Scalar hyperbolic equations

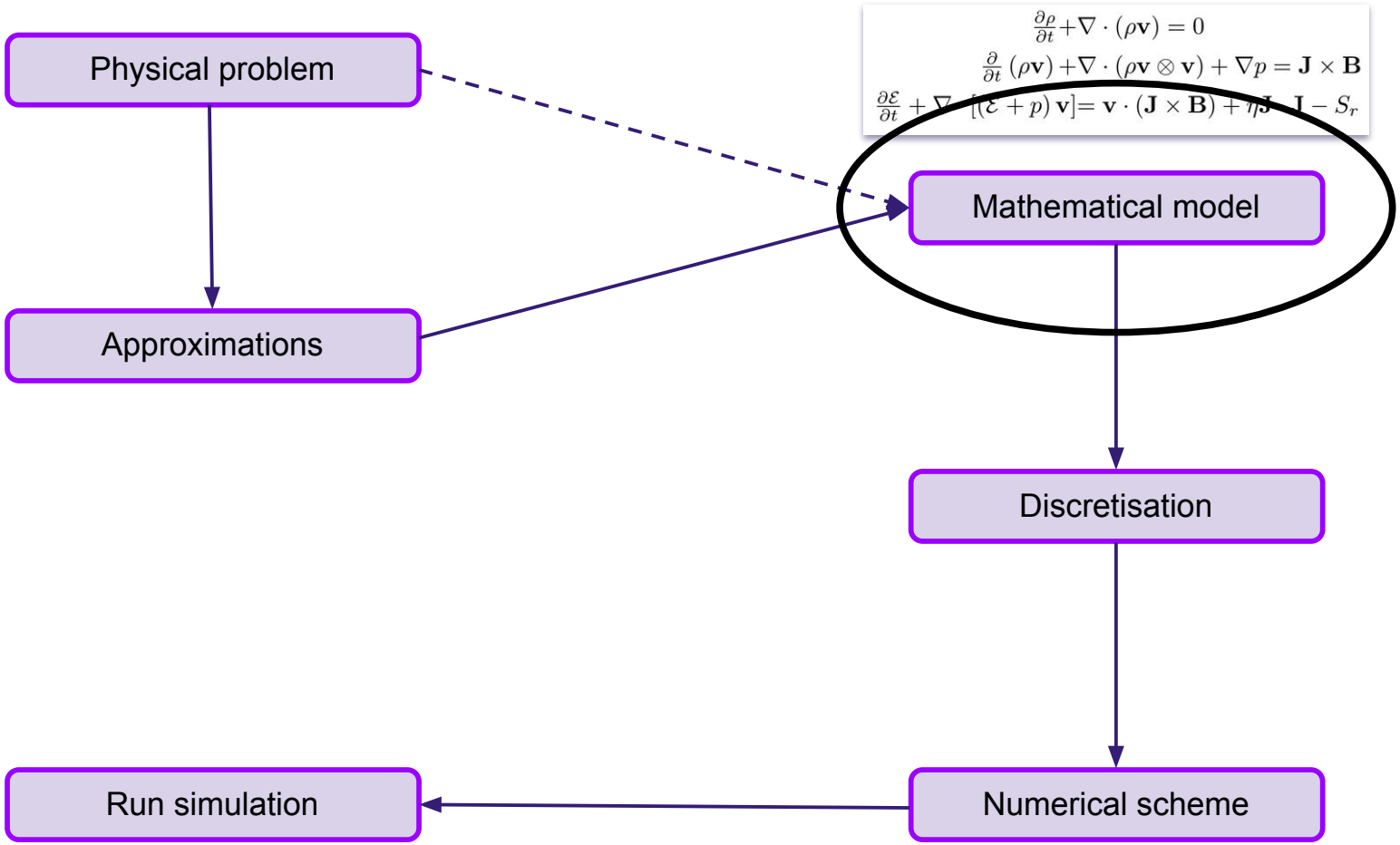
Stephen Millmore and Nikos Nikforakis

Laboratory for Scientific Computing, University of Cambridge

Outline

Outline

Properties of hyperbolic equations



Properties of hyperbolic equations

- We have so far discussed theory in the context of general hyperbolic systems of equations

$$\frac{\partial \mathbf{u}}{\partial t} + \nabla \cdot \mathbf{f}(\mathbf{u}) = 0$$

- To continue with developing a mathematical understanding of these equations, it is useful to start considering examples
- We shall start looking at specific variables, $\mathbf{u}$ and flux functions, $\mathbf{f}(\mathbf{u})$
- Starting simple, we will build up to an understanding of the principles behind non-linear coupled systems of equations

Properties of hyperbolic equations

- For each example we look at, we will consider:
  - Different forms of the governing PDEs
  - The hyperbolicity of the system
  - The **characteristics** (or wave fronts) of the problem
  - The admissible waves in the solution to the problem
- These considerations will often make use of **Riemann problems**, a class of mathematical problem we shall cover in detail
- Throughout most of this course, we shall focus on **one-dimensional** PDEs

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

- Not only does this help ensure the maths fits on the slides, the techniques we consider are directly representative of the numerical methods required to solve the complex examples shown at the start of this course

Riemann problems

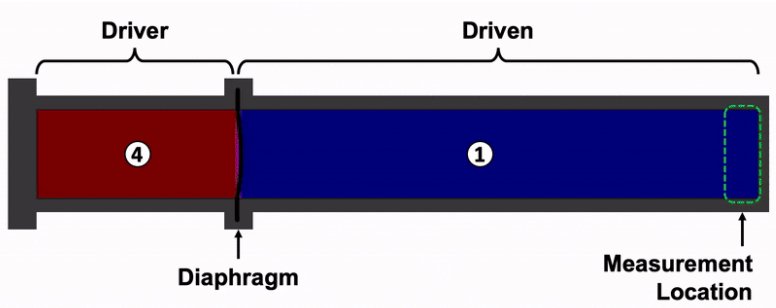
- The Riemann problem is a form of **initial value problem**, comprising a conservation equation **and** a piecewise-constant initial state with a **single** discontinuity located at $x < x_0$

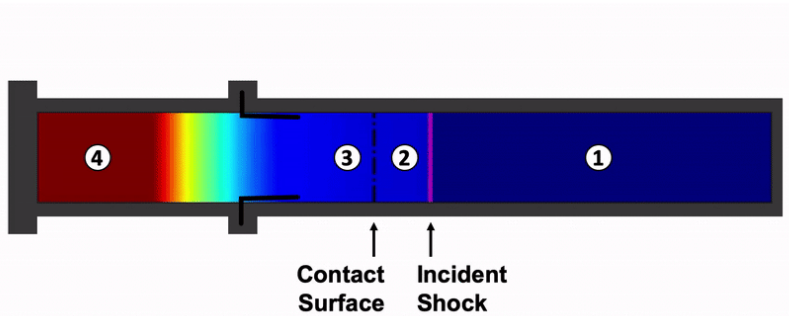
$$\frac{\partial \mathbf{u}}{\partial t} + \frac{\partial \mathbf{f}(\mathbf{u})}{\partial x} = 0, \qquad \mathbf{u}(x,0) = \begin{cases} \mathbf{u}_L & x < x_0 \\ \mathbf{u}_R & x > x_0 \end{cases}$$

- Riemann problems are important for both understanding and numerically solving PDEs
- In many complex systems, a non-trivial **exact solution** can be found for the Riemann problem, and one which contains all possible behaviour of the PDE (exact solutions for these systems are rare)
- Because they have an exact solution, Riemann problems make excellent **validation cases** for numerical methods - we can test whether our method gets the answer we expect

Riemann problems and shock tube tests

- Sometimes Riemann problems are also known as **shock tube tests**
- This comes from experimental applications
- A shock tube is a descriptive name - a tube that is filled with gas (or liquid) such that a shock wave is generated
- Typically this is achieved by setting up a Riemann problem - one part of the tube is an ambient constant state, and the other is a high-pressure, or high-density, constant state

