MAST90026 Computational Differential Equations Notes

Notes taken in Steven Carnie's 2012 and 2014 Classes
edited/adapted by James Osborne(2018), Hailong Guo (2020, 2022)
and Jesse Collis (2024)

SM1 2024

Preface

These notes for the subject MAST90026 Computational Differential Equations were originally produced by a student in the 2012 class, Eddy Qiu. He passed them on to Steven Carnie in December 2013 who turned them into notes for the next class — the class of 2014. Eddy ran out of steam just as time entered the picture. The remaining lectures were transcribed into LaTex by Cameron Patrick, Ashley Dyson and Gala Camacho Ferrari of the 2014 class, and James Osborne have used these to make notes for the 2018 class. Hailong Guo modified these notes for the 2020 and 2022 editions of this class and I (Jesse Collis) have further modified for the 2024 edition of the class.

The notes are inspired by the following works:

- - Ascher, Mattheij & Russell, Numerical solution of boundary value problems for ordinary differential equations, Prentice-Hall, 1988.
- G Gockenbach, *Understanding and implementing the finite element method*, SIAM, 2006.
- I Iserles, *A first course in the numerical analysis of differential equations*, 2nd ed., CUP, 2008.
- - Kelley, *Iterative Methods for Linear and Nonlinear Equations* , SIAM, 1995.
- L Leveque, *Finite Difference Methods for Ordinary and Partial Differential Equations: Steady-State and Time-dependent problems*, SIAM, 2007.
- - Trefethen, Spectral Methods in MATLAB, SIAM, 2000.

Jesse Collis

February, 2024

Contents

0 Introduction

In this course we aim to cover numerical techniques for solving

- Boundary value problems (BVPs) for ODEs, (MATLAB commands: `bvp4c/bvp5c`)
- Simple PDEs
  - 2D Elliptic PDEs
  - 1+1 (1 space dimension, 1 time dimension) Parabolic (MATLAB command: `pdepe`), Hyperbolic

For 2D PDEs there is a toolbox in MATLAB `pdetool`, but there is no toolbox for 3D PDEs!

The sorts of problems we are aiming to deal with include (in decreasing order of difficulty):

Example 0.1 (Incompressible Navier-Stokes equations)**.**

$$\begin{aligned} \mathbf{u}_t + (\mathbf{u}\cdot\nabla)\mathbf{u} &= -\nabla p + \mu\nabla^2\mathbf{u}, \\ \nabla\cdot\mathbf{u} &= 0. \end{aligned}$$

Example 0.2 (Black-Scholes equation)**.**

$$v_t + \frac{1}{2}\sigma^2 S^2 v_{SS} + rSv_S - rv = 0.$$

Example 0.3 (Convection-diffusion equation)**.**

$$u_t + \mathbf{v} \cdot \nabla u = \nabla \cdot (\mathrm{D}\, \nabla u) + f.$$

The convection-diffusion equation simplifies in 1D to

$$u_t + v u_x = \mathrm{D}\, u_{xx} + f.$$

a) Letting $v = 0$ we get the diffusion/heat equation:

$$u_t = \mathrm{D}\, u_{xx} + f,$$

which is *Parabolic*.

b) Or letting $\mathrm{D} = 0$ we get the advection/transport/1-way wave equation.

$$u_t + v u_x = s,$$

which we call *Hyperbolic* even though it is first order, because it shares many of the properties of second order Hyperbolic equations.

c) If we solve for the steady state solution $(u_t = 0)$ we get:

$$\mathbf{v} \cdot \nabla u - \nabla \cdot (\mathrm{D}\, \nabla u) = f,$$

which is a PDE in space only.

If $\mathbf{v} = 0$, we get

$$-\nabla \cdot (\mathrm{D}\, \nabla u) = f.$$

which is an *Elliptic* PDE.

d) Finally in one dimension, we get

$$v u_x - \mathrm{D}\, u_{xx} = f,$$

an ODE with boundary conditions – a boundary value problem!

In discussing numerical methods, we will start with the simplest problems — BVPs — and work our way back up. We probably won't get to solving the Navier-Stokes equations, which is typically done with commercial software for Computational Fluid Dynamics (CFD), but I will show you some examples. For more details, take ENGR90024 Computational Fluid Dynamics which uses the OpenFoam package.

We will concentrate on methods for solving linear BVPs that generalize to solving PDEs and to nonlinear problems.

There are a few general issues we will need to face:

- How do we step in time (for IVPs)?
- How do we we represent the solution in space (for BVPs/PDEs)?
- How do we handle higher dimensionality?
- How do we handle non-linearity?

0.1 ODE IVP Revision

Before we look at BVPs we will do a quick Initial Value Problem revision example.

Exercise 0.4. Use MATLAB's inbuilt solvers (or code up your own method) to solve the following system of first order ODE's

$$\begin{aligned}
\frac{\mathrm{d}S}{\mathrm{d}t} &= \Pi - \beta SZ - \delta S, \\
\frac{\mathrm{d}Z}{\mathrm{d}t} &= \beta SZ + \zeta R - \alpha SZ, \\
\frac{\mathrm{d}R}{\mathrm{d}t} &= \delta S + \alpha SZ - \zeta R,
\end{aligned}$$

subject to $S(0) = 1000$, $Z(0) = 0$, and $R(0) = 0$. Where $\Pi = 0$, $\beta = 0.0095$, $\delta = 0.0001$, $\zeta = 0.1$, and $\alpha = 0.005$.

1 1D Boundary Value Problems

First we will look at linear scalar 2^{nd} order BVPs. They are of the form

$$u'' + p(x)u' + q(x)u = r(x),$$

with *linear separated* boundary conditions at $x = a$ and $x = b$:

$$\begin{aligned} A_{11}u(a) + A_{12}u'(a) &= \beta_1, \\ A_{21}u(b) + A_{22}u'(b) &= \beta_2. \end{aligned}$$

These are sometimes called *Robin* boundary conditions.

If $A_{12} = A_{22} = 0$:

$$\begin{aligned} A_{11}u(a) &= \beta_1, \\ A_{21}u(b) &= \beta_2, \end{aligned}$$

then we get *Dirichlet* boundary conditions. Alternatively, if $A_{11} = A_{21} = 0$:

$$\begin{aligned} A_{12}u'(a) &= \beta_1, \\ A_{22}u'(b) &= \beta_2, \end{aligned}$$

we get *Neumann* boundary conditions. Note that our BVP may not be nearly as nice as this — both our equation and our boundary conditions could be non-linear.

