < 0 & \text{the equation yields an ellipse.} \end{cases}$$

The general second-order partial differential equation in two variables:

$$a\Phi_{\chi\chi} + b\Phi_{\chi\zeta} + c\Phi_{\zeta\zeta} + f(\Phi, \Phi_\chi, \Phi_\zeta, \chi, \zeta) = 0$$

is easily transformed to the hyperbolic equation $q_{tt} - \gamma^2 q_{xx} - f(x) = 0$

in the case $b^2 - 4ac > 0$.

The other two cases are more difficult to demonstrate.

Examples of Partial Differential Equations in one spatial dimension

elliptic equations

$$\text{Poisson's equation: } q_{xx} - f(x) = 0$$

conditions at boundaries determine the solution everywhere and simultaneously, no time dependence

parabolic equations

$$\text{heat equation: } q_t - \alpha^2 q_{xx} - f(x) = 0$$

inhomogeneities diffuse away irreversibly, leading to a steady state

hyperbolic equations

$$\text{wave equation: } q_{tt} - \gamma^2 q_{xx} - f(x) = 0$$

all physics is local and dynamic; waves that are generated propagate away from the source

Features of the three types of PDEs

	hyperbolic	parabolic	elliptic
example	$q_{tt} - \gamma^2 q_{xx} = 0$	$q_t - \alpha^2 q_{xx} = 0$	$q_{xx} + q_{yy} = 0$
eigenvalues	real	zero	complex
nature of solutions	wave-like, energy-conserving	damping, diffusion, irreversibility	steady-state, no waves
Types of boundary conditions	Cauchy (initial value problem)	Cauchy plus Neumann or Dirichlet	Neumann or Dirichlet (edges)

Features of the three types of PDEs

	hyperbolic	parabolic	elliptic
example	$q_{tt} - \gamma^2 q_{xx} = 0$	$q_t - \alpha^2 q_{xx} = 0$	$q_{xx} + q_{yy} = 0$
eigenvalues	real	zero	complex
nature of solutions	wave-like, energy-conserving	damping, diffusion, irreversibility	steady-state, no waves
Types of boundary conditions	Cauchy (initial value problem)	Cauchy plus Neumann or Dirichlet	Neumann or Dirichlet (edges)

Features of the three types of PDEs

This course

	hyperbolic	parabolic	elliptic
example	$q_{tt} - \gamma^2 q_{xx} = 0$	$q_t - \alpha^2 q_{xx} = 0$	$q_{xx} + q_{yy} = 0$
eigenvalues	real	zero	complex
nature of solutions	wave-like, energy-conserving	damping, diffusion, irreversibility	steady-state, no waves
Types of boundary conditions	Cauchy (initial value problem)	Cauchy plus Neumann or Dirichlet	Neumann or Dirichlet (edges)

Systems of first-order equations

Although the classification of PDEs into hyperbolic, parabolic, and elliptic was designed around the appearance of second-order equations, we apply it to systems of first-order equations.

For example, the linearised shallow-water (or tsunami) equations:

$$h_t(x,t) + Dv_x(x,t) = 0$$
$$Dv_t(x,t) + gDh_x(x,t) = 0$$

Here h is the wave height, v is the particle speed, D is the ocean depth, and g is the acceleration due to gravity. We'll use different symbols when we derive this later.

are a hyperbolic set, as can be seen from the derived wave equation:

$$h_{tt} - gDh_{xx} = 0$$

The methods we speak of in this course are aimed at the solution of systems of first-order equations like these.

A little on notation, following Leveque

Use subscript notation to refer to partial derivatives:

$$q_x \equiv \frac{\partial q}{\partial x}; \;\; q_y \equiv \frac{\partial q}{\partial y}; \;\; q_z \equiv \frac{\partial q}{\partial z}; \;\; q_t \equiv \frac{\partial q}{\partial t};$$

x, y, and z usually refer to the Cartesian coordinates; t to time.

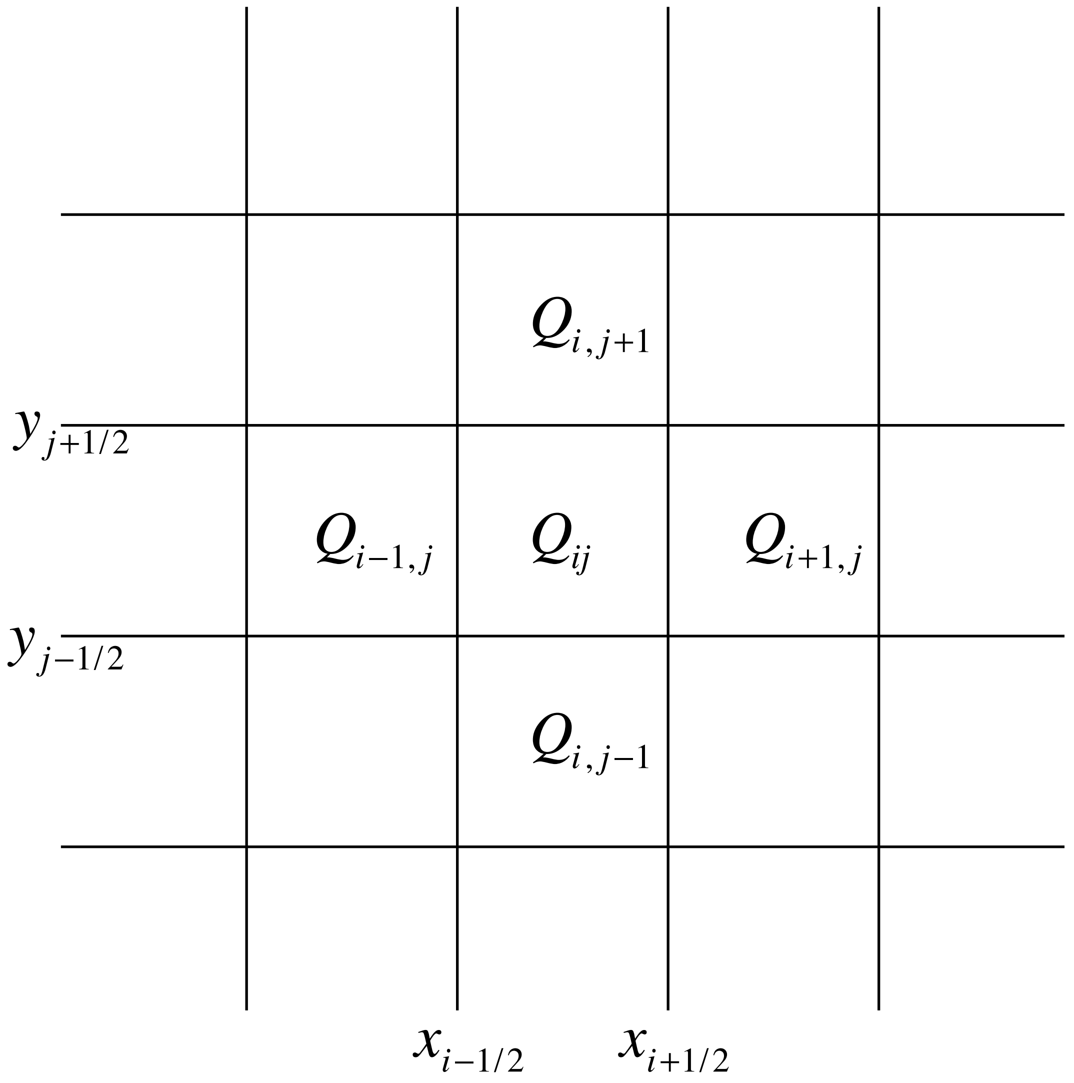
q is some quantity of interest whose value we need to know: the true solution to the partial differential equation under study. In general, q represents a *vector* of quantities, the components of which are denoted by superscripts as q^p. An $m \times m$ system of equations has eigenvectors r^p and eigenvalues λ^p.

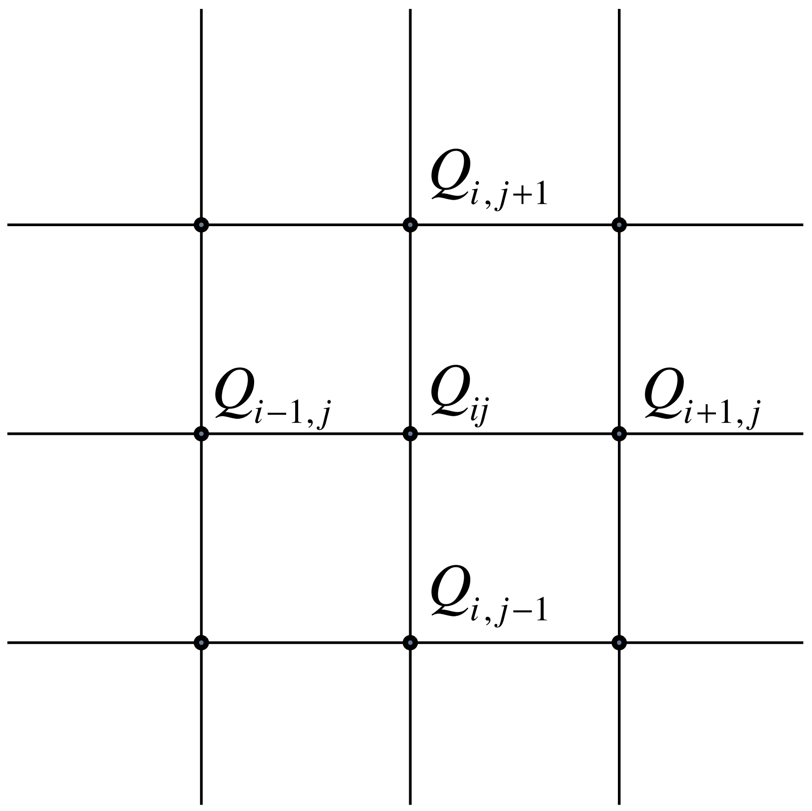
Q_i^n is the numerical approximation to q in the i th grid cell at the nth time step, and the time at the nth time step is denoted t_n.

$F_{i+1/2}^n$ is the numerical approximation to the *flux* of quantity q from cell i to cell $i+1$ at the nth time step.

For two dimensions, we use the additional cell-index subscript j, and the additional flux approximation $G_{j+1/2}^n$.

Finite Volume *vs* Finite Difference





Approximate values are *averaged* over cells:

$$Q_{ij}^{n} \approx \frac{1}{\Delta x \Delta y} \int_{y_{j-1/2}}^{y_{j+1/2}} \int_{x_{i-1/2}}^{x_{i+1/2}} q(x,y,t_n)\,dx\,dy$$

Approximate values are *evaluated* on a grid:

$$Q_{ij}^{n} \approx q(x_i, y_j, t_n)$$

Finite Difference, Finite Volume, and Finite Element Methods

Finite difference methods *sample* the solution to form the approximation.

Finite volume methods *average* the solution to form the approximation.

The approximation is the *primary representation*, the evolution uses piecewise-polynomial mappings.

Finite element methods use piecewise-polynomial mappings as the primary representation, and evolve them directly.

They are advantageous for complex geometries, but have difficulty in dealing with evolving discontinuities like shocks and mixing.

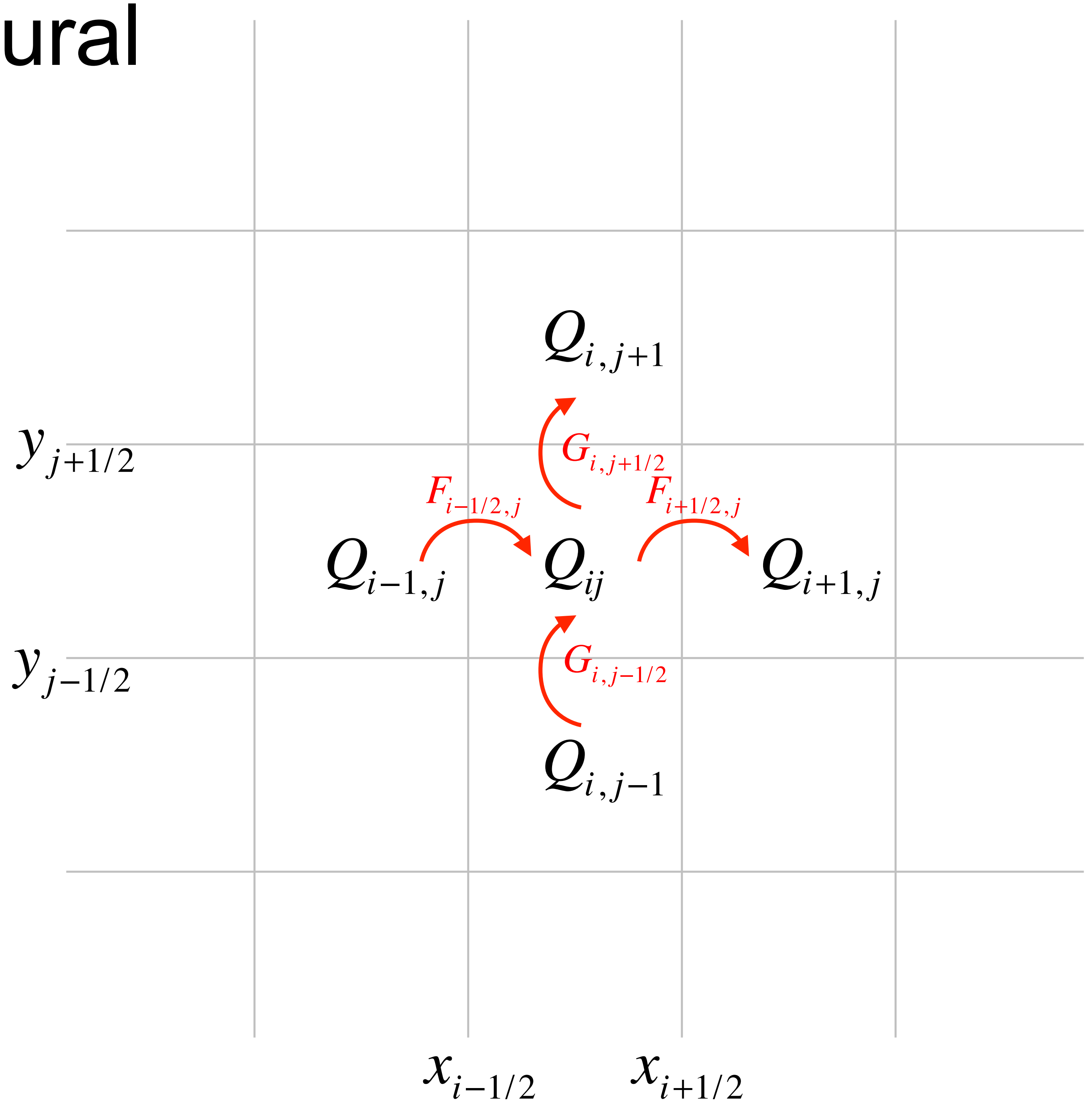
They are the subject of FYS-GEO4510, given by Marcin Dabrowski in the spring.

For conservation laws, finite volume methods are natural

If the quantity of interest is a *conserved* quantity, we update it by keeping track of the *fluxes* into (and out of) each cell

We can also allow for sources and sinks within the cells

We will use something like this to update Q for the next time step:



$$Q_{ij}^{n+1} \approx Q_{ij}^{n} - \frac{\Delta t}{\Delta x}\left(F_{i+1/2,j}^{n} - F_{i-1/2,j}^{n}\right) - \frac{\Delta t}{\Delta y}\left(G_{i,j+1/2}^{n} - G_{i,j-1/2}^{n}\right)$$

Advantages and disadvantages of finite volume methods

Finite volume methods: high resolution, but low order (2nd at best, in general)

Best for cases with discontinuities (sharp material interfaces, shocks)

Best for any kind of waves: acoustic, seismic, water, electromagnetic, etc.

Good for highly compressible media

Good for high-speed flows

In general, best for hyperbolic problems

But…

Not good for slow, viscosity-dominated processes

Poor for parabolic problems (diffusion-dominated processes)

Difficult for elliptic problems (LaPlace or Poisson type equations)

For these latter cases, linear methods, high-order finite difference methods, or finite element methods are better suited.

Introduction to Conservation Laws (Chapters 1 & 2 in Leveque)

Numerical solution of differential equations

The equation $\frac{\partial q}{\partial t} = -A\frac{\partial q}{\partial x}$ has a discrete analogue $\frac{\Delta q}{\Delta t} = -A\frac{\Delta q}{\Delta x}$

which can in principle be solved by time stepping: $q_i^{n+1} = q_i^n - A\left(\frac{\Delta q}{\Delta x}\right)_i \Delta t$

The difficulty is in suitably defining the numerical approximation of the spatial derivative, and ensuring that the solution is stable and accurate. A naive implementation of the above is highly unstable!

If the differential equation can be expressed as a *conservation law*, the spatial derivative can be interpreted as a *flux* of the conserved quantity.

Straightforward analysis then produces robust numerical techniques in which it is easier to guarantee stability and accuracy.

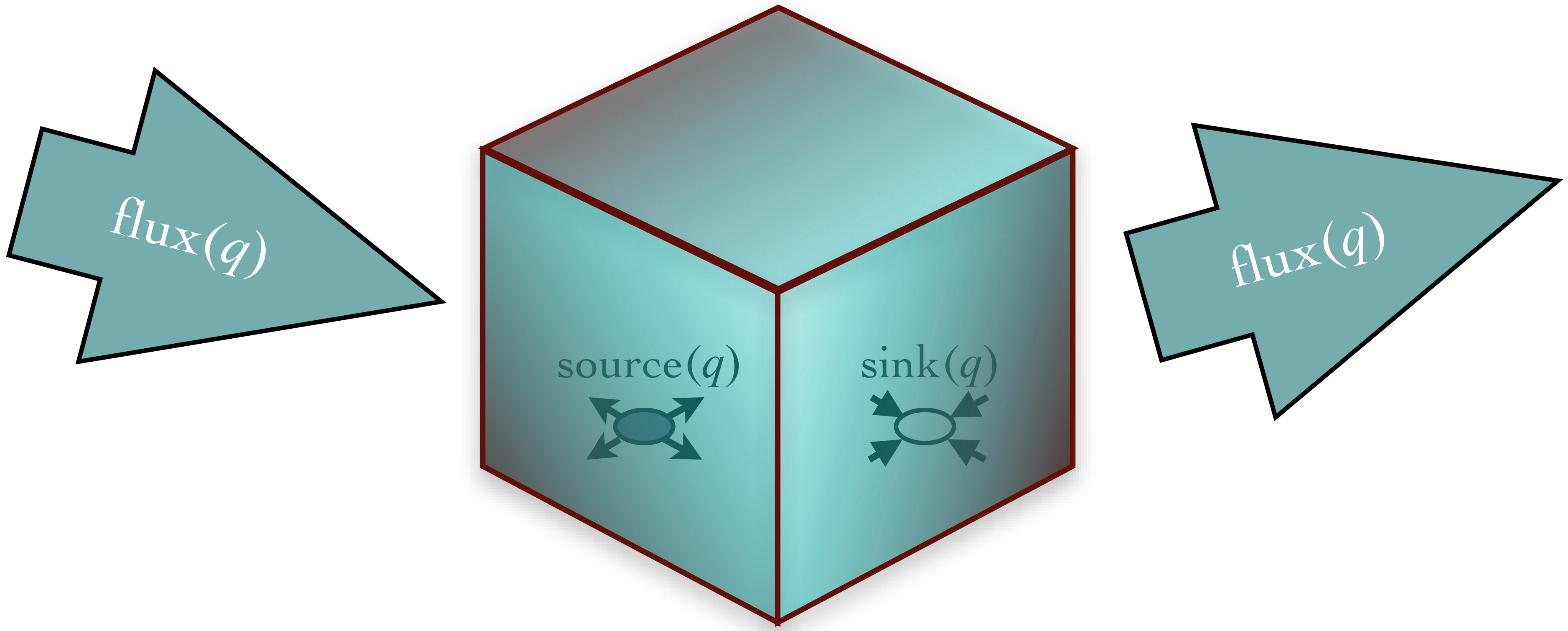
Conservation laws

Many of the fundamental physical laws are conservation laws:

conservation of mass, energy, momentum, entropy (sometimes)…

For any vector of conserved quantities q:

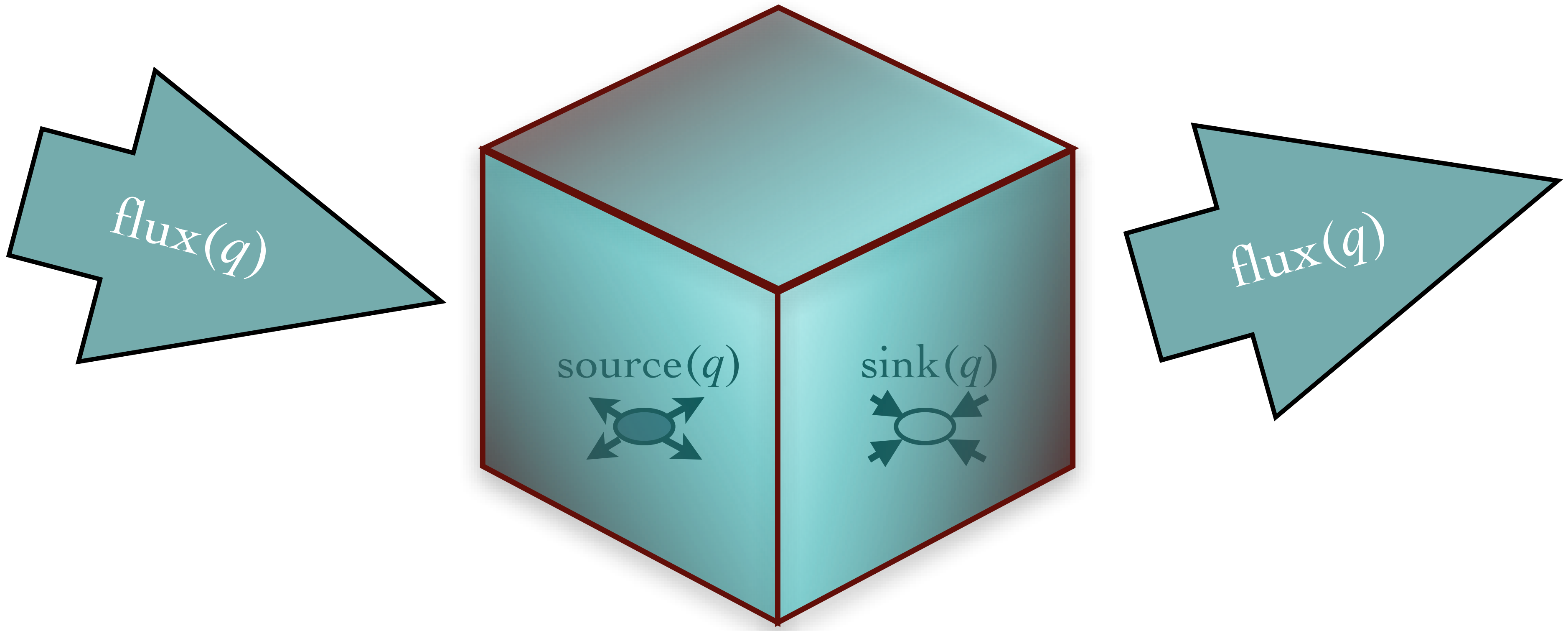
The change with time of q in a volume is due to the net flux of q into or out of the volume and the net amount of q created or destroyed within the volume:



General integral form for a three-dimensional conservation law

$$\frac{d}{dt}\iiint_{Volume} q(x,y,z;t)\,dV = -\oiint_{Surface} \vec{F}(q)\,dS - \iiint_{Volume} R(q)\,dV$$

$R(q)$ represents all sources and sinks for the quantity q in the volume V, and $\vec{F}(q)$ represents the net flux into the volume through its surface.