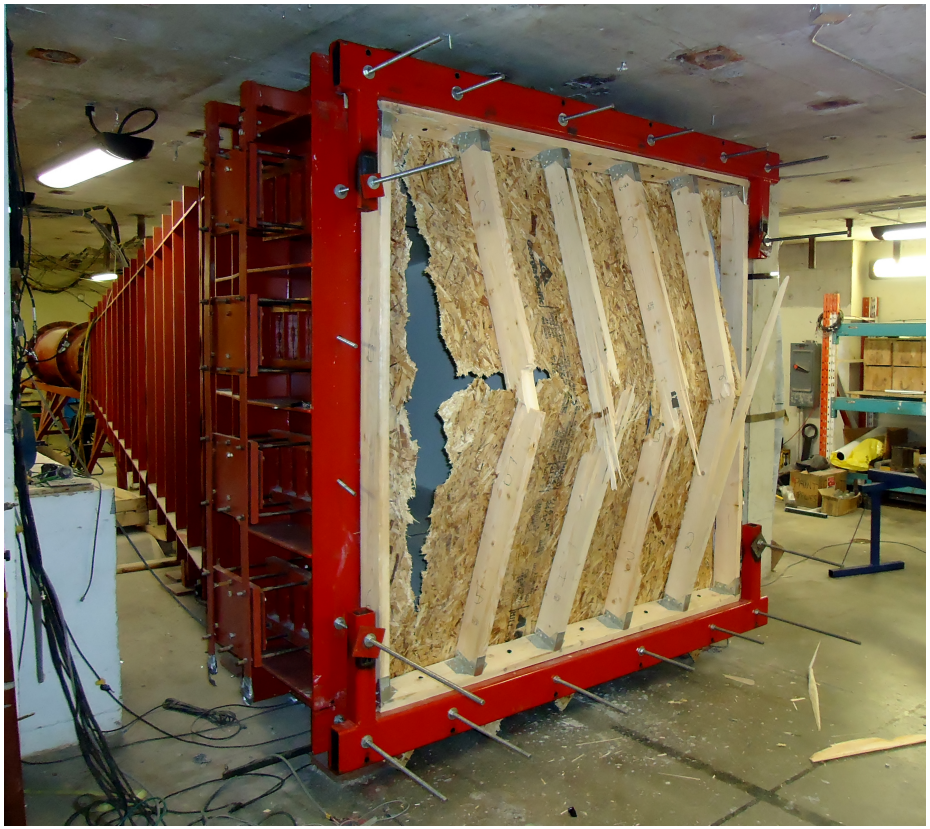




Source: Hanson Research group, Stanford Laboratory

Shock tube tests

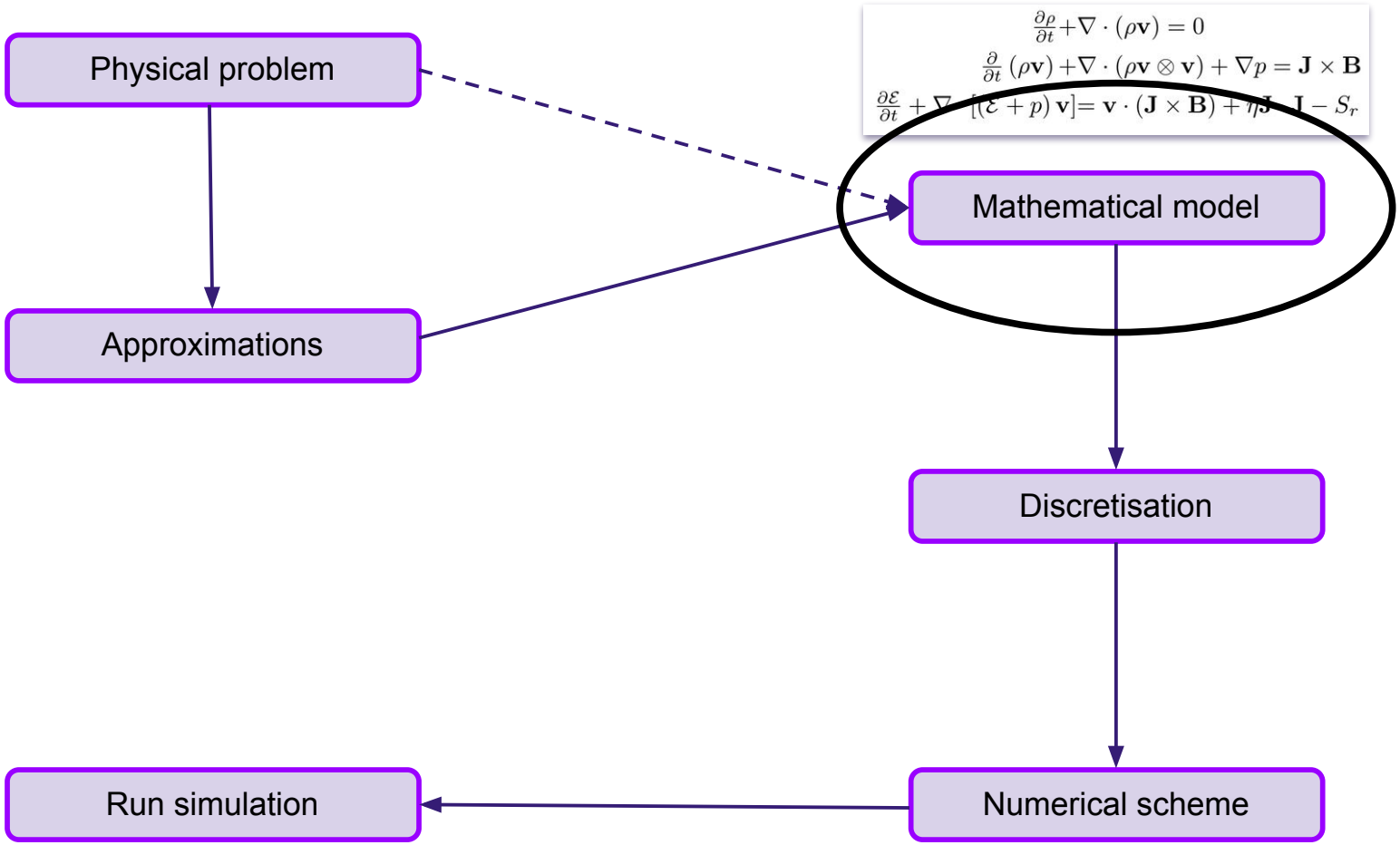
- Shock tubes are not small...
- And, unsurprisingly, deliberately creating large shock waves is a little destructive
- The size means that behaviour at the **boundaries** of the tube will not affect most of the **bulk flow**
- By the end of this course, we will be simulating shock tubes



Source: wikipedia

Outline

The linear advection equation



The linear advection equation

$$\frac{\partial u}{\partial t} + a\frac{\partial u}{\partial x} = 0, \qquad \frac{\partial u}{\partial t} + \frac{\partial (au)}{\partial x} = 0$$

- Here, u is a **scalar** variable, and $a = \text{const}$
- Recall this was a constant-temperature, constant-velocity reduction of the Navier-Stokes equations
- This equation is a very simple PDE, advection equation is the common name in the hyperbolic equations literature, elsewhere convection or transport equation are more popular
- However, these names are not used consistently and is sometimes seen under this name with additional diffusion terms (second order derivatives)

The linear advection equation - hyperbolicity

$$\frac{\partial u}{\partial t} + \frac{\partial (au)}{\partial x} = 0$$

- The advection equation is straightforward to identify as hyperbolic
- Recall that the properties of the Jacobian, $\partial \mathbf{f} / \partial \mathbf{u}$, classify the equation
- For scalar equations, the Jacobian is also a scalar, in this case a
- A scalar is its own eigenvalue, and its own single-entry diagonal matrix
- In other words - we know that the advection equation is hyperbolic just by looking at it (providing a is real)

The linear advection equation - characteristics

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

- It is equally easy to identify the characteristic form for this equation - the form given above
- We now consider what this means
- An equation in characteristic form can be used to transform a partial differential equation to a system of ordinary differential equations
- This **will** help us identify the behaviour of the PDE, and **might** even allow us to solve it analytically

The method of characteristics

- The idea behind this method is that for a PDE, we can identify variables along which the characteristic variables, $\boldsymbol{\mathcal{V}}$ are constant

$$\frac{\mathrm{d}}{\mathrm{d}s}\boldsymbol{\mathcal{V}}\left(x(s),t(s)\right)=0$$

- We use the chain rule, noting that in this case, we have a scalar $\mathcal{V}$

$$\frac{\partial}{\partial s}\mathcal{V}\left(x(s),t(s)\right)=\frac{\partial\mathcal{V}}{\partial x}\frac{\mathrm{d}x}{\mathrm{d}s}+\frac{\partial\mathcal{V}}{\partial t}\frac{\mathrm{d}t}{\mathrm{d}s}=0$$

- This can be compared to our original equation,

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

The method of characteristics

$$\frac{\partial \mathcal{V}}{\partial x}\frac{\mathrm{d}x}{\mathrm{d}s} + \frac{\partial \mathcal{V}}{\partial t}\frac{\mathrm{d}t}{\mathrm{d}s} = 0 \qquad \frac{\partial u}{\partial t} + a\frac{\partial u}{\partial x} = 0$$

- We already know that for the linear advection equation, $\mathcal{V} = u$
- It is then easy to find relationships between t and s, and x and s

$$\frac{\mathrm{d}t}{\mathrm{d}s} = 1 \quad \text{and} \quad \frac{\mathrm{d}x}{\mathrm{d}s} = a$$

- These are two ODEs, which, with initial conditions, are straightforward to solve
- Without loss of generality, we can say that $t(0) = 0$, meaning the first of these ODEs gives

$$t = s$$

The method of characteristics

$$t = s \quad \text{and} \quad \frac{\mathrm{d}x}{\mathrm{d}s} = a$$

- We are interested in solving the characteristic equation from an arbitrary position in space, giving us an initial condition for the second ODE, $x(0) = x_0$
- We then have

$$\frac{\mathrm{d}x}{\mathrm{d}s} = \frac{\mathrm{d}x}{\mathrm{d}t} = a \implies x = at + x_0$$

- Here we note that in the general case, we may not be able to solve this ODE analytically