Example 1.1 (General scalar 2^{nd} order BVPs.)**.**
They are of the form

$$u'' = f(x, u, u'),$$

with *separated* boundary conditions at $x = a$ and $x = b$:

$$\begin{aligned} g_1(u(a), u'(a)) &= B_1, \\ g_2(u(b), u'(b)) &= B_2. \end{aligned}$$

In more generality, we can have BVPs written as first order systems:

$$\mathbf{u}' = \mathbf{f}(x, \mathbf{u}),$$

with non-separated boundary conditions

$$\mathbf{g}(\mathbf{u}(a), \mathbf{u}(b)) = \mathbf{0}.$$

There are many methods for solving boundary value problems:

- **Shooting:** shooting treats the BVP as a series of initial value problems, then uses the large number of readily available IVP solvers to solve the problem.
  - Simple shooting
  - Multiple shooting
- **Relaxation:** relaxation reduces the problem into a series of simpler problems.
  - Finite difference methods (FDM): reduce the problem to a set of algebraic equations (linear or non-linear).
  - Weighted residuals: search for a solution in a simpler function space (i.e. the set of piecewise linear functions)
    * Galerkin (finite element methods (FEM))
    * Collocation (pseudospectral)
  - Finite volume methods (FVM)

If we look at the boundary volume problem as

$$\mathrm{L}\,u = r(x),$$

then in the FDM we approximate the differential operator L and in weighted residual methods we approximate the solution u. In FVM (not covered in this subject), we must have a problem written as a conservation law and approximate the fluxes. Note that in 1D, FVM and FDM are the same but in higher dimensions they're not.

Note that in general it is much harder to guarantee existence or uniqueness of a solution for a BVP as compared to an IVP e.g. for linear BVPs we can get 0, 1 or ∞ solutions.

1.1 Finite Difference Methods *Leveque, Chapters 1,2*

In finite difference methods we aim to approximate $\mathrm{L} = \mathrm{D}^2 + p(x)\,\mathrm{D} + q(x)\,\mathrm{I}$.

1.1.1 Finite Difference Formulae

To do this we approximate u' and u'' in terms of sampled values of $u(x)$, with each approximation being associated with a *discretisation error*. For example:

$$
\begin{aligned}
u'(x) &= \frac{u(x+h) - u(x)}{h} + \text{error}, && \text{Forward difference formula;} \\
u'(x) &= \frac{u(x) - u(x-h)}{h} + \text{error}, && \text{Backward difference formula;} \\
u'(x) &= \frac{u(x+h) - u(x-h)}{2h} + \text{error}, && \text{Central difference formula; } \textit{and} \\
u''(x) &= \frac{u(x+h) - 2u(x) + u(x-h)}{h^2} + \text{error}, && \text{Central difference formula.}
\end{aligned}
$$

In general we obtain an approximation for $u^{(k)}$:

$$u^{(k)}(x) = \sum_{i=1}^{n} c_i u(x + h_i) + \text{error}.$$

called a *Finite Difference Formula.*

The positions $\{h_i\}$ are determined by the choice of *mesh*, or the values where we are approximating u. The weights $\{c_i\}$ are called the *stencil* and are chosen to achieve a certain error. n determines how many values we are approximating each point with.

Example 1.2 (Forward Difference)**.**
$n = 2$, $c_1 = -\frac{1}{h}$, $c_2 = \frac{1}{h} \to \text{Error} = O(h)$.

Example 1.3 (Central Difference (for u'))**.**
$n = 3$, $c_1 = -\frac{1}{2h}$, $c_2 = 0$, $c_3 = \frac{1}{2h} \to \text{Error} = O(h^2)$.

If we approximate $u^{(k)}$ with n values our worst-case error is $O(h^{n-k})$, so we would definitely like $n > k$ so that our error decreases as we decrease the step size!

Derivation

To determine the coefficients (the weights) we fix $\{h_i\}$ and expand $u(x+h_i)$ in Taylor Series. From this we obtain a system of linear equations for $\{c_i\}$.

Alternatively, it is also possible to use interpolating polynomials. If we take the unique interpolating polynomial through $\{(x+h_i, u(x+h_i))\}$, differentiate the polynomial and evaluate at each $x + h_i$ we obtain the weights for our approximation. Leveque provides MATLAB code which generates the required weights (`fdcoeffF`).

Exercise 1.4. Use the MATLAB function `fdcoeffF` to calculate the coefficients in the above examples. If you have time, experiment with non uniform meshes.

1.1.2 Discretisation Errors

To find the errors for our approximations, we use Taylor Series:

Example 1.5 (Central Difference Error)**.**

$$
\begin{aligned}
&\frac{u(x+h)-u(x-h)}{2h}\\
=&\frac{1}{2h}\left(u(x)+hu'(x)+\frac{1}{2}h^2u''(x)+O(h^3)\right)-\\
&\frac{1}{2h}\left(u(x)-hu'(x)+\frac{1}{2}h^2u''(x)+O(h^3)\right)\\
=&\frac{2hu'(x)+O(h^3)}{2h}\\
=&u'(x)+O(h^2),
\end{aligned}
$$

where we have assumed that u is sufficiently smooth (e.g. for the above example we assumed that $u \in C^3$).

- The forward difference error is first order.
- The backward difference error is also first order.
- As we saw above, the central difference error is second order.
- The central difference approximation for u'' has a second order error if the mesh used is equal spacing, otherwise it is first order in general.

Exercise 1.6. Use the Matlab function `fdstencil` to calculate the dominant terms in the error for the above examples.

Exercise 1.7. Run the Matlab script `chap1example1` to see how the error of a set of approximations to u'. Note 'Dpu' is the Forward difference formula, 'Dmu' is the Backward difference formula, 'D0u' is the central difference formula, and 'D3u' is the following approximation

$$D_3u(x) = \frac{1}{6h}[2u(x+h) + 3u(x) - 6u(x-h) + u(x-2h)].$$

What is the accuracy of D_3u?

1.1.3 Solution off the Mesh

We end up with a system of linear equations for each of the points on our mesh which we can solve. To find the solution off the mesh interpolation is required. The most common form is linear interpolation, but more accurate methods (like splines) could be used, see MAST30028.

1.1.4 A linear BVP with Dirichlet BCs

We will examine a linear BVP with Dirichlet boundary conditions on a uniform mesh with N internal mesh points.

$$\begin{aligned}&u''(x) + p(x)u' + q(z)u = r(x),\\ &u(a) = \alpha, u(b) = \beta.\end{aligned}$$

Let $h = \frac{b-a}{N+1}$, $x_j = a + jh$. Then, using a central difference approximation for u' and u'', at x_j:

$$\frac{u_{j+1} - 2u_j + u_{j-1}}{h^2} + p(x_j)\frac{u_{j+1} - u_{j-1}}{2h} + q(x_j)u_j = r(x_j).$$

Letting $p(x_j) = p_j$, $q(x_j) = q_j$ and $r(x_j) = r_j$ we obtain the following system of linear equations:

$$\begin{aligned}u_0 &= \alpha,\\ \frac{u_0 - 2u_1 + u_2}{h^2} + p_1\frac{u_2 - u_0}{2h} + q_1u_1 &= r_1,\\ \frac{u_1 - 2u_2 + u_3}{h^2} + p_2\frac{u_3 - u_1}{2h} + q_2u_2 &= r_2,\\ &\vdots\\ \frac{u_{N-1} - 2u_N + u_{N+1}}{h^2} + p_N\frac{u_{N+1} - u_{N-1}}{2h} + q_Nu_N &= r_N,\\ u_{N+1} &= \beta.\end{aligned}$$

We can eliminate u_0 and $u_N + 1$ easily, which would eliminate the first and last equations and change the second and second last equation to:

$$\begin{aligned}\frac{-2u_1 + u_2}{h^2} + p_1\frac{u_2}{2h} + q_1u_1 &= r_1 - \frac{\alpha}{h^2} + p_1\frac{\alpha}{2h},\\ \frac{u_{N-1} - 2u_N}{h^2} + p_N\frac{-u_{N-1}}{2h} + q_Nu_N &= r_N - \frac{\beta}{h^2} - p_N\frac{\beta}{2h},\end{aligned}$$

or we could just keep the first and last trivial equations and solve a slightly larger system.