We start with a one-dimensional system

The integral form of the general one-dimensional conservation law over an interval (x_L, x_U), ignoring sources and sinks, is:

$$\frac{d}{dt}\int_{x_L}^{x_U} q(x,t)\,dx = f(q(x_L,t)) - f(q(x_U,t))$$

and the corresponding differential equation is:

$$q_t(x,t) + f(q(x,t))_x = 0$$

The integral form is more general and more fundamental. Finite volume methods are designed to solve the integral form.

The differential form is more compact, but is not valid at discontinuities (shocks or contact surfaces, for example).

We write the equations in differential form for convenience, and for constructing the matrix representation of a system of partial differential equations.

The finite volume method solves the integral form of the equation.

The world has three space dimensions!

True!

But an advantage of *hyperbolic* systems is that it is straightforward to extend the solution method to two, three, and even more dimensions.

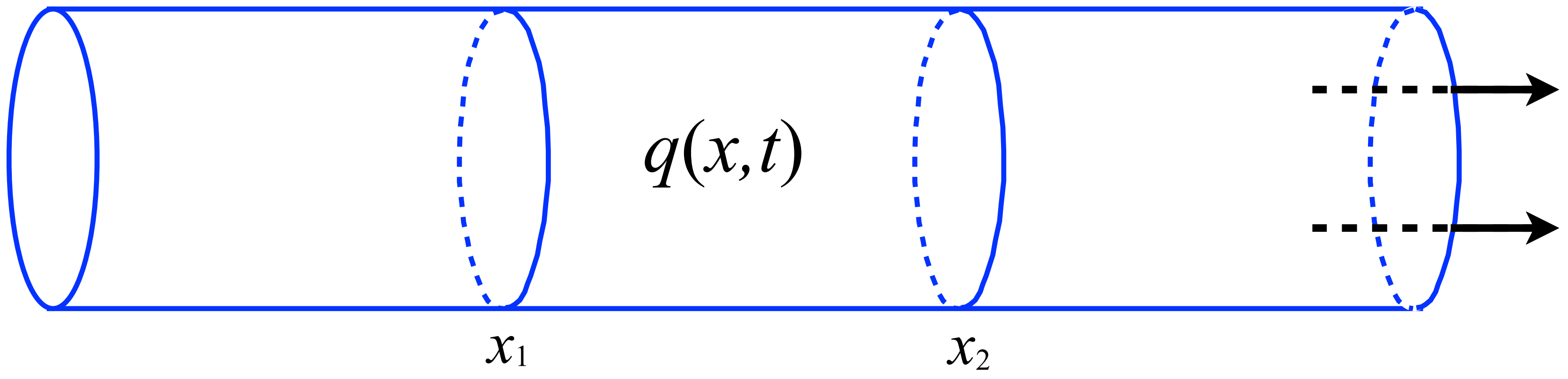
The solutions are wavelike, and waves interact with each other in predictable ways. Added complications include corner transport and boundary conditions.

In the book, two dimensional systems are covered starting in Chapter 18. We'll get there in October.

But first we have a lot of work to do to understand the one-dimensional solution.

Derivation of the conservation law

Assume we have a fluid of density $q(x,t)$ flowing through a pipe.



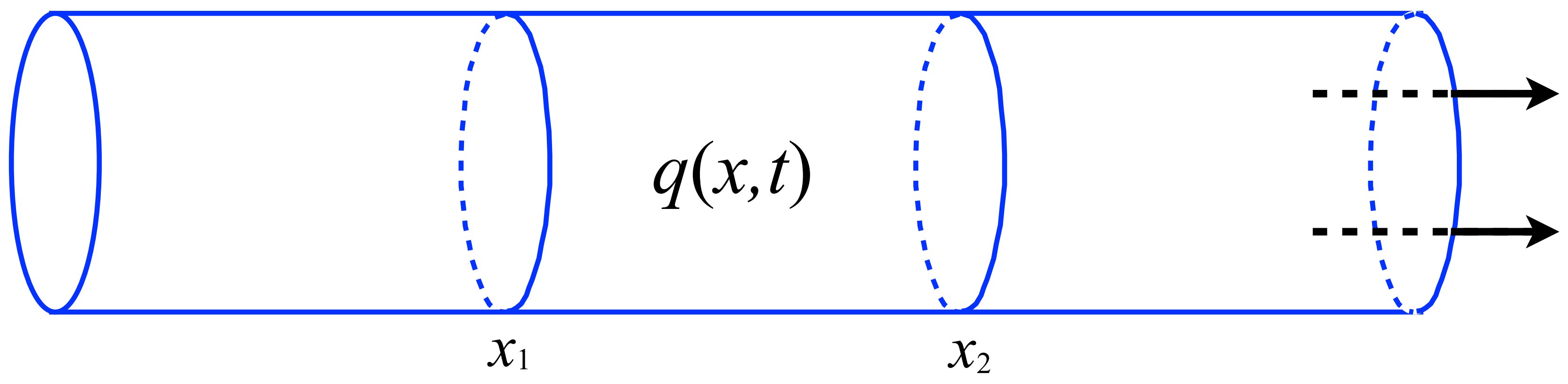
The total mass of the fluid between the positions x_1 and x_2 is:

$$\int_{x_1}^{x_2} q(x,t)\,dx$$

$q(x,t)$ is measured in units of mass per unit length.

This mass changes only because of the flux of the fluid through the left or right ends of the interval.

Conservation law, continued



This mass in the interval changes only because of the flux of the fluid through the left or right ends of the interval:

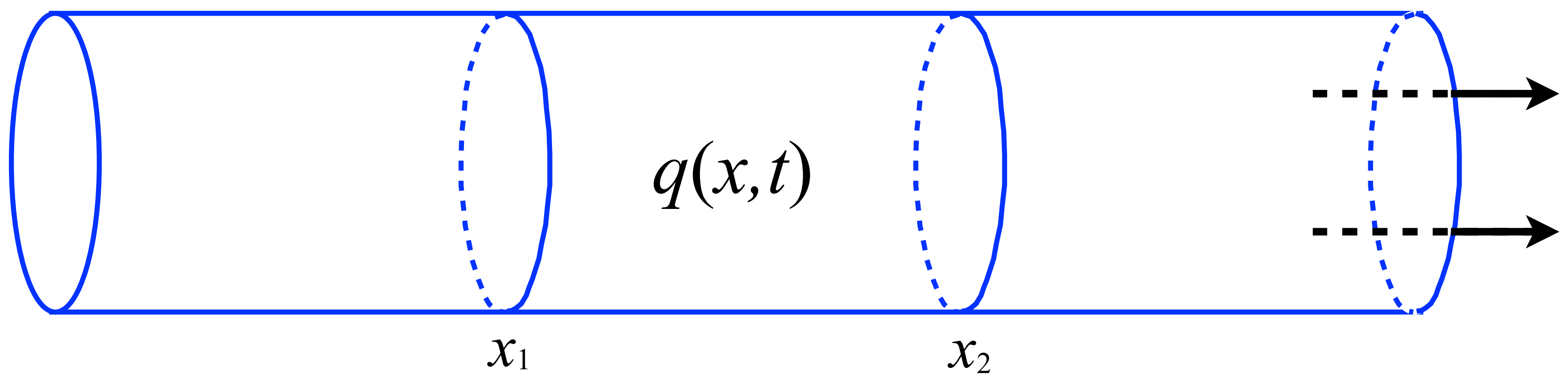
$$\frac{d}{dt}\int_{x_1}^{x_2} q(x,t)\,dx = f(q(x_1,t)) - f(q(x_2,t))$$

where $f(q)$ is the flux function. For a fluid of density q flowing at a velocity $u(x,t)$, the flux function is

$$f(q(x,t)) = u(x,t)q(x,t)$$

This is the **integral** form of the one-dimensional conservation law.

Differential form of conservation law



If q is **smooth** enough, we can rewrite:

$$\frac{d}{dt}\int_{x_1}^{x_2} q(x,t)\,dx = f(q(x_1,t)) - f(q(x_2,t))$$

as $\int_{x_1}^{x_2} q_t\,dx = -\int_{x_1}^{x_2} f(q)_x\,dx$ or $\int_{x_1}^{x_2}\left(q_t + f(q)_x\right)dx = 0$

since this must be true for all x_1 and x_2, then: $q_t + f(q)_x = 0$

This is the **differential** form of the one-dimensional conservation law.

So we have two representations

The integral form of the general one-dimensional conservation law, *valid everywhere*:

$$\frac{d}{dt}\int_{x_L}^{x_U} q(x,t)\,dx = f(q(x_L,t)) - f(q(x_U,t)),$$

and the differential form, valid where the function is *smooth*:

$$q_t(x,t) + f(q(x,t))_x = 0.$$

Because the differential form is not valid at discontinuities, it must be supplemented there by the Rankine-Hugoniot *jump conditions:*

$$f_r - f_l = s(q_r - q_l),$$

where the subscripts r and l refer to right and left states of the solution and the flux function, and s is the speed with which the discontinuity moves. We will derive this later.

Differential vs. Integral forms

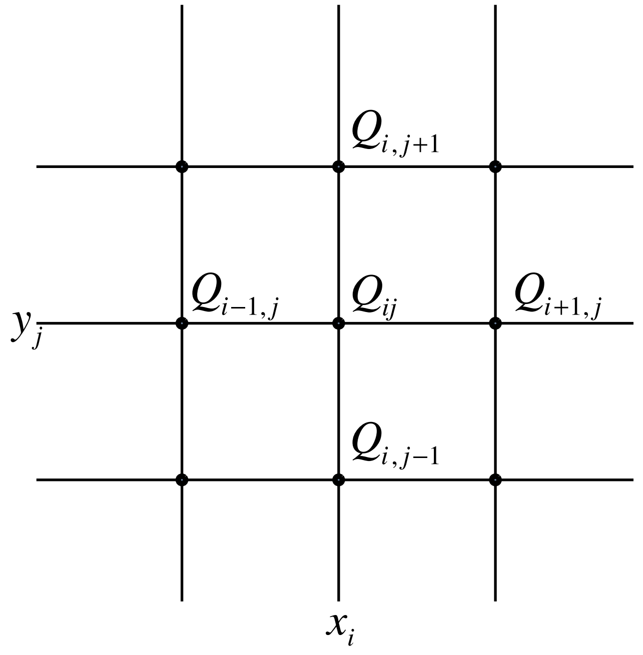
With **finite difference** methods:

approximate pointwise values, i.e. $Q_i^n \approx q(x_i, t_n)$

derivatives are approximated by differences

use **differential form** of conservation law, assume smoothness

$$q_t + f(q)_x = 0$$

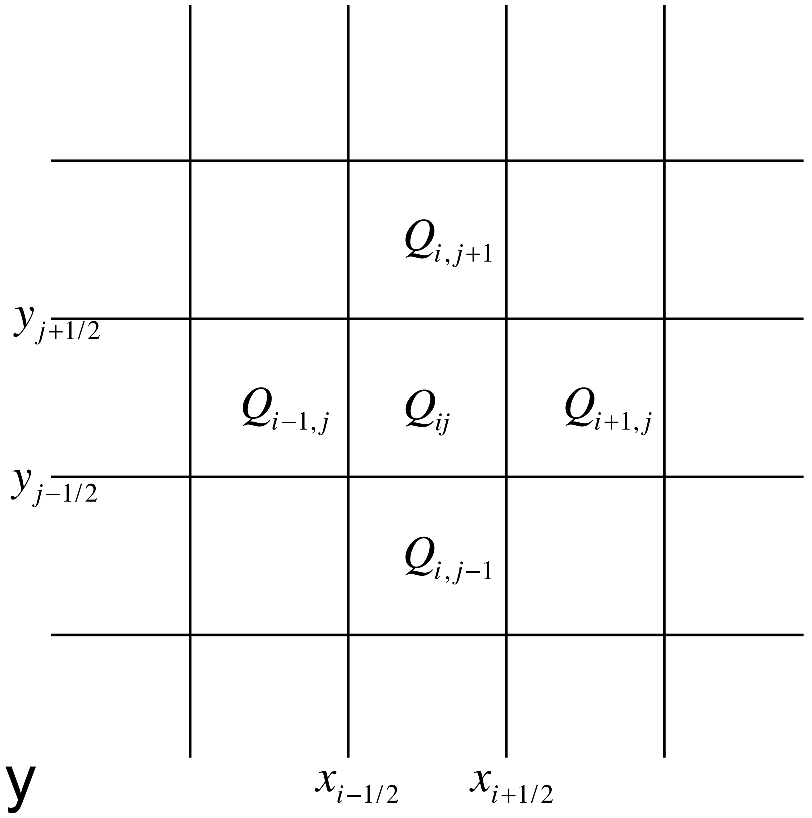


With **finite volume** methods:

approximate cell averages, i.e. $Q_i^n \approx \frac{1}{\Delta x}\int_{x_{i-1/2}}^{x_{1+1/2}} q(x, t_n) dx$

use **integral form** of the conservation law

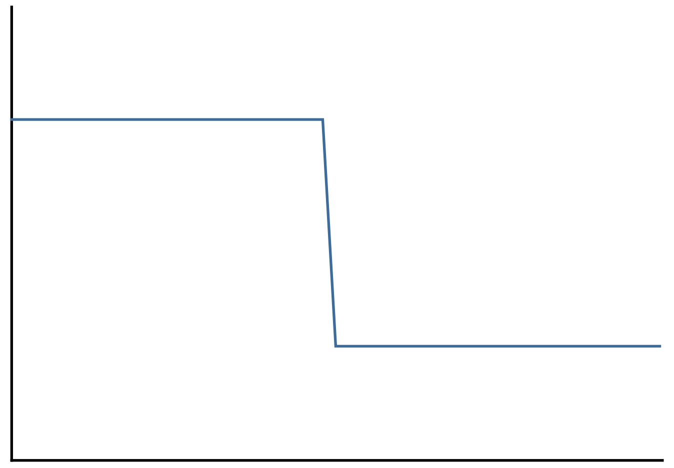
$$\frac{d}{dt}\int_{x_{i-1/2}}^{x_{i+1/2}} q(x,t)\,dx = f(q(x_{i-1/2},t)) - f(q(x_{i+1/2},t))$$



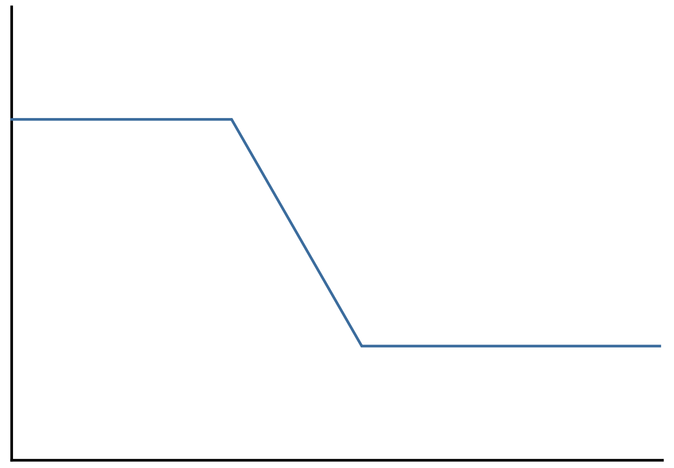
This formulation does not assume smoothness, and leads directly to the numerical method.

Discontinuities occur in real problems

Discontinuities in function values:
shocks, contacts, edges, etc.



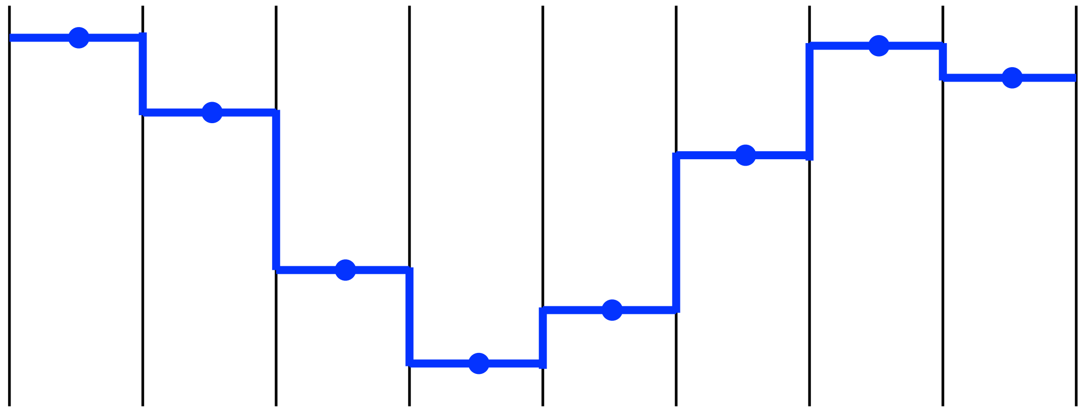
or discontinuities in function derivatives:
rarefactions, reaction fronts, etc.



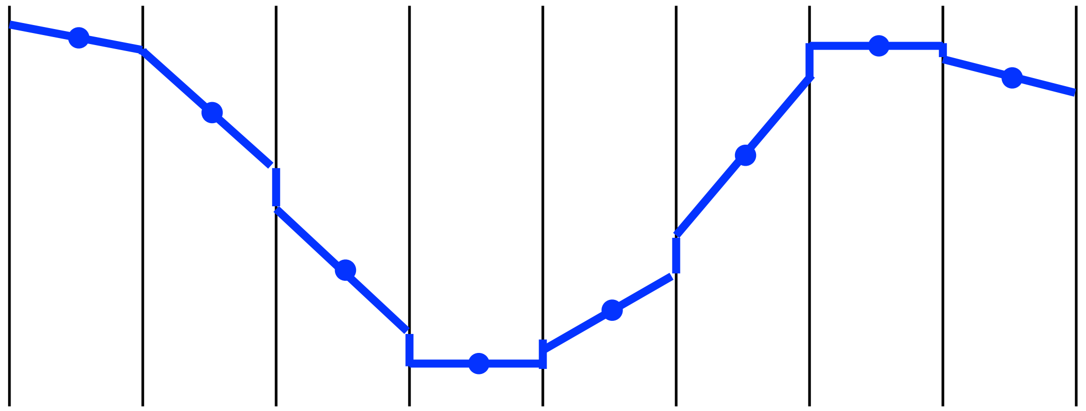
In finite volume methods *Riemann problems* are solved to calculate the propagation of waves away from discontinuities.

These waves are the *characteristics*, and their speeds are the *eigenvalues* of the system of differential equations.