The method of characteristics

- For the linear advection equation, the method of characteristics now gives us one more equation we can solve

$$\frac{\mathrm{d}u}{\mathrm{d}s} = 0$$

- Note that although we established $s = t$, this gives us $u = u(x(t), t)$, hence

$$\frac{\mathrm{d}u}{\mathrm{d}t} \neq \frac{\partial u}{\partial t}$$

- Our initial data for u is an arbitrary function, $u(0) = f(x_0)$, meaning the solution to the ODE is

$$u(x(s), t(s)) = u(x(t), t) = f(x_0)$$

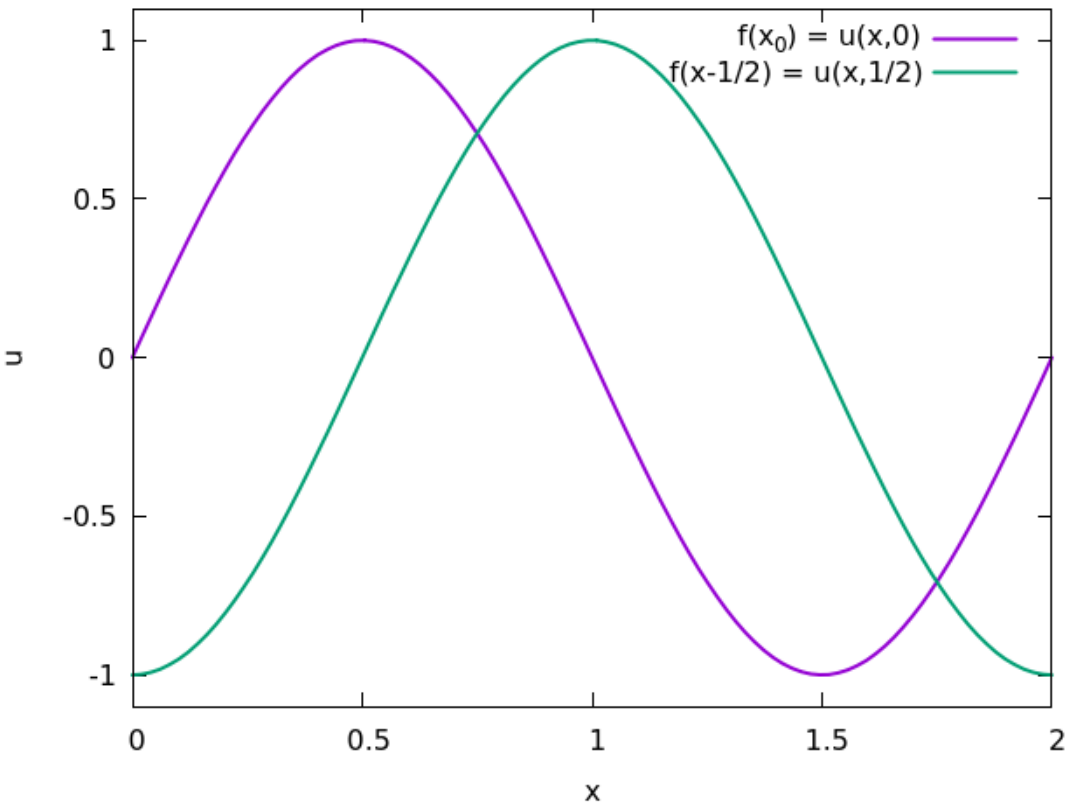
- We have also established that $x = at + x_0$, meaning

$$u(x, t) = f(x - at)$$

- Given a function $f(x_0) = u(x, 0)$, we can plot the solution at any point in time - we have solved the PDE

An example solution

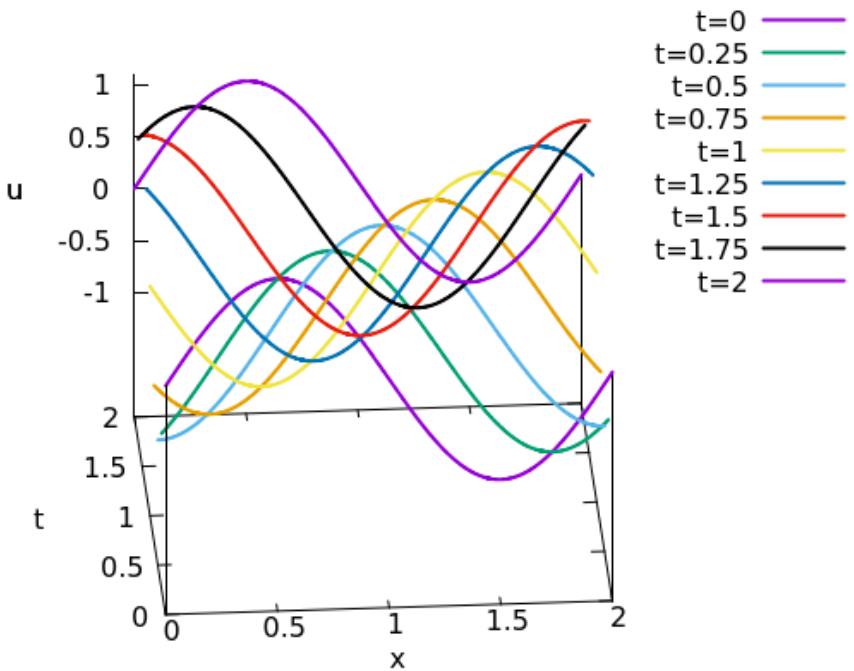
- $u(x, 0) = \sin(\pi x)$, $a = 1$

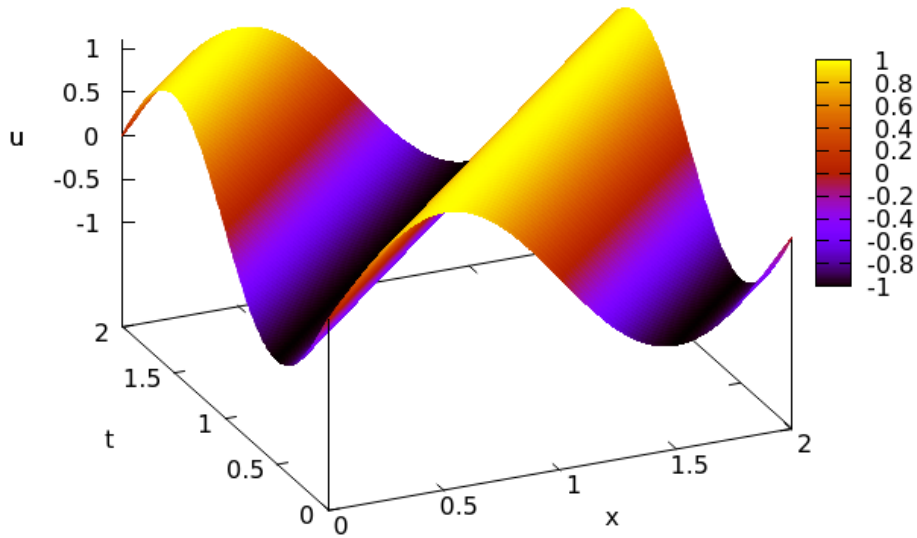


- Note: we have specified behaviour at $x = 0$ and $x = 2$ - **boundary conditions**

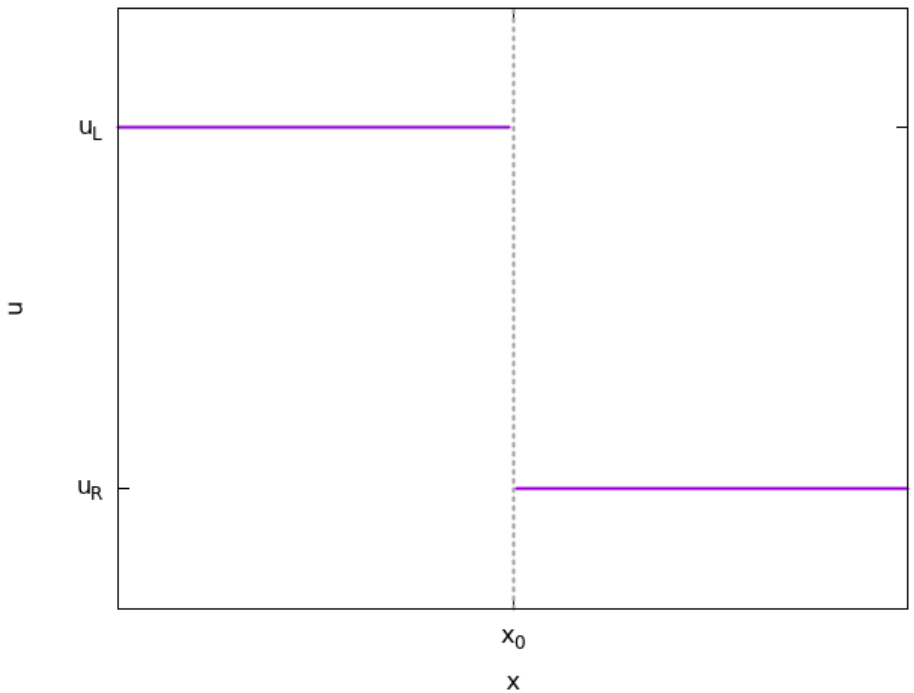
An example solution

- $u(x, 0) = \sin(\pi x)$, $a = 1$
- It is often useful to visualise data by plotting with space and time as two axes, to see how a solution evolves





The Riemann problem for the advection equation



- The advection equation is a conservation equation

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

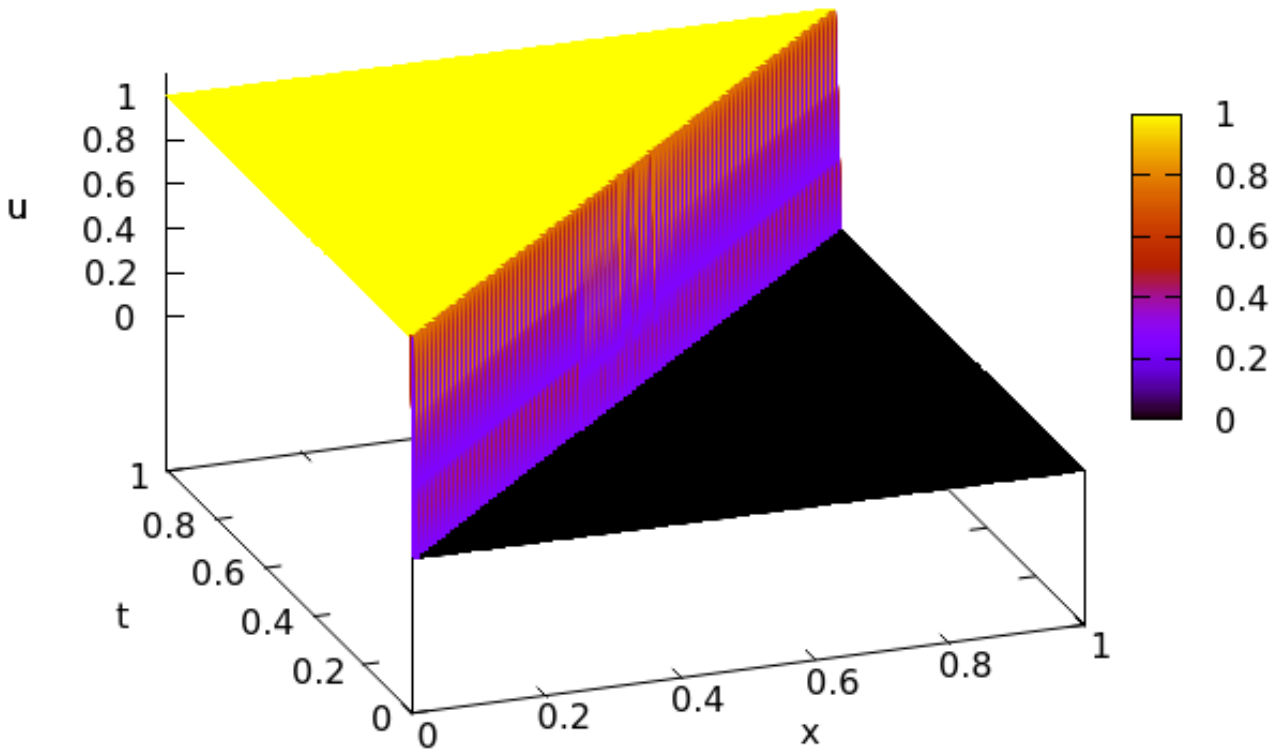
- If we give it initial data:

$$u(x,0) = u_0(x) = \begin{cases} u_L & x < x_0 \\ u_R & x > x_0 \end{cases}$$

then we have a Riemann problem

- As we mentioned, Riemann problems often have exact solutions, and this is fairly obvious for the advection equation...

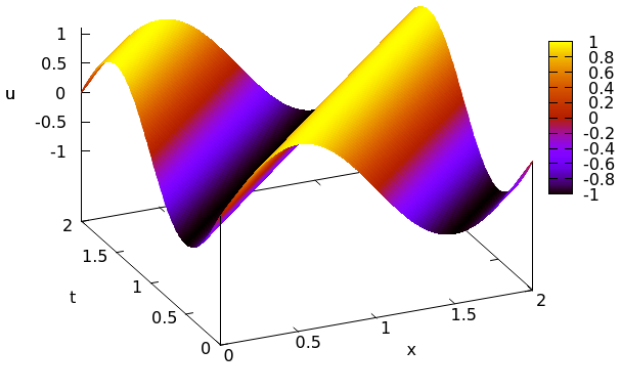
Riemann problem solution for the advection equation

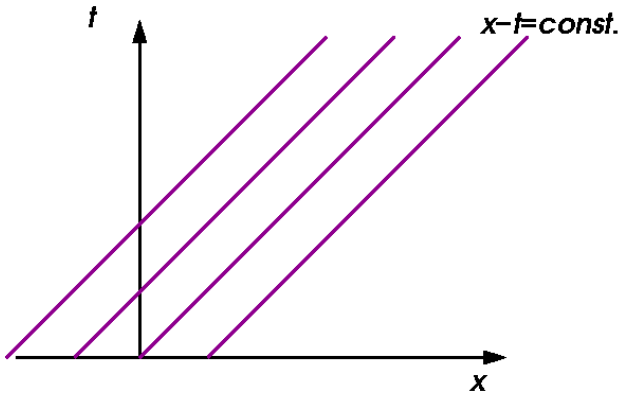


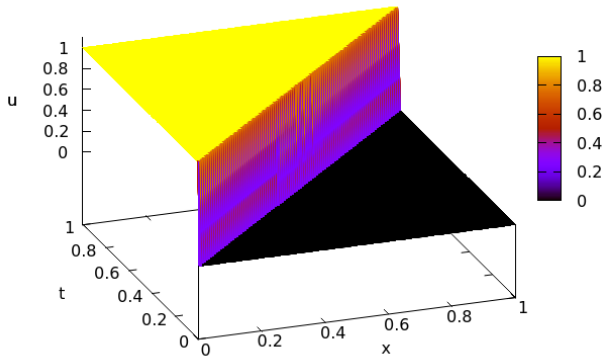
- Note - other hyperbolic equations will show a bit more interesting behaviour

The x-t diagram

- This is a common way of representing solutions to hyperbolic PDEs which **may** be helpful in visualising the solution
- For initial data at a given point in space, a curve is plotted on a time axis along which this variable is constant

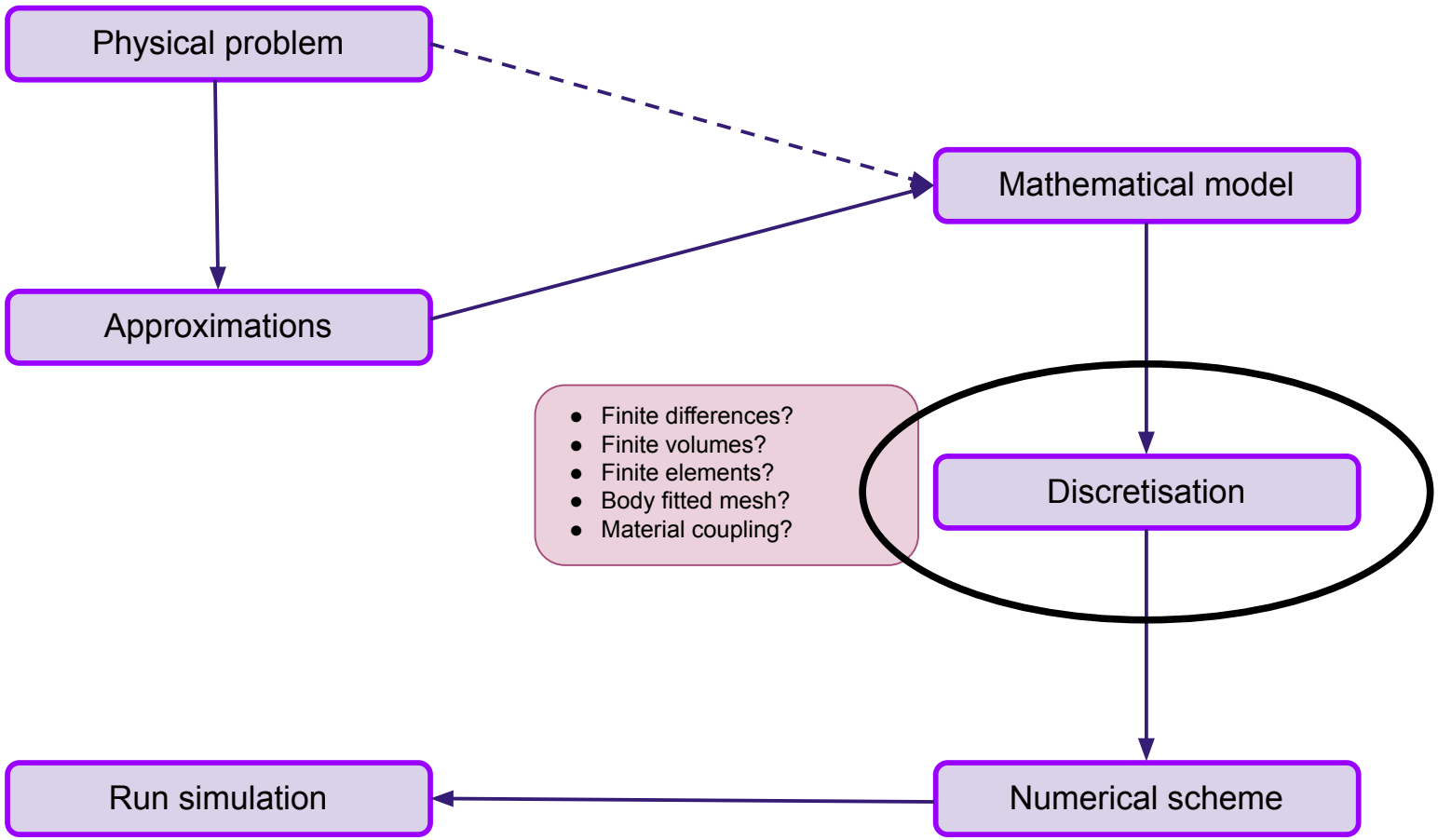






Outline

Numerical discretisation



Why do we want to solve this equation numerically?