We can write this in matrix form:

$$\frac{1}{h^2}\begin{pmatrix} -2+h^2q_1 & 1+h\frac{p_1}{2} & 0 & 0 \\ 1-h\frac{p_2}{2} & \ddots & \ddots & 0 \\ 0 & \ddots & \ddots & 1+h\frac{p_{N-1}}{2} \\ 0 & 0 & 1-h\frac{p_N}{2} & -2+h^2q_N \end{pmatrix}\begin{pmatrix} u_1 \\ u_2 \\ \vdots \\ u_N \end{pmatrix}$$
$$= \begin{pmatrix} r_1 - \frac{\alpha}{h^2} + p_1\frac{\alpha}{2h} \\ r_2 \\ \vdots \\ r_N - \frac{\beta}{h^2} - p_N\frac{\beta}{2h} \end{pmatrix}$$

or $A\mathbf{u} = \mathbf{b}$, which can be be solved using the backslash command in MATLAB (denoted \ in the text from now on): `u=A\b`.

Once we have reduced the problem to a linear system, some obvious questions are:

1. Does this have a solution?
2. How hard is it to solve?
3. How well does $\mathbf{u}$ approximate $u(x)$ for a given h?
4. Does $\mathbf{u} \to u(x_j)$ as $h \to 0$?

The answers to these questions are:

1. A solution exists as long as our matrix is non-singular. A singular matrix is possible if the underlying boundary value problem has no unique solution, e.g. a BVP with Neumann boundary conditions on both sides.

   But it can be shown that if the matrix is *strictly diagonal dominant* ($|a_{jj}| > \sum_{i \neq j} |a_{ij}|$), then our matrix will not be singular (from Linear Algebra).

   For the above matrix this is generally true if h is small and always true if $p = 0$ and $q < 0$.

2. A general dense $n \times n$ matrix requires about $\frac{1}{3}n^3$ operations to solve (Gaussian elimination with partial pivoting) or more.

   Luckily our matrix is tridiagonal (non zero entries along only the central diagonal and two diagonals above and below it). Without pivoting this takes $5n + 4$ operations and with pivoting it takes $O(n^2)$ operations. It turns out that if a matrix is strictly diagonal dominant, then we can guarantee we will not need pivoting, so our solution will be extremely quick.

   This happy fact is not so true for 2D problems, and will lead us to consider other methods for solving linear systems — iterative methods.

3/4. To answer Questions 3 and 4, we need to do some error analysis for BVPs.

1.1.5 Error analysis

In solving the linear system

$$A\mathbf{x} = \mathbf{b},$$

we get the rule of thumb

relative forward error $\sim$ condition # $\times$ relative backward error (residual).

The condition number is a property of the problem and measures how sensitive the problem output is to a small change in the problem inputs. The backward error is a property of the algorithm and gives how close is the nearby problem it actually solved. As an equation we say

$$\frac{\|\boldsymbol{\Delta}\mathbf{x}\|}{\|\mathbf{x}\|} \lesssim \kappa(A)\left(\frac{\|\Delta A\|}{\|A\|} + \frac{\|\boldsymbol{\Delta}\mathbf{b}\|}{\|\mathbf{b}\|}\right),$$

where $\kappa(A)$ is the condition number (for inversion), the forward error $\boldsymbol{\Delta}\mathbf{x}$ is the error in the solution, and the backward error ΔA (and $\boldsymbol{\Delta}\mathbf{b}$) is the backward error due to the solution algorithm. Notice that here we use vector norms and subordinate matrix norms to measure the errors.

For initial value problems, we can show (see MAST30028)

convergence $\sim$ 0-stability + consistency (residual),

where convergence measures the accuracy of our solution, 0-stability measures how much errors of the discrete problem stay bounded as $h \to 0$ (how stable our algorithm is), and consistency relates to how well we approximate the problem (or if we put in the real solution, what kind of residual we obtain).

We aim for a similar result for BVPs, in the form

convergence $\sim$ stability + consistency (residual).

To measure the error we cannot use traditional vector norms because as we refine our mesh our vector grows so even though the accuracy may remain the same or get better the size of our error vector may grow. Instead we use grid function (i.e. it is only defined on the grid/mesh) norms:

$$\begin{aligned}\|\Sigma\|_1 &= h\sum |\Sigma_i|,\\ \|\Sigma\|_2 &= \left(h\sum |\Sigma_i|^2\right)^{\frac{1}{2}},\\ \|\Sigma\|_\infty &= \max|\Sigma_i|.\end{aligned}$$

(Note that the infinity norm for grid functions is the same as the vector infinity norm). By convergence, we mean the accuracy of the solution or the *global error* which is defined as:

$$\|E\| = \|\mathbf{u} - \hat{\mathbf{u}}\|,$$

the difference between the computed solution $\mathbf{u} = \{u_j\}$ and the true solution $\hat{\mathbf{u}} = \{\hat{u}(x_j)\}$ at the grid points, measured with a grid function norm. By residual, we mean the Local Truncation Error. This leads to the following definitions.

Definition 1.8 (Convergence).
A method with global error $\|E\| = O(h^p)$ *as* $h \to 0$ *is convergent of order* p.

Definition 1.9 (Local Truncation Error (LTE)).
The local truncation error τ_j *is the residual at grid point* x_j *when the true solution is put into the finite difference formula*

The LTE for our system is given as

$$\tau_j = \frac{1}{h^2}(u(x_{j+1}) - 2u(x_j) + u(x_{j-1})) + p(x_j)\frac{u(x_{j+1}) - u(x_{j-1})}{2h} + q(x_j)u(x_j) - r(x_j).$$

Now to find the size of τ_j we can use Taylor series (assuming $u \in C^3$) to get the following:

$$\begin{aligned}\tau_j &= u''(x_j) + p(x_j)u'(x_j) + q(x_j)u(x_j) - r(x_j) + Ch^2u^{(3)} + O(h^4)\\ &= Ch^2u^{(3)}(\xi) + O(h^4),\end{aligned}$$

so $\tau_j = O(h^2)$. Define $\tau = (\tau_1, \dots \tau_N)^T$ as a grid function.

Exercise 1.10. Using taylor expansion derive the form for τ_j given above.

Definition 1.11 (Consistency). *A method is consistent of order p if $\|\tau\| = O(h^p)$ as $h \to 0$.*

Example 1.12. For the central difference formulae above, $\|\tau\| = O(h^2)$, so the method is consistent of order 2.