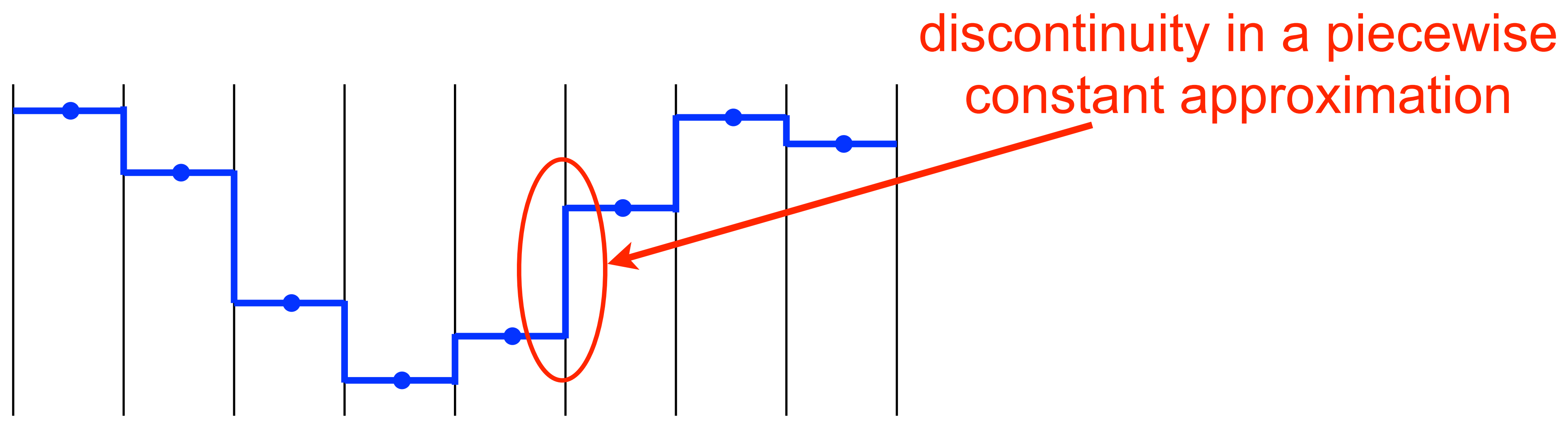
Artificial discontinuities occur in the numerical solution of equations



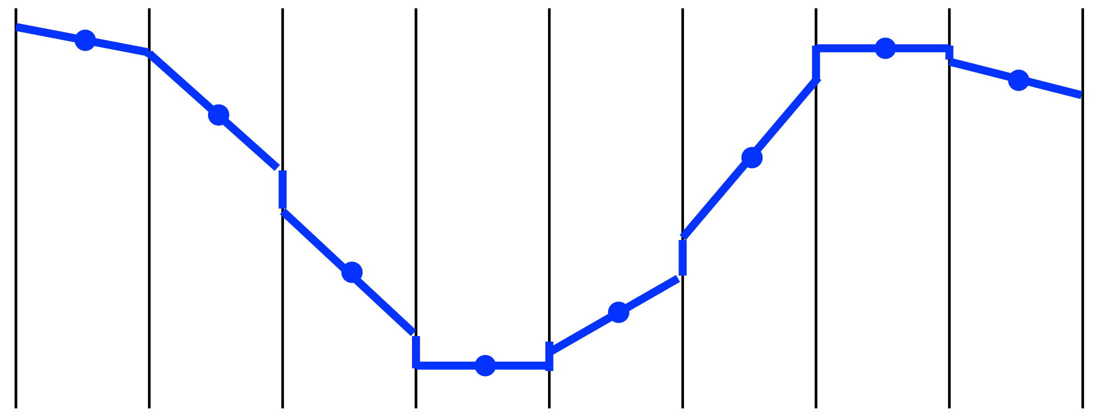
Discretising a problem for numerical solution involves the creation of many small discontinuities; therefore understanding the *characteristics* and solving *Riemann problems* is essential to accurate numerical work.



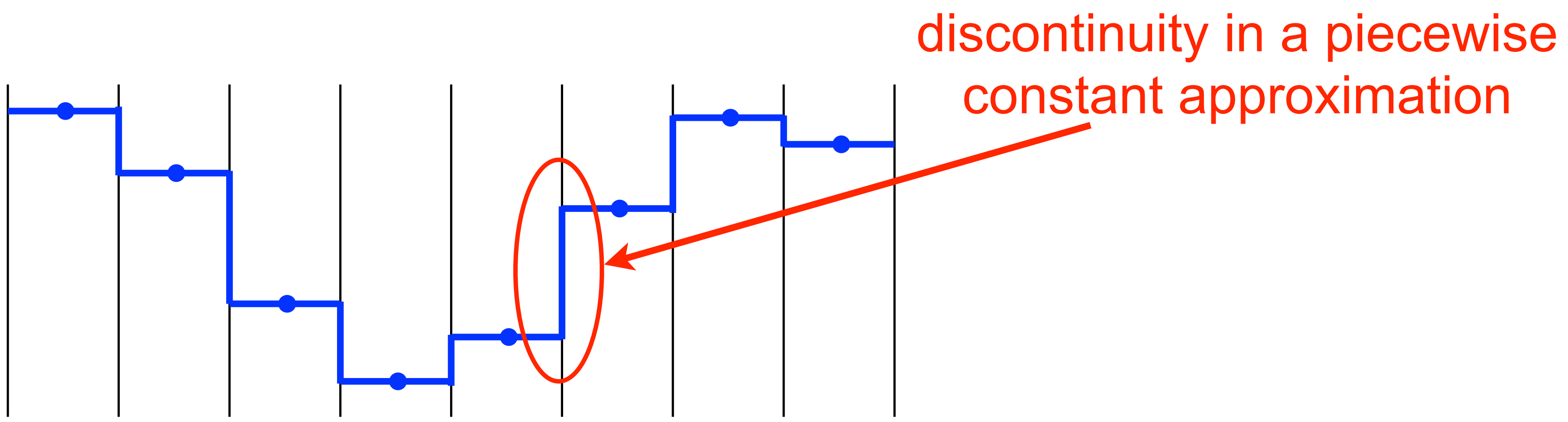
Artificial discontinuities occur in the numerical solution of equations



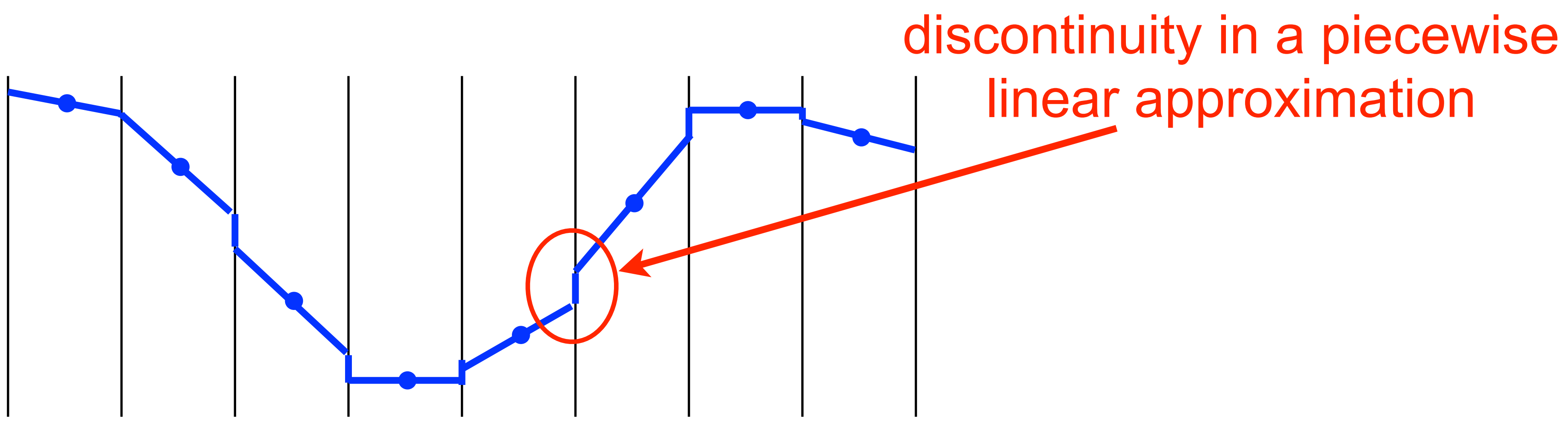
Discretising a problem for numerical solution involves the creation of many small discontinuities; therefore understanding the *characteristics* and solving *Riemann problems* is essential to accurate numerical work.



Artificial discontinuities occur in the numerical solution of equations



Discretising a problem for numerical solution involves the creation of many small discontinuities; therefore understanding the *characteristics* and solving *Riemann problems* is essential to accurate numerical work.

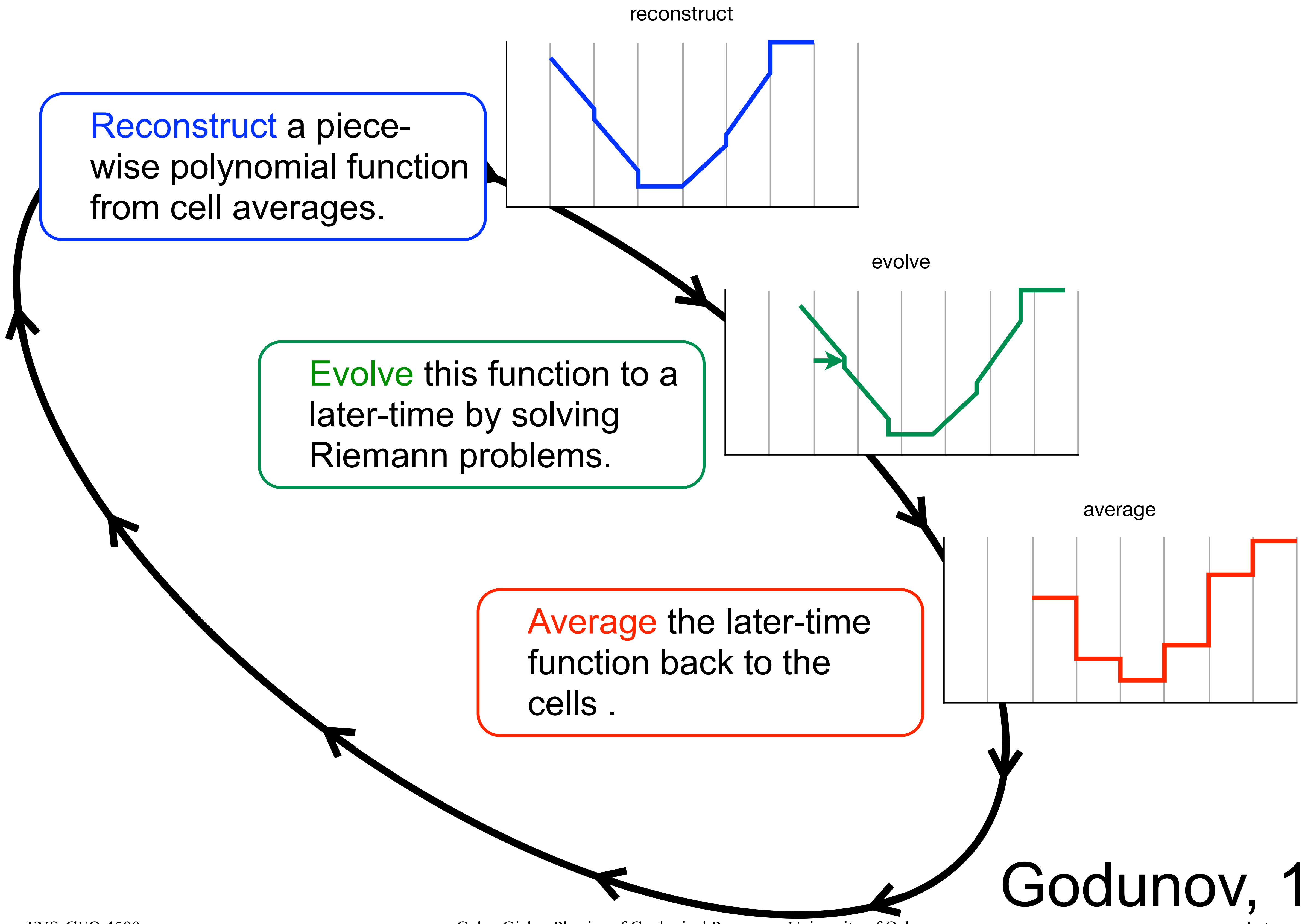


Reconstruct - Evolve - Average -(REA)

The scheme we use:

Godunov, 1959

Reconstruct - Evolve - Average -(REA)



Reconstruct a piece-wise polynomial function from cell averages.

Evolve this function to a later-time by solving Riemann problems.

Average the later-time function back to the cells .

Godunov, 1959

The advection equation — the simplest hyperbolic differential equation

With $q(x,t)$ a tracer concentration, and flow velocity

$$u(x,t) = u = \text{constant},$$

The flux function is:

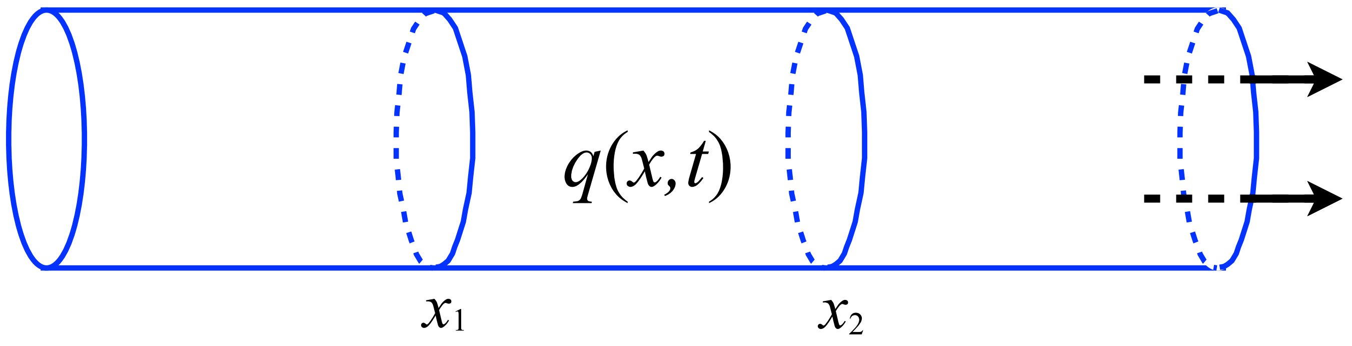
$$f(x,t) = uq(x,t)$$

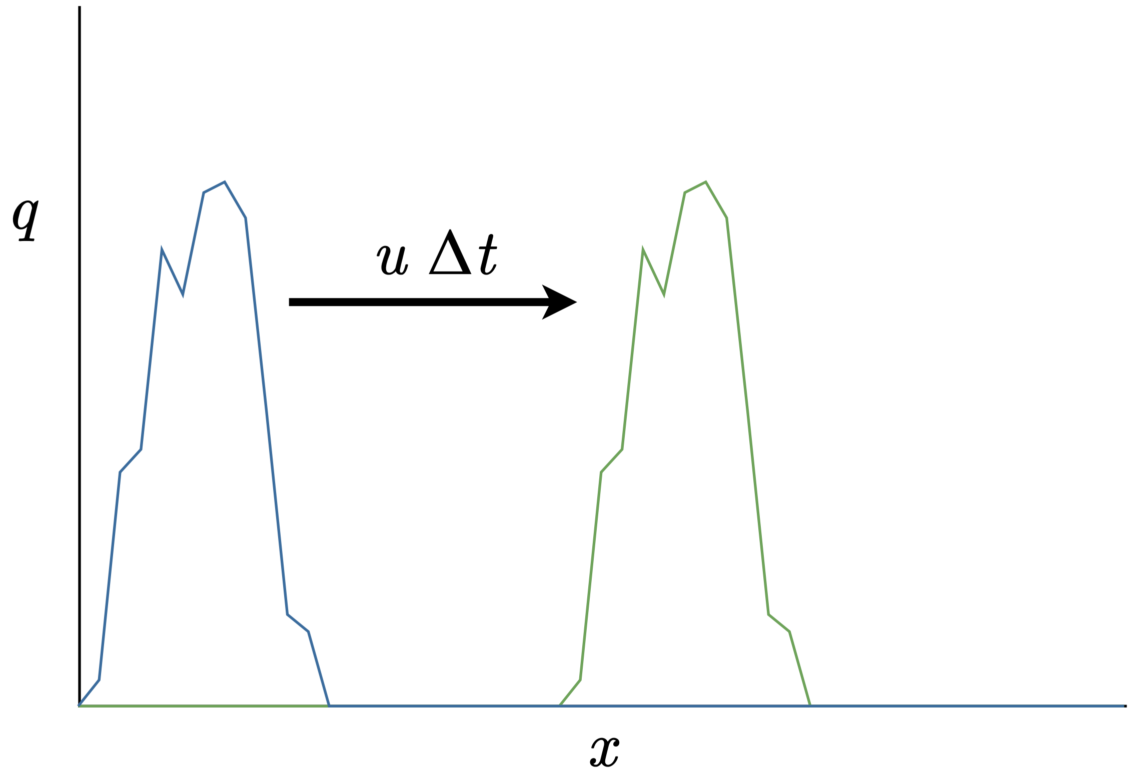
The conservation law is:

$$q_t(x,t) + uq_x(x,t) = 0$$

And the solution is:

$$q(x,t) = q(x-ut,0)$$





The advection equation is also called the *scalar wave equation*, because the solution is a one-way wave, propagating to the right with velocity u.

The advection equation — the simplest hyperbolic differential equation

With $q(x,t)$ a tracer concentration, and flow velocity

$$u(x,t) = u = \text{constant},$$

The flux function is:

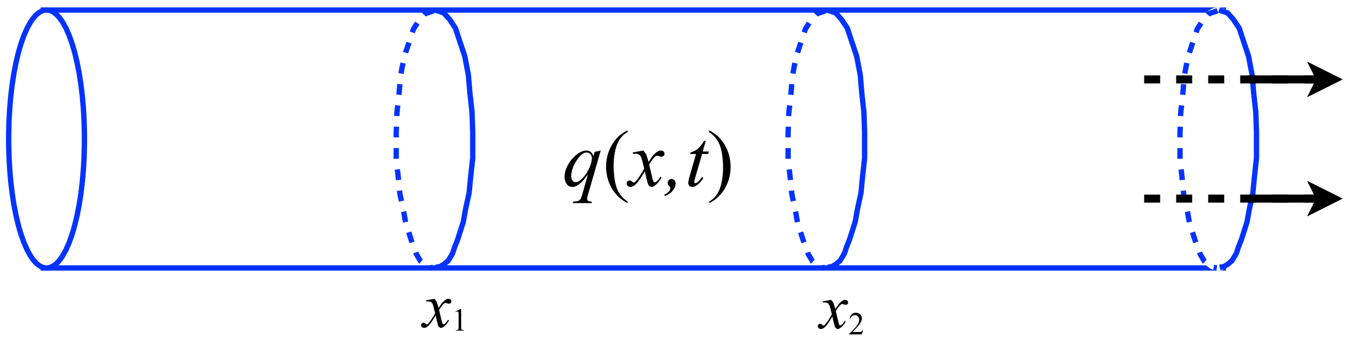
$$f(x,t) = uq(x,t)$$

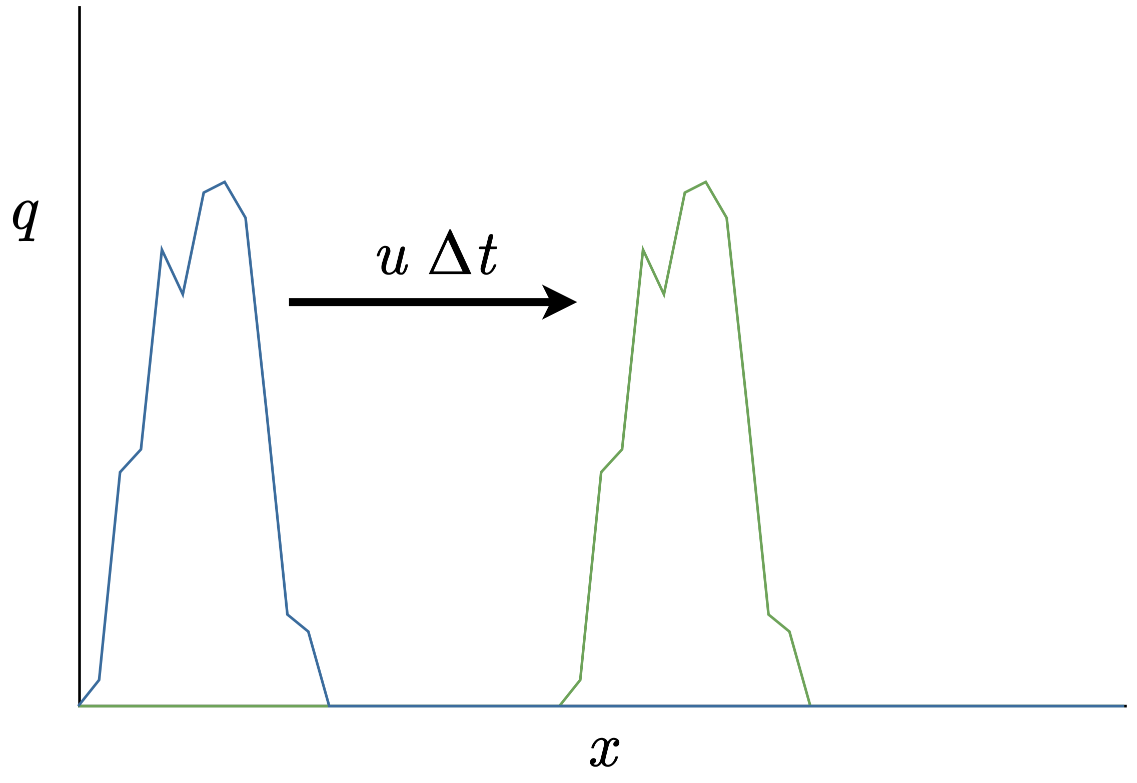
The conservation law is:

$$\boxed{q_t(x,t) + uq_x(x,t) = 0}$$

And the solution is:

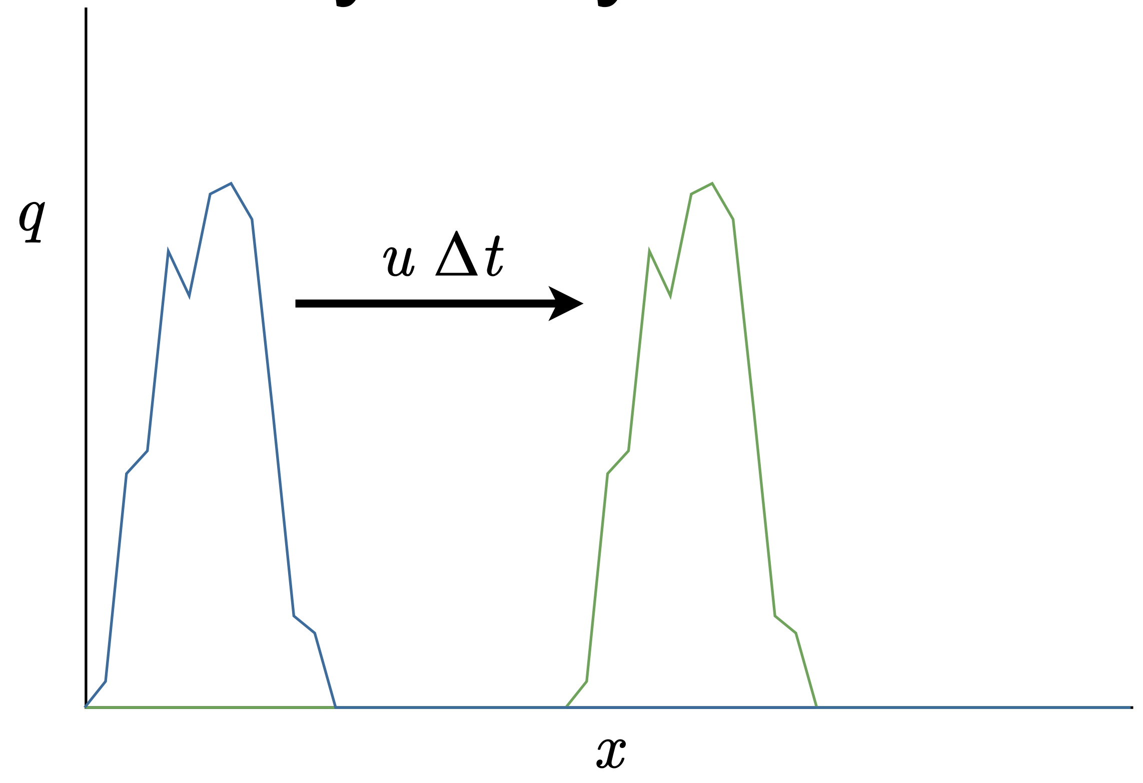
$$q(x,t) = q(x-ut,0)$$





The advection equation is also called the *scalar wave equation*, because the solution is a one-way wave, propagating to the right with velocity u.

Advection is easy; why bother?



If advection is simply the translation of a profile in space, we can do it trivially!

So why bother?

Actually, it's not that easy … AND …

Advection is always involved in more complex problems:
  material is advected through shocks, for example: and
  advection *always* occurs when fluids move, and fluids move in complex ways.

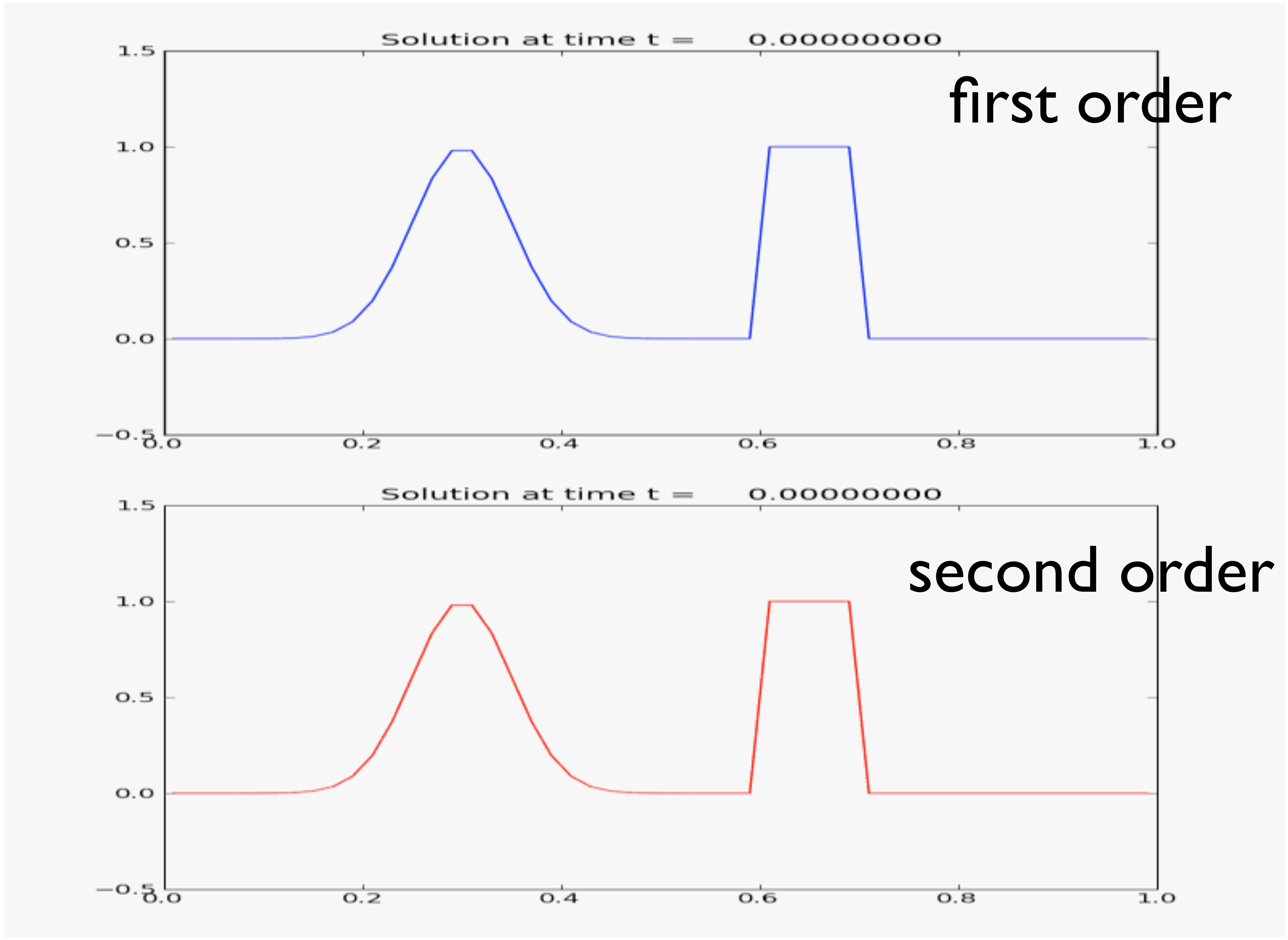
We will find that the solution to more general problems is a superposition of advection solutions.

Advection is difficult!

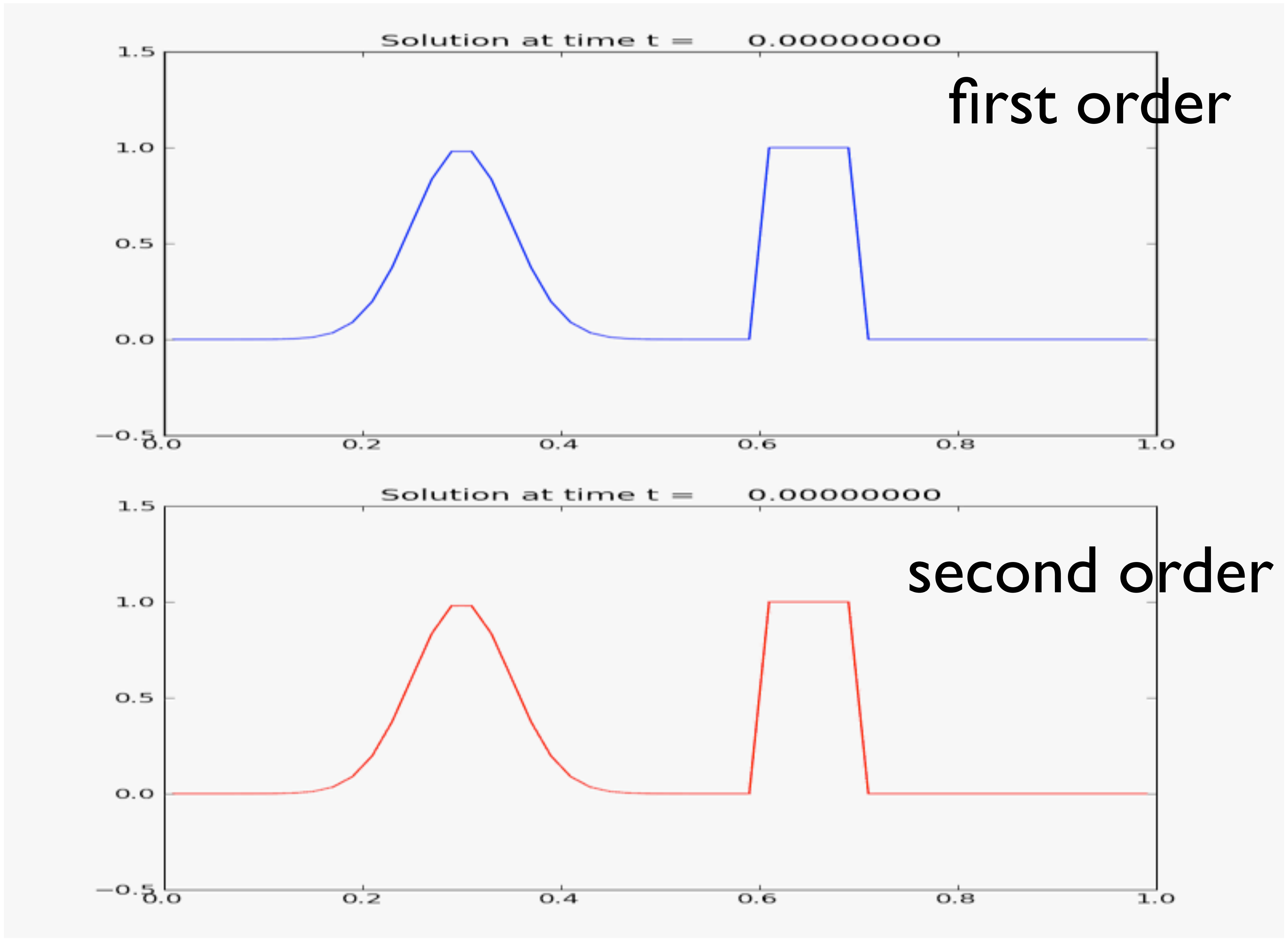
first order

second order

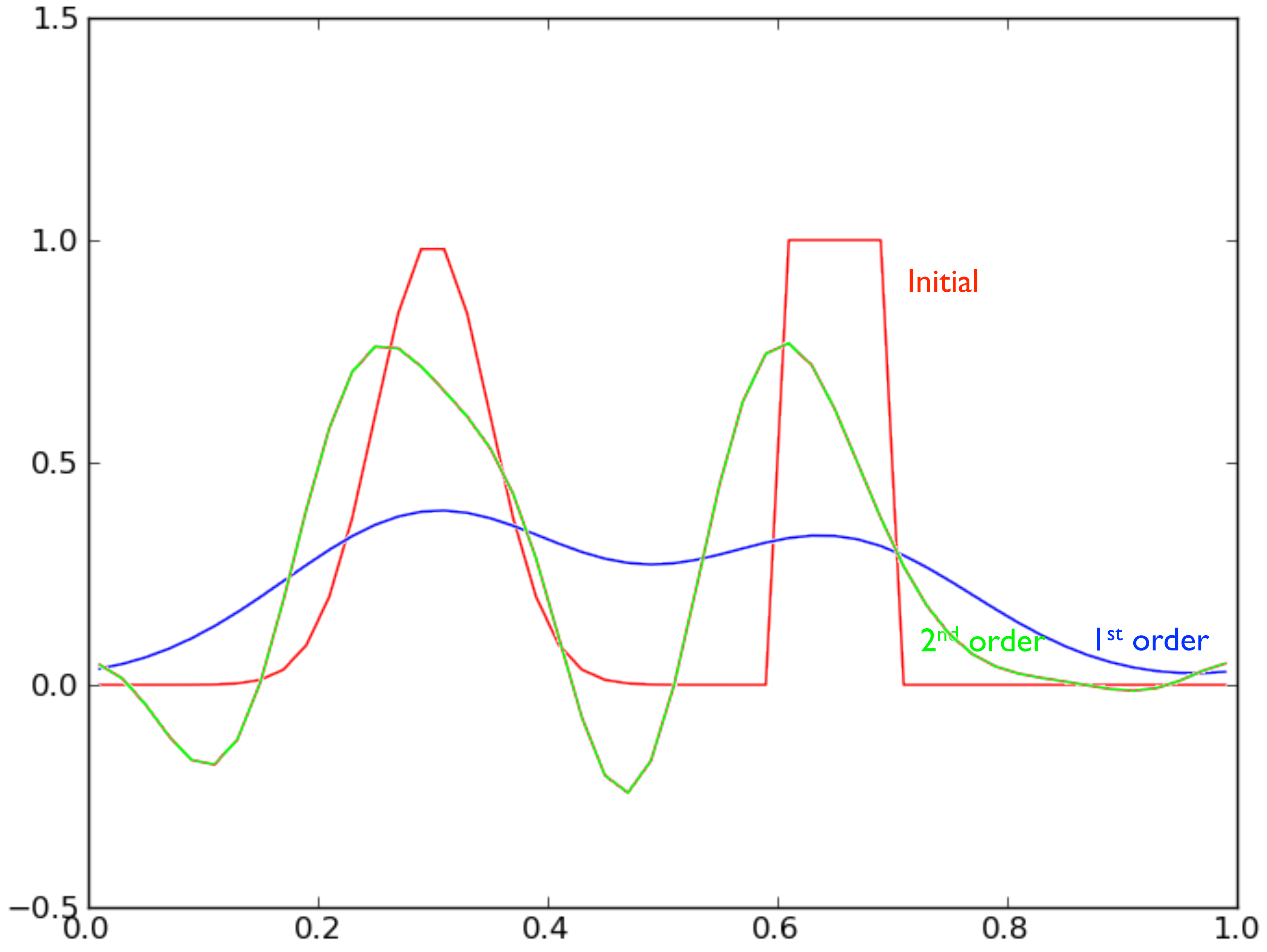
Advection is difficult!



Advection is difficult!

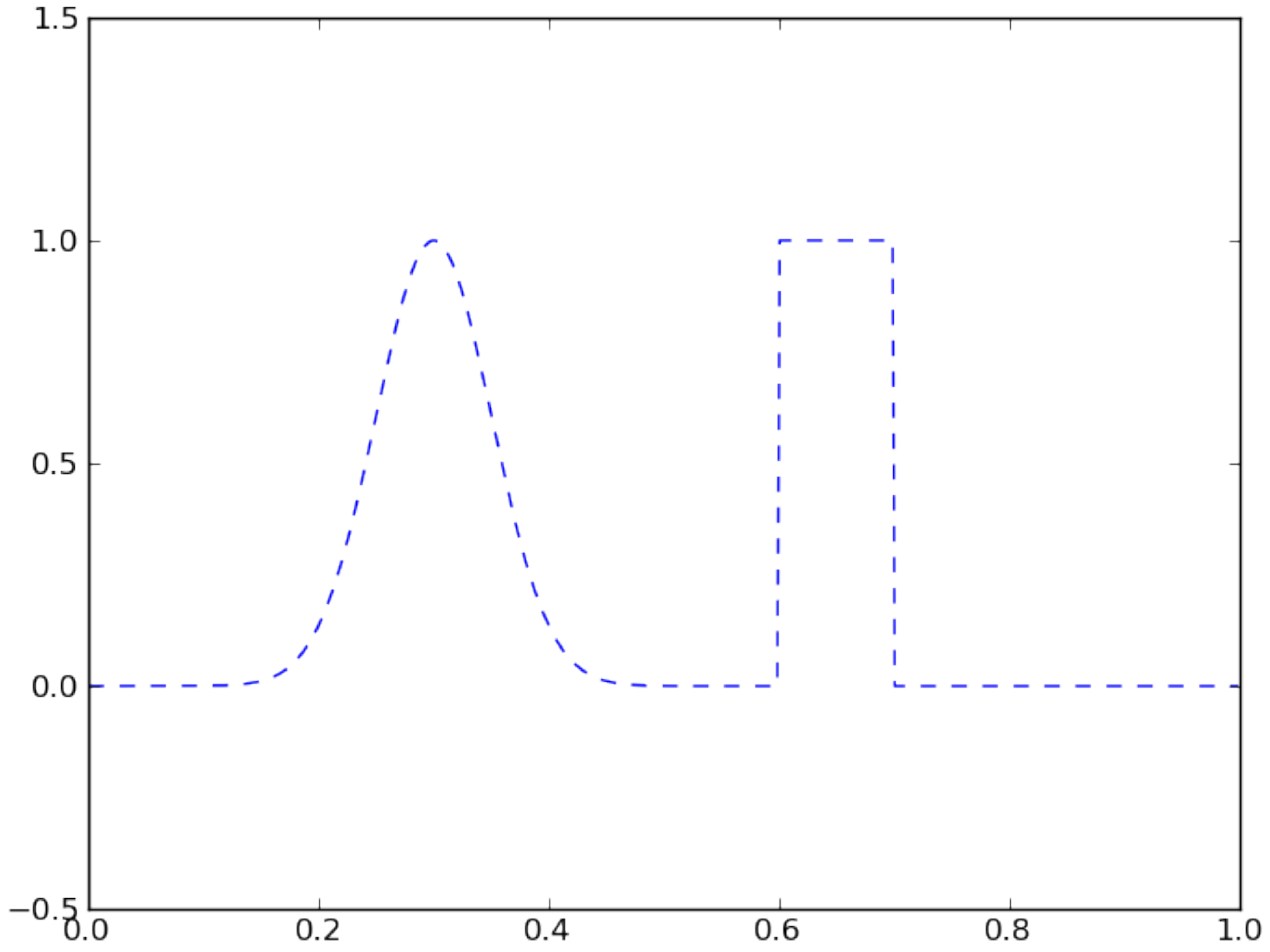


Shapes are not well preserved, but volume is conserved



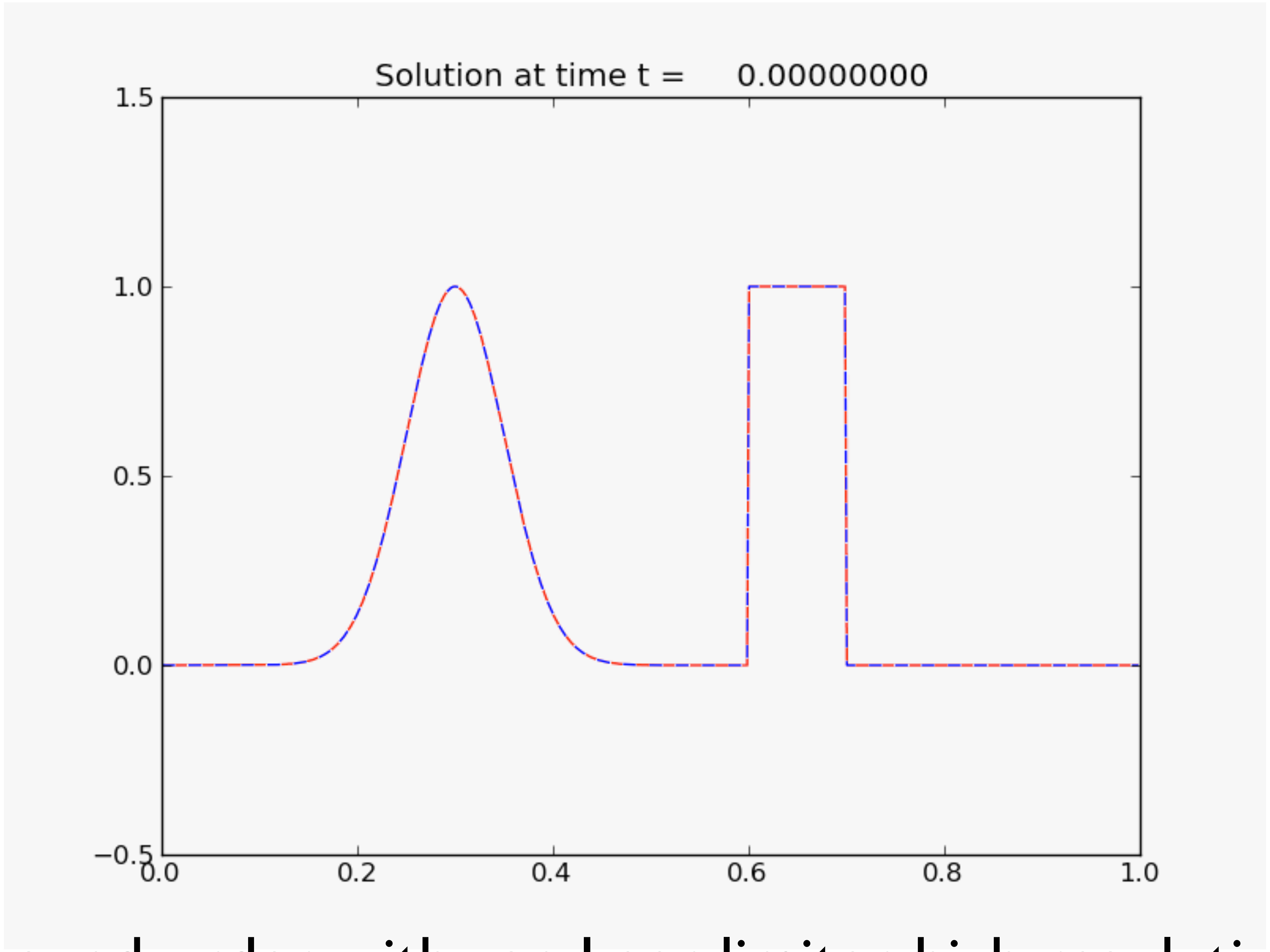
after 4 revolutions

We can do much better!



second order with van Leer limiter, high resolution

We can do much better!



second order with van Leer limiter, high resolution

Advection is difficult. Why?