- Throughout this course, we will consider numerical solutions to partial differential equations, using the advection equation as our first example
- A numerical solution is an approximation made using a **discrete representation** of the quantities being solved
- As a result, we will only ever know the solution at discrete points in time and space
- Since we already know the solution to the advection equation, this is, of course, unnecessary
- However, the same techniques for solving this equation can be used to solve systems where we do not know the solution analytically
- When we use a numerical method to solve something **to which we already know the answer**, we are **validating** our implementation of the method

Space discretisation and equation discretisation

- In a discrete representation of a PDE, these are two related concepts
- Discretisation of space (and time) is often known as **meshing** or **gridding**, breaks up a continuum domain into discrete units
- Equation discretisation then describes the PDE, and the quantities being evolved, with respect to this underlying mesh, and is often known as the **numerical method**
- In some cases, numerical methods and meshing are directly linked
- Here, we focus first on the spatial discretisation, in particular for non-linear hyperbolic equations

Space discretisation and equation discretisation

- Perhaps the first choice we have to make is whether we use a Lagrangian or an Eulerian framework
  - **Lagrangian**: Consider units of fixed '**mass**' which move with material flow, and change in shape
  - **Eulerian**: Consider units of fixed **volume**, through which material moves
- Which is the best for our application?
- A related question - if you pick up any of the recommended books at the start of this course, they will go into very little detail about how space is discretised - why?
- To answer this second question, we need to know about the behaviour permitted in the solution - this is something we will cover in more detail later

Non-linear compressive waves

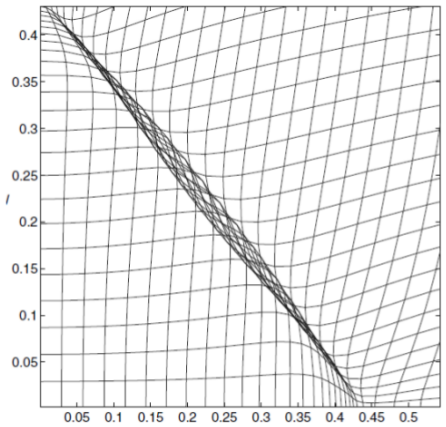
- Some of these concerns are somewhat irrelevant for the advection equation, and we want to start implementing numerical solutions for simple situations quickly
- However, we still want to do so such that they are useful for the non-linear compressible systems we are aiming towards
- Therefore, a couple of features of these more-complex systems that we need to be aware of now, before we chose our discretisation approach

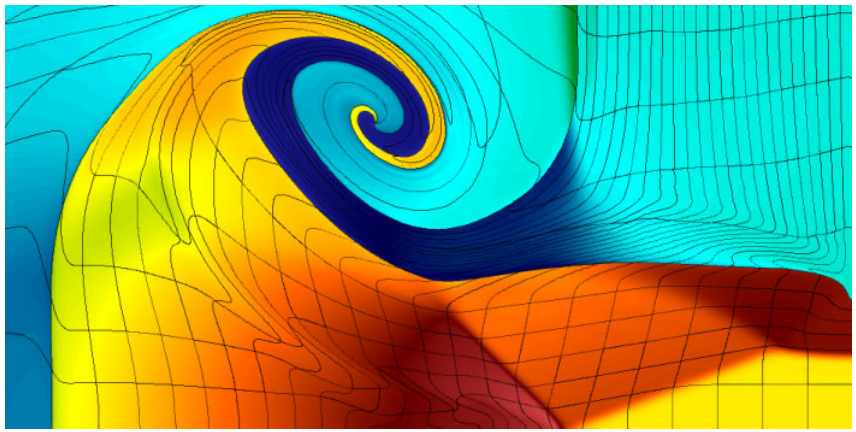
1. **Compression and expansion** - shock waves and expansion waves
2. **Vorticity** - $\nabla \cdot \mathbf{v} \neq 0$ means vortices are able to form

- Do these pose a problem for some methods of discretisation?

Distorted mesh

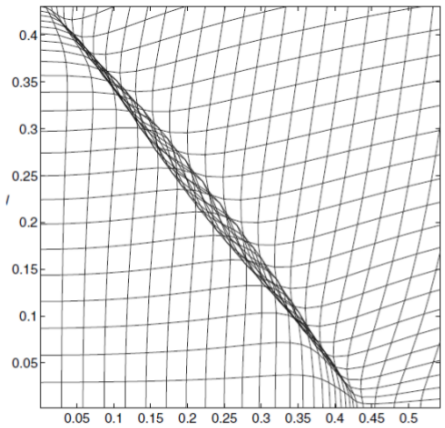
- Compression, expansion and vorticity certainly have an impact for a Lagrangian mesh

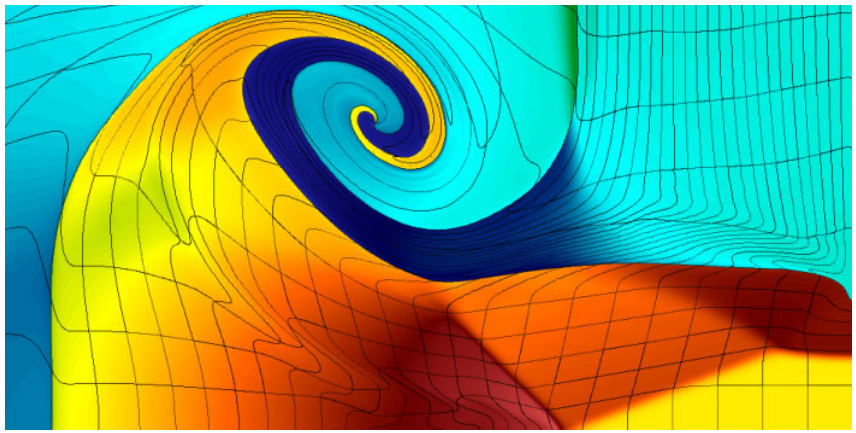




- Lagrangian meshes can suffer from very small volumes (values very close to zero) or very distorted ones (mapping techniques fail)
- These can be dealt with, but might require **re-meshing techniques**, which can be computationally expensive

Effect of volume size

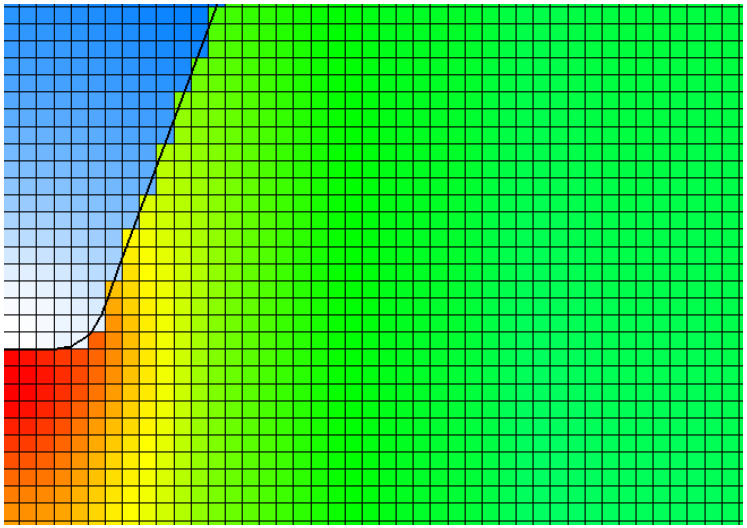


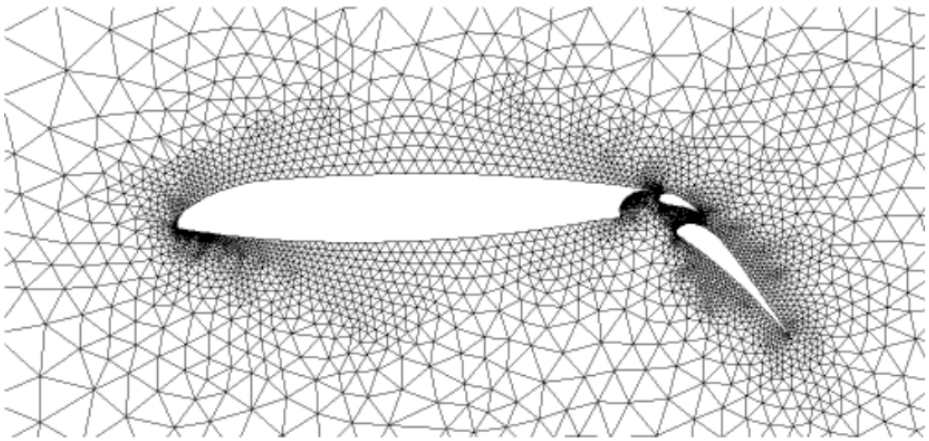


- Another aspect of our non-linear system of equations is that numerical techniques which solve these equations are dominated by the **smallest unit of the mesh**
- Affects both efficiency and accuracy
- Eulerian meshes remain unchanged in structure by the behaviour of the flow
- They are most commonly encountered method for equations and applications presented in this course

Structured or unstructured mesh?

- Within an Eulerian framework, you still have choices on how to make your mesh
- This could range from a simple Cartesian mesh to a complex body-fitting structure
- The restrictions based on the smallest unit of the mesh are relevant again here
- For non-linear hyperbolic equations, a regular Cartesian mesh is commonly encountered

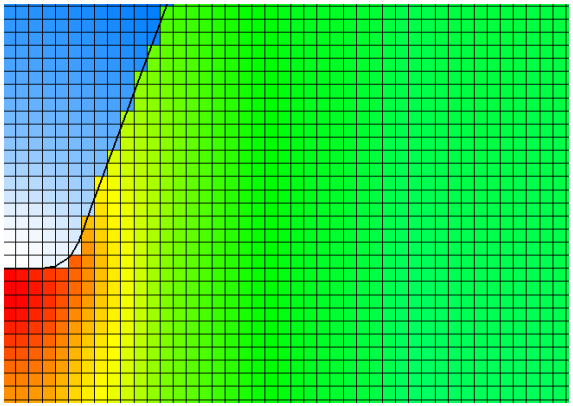




Advantages of a regular, Eulerian, Cartesian mesh

1. **Unpredictable behaviour** - it is not always possible to predict where a non-linear feature will arise - a regular mesh means it is not an area of low accuracy
    - Of course, there could be disadvantages to having high-accuracy everywhere
2. **Connectivity** - behaviour in one mesh unit is dependent on its neighbours, and our numerical method will need to access this information frequently - identifying these neighbours is straightforward in a computational storage structure
    - Connectivity helps coding as well as efficiency
3. **Simplicity** - Easy to set up, and assess quality, based on length-scales of the problem of interest, and does not need recomputing
    - Useful when first implementing a numerical method