For a method to be consistent, it just means you've used a sensible discretisation of the original problem; usually consistency is not an issue. We now need to connect the global error and the LTE.

The exact solution solves the problem

$$\mathrm{L}\,\hat{u} = r(x).$$

By definition of the LTE, it satisfies for the discretized operator $\mathrm{L_h}$ the equation

$$\mathrm{L_h}\,\hat{u} = r(x) + \tau,$$

or, when evaluated at the grid points

$$A\hat{\mathbf{u}} = \mathbf{b} + \tau.$$

The computed solution satisfies

$$A\mathbf{u} = \mathbf{b}.$$

so, subtracting we get

$$A(\mathbf{u} - \hat{\mathbf{u}}) = AE = -\tau,$$
$$\implies E = -A^{-1}\tau,$$

where E is the global error, so

$$\|E\| = \|A^{-1}\tau\| \leq \|A^{-1}\|\|\tau\|.$$

This is our connection between the global error and the LTE.

To make sure the global error becomes small like the LTE, we want $\|A^{-1}\| \leq c$ $\forall h \leq h_0$.

Definition 1.13 (Stability). *A method is stable if A^{-1} exists and $\|A^{-1}\| \leq c$ $\forall h < h_0$*

Finding such a constant c is called finding a *stability bound.* So a stable discrete matrix stays well-conditioned as $h \to 0$.

Theorem 1.14 (Convergence)**.** *Consistency + Stability $\implies$ Convergence.*

Proof:

$$\begin{aligned} & AE = -\tau, \\ \implies & E = -A^{-1}\tau, \\ \implies & \|E\| \le \|A^{-1}\|\|\tau\| \le cCh^2 = O(h^2). \end{aligned}$$

Proving stability is often hard since finding A^{-1} is often harder than solving the problem. Also A increases in size as $h \to 0$.

Example 1.15. *Levesque* §*2.10, 2.11* proves stablility for $u'' = f$ on $[0,1]$ with Dirichlet boundary conditions by proving

$$\|A^{-1}\|_2 \le \frac{1}{\pi^2} \quad \text{and} \quad \|A^{-1}\|_\infty \le 3.$$

1.1.6 Other boundary conditions

- Neumann boundary condition at one boundary, Dirichlet at the other (*mixed boundary conditions*)
- *Robin boundary conditions*: $A_1 u(a) + A_2 u'(a) = B_1$ etc.

For these problems we can use:

- 1^{st} Order Method:

  We can simply let $u'(x_0) = \alpha \implies \frac{u_1 - u_0}{h} = \alpha$ so our error is now $O(h)$ and we have lost $O(h^2)$ behaviour.

- 2^{nd} Order Methods:

  – Ghost nodes:

  $$\frac{u_{-1} - 2u_0 + u_1}{h^2} + p_0 \left(\frac{u_1 - u_{-1}}{2h} \right) + q_0 u_0 = r_0,$$

  where u_{-1} is a ghost node. Then, to eliminate u_{-1} we apply our boundary conditions, using a central difference to ensure $O(h^2)$:

  $$u'(x_0) = \alpha \implies \frac{u_1 - u_{-1}}{2h} = \alpha, \implies u_{-1} = u_1 - 2\alpha h.$$

  So we obtain for the first row in our matrix:

  $$\frac{-2u_0 + 2u_1}{h^2} + q_0 u_0 = r_0 - p_0 \alpha + \frac{2\alpha}{h}.$$

  – Alternatively we can use a 1-sided second order forward difference formula for $u'(x_0)$

  $$u'(x_0) = \frac{-3u_0 + 4u_1 - u_2}{2h} + O(h^2),$$

  where we have retained our $O(h^2)$ behaviour. Our first row now has 3 elements instead of 2 so our matrix is no longer tridiagonal, but it is still banded.

1.2 Method of Weighted Residuals

We let u be a vector in some function space, called the trial space, with basis $\{\phi_j\}$ and coefficients c_j.

$$u_N = \sum_{j=0}^{N} c_j \phi_j.$$

There are then two aspects of our solution where we can make choices.

1. There are two common ways to choose ϕ_j:
    - ϕ_j are global basis functions. So $\phi_j \neq 0$ over $[a,b]$ e.g. $\sin(jx)$, polynomials.
    - ϕ_j are bases of a piecewise polynomial space. So ϕ_j has a local support and is only non zero over a compact subset of $[a,b]$ e.g. 'hat functions' form a basis for piecewise linear functions.
2. And then are a few ways to choose a criterion to find our coefficients to minimize the size of the residual $\mathrm{L}\, u_N - r$.
    - We can force the residual to be equal to 0 at certain $\{x_j\}$ in our domain. This is called *collocation*.
    - Or we can try and make the residual orthogonal to the function space of our approximate solution u_N (or *trial space*). We call this a *Galerkin* method. This is equivalent to finding a solution in the trial space that satisfies the *weak form* of the boundary value problem.

Aside. We can also try and make our residual orthogonal to a (different) *test space*. These methods are called *Petrov-Galerkin* methods.

These choices give the following methods:

	Global Basis	Piecewise polynomial / local support
Collocation	Spectral Collocation / Pseudospectral	Spline Collocation
Galerkin	Spectral Galerkin	Finite Element Methods

We will not cover Spectral Galerkin methods here.

1.3 Collocation methods

1.3.1 Spectral Collocation

In spectral collocation we first have a trial space:

1. We can choose a trial space of trigonometric functions:

$$u_N(x) = \sum_{j=0}^{N} a_j \cos(jx) + \sum_{j=1}^{N} b_j \sin(jx) = \sum_{j=-N}^{N} c_j e^{ijx},$$

   which is the same as representing our solution by a truncated Fourier series (hence the term *spectral* in spectral collocation).

   This choice is good for problems with periodic boundary conditions, e.g. $u(0) = u(2\pi)$ (which is not separated), since there is a theorem that states we can represent all periodic L^2 functions with Fourier series.

2. We can also choose a trial space of polynomials of degree N (order $N+1$).

$$u_N(x) = p_N(x) \in \mathbb{P}_{N+1}.$$

   This is more common for non periodic problems on a domain $[a, b]$.

   We still have more questions to answer:

   (a) What basis functions should we use over the polynomials?

   (b) What collocation points should we use?

   - We can choose a basis of Legendre polynomials $P_n(x)$ $(n = 0, \ldots, N)$. Then choose as collocation points the roots of P_{N+1}(*orthogonal collocation*).
   - Or we can first choose to collocate on the *Chebyshev points* (of the second kind): the maxima/minima of $T_N(x)$ (i.e. the Chebyshev polynomial of the first kind)

$$\{x_k\} = \left\{ \cos\left(\frac{\pi k}{N}\right) \middle| k = 0, \ldots, N \right\},$$

     And then choose the *cardinal basis* (or *nodal basis*) as basis functions:

$$\ell_j(x_i) = \delta_{ij},$$

i.e.

$$\ell_j(x) = \prod_{k \neq j} \frac{(x - x_k)}{(x_j - x_k)},$$

which is called the *Lagrange interpolating polynomial.*

With this choice of basis, we have

$$p_N(x_i) = \sum_{j=0}^{N} u_j \ell_j(x_i) = c_i,$$

so we can let $c_i = u(x_i) = u_i$ which simplifies things somewhat.