Advection, by itself, is **notoriously difficult** to do correctly in a numerical problem:

The fluid is discretised into finite volumes

And the velocity is different from the *discretisation ratio*: $\left(\frac{\text{spatial grid size}}{\text{time step}}\right)$
so every time step advance requires **interpolation**

But advection is a good, challenging, test problem to start with, so:
we will develop techniques using this problem
then use the same techniques to solve more difficult ones.

Problems we will be able to solve: acoustic, seismic, and electromagnetic wave propagation, gas dynamics (airplanes, wind turbines, hydrothermal vents, volcanic jets), water waves, tsunamis, traffic flow.

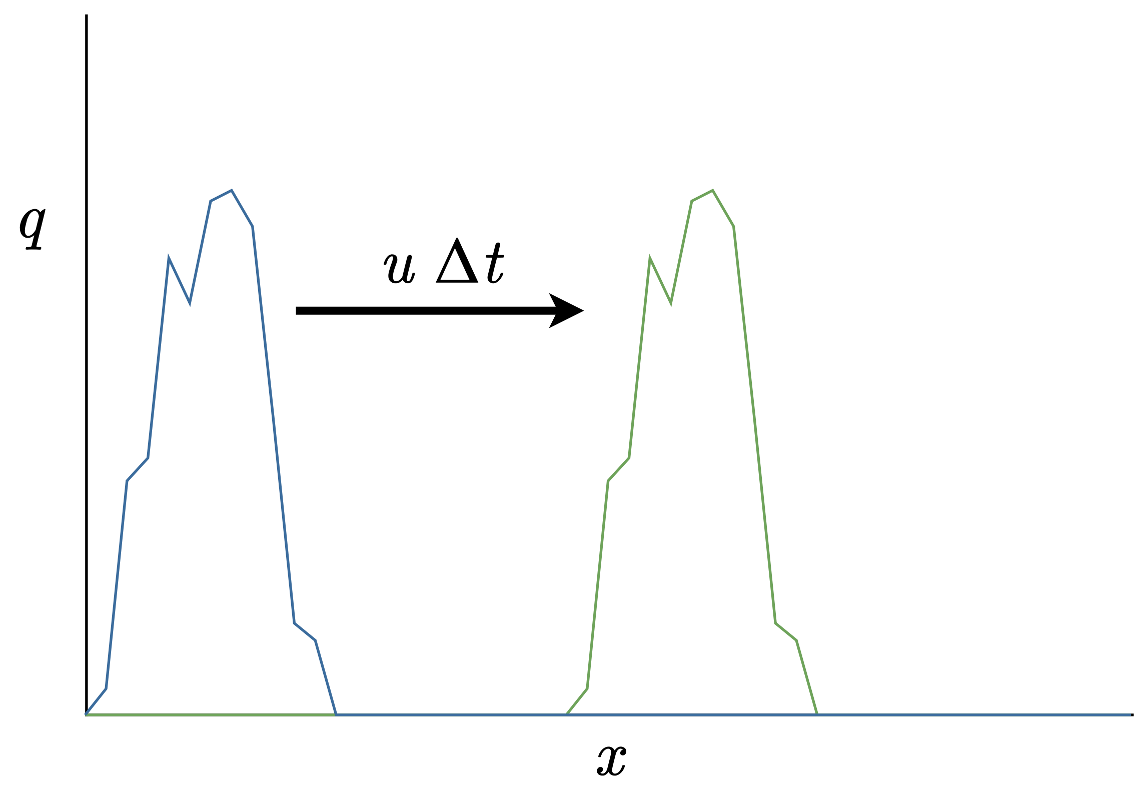
We need to learn about *characteristics*

The one-dimensional advection equation

$$q_t(x,t) + u q_x(x,t) = 0$$

has the solution

$$q(x,t) = \eta(x - ut)$$



The function $q(x,t)$ is constant along any space-time ray for which

$$x - ut = \text{constant} = x_0$$

Characteristics for the advection equation

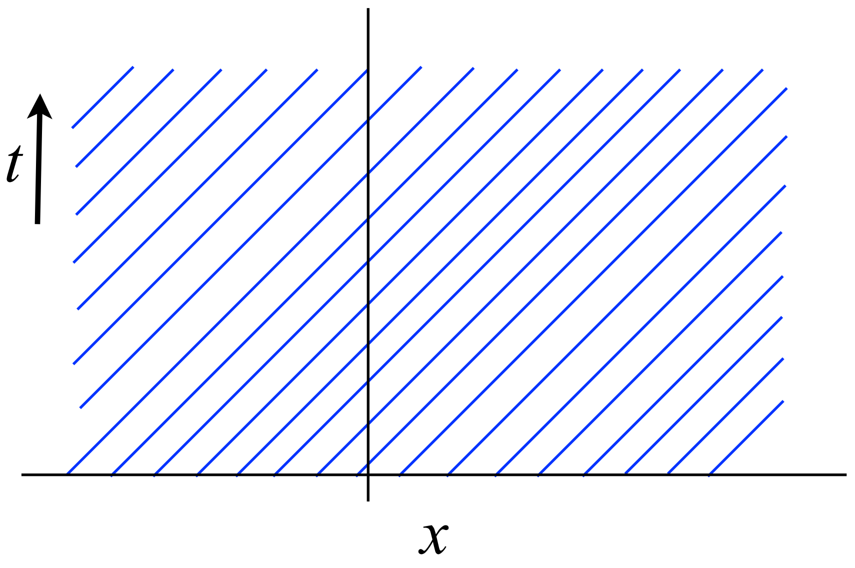
The solution $q(x,t) = \eta(x - ut)$ implies that $q(x,t)$ is constant along lines

$$X(t) = x_0 + ut,\ \ t \ge 0$$

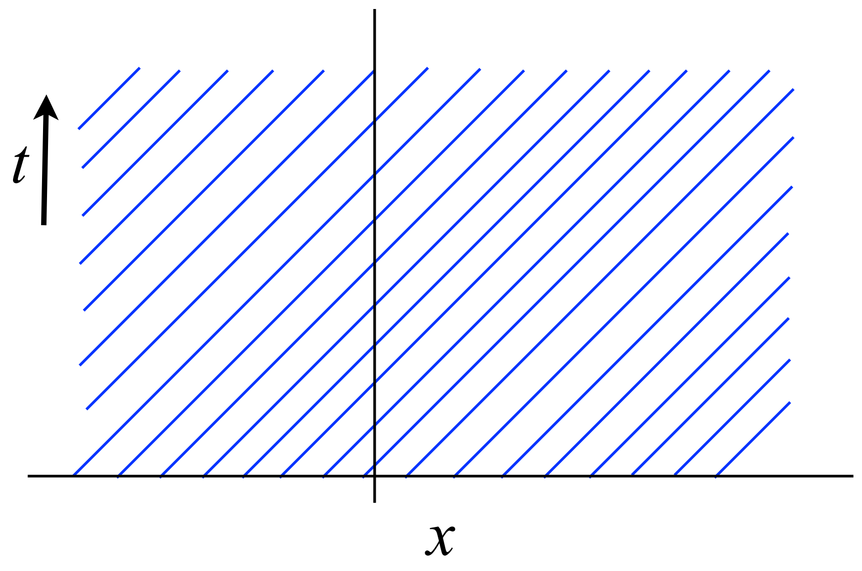
Proof:

$$\begin{aligned}\frac{d}{dt} q\big(X(t),t\big) &= q_x\big(X(t),t\big)X'(t) + q_t\big(X(t),t\big) \\ &= q_x\big(X(t),t\big)u + q_t\big(X(t),t\big) \\ &= q_t + uq_x = 0\end{aligned}$$

$X(t)$ are the wave fronts, or *characteristics*. In the x–t plane they are lines of constant slope:



Initial conditions



If we solve the advection equation on an infinite one-dimensional domain:

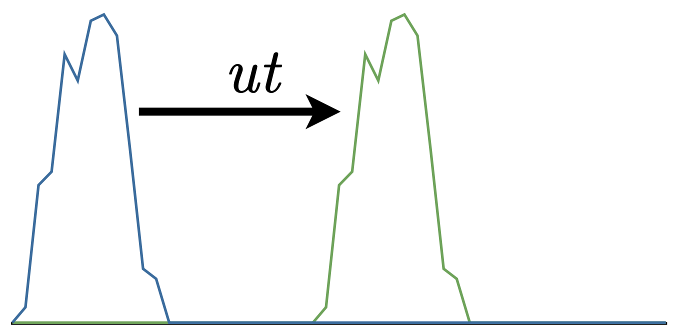
$$q_t(x,t) + u q_x(x,t) = 0 \qquad -\infty < x < \infty, \quad t \geq 0$$

Then we need the initial condition (initial data):

$$q(x,0) = \eta(x), \qquad -\infty < x < \infty$$

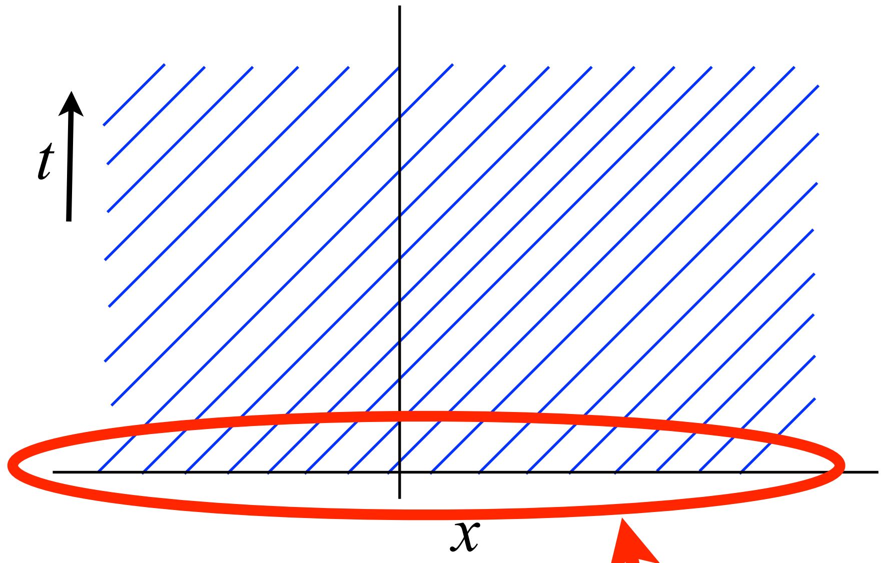
And we have the solution:

$$q(x,t) = \eta(x - ut) \qquad -\infty < x < \infty, \quad t \geq 0$$



This is an Initial Value Problem (IVP) or Cauchy problem

Initial conditions



If we solve the advection equation on an infinite one-dimensional domain:

$$q_t(x,t) + uq_x(x,t) = 0 \qquad -\infty < x < \infty, \;\; t \geq 0$$

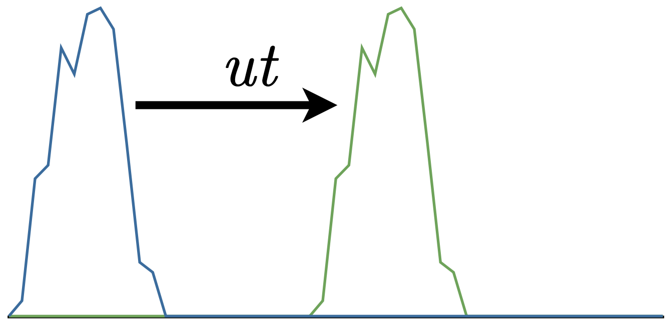
Cauchy problem: all characteristics originate at $t=0$

Then we need the initial condition (initial data):

$$q(x,0) = \eta(x), \qquad -\infty < x < \infty$$

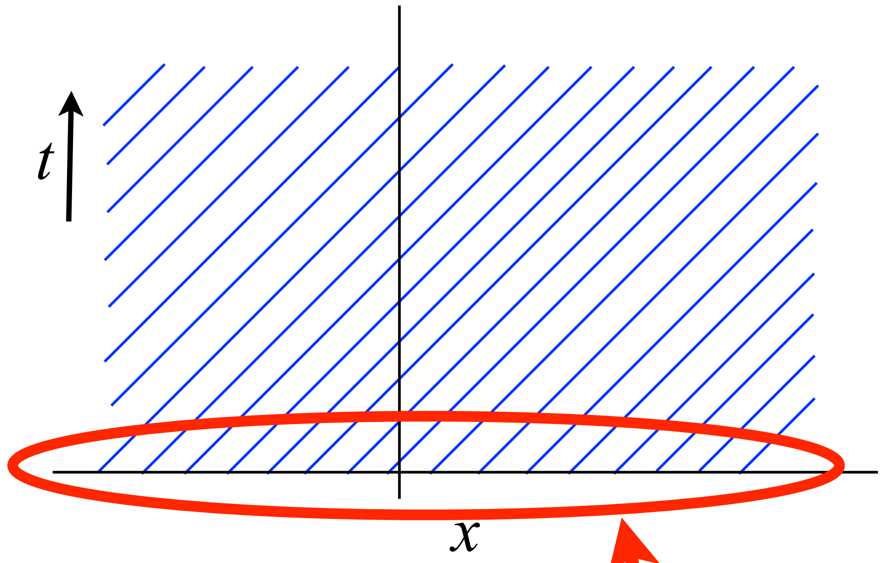
And we have the solution:

$$q(x,t) = \eta(x - ut) \qquad -\infty < x < \infty, \;\; t \geq 0$$



This is an Initial Value Problem (IVP) or Cauchy problem

Initial conditions



If we solve the advection equation on an infinite one-dimensional domain:

$$q_t(x,t) + uq_x(x,t) = 0 \qquad -\infty < x < \infty, \;\; t \geq 0$$

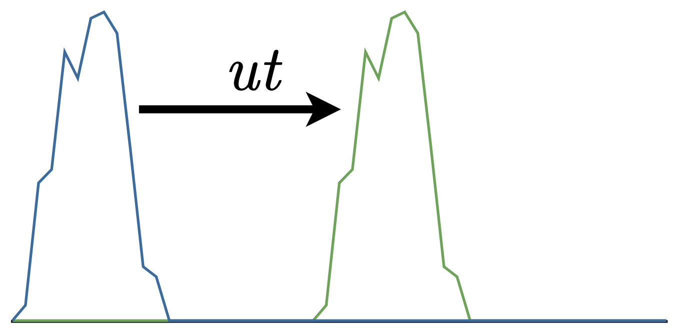
Cauchy problem: all characteristics originate at $t=0$

Then we need the initial condition (initial data):

$$q(x,0) = \eta(x), \qquad -\infty < x < \infty$$

And we have the solution:

$$q(x,t) = \eta(x - ut) \qquad -\infty < x < \infty, \;\; t \geq 0$$



This is an Initial Value Problem (IVP) or Cauchy problem

Boundary Conditions

If we need to solve the same equation on a finite domain:

$$q_t(x,t) + uq_x(x,t) = 0 \qquad a < x < b, \;\; t \geq 0$$

Then we need initial data:

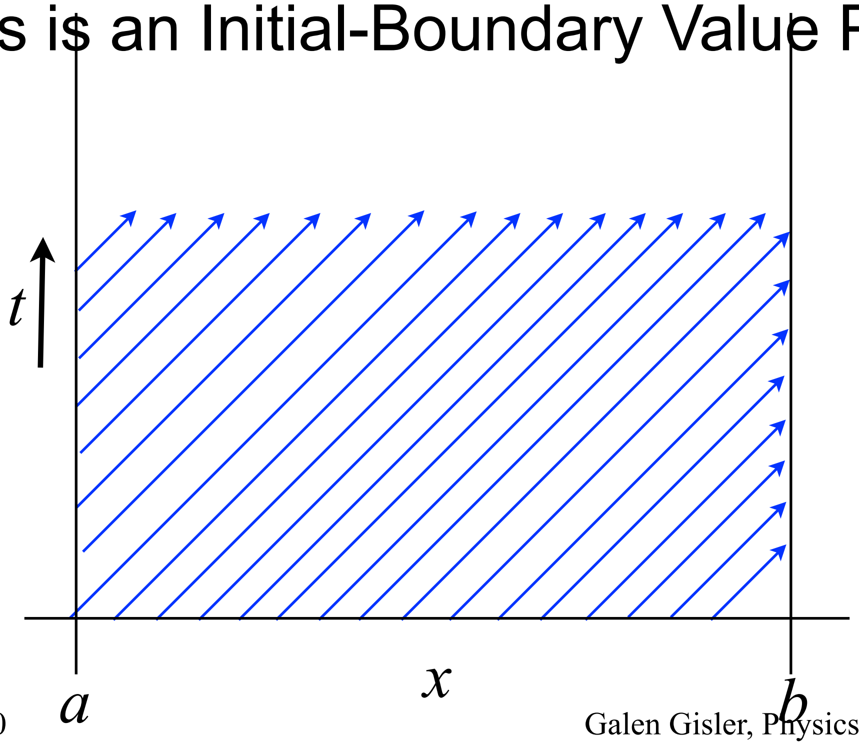
$$q(x,0) = \eta(x), \qquad a < x < b$$

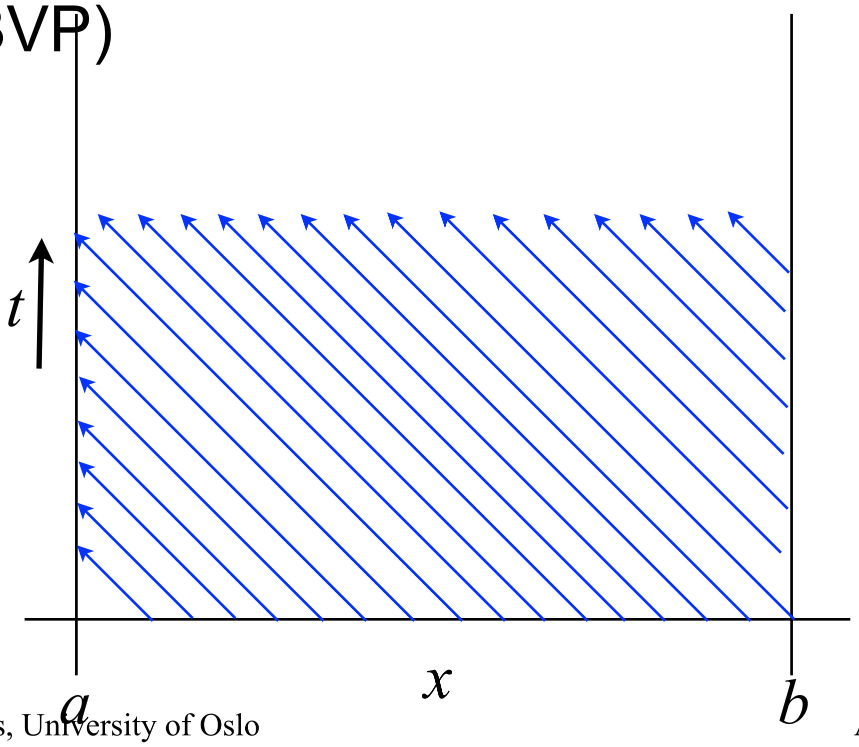
And boundary data at the *inflow* boundary (no boundary condition on *outflow*):

$$q(a,t) = g(t), \quad t \geq 0 \qquad \underline{\textbf{or}} \qquad q(b,t) = g(t), \quad t \geq 0$$

$$\text{if } u > 0 \qquad\qquad\qquad \text{if } u < 0$$

This is an Initial-Boundary Value Problem (IBVP)