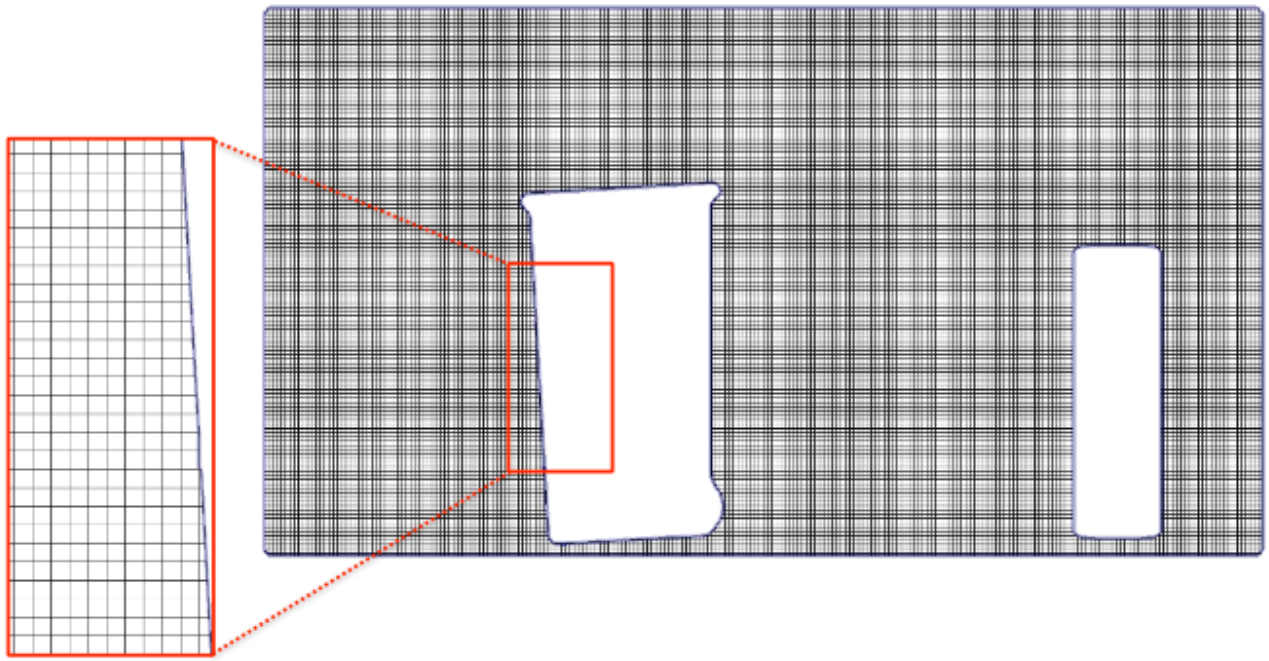
Regular, Eulerian, Cartesian meshes



- The simple structure of a regular Cartesian mesh means that it is frequently referred to as a **grid**
- But how do we apply a Cartesian mesh to a non-rectangular body?
- Could a uniform mesh be inefficient sometimes?

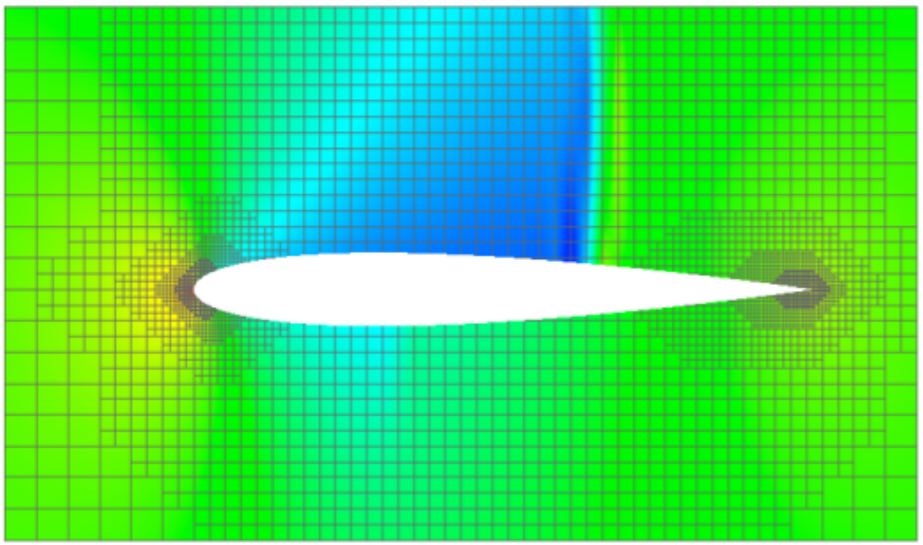
Immersed boundaries and cut cells

- No boundary is truly rigid, we could include a material model **inside the body** as well as outside (see "Multiphysics Modelling for Four States of Matter")
- This might be inefficient - complex geometries can be incorporated within a regular Cartesian grid through **cut cell methods**
- Nandan Gokhale will go into more detail in "Simulation of Matter under Extreme Conditions"



Adaptive mesh refinement

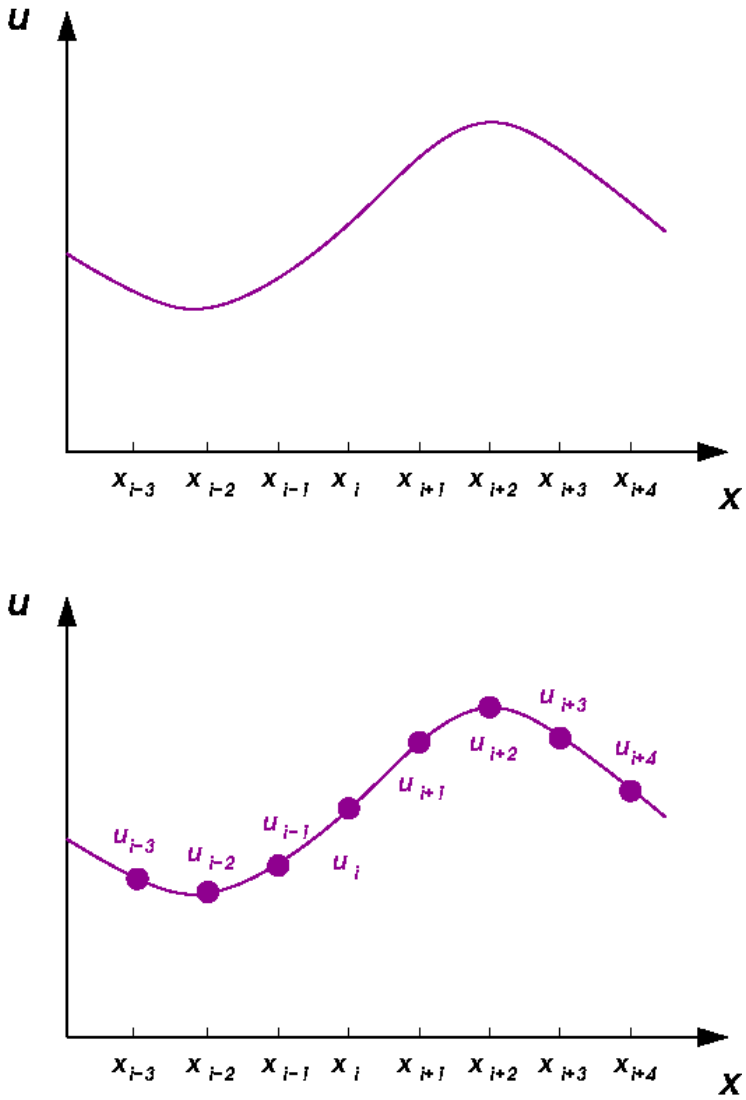
- It is possible to reduce the spacing of a regular Cartesian grid to increase accuracy of a solution
- This can be done only when and where required - **Adaptive Mesh Refinement**
- Because a uniform underlying structure exists, refinement can happen in space and in time - **hierarchical** adaptive mesh refinement
- Philip Blakely will talk more about this (after exams)



What about equation discretisation?

- For the remainder of this course, we shall assume a **regular Cartesian grid** structure
- This determines discrete regions in space where our solution is defined
- We can also consider the grid 'extending' along the time axis, i.e. we consider discrete **instances in time** for which our solution is defined
- How we proceed from this point is now determined by how we discretise our governing equations
- To do this, we first need to consider how our solution, u, exists on the grid
- We can then consider how we calculate its derivatives

Discretising a function



- For a function $u(x)$, we first discretise the spatial domain to a number of values we are interested in
- In a numerical solution, this is done for the **initial data**, $u_0(x) = u(x, 0)$
- Discretisation does not need to be uniform
- At each of these points, we assign a value to the function
- Is this as simple as $u_i = u(x_i)$?

Finite differences, elements and volumes

- The discretisation of the domain is directly related to the discretisation of the partial differential equation
- The method governs what information it needs from the underlying data
- **Finite difference** - data defined at **points**; function values are known only at specific points, i.e. $u_i = u(x_i)$
- **Finite elements** - data defined at **nodes**, elements are enclosed within nodes; data within the elements is reconstructed from the nodes
- **Finite volumes** - data defined within **volumes**; function values are integral averages over the volume centred at x_i

Finite differences, elements and volumes

- Finite volume methods are a natural way to treat conservation laws
- This is due to their ability to deal with discontinuous data
- By discretising through integral quantities, a volume can contain a discontinuity
- There is also an overlap between how finite difference and finite volume methods approximate derivatives
- In fact, for simple numerical methods, and simple equations, the methods may even look identical
- As a result, we won't make the distinction in our methods for a lecture or two
- Finite elements are, however, completely different, and actually use integral forms of the equations - we will not cover these

Discretisation notation (1)