Aside. The polynomial interpolant through $\{x_j, u_j\}$, $(j = 0, \ldots, N)$ is:

$$p_N(x) = \sum_{j=0}^{N} u_j \ell_j(x),$$

where $\ell_j(x_i) = \delta_{ij}$ as above.

Exercise 1.16. Call the function `rungeinterp`, to see an example of polynomial fitting on equally spaced points, note the Runge Phenomena at the edges.

Exercise 1.17. Call the function `chebRunge`, to see an example of polynomial fitting on Chebyshev points, note the error at the the edges.

The main advantage of spectral methods is that under the right conditions (i.e. u is smooth enough) we can get *spectral convergence.*

i.e. $e_N \sim e^{-cN}$ instead of $e_N \sim N^{-p}$ as in FDM and FEM.

So N does not have to be very large for great accuracy. The trade off is that the matrices generated by spectral collocation are dense.

Exercise 1.18. Run the script `p12` what does this tell you about the error?

Example 1.19. Consider the BVP $u''(x) + p(x)u'(x) + q(x) = r(x)$ over $[a, b]$.

First, map the interval $[a, b]$ to $[-1, 1]$. $\rightarrow u''(t) + \bar{p}(t)u'(t) + \bar{q}u(t) = \bar{r}(t)$.

Let

$$u_N(x) = \sum_{j=0}^{N} c_j \phi_j(x),$$

and collocate at the Chebyshev points $x_k = \cos(\frac{\pi k}{N})$, $(k = 0, \ldots, N)$. Let $\{\phi_j\}$ = Lagrange polynomials based on the Chebyshev points then, $c_j = u_j$. So,

$$u'_N(x) = \sum_{j=0}^{N} u_j \phi'_j(x),$$

$$u''_N(x) = \sum_{j=0}^{N} u_j \phi''_j(x).$$

Since we are collocating, we only need to find u' and u'' at the collocation points.

$$u'_N(x_k) = \sum_{j=0}^{N} u_j \phi'_j(x_k) = \sum_{j=0}^{N} D_{kj} u_j, \quad \text{and}$$

$$u''_N(x_k) = \sum_{j=0}^{N} u_j \phi''_j(x_k) = \sum_{j=0}^{N} D^{(2)}_{kj} u_j,$$

where $D_{kj} \equiv \frac{d}{dx}\phi_j(x)\Big|_{x_k}$ and $D^{(2)}_{kj} = \frac{d^2}{dx^2}\phi_j(x)\Big|_{x_k}$. But since $\mathbf{u}'' = D\mathbf{u}' = D(D\mathbf{u}) = D^2\mathbf{u}$, $D^{(2)} = D^2$.

The *differentiation matrix* D is given by the M-file `cheb` due to Trefethen (see lab materials).

```
% CHEB  compute D = differentiation matrix, x = Chebyshev grid

  function [D,x] = cheb(N)
  if N==0, D=0; x=1; return, end
  x = cos(pi*(0:N)/N)';
  c = [2; ones(N-1,1); 2].*(-1).^(0:N)';
  X = repmat(x,1,N+1);
  dX = X-X';
  D  = (c*(1./c)')./(dX+(eye(N+1)));      % off-diagonal entries
  D  = D - diag(sum(D'));                 % diagonal entries
```

Don't bother trying to understand how it works — just take it that D is available.

So now we collocate at the Chebyshev points and

$$u_N''(x_k) + \bar{p}(x_k)u_N'(x_k) + \bar{q}(x_k)u_N(x_k) = \bar{r}(x_k),$$

becomes

$$D^2\mathbf{u} + \begin{pmatrix} \bar{p}_0 & 0 & \cdots & 0 \\ 0 & \bar{p}_1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \bar{p}_N \end{pmatrix} D\mathbf{u} + \begin{pmatrix} \bar{q}_0 & 0 & \cdots & 0 \\ 0 & \bar{q}_1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \bar{q}_N \end{pmatrix} \mathbf{u} = \begin{pmatrix} \bar{r}_0 \\ \bar{r}_1 \\ \vdots \\ \bar{r}_N \end{pmatrix} = \bar{\mathbf{r}}.$$

And you can solve this for $\mathbf{u}$.

How do we add in (Dirichlet) boundary conditions?

Since we know the value of u at $x = \pm 1$, $(k = 0, N)$, we don't need to collocate at these points. So we do not need the first and last rows of D and D^2.

So for the rest of our rows we have:

$$\begin{aligned} u'_k &= \sum_{j=0}^{N} D_{kj} u_j \qquad (k = 1, \dots, N-1) \\ &= \sum_{j=1}^{N-1} D_{kj} u_j + u_0 D_{k0} + u_N D_{kN}. \end{aligned}$$

Alternatively, you could just add the trivial equations for u_0, u_N to the system.

Exercise 1.20. Run the script `p13mod` to see an example of spectral colocation. Note this uses Barycentric interpolation which is an improved version of Lagrange interpolation. If you're interested details can be found in the Trefethen book, or the associated online materials.

Adding in Neumann boundary conditions etc. is possible but more involved. See `dmsuite` on MATLAB central.

Most of these commands have been made redundant with the release of the package `chebfun` availiable at **`http://www.chebfun.org/`**.

1.4 Galerkin Methods

Iserles §9.1

We consider the second order self-adjoint BVP:

$$-(D(x)u')' + q(x)u = f(x),$$

subject to Dirichlet boundary conditions $u(a) = \alpha$, $u(b) = \beta$. We also require $D(x) > 0, q(x) \geq 0$.

In Galerkin methods we change the formulation of our DE by projection (using an inner product) onto some vector space V, e.g. $C^1[a, b]$.

We require the projection of the residual onto V to vanish. This is equivalent to making the residual orthogonal to the space V using the usual inner product $(f, g) = \int_a^b fg\, dx$.

So we require

$$\int_a^b [-(D(x)u')' + q(x)u - f(x)]v(x)\, dx = 0, \qquad \forall v \in V,$$

where V is called the *test space.*

1.4.1 Weak Formulation

<u>Idea:</u> We integrate by parts (if possible) and obtain the equation:

$$[-D(x)u'v]_a^b + \int_a^b D(x)u'v'\, dx + \int_a^b q(x)uv\, dx = \int_a^b fv\, dx, \qquad \forall v \in V.$$

We call $D(x)u'(x)$ the *flux.* Note that our first term depends on boundary conditions.

For Neumann boundary conditions, the flux or u' is known on the boundaries so this term is known.

For Dirichlet boundary conditions we do not know the flux on the boundary. So to remove the boundary term we require $v = 0$ on the boundaries, this is the "don't test where you know u" principle (Hipfmaier). So now $v \in V_0$ where the 0 denotes $v = 0$ on the boundaries.