Boundary Conditions

If we need to solve the same equation on a finite domain:

$$q_t(x,t) + uq_x(x,t) = 0 \qquad a < x < b, \;\; t \geq 0$$

Then we need initial data:

$$q(x,0) = \eta(x), \qquad a < x < b$$

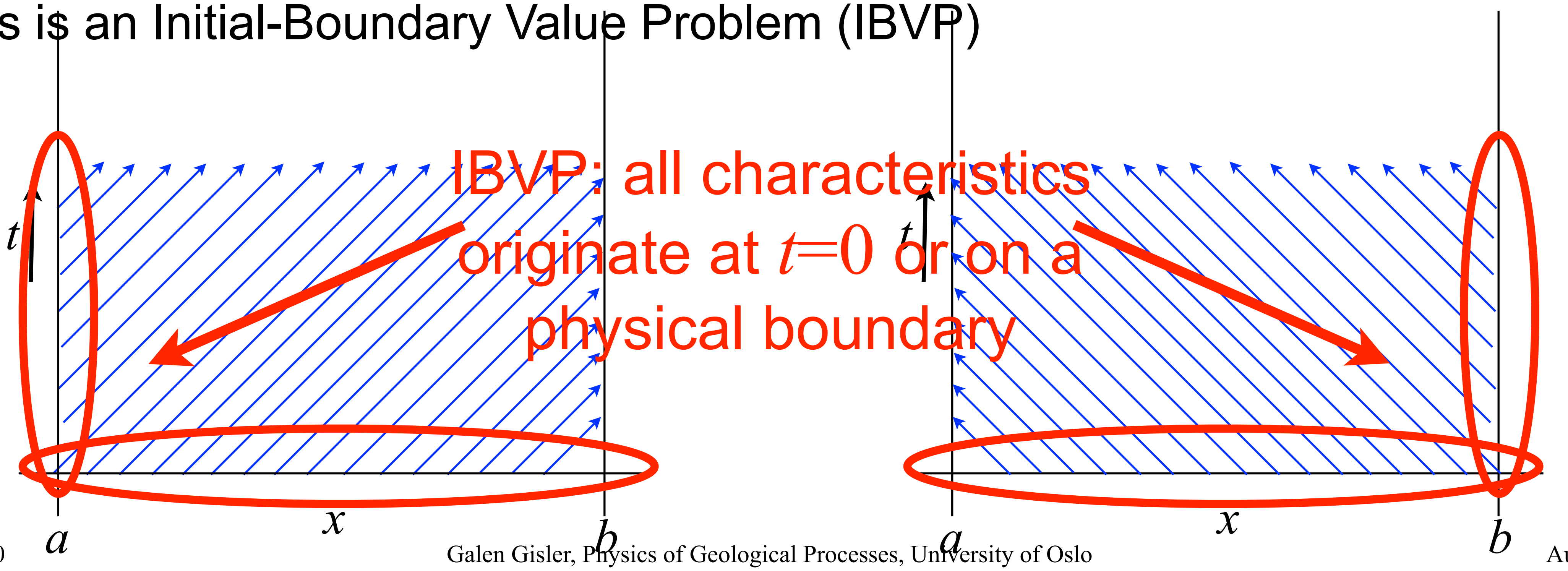
And boundary data at the *inflow* boundary (no boundary condition on *outflow*):

$$q(a,t) = g(t), \qquad t \geq 0 \qquad \text{if } u > 0$$

or

$$q(b,t) = g(t), \qquad t \geq 0 \qquad \text{if } u < 0$$

This is an Initial-Boundary Value Problem (IBVP)



IBVP: all characteristics originate at t=0 or on a physical boundary
t
a
x
b
t
a
x
b

Periodic Boundary Conditions

The easiest boundary conditions to implement numerically are periodic boundary conditions. Under certain conditions this mimics an infinite domain, but beware the inherent periodicity. For periodic boundary conditions:

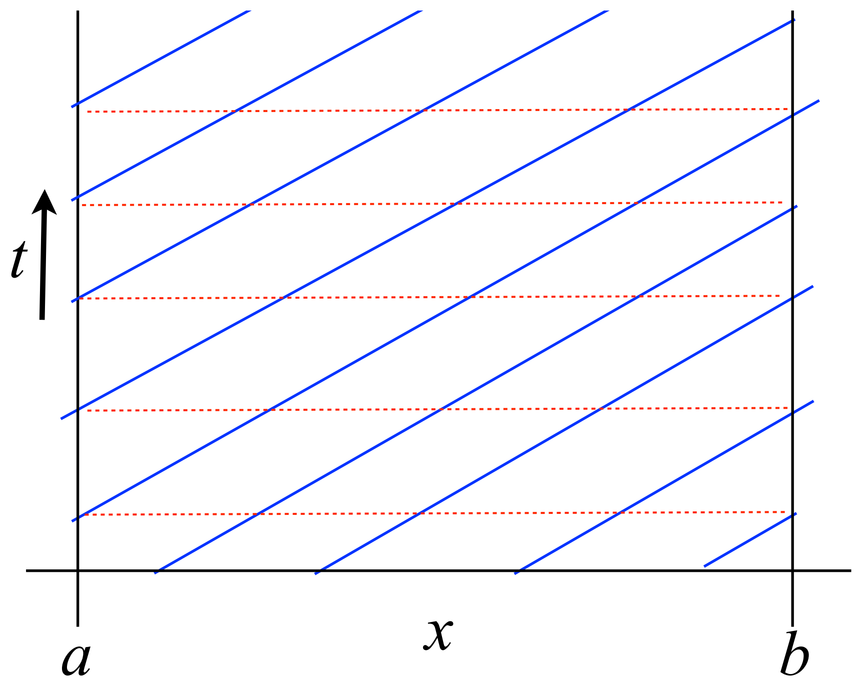
$$q(a,t) = q(b,t), \qquad t \geq 0$$

In this case, the solution is:

$$q(x,0) = \eta\big(X_0(x,t)\big)$$

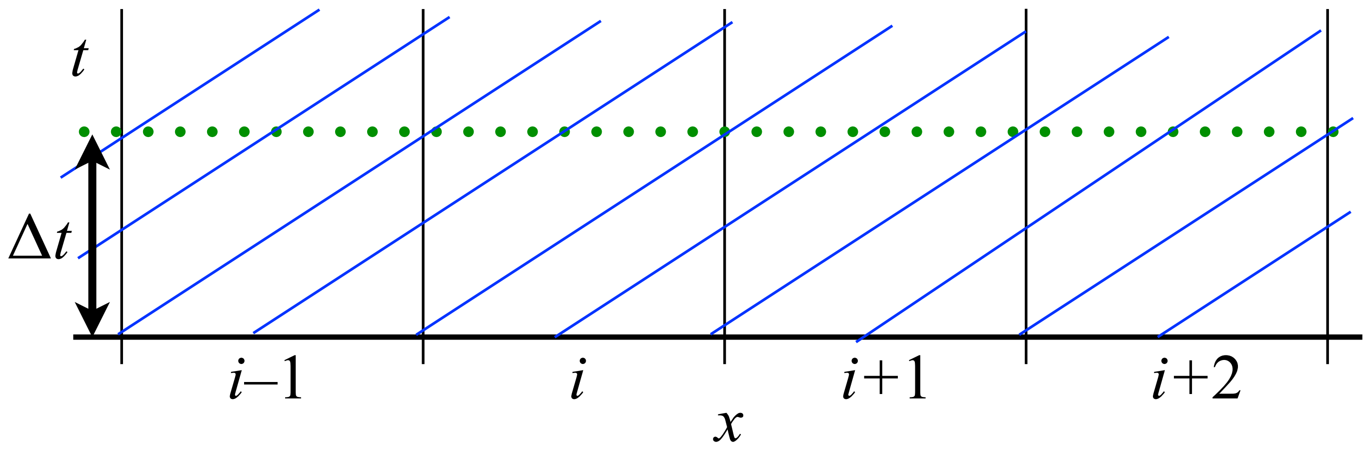
And the characteristics are:

$$X_0(x,t) = a + \mathrm{mod}(x - ut - a,\ b - a)$$



Characteristics lead to the concept of "domain of influence"

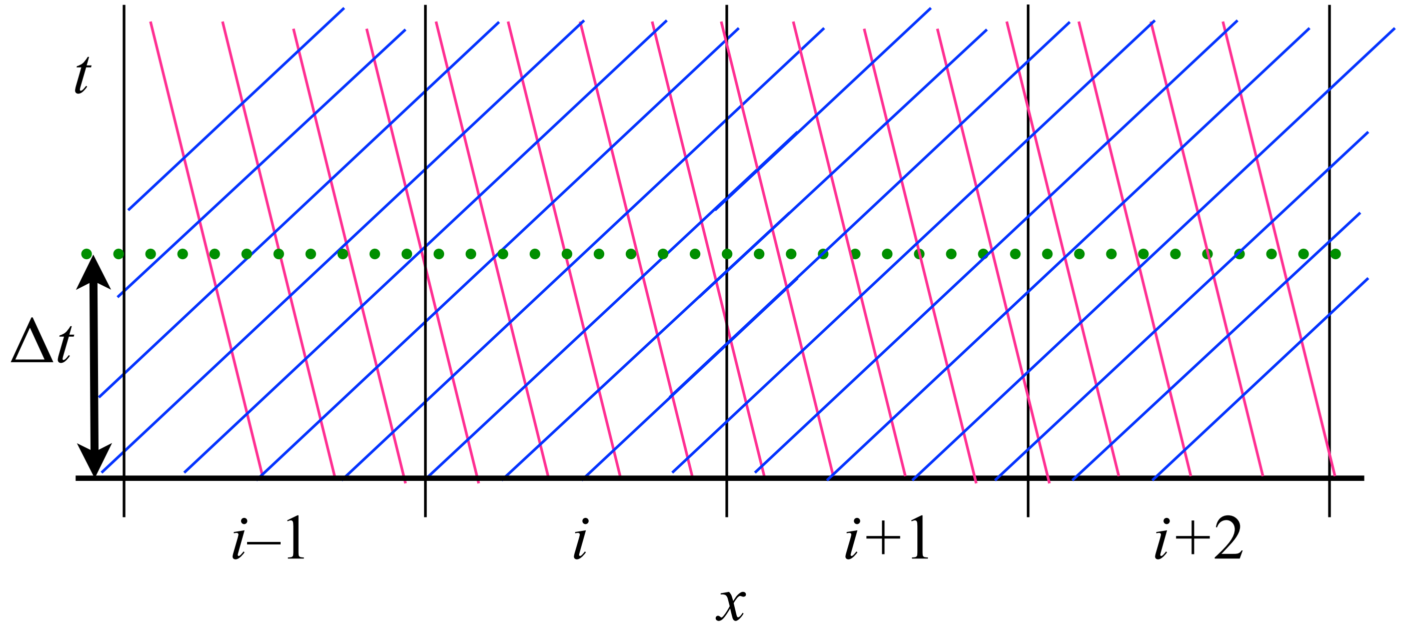
The *domain of influence* is the region of space that can be physically affected by a source.



In this example, cell i can influence **all** of cell $i+1$ in the time Δt, and cell i is in turn influenced by cell $i-1$.

For stability, a cell must not attempt to influence more than its immediate neighbours during a single time step of a numerical method.

Characteristic lines for systems of equations



Systems of equations can have as many characteristics as there are equations. These characteristics can propagate in both directions.

Every point in the accessible space-time is crossed by all of the characteristics.

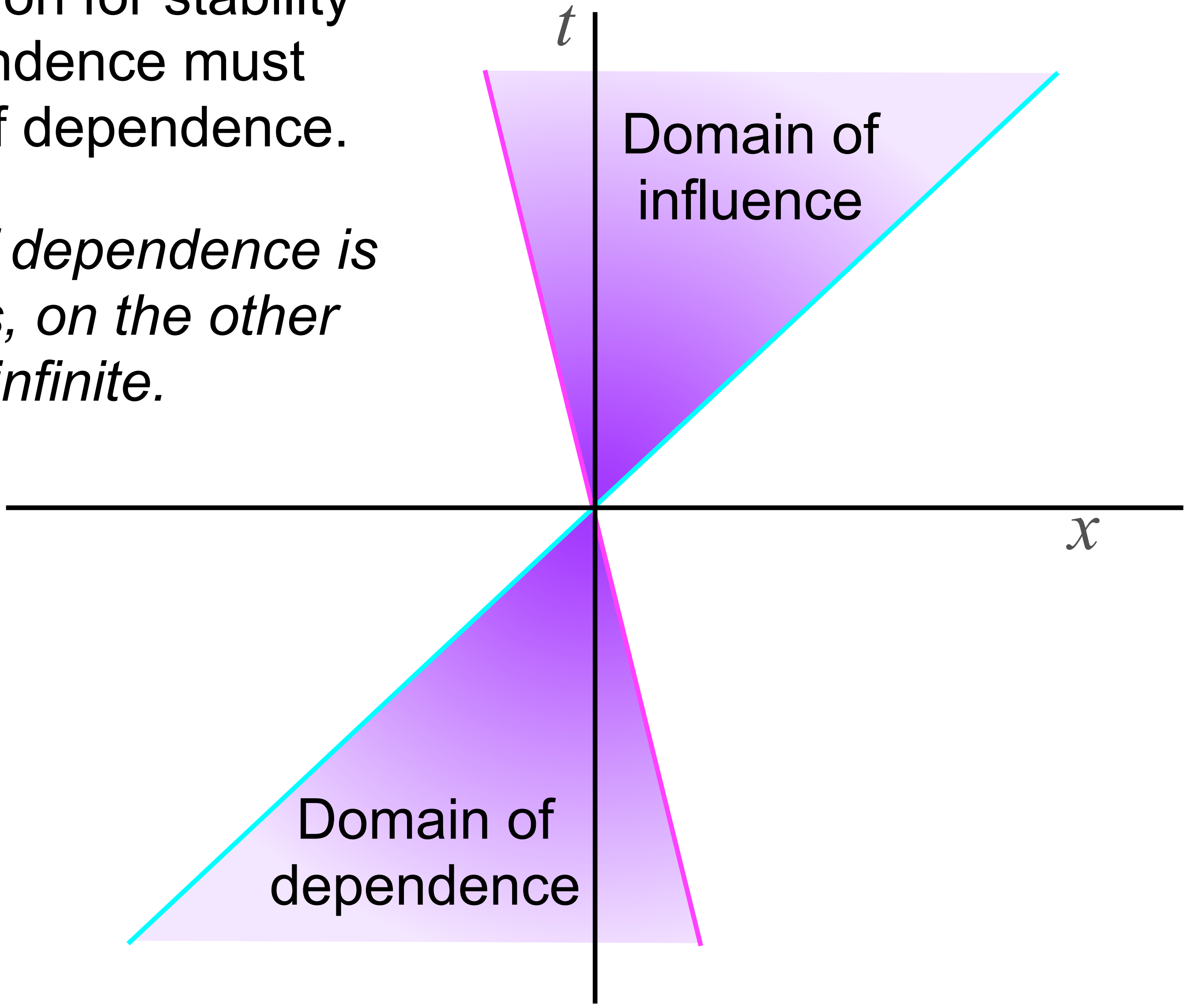
In this example, following the blue characteristics, cell i can influence most of cell $i+1$ in the time Δt, and cell i is in turn influenced by cell $i–1$. Following the red characteristics, cell $i+1$ influences part of cell i, and cell i influences part of cell $i–1$.

Causal domains in space-time

The domains of influence and dependence depend on the characteristics of the equations.

The Courant-Friedrich-Lewy (CFL) condition for stability states that the *numerical* domain of dependence must completely contain the *physical* domain of dependence.

For any hyperbolic system, the domain of dependence is bounded. For elliptic or parabolic systems, on the other hand, the domain of dependence can be infinite.

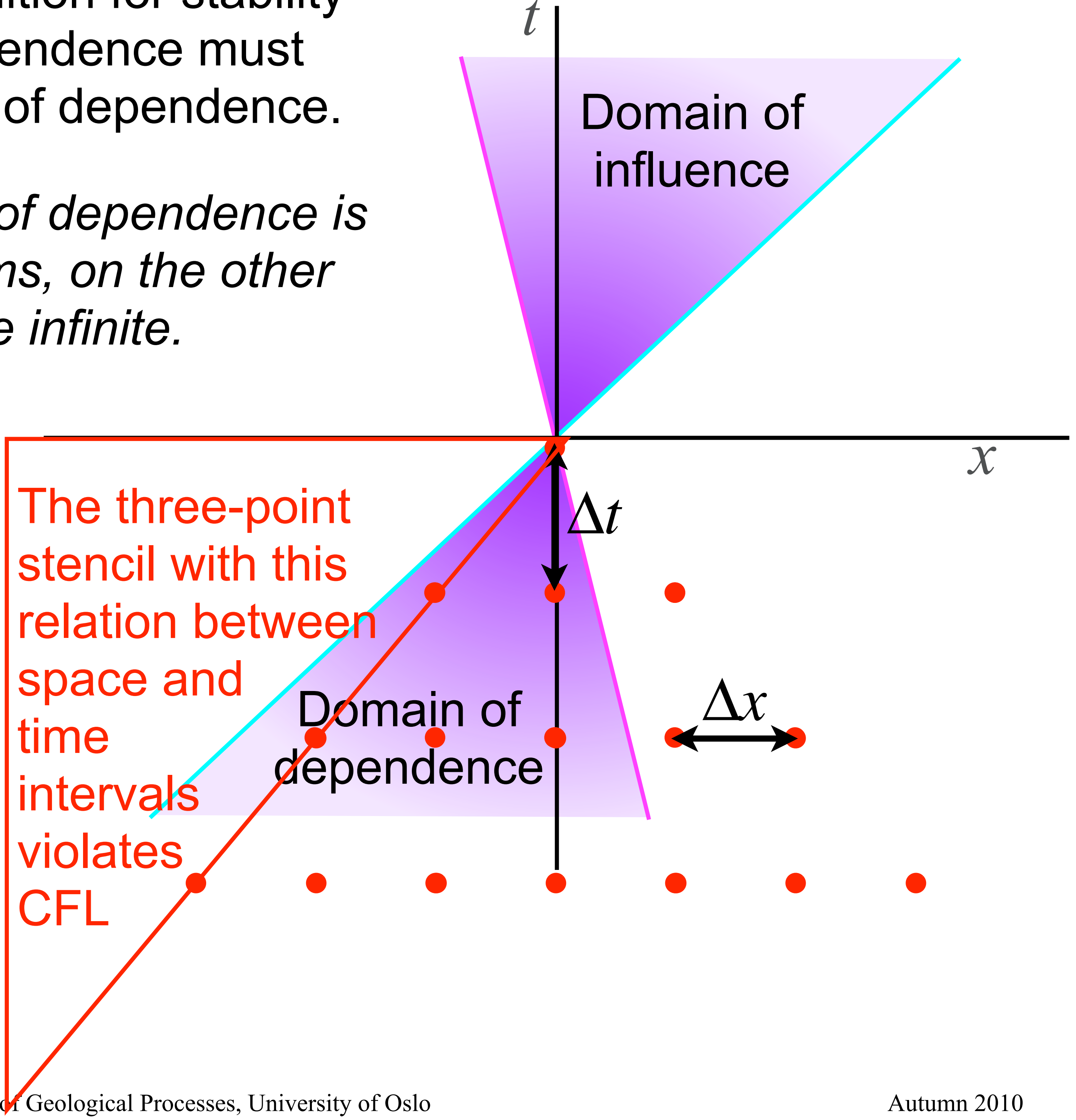


Causal domains in space-time

The domains of influence and dependence depend on the characteristics of the equations.

The Courant-Friedrich-Lewy (CFL) condition for stability states that the *numerical* domain of dependence must completely contain the *physical* domain of dependence.

For any hyperbolic system, the domain of dependence is bounded. For elliptic or parabolic systems, on the other hand, the domain of dependence can be infinite.

