PHAS0102: Techniques of High-Performance Computing

We'll start at about 10:05 to give people time to join

Feedback

menti.com/al3qxshz4ppy
(I will post this in chat)

Finite differences

$$\frac{\mathrm{d}y}{\mathrm{d}x} \approx \frac{y_1 - y_0}{x_1 - x_0}$$

Differential equations → Simultaneous equations / matrix-vector problems

Time dependent problems

$$\frac{\mathrm{d}y}{\mathrm{d}t} \approx \frac{y(t_1) - y(t_0)}{t_1 - t_0}$$

$$\frac{\mathrm{d}y}{\mathrm{d}t} \approx \frac{y(t + \Delta t) - y(t)}{\Delta t}$$

Forward Euler

Backward Euler

$$\frac{\mathrm{d}y}{\mathrm{d}t} \approx \frac{y(t) - y(t - \Delta t)}{\Delta t}$$

$$\frac{\mathrm{d}y}{\mathrm{d}t} \approx \frac{y(t + \Delta t) - y(t - \Delta t)}{2\Delta t}$$

Forward Euler

$$\frac{\mathrm{d}y}{\mathrm{d}t} \approx \frac{y(t+\Delta t) - y(t)}{\Delta t}$$

$$\frac{\mathrm{d}y}{\mathrm{d}t} = f(y,t) \longrightarrow y(t+\Delta t) = y(t) + \Delta t\, f(y(t),t)$$

$$y_{n+1} = y_n + \Delta t\, f(y_n, t)$$

Backward Euler

$$\frac{\mathrm{d}y}{\mathrm{d}t} \approx \frac{y(t) - y(t - \Delta t)}{\Delta t}$$

$$\frac{\mathrm{d}y}{\mathrm{d}t} = f(y, t) \longrightarrow y(t + \Delta t) = y(t) + \Delta t\, f(y(t + \Delta t), t)$$

$$y_{n+1} = y_n + \Delta t\, f(y_{n+1}, t)$$

Forward Euler

$$y_{n+1} = y_n + \Delta t\, f(y_n, t)$$

(Usually) easy to solve
Explicit
Less stable

Backward Euler

$$y_{n+1} = y_n + \Delta t\, f(y_{n+1}, t)$$

Harder to solve
Implicit
More stable

Finite elements

A more powerful method for turning PDEs into matrix-vector problems.

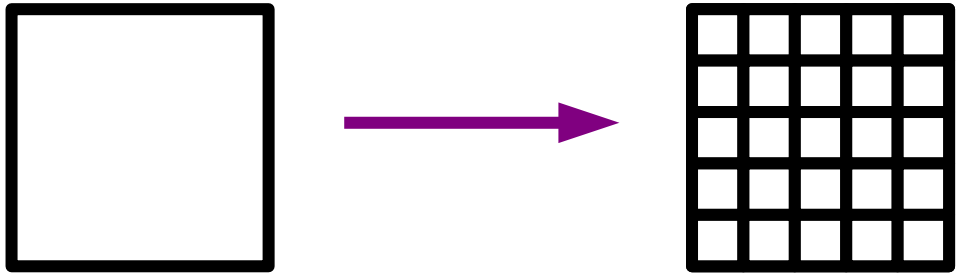
Finite elements

1. Write differential equation in variational form

$$-\Delta u = f \longrightarrow \text{Find } u \in V \text{ such that}$$

$$\int \nabla u \cdot \nabla v = \int v f \quad (\text{for all } v \in V)$$

2. Pick a finite dimensional subspace of V.



Functions that are continuous and polynomial of max degree n on each square in the mesh

Finite elements

Turning the PDE into a matrix-vector problem is called discretisation

The discretisation is done in such a way that leads to sparse matrices.

Open source software

There is a lot of open source software for finite element methods. Most is developed at universities. Eg:

- FEniCS (Cambridge, Luxembourg, Simula (Oslo), UCL, others)
- Firedrake (Imperial, Oxford, others)
- Deal.II (Colorado State, others)
- Dune
- MFEM

There is a lot of research work currently taking place based on developing fast FEM libraries and building new methods on top of them.

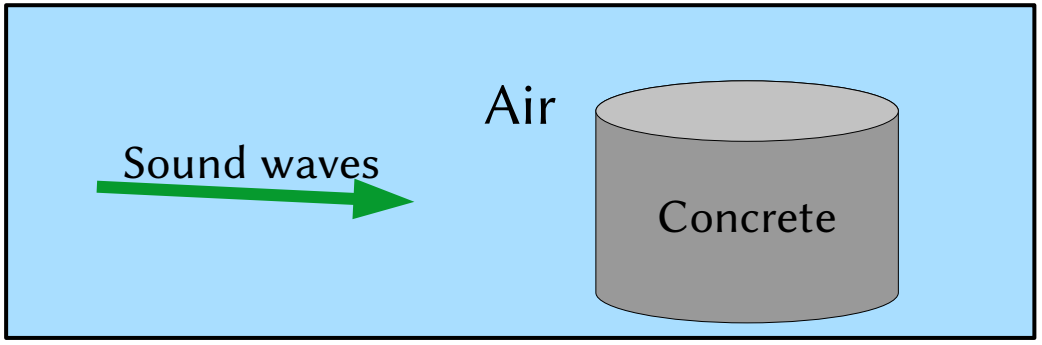
FEniCS

During my previous job in Cambridge, I was working on developing FEniCS.

If you ever need to use a finite element solver in the future, get in touch.

Domain decomposition

Many real world problems involve domain decomposition, eg:



These kind of problems are possible to parallelise: get one process to solve in the air and other in the concrete.

Space-time parallelisation

Problems involving time are harder to parallelise, as the whole solution at the current time is needed before starting on the next point in time.

Alternative method: make a 4D mesh (3D space and 1 time dimension) and solve at all times at once – this can be parallelised much more effectively.

Solvers

Effective solvers for sparse matrices arising from FEM is a big area of research:

- Multigrid and variants of multigrid
- Preconditioning

Randomised linear algebra

In some applications, randomised linear algebra can be very effective.

eg picking some random vectors and multiplying them with a matrix can be a fast way to approximate eigenvalues.

Machine learning

- Huge matrices of data
  - Matrix storage and compression
  - Fast solvers
- Randomisation, trial and error
- Parallelisation

Some advice for the future

Keep programming

- The best way to become a better programmer is to practice.
- Programming puzzles you can do to practice:
  - Advent of Code
  - Project Euler

Keep programming

- Learn more languages
  - If you want to do fast computation: C++ or Rust
  - If you want to do web development: Javascript or PHP
  - If you want to work at Google: Go
- Once you've learned a couple of languages, picking up more is very easy.

Reproducibility

- It is very important to do everything we can to make scientific experiment reproducible
- Whenever you publish anything based on code:
  - Make an archive of the current state of the code
  - Make and archive a docker image in which you can run the code

Embrace open source

- There are very few downside of making every piece of code you write open source.
  - Should I share my code? Chalkdust Magazine (http://chalkdustmagazine.com/features/o%cf%80nions-should-i-share-my-code/)
- Learn git. Put any code you write on GitHub.
- Use other people's code. If it doesn't do quite what you want, email them and help develop new features.

Two final pieces of advice

- Watch out for off-by-one errors

Thank you!

- Thanks for being great students. I've really enjoyed working with you.

- If anyone would like to ask anything "on their way out", I'm going to drop into the Gather Town space for the next 5 or so minutes (link on Moodle and in chat).