This is the *weak formulation* of the ODE. A solution of this weak formulation is called a *weak solution.* A weak solution in C^2 is a *classical solution* since it also satisfies the original equation. It is easier to prove things with the weak formulation because we are working with a larger space of functions.

1.4.2 Galerkin Equations

Principle 1: Make the residual orthogonal to a test space V. (Galerkin 1915)

Principle 2: Integrate by parts to lower smoothness requirements of the solution. (Seek a weak solution of the BVP).

Principle 3: Satisfy boundary conditions

- For Neumann boundary conditions there is no need to restrict the test space, so sometimes these boundary conditions are also called *natural boundary conditions.*
- For Dirichlet boundary conditions, remove boundary terms by restricting test space to have 0 on the boundaries. e.g. if the BCs are $u(a) = \alpha$, $u(b) = \beta$, then let $v \in C_0^1[a, b]$ so $v = 0$ at a and b. These are called *essential boundary conditions.*
- So we have the test space satisfying zero boundary conditions and the trial space satisfying specific boundary conditions for Dirichlet boundary conditions but no restrictions for Neumann boundary conditions.

Principle 4: Choose trial and test space to be finite dimensional function spaces U_N, V_N.

- For Dirichlet boundary conditions: $V = V_N^0$, $u_N = \phi_0 + \bar{u}$ where $\bar{u} \in V_N^0$ and ϕ_0 satisfies the boundary conditions. The space u_N exists and is an *affine space.*

  For V_N^0 choose any vector space of functions with zero boundary conditions.
- For Neumann boundary conditions $v \in V_N$, $u \in U_N$ where U_N and V_N are any finite dimensional vector spaces of functions.

Principles 1 to 4 combine to give a *Galerkin method.*

So if we look back at our original problem, let $u = \sum_{j=1}^{N} c_j \phi_j(x)$.

$$\begin{aligned} &(Du', v') + (qu, v) = (f, v) + D(b)u'(b)v(b) - D(a)u'(a)v(a) \quad \forall v \in V \\ \implies &(Du', \phi_k') + (qu, \phi_k) = (f, \phi_k) + D(b)u'(b)\phi_k(b) - D(a)u'(a)\phi_k(a) \quad k = 1, \dots, N \\ \implies &(D\sum_{j=1}^{N} c_j\phi_j'(x), \phi_k') + (q\sum_{j=1}^{N} c_j\phi_j(x), \phi_k) = (f, \phi_k) + B_k \quad k = 1, \dots, N. \end{aligned}$$

So we have a system of equations $A\mathbf{c} = \mathbf{f}$, where

$$\begin{aligned} A_{kj} &= \int_a^b D(x)\phi_k'(x)\phi_j'(x) + q(x)\phi_k(x)\phi_j(x)\,dx, \\ \mathbf{c} &= (c_1, \dots, c_N)^T, \\ f_k &= \int_a^b f(x)\phi_k(x)\,dx + B_k. \end{aligned}$$

These are the *Galerkin equations.* We call $\mathbf{f}$ the (global) load vector.

Note: $A = K + M$ where $K_{kj} = (D\phi_k', \phi_j')$ and $M_{kj} = (q\phi_k, \phi_j)$. We call K the stiffness matrix and M the mass matrix.

Eventually in MATLAB we will solve our system with the command `c = A \ f`.

There are choices for ϕ_0, V_N^0 and V_N that we can make.

For ϕ_0 and V_N^0 we can either choose

- Spaces with global support e.g. polynomials or trigonometric functions. These are called *Spectral Galerkin methods* (and ones that use Legendre polynomials specifically are called *nodal continuous Galerkin methods*; for more on this, see Kopriva). The big problem with these is that calculating the integrals is computationally expensive.

- We can instead choose to work in piecewise polynomial spaces. So ϕ_j will have local support. We call these spaces *finite element spaces.*

A key feature of any Galerkin method is the following property: (assuming $D(x) > 0$, $q(x) \geq 0$)

$$\|u - u_h\|_{H^1} \leq \min_{w_h \in V_h} \|u - w_h\|_{H^1},$$

where u is the exact solution, u_h is the Galerkin solution, w_h is any function in V_h and $\|\cdot\|_{H^1}$ is the norm of the Sobolev space H^1. This is the "correct" space to be working in: it is the space of functions in L^2 with weak derivatives in L^2 and its norm measures the distance between functions and their derivatives in L^2.

$$\|V\|_{H^1} = \int_a^b V^2 + V'^2 \, dx.$$

Motivation for weak derivative: If a function $f(x)$ is differentiable on $[a, b]$, then after integration by parts, for any smooth function $v(x)$ satisfying $v(a) = v(b) = 0$, we have

$$\int_a^b f(x)v'(x)dx = -\int_a^b f'(x)v(x)dx.$$

The function $f(x) = |x|$ is not differentiable but we can define its weak derivative as the step function $g(x) = \begin{cases} 1 & x \geq 0 \\ -1 & x < 0 \end{cases}$

Definition 1.21 (Weak Derivative).
A function $g(x)$ is defined to be the weak derivative of on $[a, b]$ if it satisfies

$$\int_a^b f(x)v'(x)dx = -\int_a^b f'(x)v(x)dx, \quad \forall v(x) \in C_0^\infty([a, b]),$$

where $C^\infty([a, b])$ is the set of functions with derivatives of any order being continuous and $C_0^\infty([a, b])$ denotes that functions should also vanishing at the boundaries $x = a$ and $x = b$.

This is the *best approximation property* or *Cea's lemma.* It says that the Galerkin solution is the best approximation to u from all the functions in V_h, when measured in the H^1-norm.

We also get

$$\|u - u_h\|_E = \min_{w_h \in V_h} \|u - w_h\|_E,$$

where $\|\cdot\|_E$ is the *energy norm.* $\|V\|_E^2 = \int_a^b D(x)|V'|^2 + q(x)V^2 \, dx$. The energy norm is the natural norm associated with the inner product defined by the ODE

$$(u, v)_a = \int_a^b D(x)u'v' + q(x)uv \; dx.$$

Using this inner product, the error is orthogonal to the space V_h:

$$(u - u_h, v_h)_a = 0 \quad \forall v_h \in V_h,$$

a property called *Galerkin orthogonality.*

If we have Dirichlet BCs then

$$\|u - u_h\|_{H_0^1} \leq \min_{w_h \in V_h} \|u - w_h\|_{H_0^1},$$

where H_0^1 is the Sobolev space of functions with 0 boundary conditions.

1.4.3 Finite Element Methods

Refs: Iserles, Gockenbach, Silvester, QSS, Suli

Principle 5: Choose ϕ_j to have local support. So $(\phi_j, \phi_k) = 0$ for most j, k and K, M and A are sparse. This is called the *finite element method.*

In FEM we construct V_h using piecewise polynomials in C^0, giving the X_h^k finite element spaces. (The pieces join on sets of measure 0 so we will have weak derivatives). We partition $[a, b]$ with a mesh consisting of $k_j = [x_j, x_{j+1}]$, $h_j = x_{j+1} - x_j$, with n subintervals which we call *elements.*

For X^k each piece is a polynomial of degree k. So we have $n(k+1)$ parameters to solve for. There are $(n-1)$ matching conditions at the mesh points which leaves us with $(nk+1)$ free parameters giving us the dimension of the space we are in. So we seek $(nk+1)$ basis functions for our space. We choose a cardinal basis/nodal basis $\phi_j(x_i) = \delta_{ij}$ with local support. We then map each element $k_j = [x_j, x_{j+1}]$ to a master element on $[0, 1]$.