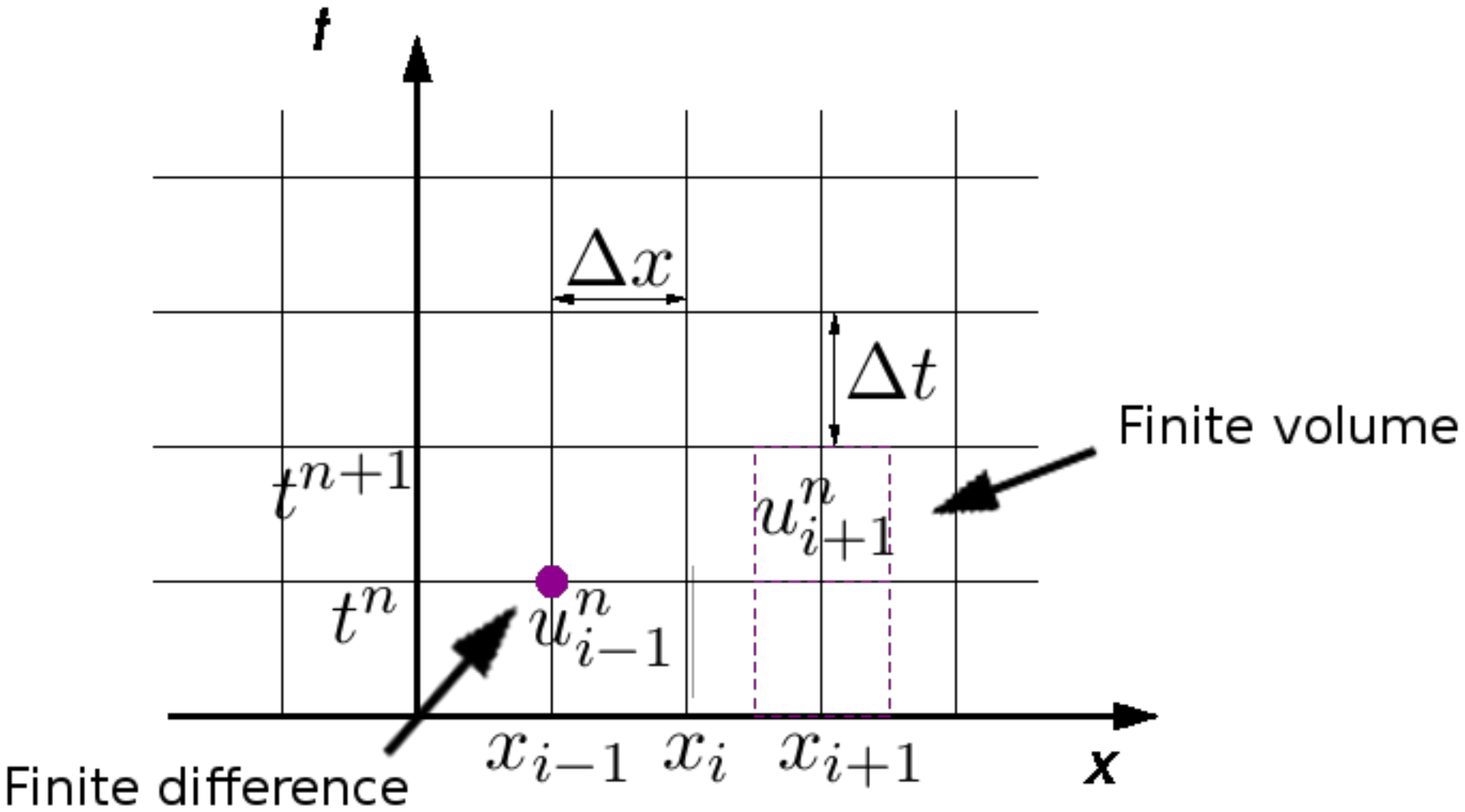
- Before considering any specific methods, we introduce the notation for discretised equations and the corresponding numerical methods
- For the solution to a partial differential equation, we discretise in **time** as well as **space**
- We saw previously that we split our space domain up into values x_i, similarly we consider the solution at discrete points in space, t^n
- **Convention:** subscript values correspond to a spatial discretisation, typically i, j and k correspond to a position in x, y and z respectively
- **Convention:** superscript values correspond to a temporal discretisation (**not** a power), typically n is used
- The solution to the function $u(\mathbf{x}, t)$ at a discrete point, (x_i, y_j, z_k, t^n) can be written

$$u^n_{i,j,k}$$

Discretisation notation (2)

- In addition to the notation for a discrete representation of the function u, we also introduce means to relate points neighbouring in both time and space
- For a domain $x \in [x_1, x_2]$ which we wish to solve our PDE, it is split into M points (or cells or elements) - this is our **mesh** or **grid**
- Between each point, we have a **grid spacing**, $\Delta x = x_i - x_{i-1}$
- Throughout this course, we shall assume Δx is uniform across the entire mesh, but this is not necessary (or desirable) for all applications
- Two different times are separated by a **time step**, $\Delta t = t^{n+1} - t^n$
- It is not necessary (and is often actually undesirable) for time steps to be constant across all time
- However, all points in space exist at the same time (numerical methods get very confusing otherwise)

Representation of discrete quantities



Discretising the equations

- Recall how Taylor series lead to approximations to derivatives
- First order **forward difference** -

$$\left.\frac{\partial u}{\partial x}\right|_{x_i} = \frac{u_{i+1} - u_i}{\Delta x} + \mathcal{O}(\Delta x)$$

- First order **backwards difference** (Taylor expansion at $x - \Delta x$)

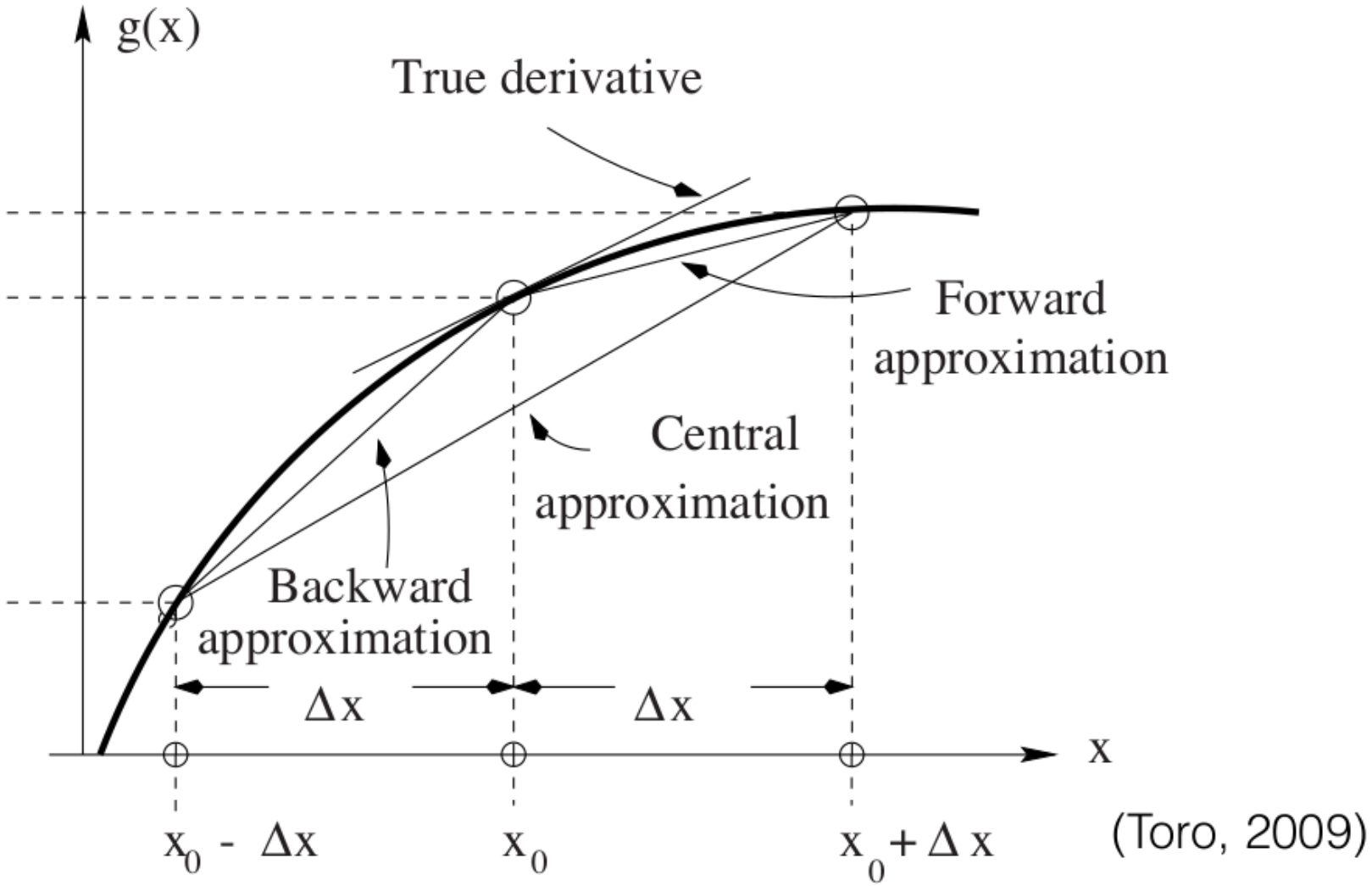
$$\left.\frac{\partial u}{\partial x}\right|_{x_i} = \frac{u_i - u_{i-1}}{\Delta x} + \mathcal{O}(\Delta x)$$

- Second order **central difference**

$$\left.\frac{\partial u}{\partial x}\right|_{x_i} = \frac{u_{i+1} - u_{i-1}}{2\Delta x} + \mathcal{O}(\Delta x)^2$$