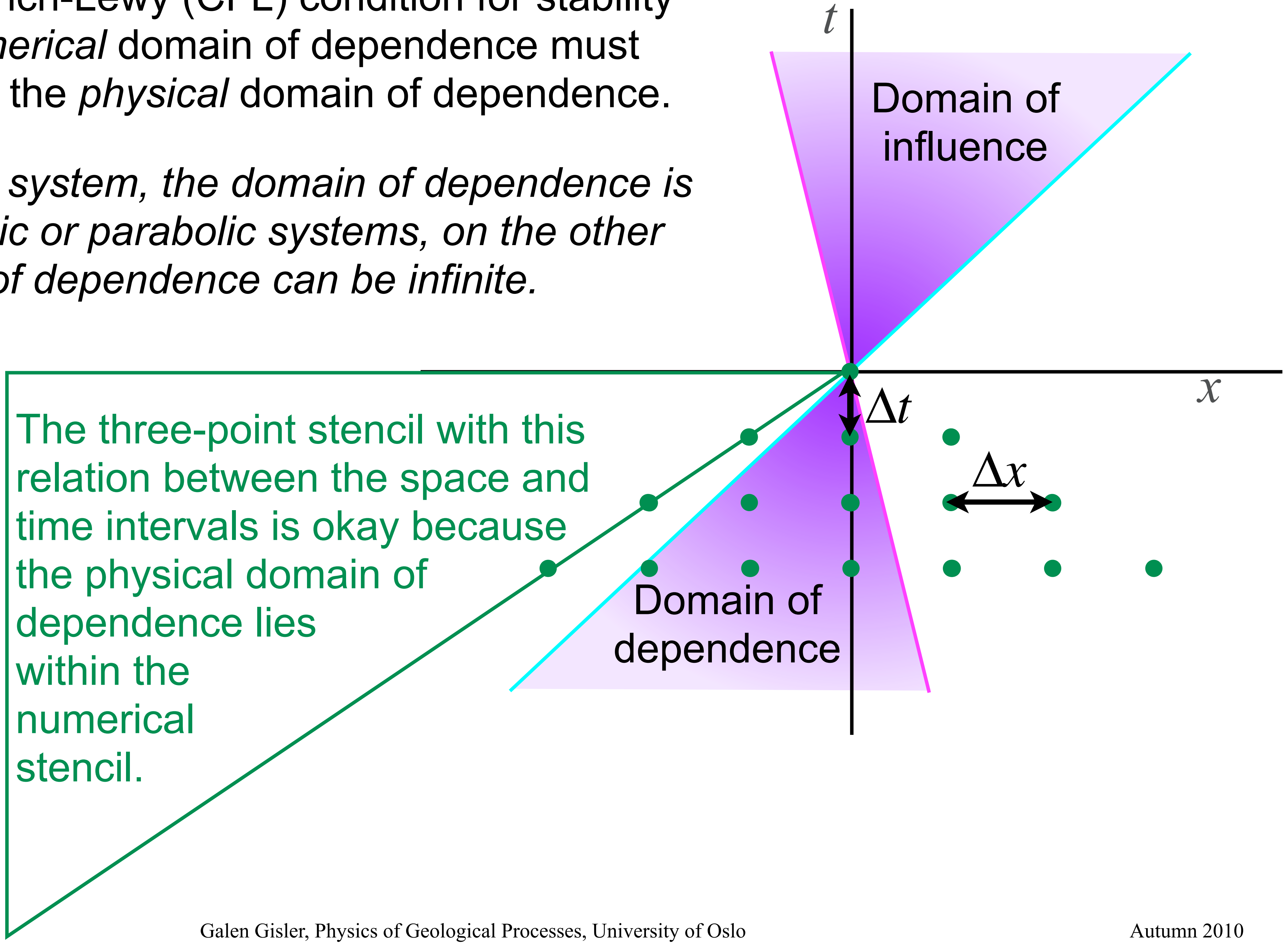

t
Domain of influence
x
The three-point stencil with this relation between space and time intervals violates CFL
Δt
Domain of dependence
Δx

Causal domains in space-time

The domains of influence and dependence depend on the characteristics of the equations.

The Courant-Friedrich-Lewy (CFL) condition for stability states that the *numerical* domain of dependence must completely contain the *physical* domain of dependence.

For any hyperbolic system, the domain of dependence is bounded. For elliptic or parabolic systems, on the other hand, the domain of dependence can be infinite.



The three-point stencil with this relation between the space and time intervals is okay because the physical domain of dependence lies within the numerical stencil.

Variable coefficients

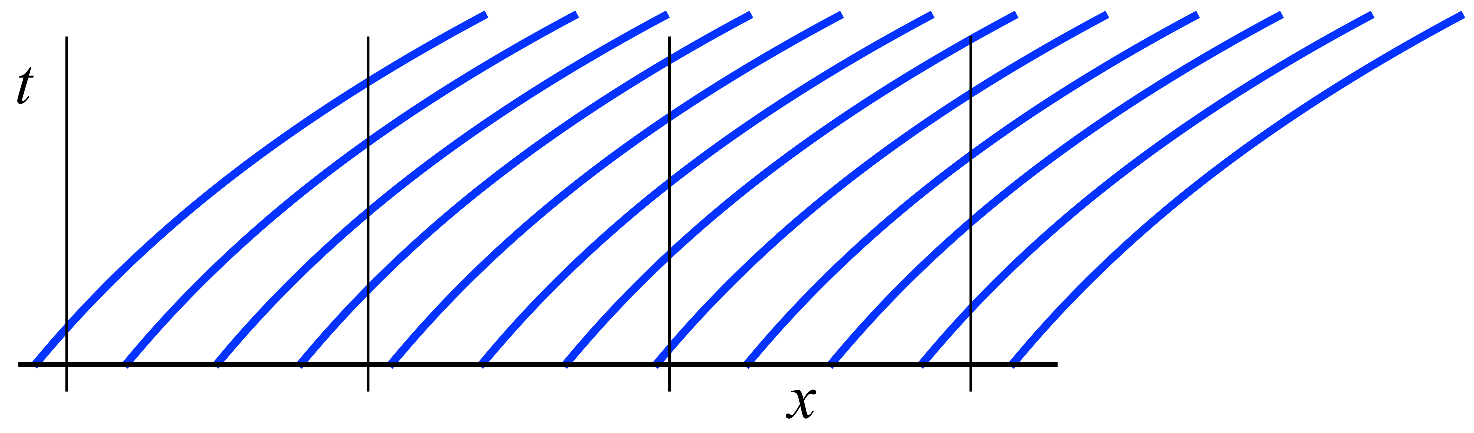
If the fluid velocity varies with x, then the conservation law is:

$$q_t + \left(u(x)q\right)_x = 0$$

And then the characteristics are not straight lines, but *curves* which are found by solving the ODE:

$$X' = u\left(X(t)\right)$$

In this case, q is not constant along the curves, but the curves still track material particles.



The *material derivative* $\dfrac{\partial}{\partial t} + u\dfrac{\partial}{\partial x}$ tracks changes observed by someone

moving along with the fluid (along a characteristic curve).

A little gas dynamics: the continuity equation

A special kind of variable-coefficient equation arises from considering the flow of a gas. Since gasses are compressible, the density and velocity can both vary during the flow. Then the conservation equation derived before, expressed in terms of density ρ is:

$$\rho_t + f(\rho)_x = 0$$

With the flux function $f(x,t) = \rho(x,t)u(x,t)$

This becomes the equation for the conservation of mass, called the continuity equation:

$$\rho_t + (\rho u)_x = 0$$

Conservation of momentum

The continuity equation by itself isn't sufficient to solve the flow of a gas, so we need additional conservation laws.

The product $\rho(x,t)u(x,t)$ is the density of momentum, and we can derive a conservation law equation from it, remembering that pressure p contributes to change of momentum:

$$(\rho u)_t + \left(\rho u^2 + p\right)_x = 0$$

And then we would need another conservation law for energy and an *equation of state* relating energy to both p and ρ. As a short-cut, we could simply use an equation relating p to ρ. This is sometimes sufficient.

One example is the polytropic equation $p = K\rho^\gamma$, which is the isothermal condition for $\gamma = 1$. More generally we may use the barotropic relation $p = P(\rho)$.

Then we have a closed system of two equations. If $P'(\rho) > 0$ for positive ρ, the system is hyperbolic.

The basic system of equations for a barotropic fluid (simplified gas dynamics)

Our system is
$$\rho_t + (\rho u)_x = 0$$
$$\left(\rho u\right)_t + \left(\rho u^2 + P(\rho)\right)_x = 0$$

Which we can write as $q_t + f(q)_x = 0$, if we define

$$q = \begin{bmatrix} \rho \\ \rho u \end{bmatrix}, \quad f(q) = \begin{bmatrix} \rho u \\ \rho u^2 + P(\rho) \end{bmatrix}$$

If q is sufficiently smooth, we can write $q_t + f'(q) q_x = 0$ (quasilinear form)

Where $f'(q) = \begin{bmatrix} \dfrac{\partial f^1}{\partial q^1} & \dfrac{\partial f^1}{\partial q^2} \\ \dfrac{\partial f^2}{\partial q^1} & \dfrac{\partial f^2}{\partial q^2} \end{bmatrix}$ is the *Jacobian* matrix.

Linear acoustics

This will be a useful example that we'll return to often. We linearise the barotropic system by examining only a perturbation to the (constant) background state. Let

$$q(x,t) = q_0 + \tilde{q}(x,t),\ \tilde{q} = \begin{bmatrix} \tilde{\rho} \\ \widetilde{\rho u} \end{bmatrix}$$

The conservation law $q_t + f(q)_x = 0$ becomes the constant-coefficient linear system $\tilde{q}_t + f'(q_0)\tilde{q}_x = 0$ since we discard powers of the perturbed quantity.

The Jacobian of the perturbed barotropic system then becomes

$$A = f'(q) = \begin{bmatrix} \frac{\partial f^1}{\partial q^1} & \frac{\partial f^1}{\partial q^2} \\ \frac{\partial f^2}{\partial q^1} & \frac{\partial f^2}{\partial q^2} \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ -u^2 + P'(\rho) & 2u \end{bmatrix}$$

Linear acoustics

The system just obtained can be written out as:

$$\tilde{\rho}_t + (\widetilde{\rho u})_x = 0$$

$$(\widetilde{\rho u})_t + \left(-u_0^2 + P'(\rho_0)\right)\tilde{\rho}_x + 2u_0(\widetilde{\rho u})_x = 0$$

Then writing $P'(\rho_0)\tilde{\rho} = \tilde{p}$, $\widetilde{\rho u} = u_0\tilde{\rho} + \rho_0\tilde{u}$, and $K = \rho_0 P'(\rho_0)$

We obtain the linear acoustics equations

$$\tilde{p}_t + u_0\tilde{p}_x + K\tilde{u}_x = 0$$

$$\rho_0\tilde{u}_t + \tilde{p}_x + \rho_0 u_0\tilde{u}_x = 0$$

With $u_0 = 0$, this becomes the system (dropping the tildes):

$$q_t(x,t) + Aq_x(x,t) = 0$$

$$q = \begin{bmatrix} p \\ u \end{bmatrix}, \quad A = \begin{bmatrix} 0 & K \\ \frac{1}{\rho_0} & 0 \end{bmatrix}, \qquad \text{so} \quad \begin{array}{l} p_t + Ku_x = 0 \\ \rho_0 u_t + p_x = 0 \end{array}$$

Sound waves

$$q_t(x,t) + Aq_x(x,t) = 0 \qquad q = \begin{bmatrix} p \\ u \end{bmatrix}, \; A = \begin{bmatrix} 0 & K \\ \frac{1}{\rho_0} & 0 \end{bmatrix}$$

The acoustics equations must produce a solution with sound waves travelling in both directions.

We try a solution of the form $q(x,t) = \eta(x - st)$ and then compute the derivatives:

$$q_t(x,t) = -s\eta'(x - st), \qquad q_x(x,t) = \eta'(x - st)$$

So from $q_t(x,t) + Aq_x(x,t) = 0$ we get

$$A\eta'(x - st) = s\eta'(x - st)$$

implying that s is an *eigenvalue* and η' the corresponding *eigenvector* of the matrix A.

Can you calculate the sound speed?

This is an important key to the methods we will develop in this course.

Reminder: eigenvalues of a 2x2 matrix

Eigenvalues λ and eigenvectors r for a matrix A are found from $Ar = \lambda r$

For the matrix $A = \begin{bmatrix} a & b \\ c & d \end{bmatrix}$, the eigenvalues are:

$$\lambda^{1,2} = \frac{a+d}{2} \pm \frac{\sqrt{4bc + (a-d)^2}}{2}$$

Hence the acoustic equation matrix $A = \begin{bmatrix} 0 & K \\ \frac{1}{\rho} & 0 \end{bmatrix}$

has eigenvalues: $\lambda^{1,2} = \pm c = \pm\sqrt{\frac{K}{\rho}}$ and eigenvectors $r^{1,2} = \begin{bmatrix} \pm\sqrt{K\rho} \\ 1 \end{bmatrix}$

Important result: ***the eigenvalues are the wave speeds***. The eigenvectors express the relation between the components of the solution vector.

One way to look at linear acoustics:

$$p_t(x,t) + Ku_x(x,t) = 0$$

$$u_t(x,t) + \frac{1}{\rho} p_x(x,t) = 0$$

The unknown functions $p(x,t)$ and $u(x,t)$ are pressure and velocity; K and ρ are the material constants (bulk modulus and density). Differentiate the first with respect to t and the second with respect to x:

$$p_{tt} - Ku_{xt} = 0$$

$$u_{xt} - \frac{1}{\rho} p_{xx} = 0$$

Then add K times the second to the first:

$$p_{tt} - \frac{K}{\rho} p_{xx} = 0$$

This is the familiar **2nd order** wave equation, and it is hyperbolic. The solution gives waves travelling in both directions at velocity $c = \sqrt{\frac{K}{\rho}}$.

Another way to look at linear acoustics:

The acoustic equations are:

$$p_t(x,t) + K u_x(x,t) = 0$$
$$u_t(x,t) + \frac{1}{\rho} p_x(x,t) = 0$$

Express in matrix notation:

$$q_t(x,t) + A q_x(x,t) = 0 \qquad q = \begin{bmatrix} p \\ u \end{bmatrix}, \quad A = \begin{bmatrix} 0 & K \\ \frac{1}{\rho} & 0 \end{bmatrix}.$$

Resolve into the eigensystem: $Ar = \lambda r,$

with eigenvalues $\lambda^{1,2} = \pm c = \pm\sqrt{\frac{K}{\rho}}$ and eigenvectors $r^{1,2} = \begin{bmatrix} \pm\sqrt{K\rho} \\ 1 \end{bmatrix}.$

The eigenvalues are the wave speeds, and the eigenvectors express relations between the components of the solution q.

Acoustic impedance

The eigenvalues and eigenvectors of linear acoustics are

$$\lambda^{1,2} = \pm c_0 = \pm\sqrt{\frac{K}{\rho_0}} \qquad\qquad r^{1,2} = \begin{bmatrix} \pm\sqrt{K\rho_0} \\ 1 \end{bmatrix} = \begin{bmatrix} \pm\rho_0 c_0 \\ 1 \end{bmatrix}.$$

The quantity $Z_0 = \rho_0 c_0$ is commonly known as the *impedance* of the medium.

Wave-propagation methods

A conservation law

$$q_t + f(q)_x = 0$$

is locally linearised into the form

$$q_t + Aq_x = 0$$

The system is *hyperbolic* if the $m \times m$ matrix A is diagonalisable with eigenvalues

$$\lambda^1, \lambda^2, \ldots \lambda^m$$

The solution to the conservation law is the *superposition of waves propagating with velocities given by the eigenvalues.*

We will calculate these waves and use them to construct the solution.

Hyperbolicity Definition

The Jacobian matrix of a linear $m \times m$ system of partial differential equations:

$$A = f'(q) = \begin{bmatrix} \frac{\partial f^1}{\partial q^1} & \cdots & \frac{\partial f^1}{\partial q^m} \\ \vdots & \ddots & \vdots \\ \frac{\partial f^m}{\partial q^1} & \cdots & \frac{\partial f^m}{\partial q^m} \end{bmatrix}$$

A is diagonalisable if it has a complete set of eigenvectors/eigenvalues, *i.e.*

$$Ar^p = \lambda^p r^p \text{ for } p = 1,2,\ldots,m$$

(the vectors r^p must be nonzero)

Strongly hyperbolic:
the matrix A is diagonalisable and has real eigenvalues

Strictly hyperbolic:
the matrix A is diagonalisable and has *distinct* real eigenvalues

Weakly hyperbolic:
the matrix A is not diagonalisable but has real eigenvalues (not a complete set)

Hyperbolic systems lead to wave methods

Our hyperbolic sytem has the locally diagonalisable Jacobian A:

$$Ar^p = \lambda^p r^p \text{ for } p = 1,2,\ldots,m$$

The matrix $R = \left[r^1 \middle| r^2 \middle| \ldots \middle| r^m \right]$ is nonsingular. Forming

$$R^{-1}AR = \Lambda = \begin{bmatrix} \lambda^1 & & & \\ & \lambda^2 & & \\ & & \ddots & \\ & & & \lambda^m \end{bmatrix},$$

we can write our conservation law as

$$\begin{aligned} R^{-1}q_t + \left(R^{-1}\Lambda R\right)R^{-1}q_x &= 0 \\ w_t + \Lambda w_x &= 0 \end{aligned}$$

with $w(x,t) = R^{-1}q(x,t)$, and then we resolve it into:

$$w_t^p + \lambda^p w_x^p = 0 \text{ for } p = 1,2,\ldots m$$

Elastic waves in solids

These are also hyperbolic systems and may be solved by the same methods

P-waves in one dimension:

$$\varepsilon_t^{11} - u_x = 0$$
$$\rho u_t - \sigma_x^{11} = 0$$

S-waves in one dimension:

$$\varepsilon_t^{12} - \frac{1}{2} v_x = 0$$
$$\rho v_t - \sigma_x^{12} = 0$$

Higher dimensional systems are covered in Chapter 22 of Leveque; let me know soon if anyone is interested in doing such problems, otherwise I won't cover them in the course.

Electromagnetic waves

A plane electromagnetic wave propagating in the x direction has electric and magnetic field given by

$$\mathbf{E} = \begin{bmatrix} 0 \\ E^2(x,t) \\ 0 \end{bmatrix}, \quad \mathbf{B} = \begin{bmatrix} 0 \\ 0 \\ B^3(x,t) \end{bmatrix}$$

Maxwell's equations for this case reduce to:

$$E_t^2 + \frac{1}{\varepsilon\mu} B_x^3 = 0$$

$$B_t^3 + E_x^2 = 0$$

The eigenvalues are $\lambda^{1,2} = \pm c = \pm \frac{1}{\sqrt{\varepsilon\mu}}$ giving the speed of light in the medium.

Review: conservation law and advection

The fundamental conservation law in one spatial dimension, expressed in differential form, is:

$$q_t(x,t) + f(q(x,t))_x = 0.$$

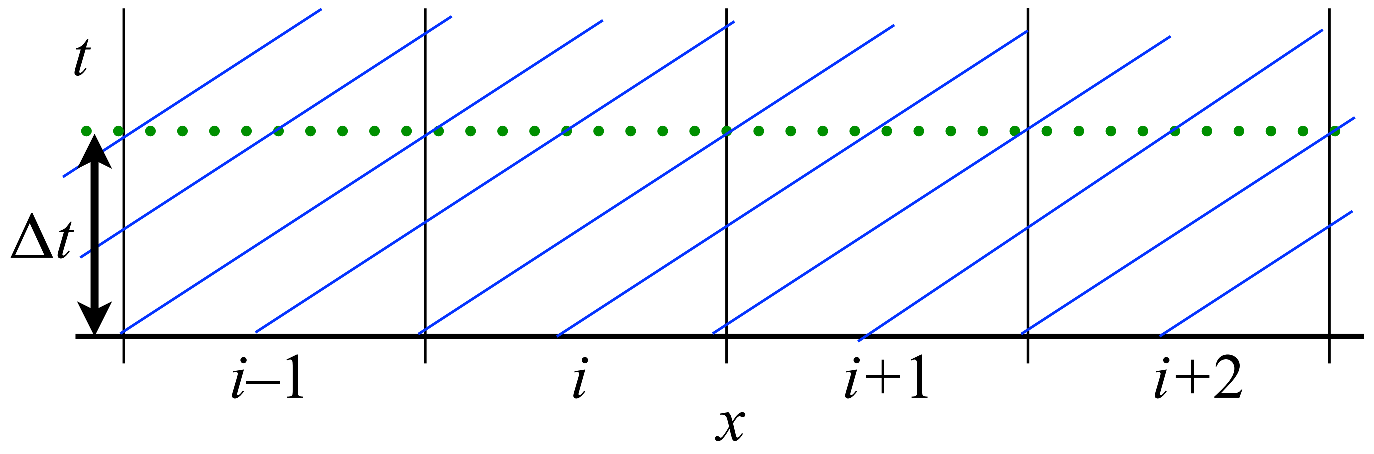
The advection equation, the simplest hyperbolic differential equation,

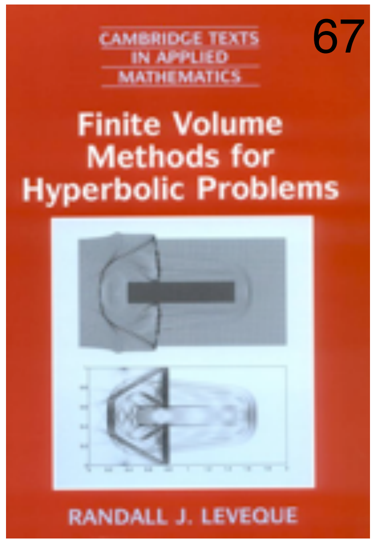
$$q_t(x,t) + uq_x(x,t) = 0,$$

is a conservation law with the flux function $f(x,t) = uq(x,t)$. Its solution is

$$q(x,t) = q(x-ut,0),$$

and this function is constant along rays in space-time (*characteristics*) with $x–ut$ = constant.




CAMBRIDGE TEXTS IN APPLIED MATHEMATICS
Finite Volume Methods for Hyperbolic Problems
RANDALL J. LEVEQUE

Assignment (due Monday 6 Sept):

Get the book! It's available at Akademika now.