Linear basis functions The master element for $n = 1$, $k = 2$ has 2 basis functions called *shape functions.* These are linear functions with the cardinal property $N_1(0) = 1$, $N_1(1) = 0$, $N_2(0) = 0$, $N_2(1) = 1$. This gives us the equations for the two functions, $N_1(\xi) = 1 - \xi$, $N_2(\xi) = \xi$.

We map back to k_j by $x_j(\xi) = x_j + \xi(x_{j+1} - x_j)$ or $\xi(x) = \frac{x - x_j}{x_{j+1} - x_j}$ which is an affine map. So there are two basic functions defined per element on k_j: we let $\phi_j = N_1(\xi(x))$, $\phi_{j+1} = N_2(\xi(x))$. So all our functions except our first and last have support on two elements while the first and last have support on only one element. The linear functions we defined are the *linear Lagrange elements* P_1. For a problem in X_0^1 we remove ϕ_1 and ϕ_{n+1} to satisfy the boundary conditions.

Quadratic basis functions For $k = 2$ we have $2n + 1$ parameters so on top of our $n + 1$ mesh points we define n extra nodes and have our basis functions cardinal on our set of $2n + 1$ points. So on our master element: $N_1 = (1 - \xi)(1 - 2\xi)$, $N_2 = 4(1 - \xi)\xi$ and $N_3 = \xi(2\xi - 1)$. We call N_2 a bubble function. So for even indices our basis functions will be bubble functions.

The support of the bubble functions ϕ_{2j} are h_j and the support of the non bubble functions ϕ_{2j+1} will be h_{j-1} and h_j. These are the *quadratic Lagrange elements* or P_2.

Theorem 1.22. *Assume $u \in H^s$ for $s \geq 2$ and $u_h \in V_h = X_h^k$. Then:*

$$\|u - u_h\|_{H_0^1} \leq Ch^l \|u\|_{H^{l+1}},$$

where $l = \min\{k, s - 1\}$.

Also,

$$\|u - u_h\|_{L^2} \leq Ch^{l+1} \|u\|_{H^{l+1}}.$$

So if u is smooth enough, $s \geq k + 1 \implies l = k$. Then $\|u - u_h\|_{H_0^1} \leq O(h^k)$ and $\|u - u_h\|_{L^2} \leq O(h^{k+1})$. Else $s \leq k \implies l = s-1$, then $\|u - u_h\|_{H_0^1} = O(h^{s-1})$ and $\|u - u_h\|_{L^2} \leq O(h^s)$. This type of behaviour is called a *regularity threshold.* These error bounds are tight (i.e. it is easy to construct functions that show these bounds, unlike the FDM estimates).

So if $k = 1$ our error is $O(h^2)$ if $s \geq 2$ and if $k = 2$ then our error is $O(h^3)$ if $s \geq 3$.

In practice, FEM implementations differ from others in 2 ways:

1. The stiffness/mass matrices and load vector are computed by doing integrals over each element, called *element stiffness matrices/vectors*, and assembled into the final global matrices/vectors.

2. Rather than treat Dirichlet boundary conditions using general functions with $u = \phi_0 + \bar{u} \quad (\bar{u} \in V_h^0)$. Choose $\phi_0 \in X^k \backslash X_0^k$ so $u \in X^k$ with non

zero end basis functions. We do integrals using $u = \sum_{j=1}^{N+1} u_h \phi_j$ then move the corresponding known Dirichlet node values to the right hand side of $A\mathbf{x} = \mathbf{b}$.

For linear Lagrange element, we have N elements and $N+1$ nodes giving global matrices of size $(N+1) \times (N+1)$. For each element we provide a list of nodes that make up the element in 1D so E_ℓ has nodes X_ℓ and $X_{\ell+1}$. The stiffness matrix and load vector are

$$\begin{aligned} K_{ij} &= \int_a^b D(x)\phi_i'\phi_j'\,dx, \quad \text{for} \quad i,j = 1,\dots,\text{degrees of freedom} \\ F_i &= \int_a^b f(x)\phi_i(x)\,dx + \text{boundary terms}, \end{aligned}$$

We could do this by looping over i,j, but we note that $K_{ij} = 0$ unless ϕ_i' and ϕ_j' overlap so there would be a lot of 0 terms. Instead we break things over the elements E_k:

$$\begin{aligned} K_{ij} &= \int_a^b D(x)\phi_i'\phi_j'\,dx \\ &= \sum_{\ell=1}^{N} \int_{x_\ell}^{x_{\ell+1}} D(x)\phi_i'\phi_j'\,dx \\ &= \sum_{\ell=1}^{N} \int_{E_\ell} D(x)\phi_i'\phi_j'\,dx, \end{aligned}$$

where each individual term in the sum is an element stiffness matrix. Similarly

$$F_i = \sum_{\ell=1}^{N} \int_{E_\ell} f(x)\phi_i\,dx.$$

The only basis functions that are non zero on element ℓ are ϕ_ℓ and $\phi_{\ell+1}$.

$$F_{E_\ell} = \begin{bmatrix} \int_{E_\ell} f(x)\phi_\ell\,dx \\ \int_{E_\ell} f(x)\phi_{\ell+1}\,dx \end{bmatrix},$$

$$K_{E_\ell} = \begin{bmatrix} \int_{E_\ell} D(x)\phi_\ell'^2\,dx & \int_{E_\ell} D(x)\phi_\ell'\phi_{\ell+1}'\,dx \\ \int_{E_\ell} D(x)\phi_{\ell+1}'\phi_k'\,dx & \int_{E_\ell} D(x)\phi_{\ell+1}'^2\,dx \end{bmatrix}.$$

So

$$\mathbf{f} = \begin{bmatrix} (f,\phi_1) \\ (f,\phi_2) \\ \vdots \\ \vdots \\ (f,\phi_{N+1}) \end{bmatrix} = \begin{bmatrix} \int_{x_1}^{x_2} f\phi_1 \\ \int_{x_1}^{x_3} f\phi_2 \\ \vdots \\ \vdots \\ \int_{x_N}^{x_{N+1}} f\phi_{N+1} \end{bmatrix}$$

$$= \begin{bmatrix} \int_{x_1}^{x_2} f\phi_1 \\ \int_{x_1}^{x_2} f\phi_2 \\ 0 \\ \vdots \\ 0 \end{bmatrix} + \begin{bmatrix} 0 \\ \int_{x_2}^{x_3} f\phi_2 \\ \int_{x_2}^{x_3} f\phi_3 \\ 0 \\ \vdots \end{bmatrix} + \cdots + \begin{bmatrix} 0 \\ \vdots \\ \vdots \\ \int_{x_N}^{x_{N+1}} f\phi_N \\ \int_{x_N}^{x_{N+1}} f\phi_{N+1} \end{bmatrix},$$

so we assemble the global load vector by putting each 2×1 element load vector into the appropriate place, with overlaps. We assemble K in a similar fashion by putting the two by two blocks on the diagonals of a $N+1$ matrix with the matrices overlapping on the corners.