- These represent one technique which can be used in a numerical solution

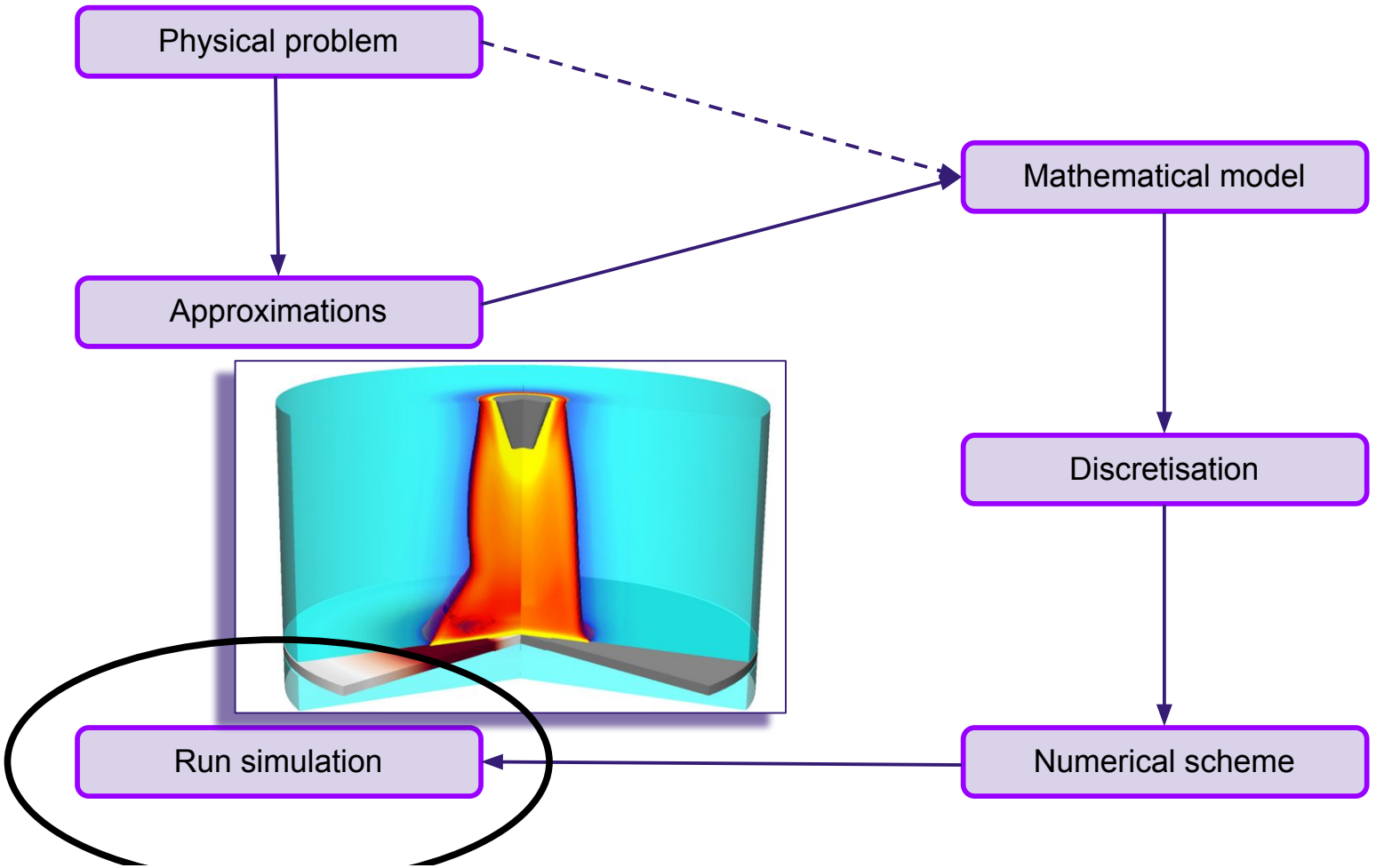
Graphical visualisation



(Toro, 2009)

Outline

Solving the advection equation



Discretisation of the advection equation

- We are now going to consider a numerical solution to the advection equation
- We have our equation:

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

- We can come up with some initial data, $u(x, 0) = u_0(x)$
- We can use a **finite difference** representation of the data, $u_i = u_0(x_i)$
- We know how to discretise the equation, for example,

$$\frac{u_i^{n+1} - u_i^n}{\Delta t} + a\frac{u_{i+1}^n - u_i^n}{\Delta x} = 0$$

- Rearranging this equation, we get an equation for u_i^{n+1} dependent **only** on data at time t^n

$$u_i^{n+1} = u_i^n - a\frac{\Delta t}{\Delta x}\left(u_{i+1}^n - u_i^n\right)$$

- This is referred to as an **explicit** numerical scheme

Constructing an advection equation code

- At the most basic, your advection code needs to consist of three steps:
  1. Set up initial data
  2. Update the data
  3. Output updated data
- Technically the last step is optional - the solution has been updated, whether you look at it or not, but unless you can see the data, you don't know whether it is correct
- We consider what is required for each of these steps

Setting up initial data - defining constants

- At the start of your simulation, there are several things that must be set:

How long is the domain?	How many points?
Start time, end time?	Time step?
Advection velocity a?	Initial data $u_0(x)$?

- Some of these are governed by the problem at hand (a, u_0 and the domain and time start and end), some of these are your choice (number of points, time step (sort of... we'll come to this later))

```
int nPoints = 100;  // Chosen such that distance
double x0 = 0.0;    // between points is 0.1
double x1 = 1.0;
double tStart = 0.0;
double tStop = 1.0;
double a = 1.0;
```

- Once we have defined these constants, we now need to discretise the domain

Setting up initial data - discretising the domain

- We can compute, or decide, discrete quantities

```
double dx = (x1 - x0) / nPoints;
// See problem sheet for more options here:
double dt = dx;
//define a vector with enough space to store discretised data
// at each point
std::vector<double> u;
u.resize(nPoints);
```

- Assuming we know $u_0(x)$, setting the initial data in this vector is easy

```
for(int i = 0; i < u.size(); i++) {
  double x = x0 + i * dx;
  u[i] = sin(x); // u_0(x) = sin(x)
}
```

- You may want to output the initial data at this point, both to check it is correct and to use it for comparison to your solution

Updating the data - looping through time

- We now want to find the solution to the advection equation at time $t = \texttt{tStop}$
- Since the time step, Δt, we chose is less that $\texttt{tStop} - \texttt{tStart}$, we need to put this in a loop

```
double t = tStart;
do {
  t = t + dt;
  //You may want to manually reduce dt if this
  // would overshoot tStop

  // Update the data
   Insert some code here...
} while (t < tStop);
```

- At some point, we need to define a second vector to store the updated data

```
std::vector<double> uPlus1;
u.resize(nPoints);
```

- This can be done before the time iteration loop, and overwritten each time step with updated data

Updating the data - looping through space

- Within the time iteration, we also need to iterate over space to update the solution at each point

```
//Update the data
for(int i = 0; i < u.size(); i++) {
  // Use the forward difference defined earlier
  uPlus1[i] = u[i] - a * (dt/dx) * (u[i+1] - u[i]);
}
// Now replace u with the updated data for the next time step
u = uPlus1;
```

- We now have an algorithm which can update data from time t^0 to time t^n for the advection equation, but will it work?
- Important question - what happens in the update formula for point 100, the last point in the domain?

Boundary conditions

```
uPlus1[i] = u[i] - a * (dt/dx) * (u[i+1] - u[i]);
```

- To update data at point i, we need information from points i and $i+1$ - this is the **stencil** of the numerical method we are using
- The numerical stencil is larger than the domain size, the method **cannot** work unless we supply **boundary conditions**
- These provide the correct data for points such as $i = 101$, and depend on the physics of the problem you consider
- In addition to providing these conditions, this means that we need the size of the vector storing u to be a bit bigger
- There are several types of boundary conditions which are commonly encountered

Boundary condition types

- **Periodic** - the domain 'wraps' - as data leaves one edge of the domain, it enters from the other
- **Transmissive** - a form of Neumann boundary - a constant gradient is assumed in u at the boundary
- **Reflective** - similar to transmissive, except some quantities are reflective (have their sign changed) - this requires at least one of your evolved variables to be e.g. velocity
- **Fixed** - a Dirichlet boundary, held at a predetermined (or computed) value - uncommon for hyperbolic equations except in the case of inflow - material being 'pumped' into the domain
- There are other possibilities, including mixed conditions along a single boundary
- For our advection example, we will assume periodic boundaries - we revisit our code in light of this

Setting up initial data - discretising the domain

- The definition of the constants does not need to change, the definition of the vectors does

```
//define a vector with enough space to store discretised data
// at each point
std::vector<double> u;
// Allow for one additional point at each end of the domain
u.resize(nPoints+2);
```

- We have defined u such that we can test forward, backward and central differences without re-sizing the vector
- We now need to be a little more careful when defining the x-location from the loop index

```
for(int i = 0; i < u.size(); i++) {
  // x_0 is at point i=1
  double x = x0 + (i-1) * dx;
  u[i] = sin(x); // u_0(x) = sin(x)
}
```

Updating the data

- Before updating any data, we need to compute the boundary conditions

```
double t = tStart;
do {
  t = t + dt;
  //You may want to manually reduce dt if this
  // would overshoot tStop

  //Periodic boundary conditions
  u[0] = u[nPoints];
  u[nPoints+1] = u[1];

  //Update the data
  //Make sure the limits on this loop correspond only to the
  // true domain
  for(int i = 1; i < nPoints+1; i++) {
    // Use the forward difference defined earlier
    uPlus1[i] = u[i] - a * (dt/dx) * (u[i+1] - u[i]);
  }
  // Now replace u with the updated data for the next time step
  u = uPlus1;
} while (t < tStop);
```

Outputting the data

- At the end of the simulation (or at any desired point during the loop) data can be output, either to the terminal or to a file
- Output to file allows you to use the data in plotting programs and produce images for reports/thesis/presentations

```
//Output the data
std::ofstream output(``advectionResults.dat'');

for(int i = 1; i < nPoints+1; i++) {
  double x = x0 + (i-1)*dx;
  output << x << " " << u[i] << std::endl;
}
```

- Now it is time to put these algorithms to the test