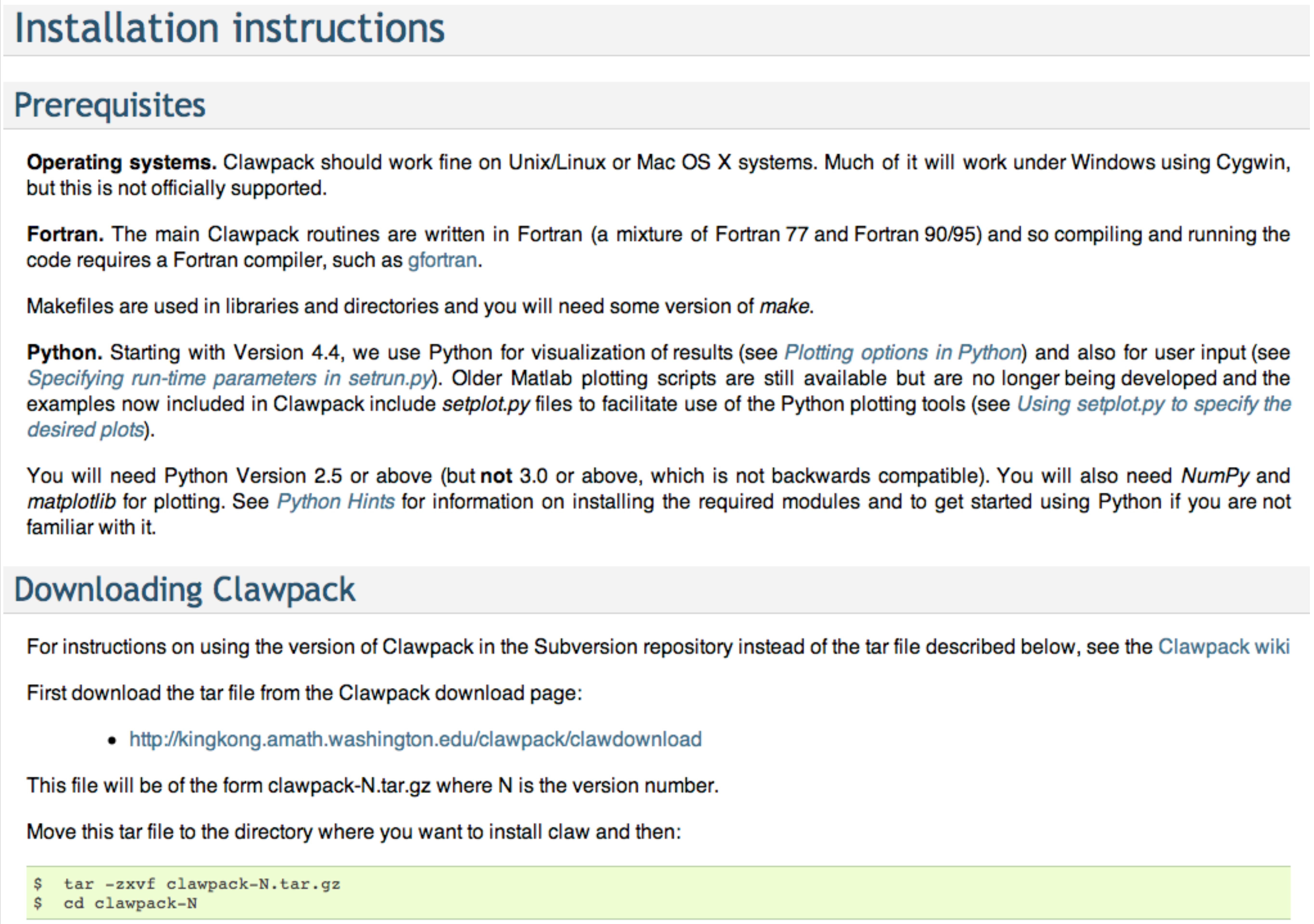
Download the *Errata* from
http://www.amath.washington.edu/~claw/book.html
and **apply the corrections**. Your copy may not need all of them since it's a newer printing.

Read all of Chapter 1, and Chapter 2 at least through section 2.11. Include 2.12 if you think you might want to do waves in elastic media (like seismic waves).

Work Exercises 2.2 (a and b parts), 2.4, and 2.8 (a part only) and hand them in to me by next Monday (the 24th). There is a file of sample solutions available at the above website, but please try them first on your own before consulting it.

Read the instructions for downloading Clawpack. These instructions are attached to the PDF of these slides, and also found on the Clawpack site at
http://kingkong.amath.washington.edu/clawpack/users/index.html.
Make sure you have access to a Unix/Linux/Mac OS X machine with a good development environment, including at a minimum Fortran 90/95 (gfortran, for example), and Python 2.5 or 2.6.

Download Clawpack and install it on your computer. Run some tests to make sure it is installed correctly. **Try to reproduce Fig. 3.1 (and 3.8) in Leveque.** Come see me if you get stuck; let's discuss difficulties this week.



Installation instructions

Prerequisites

Operating systems. Clawpack should work fine on Unix/Linux or Mac OS X systems. Much of it will work under Windows using Cygwin, but this is not officially supported.

Fortran. The main Clawpack routines are written in Fortran (a mixture of Fortran 77 and Fortran 90/95) and so compiling and running the code requires a Fortran compiler, such as gfortran.

Makefiles are used in libraries and directories and you will need some version of *make*.

Python. Starting with Version 4.4, we use Python for visualization of results (see *Plotting options in Python*) and also for user input (see *Specifying run-time parameters in setrun.py*). Older Matlab plotting scripts are still available but are no longer being developed and the examples now included in Clawpack include *setplot.py* files to facilitate use of the Python plotting tools (see *Using setplot.py to specify the desired plots*).

You will need Python Version 2.5 or above (but **not** 3.0 or above, which is not backwards compatible). You will also need *NumPy* and *matplotlib* for plotting. See *Python Hints* for information on installing the required modules and to get started using Python if you are not familiar with it.

Downloading Clawpack

For instructions on using the version of Clawpack in the Subversion repository instead of the tar file described below, see the Clawpack wiki

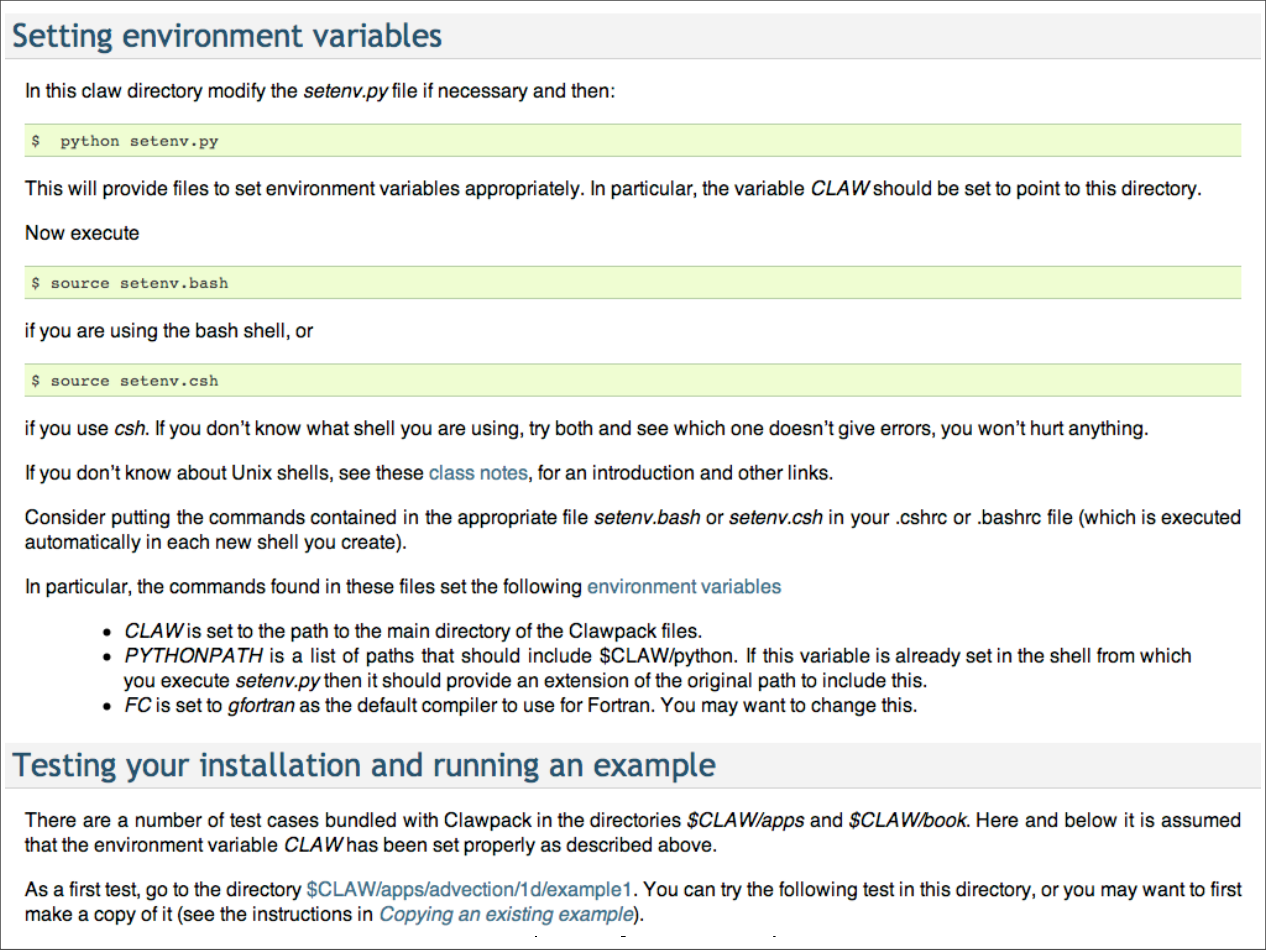
First download the tar file from the Clawpack download page:

- http://kingkong.amath.washington.edu/clawpack/clawdownload

This file will be of the form clawpack-N.tar.gz where N is the version number.

Move this tar file to the directory where you want to install claw and then:

```
$  tar -zxvf clawpack-N.tar.gz
$  cd clawpack-N
```



Setting environment variables

In this claw directory modify the *setenv.py* file if necessary and then:

```
$  python setenv.py
```

This will provide files to set environment variables appropriately. In particular, the variable *CLAW* should be set to point to this directory.

Now execute

```
$ source setenv.bash
```

if you are using the bash shell, or

```
$ source setenv.csh
```

if you use *csh*. If you don't know what shell you are using, try both and see which one doesn't give errors, you won't hurt anything.

If you don't know about Unix shells, see these class notes, for an introduction and other links.

Consider putting the commands contained in the appropriate file *setenv.bash* or *setenv.csh* in your .cshrc or .bashrc file (which is executed automatically in each new shell you create).

In particular, the commands found in these files set the following environment variables

- *CLAW* is set to the path to the main directory of the Clawpack files.
- *PYTHONPATH* is a list of paths that should include $CLAW/python. If this variable is already set in the shell from which you execute *setenv.py* then it should provide an extension of the original path to include this.
- *FC* is set to *gfortran* as the default compiler to use for Fortran. You may want to change this.

Testing your installation and running an example

There are a number of test cases bundled with Clawpack in the directories *$CLAW/apps* and *$CLAW/book*. Here and below it is assumed that the environment variable *CLAW* has been set properly as described above.

As a first test, go to the directory $CLAW/apps/advection/1d/example1. You can try the following test in this directory, or you may want to first make a copy of it (see the instructions in *Copying an existing example*).

The Makefiles are set up to do dependency checking so that in many application directories you can simply type:

```
$ make .plots
```

and the Fortran code will be compiled, data files created, the code run, and the results plotted automatically, resulting in a set of webpages showing the results.

However, if this is your first attempt to run a code, it is useful to go through these steps one at a time, both to understand the steps and so that any problems with your installation can be properly identified.

You might want to start by examining the Makefile. This sets a number of variables, which at some point you might need to modify for other examples, see *Clawpack Makefiles* for more about this. At the bottom of the Makefile is an *include* statement that points to a common Makefile that is used by most applications, and where all the details of the make process can be found.

To compile the code, type:

```
$ make .exe
```

If this gives an error, see *Trouble running "make .exe"*.

This should compile the example code (after first compiling the required library routines) and produce an executable named *xclaw* in this directory.

Before running the code, it is necessary to also create a set of data files that are read in by the Fortran code. This can be done via:

```
$ make .data
```

If this gives an error, see *Trouble running "make .data"*.

This uses the Python code in *setrun.py* to create data files that have the form **.data*. For the 1d advection example, two files are created, *claw.data* and *setprob.data*. The file *claw.data* contains standard run-time parameters of Clawpack (such as the number of grid cells *mx*, indications of what method to use, what boundary conditions to impose, etc.). The file *setprob.data* typically contains parameters specific to a particular application, in this case the advection velocity *u*.

In Clawpack 4.3 and earlier versions, the user would modify the *claw.data* and *setprob.data* files directly. Starting with Clawpack 4.4, the recommended approach is to only modify the Python function *setrun* defined in the file *setrun.py*, and use "make .data" to create the **.data* files. See *Specifying run-time parameters in setrun.py* for more details.

Once the executable and the data files all exist, we can run the code. The recommended way to do this is to type:

```
$ make .output
```

If this gives an error, see *Trouble running "make .output"*.

One could run the code by typing "./xclaw", but using the make option has several advantages. For one thing, this checks dependencies to make sure the executable and data files are up to date, so you could have typed "make .output" without the first two steps above.

Also, before running the code a subdirectory *_output* is created and the output of the code (often a large number of files) is directed to this subdirectory. This is convenient if you want to do several runs with different parameter values and keep the results organized. After the code has run you can rename the subdirectory, or you can modify the variable *OUTDIR* in the Makefile to direct results to a different directory. See *Clawpack Makefiles* for more details. Copies of all the data files are also placed in the output directory for future reference.

If the code runs successfully, you should see output like the following:

```
Reading data file, first 5 lines are comments: claw.data
 running...

Reading data file, first 5 lines are comments: setprob.data
CLAW1EZ: Frame    0 output plot files done at time t =  0.0000D+00

CLAW1... Step   1   Courant number = 5.000  dt =  0.1000D+00  t =  0.1000D+00
CLAW1 rejecting step... Courant number too large
CLAW1... Step   1   Courant number = 0.900  dt =  0.1800D-01  t =  0.1800D-01
CLAW1... Step   2   Courant number = 0.900  dt =  0.1800D-01  t =  0.3600D-01
CLAW1... Step   3   Courant number = 0.900  dt =  0.1800D-01  t =  0.5400D-01
CLAW1... Step   4   Courant number = 0.900  dt =  0.1800D-01  t =  0.7200D-01
CLAW1... Step   5   Courant number = 0.900  dt =  0.1800D-01  t =  0.9000D-01
CLAW1... Step   6   Courant number = 0.500  dt =  0.1000D-01  t =  0.1000D+00
CLAW1EZ: Frame    1 output plot files done at time t =  0.1000D+00

--- etc --- etc ---

CLAW1EZ: Frame    9 output plot files done at time t =  0.9000D+00

CLAW1... Step   1   Courant number = 0.900  dt =  0.1800D-01  t =  0.9180D+00
CLAW1... Step   2   Courant number = 0.900  dt =  0.1800D-01  t =  0.9360D+00
CLAW1... Step   3   Courant number = 0.900  dt =  0.1800D-01  t =  0.9540D+00
CLAW1... Step   4   Courant number = 0.900  dt =  0.1800D-01  t =  0.9720D+00
CLAW1... Step   5   Courant number = 0.900  dt =  0.1800D-01  t =  0.9900D+00
CLAW1... Step   6   Courant number = 0.500  dt =  0.1000D-01  t =  0.1000D+01
CLAW1EZ: Frame   10 output plot files done at time t =  0.1000D+01
```

If you don't like seeing output from every time step, you can suppress this by setting *verbosity = 0* in the file *setrun.py*. You might try doing that and then typing:

```
$ make .output
```

It should recreate the data files and rerun the code, with less output along the way.

If the code runs properly, the subdirectory *_output* should contain the following files:

```
claw.data   fort.q0003  fort.q0008  fort.t0002  fort.t0007
fort.info   fort.q0004  fort.q0009  fort.t0003  fort.t0008
fort.q0000  fort.q0005  fort.q0010  fort.t0004  fort.t0009
fort.q0001  fort.q0006  fort.t0000  fort.t0005  fort.t0010
fort.q0002  fort.q0007  fort.t0001  fort.t0006  setprob.data
```

The *fort.info* file contains information about the run just completed. The files with names of the form *fort.t000N* and *fort.q000N* contain the computed results for Frame *N*. See *fortfiles* for more information about the contents of these files.

Normally you will not want to examine these files directly, but instead will use a plotting tool to plot the results.

Plotting the results. Once the code has run and the files listed above have been created, there are several options for plotting the results.

To try the Python tools, type:

```
$ make .plots
```

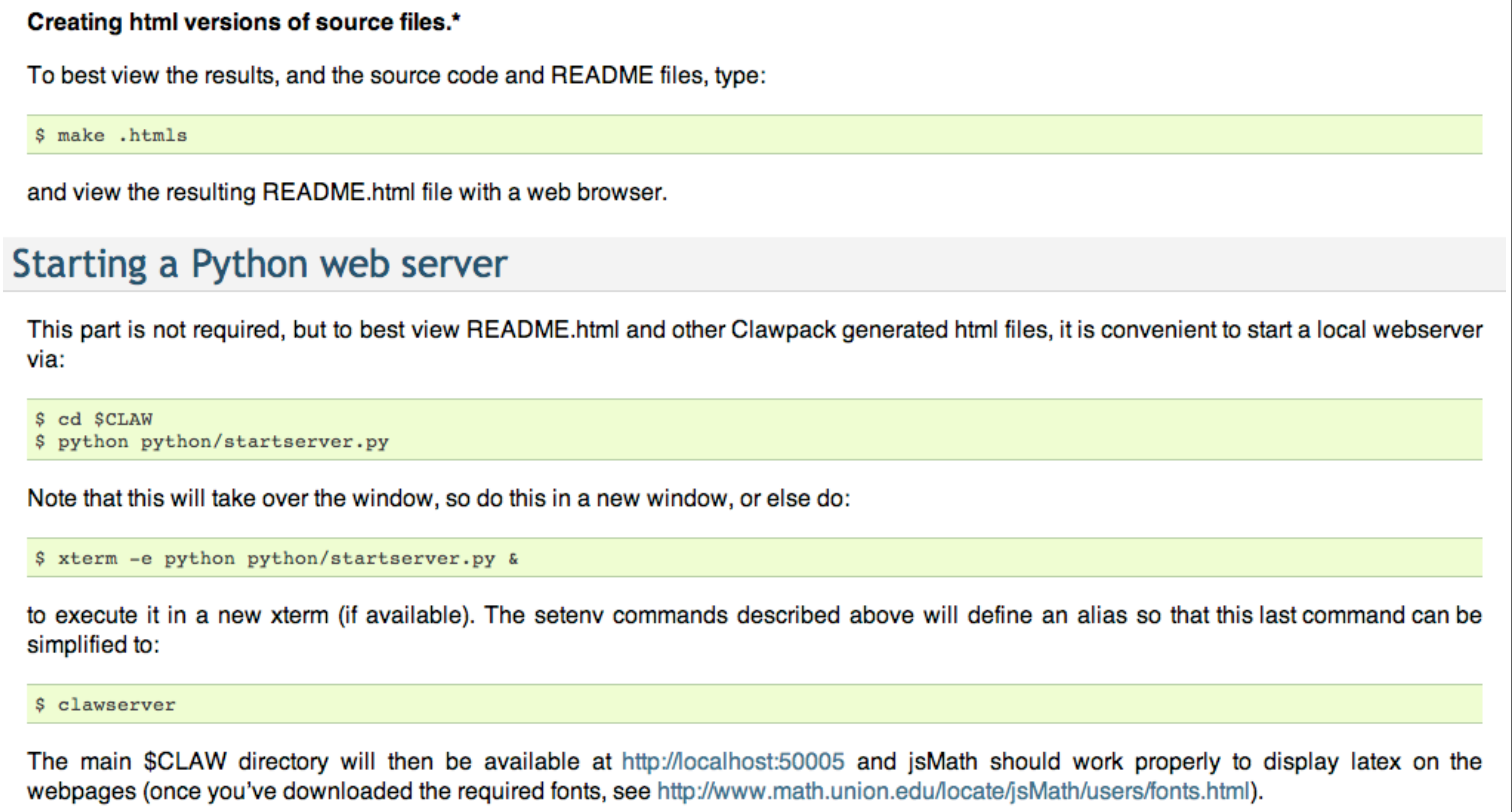
If this gives an error, see *Trouble running "make .plots"*.

If this works, it will create a subdirectory named *_plots* that contains a number of image files (the **.png* files) and a set of html files that can be used to view the results from a web browser. See *plotting_makeplots* for more details.

An alternative is to view the plots from an interactive Python session, as described in the section *Interactive plotting with Iplotclaw*.

If you wish to use Matlab instead, see *Plotting using Matlab*.

Other visualization packages could also be used to display the results, but you will need to figure out how to read in the data. See *fortfiles* for information about the format of the files produced by Clawpack.



Creating html versions of source files.*

To best view the results, and the source code and README files, type:

```
$ make .htmls
```

and view the resulting README.html file with a web browser.

Starting a Python web server

This part is not required, but to best view README.html and other Clawpack generated html files, it is convenient to start a local webserver via:

```
$ cd $CLAW
$ python python/startserver.py
```

Note that this will take over the window, so do this in a new window, or else do:

```
$ xterm -e python python/startserver.py &
```

to execute it in a new xterm (if available). The setenv commands described above will define an alias so that this last command can be simplified to:

```
$ clawserver
```

The main $CLAW directory will then be available at http://localhost:50005 and jsMath should work properly to display latex on the webpages (once you've downloaded the required fonts, see http://www.math.union.edu/locate/jsMath/users/fonts.html).

Next: Riemann Problem (Ch 3)