We usually have to do these integrals using numerical quadrature rules. We should aim for quadrature rules that don't destroy the error estimates from the finite element analysis.

It is sufficient to use quadrature rules with degree of precision $\geq 2k-2$ for the space X^k (piece wise polynomials with degree k), where the degree of precision is the degree of polynomials quadrature rules integrate exactly.

Example 1.23 (Quadrature rules)**.**

1. Trapezoid Rule: Integrates linear functions exactly so has degree of precision 1.
2. Simpson's Rule: Has degree of precision 3.
3. Gauss s-stage rules have degree of precision $2s-1$.
4. Gauss-Lobatto rules have degree of precision $2s-3$.

We map each integral to the master element and then integrate over $[0, 1]$.

$$\int_{x_j}^{x_{j+1}} f(x)\phi_j \, dx = h_j \int_0^1 f(x(\xi))N_1(\xi) \, d\xi,$$
$$\int_{x_j}^{x_{j+1}} f(x)\phi_{j+1} \, dx = h_j \int_0^1 f(x(\xi))N_2(\xi) \, d\xi.$$

Example 1.24 (Gauss rules to use)**.**

1. X^1:
   - Trapezoid rule: $\int_0^1 g(\xi) \, d\xi = \frac{1}{2}[g(0) + g(1)]$.
   - Midpoint rule: $\int_0^1 g(\xi) \, d\xi = g(\frac{1}{2})$.

   Both of the above methods have degree of precision 1 and so are fine where working with X^1.
2. X^2:
   - Simpson's rule: $\int_0^1 g(\xi) \, d\xi = \frac{1}{6}[g(0) + 4g(\frac{1}{2}) + g(1)]$.
   - Two point Gauss rule.

   Both of the above methods have degree of precision 3 and so are fine where working with X^2.
3. X^3:
   - Three point Gauss rule.

   The three point Gauss rule has degree of precision 5 and so is fine when working with X^3.

So we have an equation $A\mathbf{u} = \mathbf{f}$, we then modify the RHS by imposing boundary conditions and then remove equations for known nodes to obtain $\bar{A}\bar{\mathbf{u}} = \bar{\mathbf{f}}$. We solve the system and then post process.

Example 1.25. Consider when we are using Lagrange quadratic elements (working in X^2). If we have N elements then we will have $2N + 1$ basis functions with N bubble functions which we give even indices and $N + 1$ edge functions.

Then

$$F_{E_1} = \begin{bmatrix} (f,\phi_1) \\ (f,\phi_2) \\ (f,\phi_3) \end{bmatrix},$$

$$F_{E_2} = \begin{bmatrix} (f,\phi_3) \\ (f,\phi_4) \\ (f,\phi_5) \end{bmatrix},$$

and they will overlap in one entry in the global **f** vector.

$$K_{E_1} = \begin{bmatrix} K_{11} & K_{12} & K_{13} \\ K_{21} & K_{22} & K_{23} \\ K_{31} & K_{32} & K_{33} \end{bmatrix},$$

$$K_{E_2} = \begin{bmatrix} K_{33} & K_{34} & K_{35} \\ K_{43} & K_{44} & K_{45} \\ K_{53} & K_{54} & K_{55} \end{bmatrix},$$

where the element stiffness matrices will overlap in the upper left and lower right entries of the global stiffness matrix.

1.5 Handling nonlinearities

There are two main ways of dealing with non linearities as shown in the following diagram:

$$\begin{tikzcd}
\text{Non-linear BVP} \arrow[r, "\text{discretize}", "\text{FDM, collocation, FEM}"'] \arrow[d, "\text{quasilinearise}"'] & \text{Non-linear algebraic equations} \arrow[d, "\text{linearise}"] \\
\text{Sequence of linear BVPs} \arrow[r, "\text{discretize}"] & \text{Solve a sequence of linear systems}
\end{tikzcd}$$

We will first examine the top route, later we will look at the bottom route.

First consider a simple non-linear BVP:

$$u'' + 1 - u^2 = 0, \qquad u(0) = 0, u(1) = 1,$$

with $N + 1$ mesh points or $N - 1$ internal points.

If we approximate u'' with Taylor series using a central difference method as we did in the FDM we obtain:

$$\frac{u_{j-1} - 2u_j + u_{j+1}}{h^2} + 1 - u_j^2 = 0, \qquad j = 2, \dots N, \qquad u_1 = 0,\, u_{N+1} = 1.$$

Which gives the system of nonlinear equations:

$$\frac{1}{h^2}\begin{pmatrix} -2 & 1 & 0 & \cdots & \cdots \\ 1 & -2 & 1 & 0 & \vdots \\ 0 & 1 & \ddots & \ddots & \vdots \\ \vdots & \vdots & \ddots & \ddots & \vdots \\ \vdots & \cdots & \cdots & \cdots & -2 \end{pmatrix} \begin{pmatrix} u_2 \\ \vdots \\ \vdots \\ \vdots \\ u_N \end{pmatrix} + \begin{pmatrix} 1 - u_2^2 \\ \vdots \\ \vdots \\ \vdots \\ 1 - u_N^2 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ \vdots \\ \vdots \\ -\frac{1}{h^2} \end{pmatrix}$$

or `A*u + 1-u.^2=b` in MATLAB.

This is a nonlinear system of algebraic equations

$$\mathbf{F}(\mathbf{u}) = A\mathbf{u} + 1 - \mathbf{u} \odot \mathbf{u} - \mathbf{b} = \mathbf{0}.$$

1.5.1 Nonlinear equations

Popular methods to solve a single nonlinear equation, $f(x) = 0$, include:

- fixed point or Picard iteration (easy to implement but usually linearly convergent)
- Newton's Method (need f' but quadratically convergent)
- bisection (globally convergent but slow)
- secant method (don't need f' but super linear convergence)

We measure the error by letting $e_n = x_n - x^*$. If $e_{n+1} \leq Ce_n$ and $|C| < 1$ (required for convergence) then we have linear convergence. Better convergence is called superlinear convergence. Generally, fixed point iteration will converge linearly. If $e_{n+1} \leq Ce_n^2$ we have quadratic convergence.

In this course we will look at:

- fixed point iteration (or Picard iteration); and
- Newton's method (and variants).

1.5.2 Fixed point (Picard) iteration

In 1D, rewrite $f(x) = 0$ in fixed point form $x = g(x)$ (this form will not be unique) and let $x_{n+1} = g(x_n)$. If the sequence converges to some fixed point of $x^* = g(x^*)$ then $f(x^*) = 0$. If we have a *contractive mapping* (i.e. $|g(x) - g(y)| < |x - y|$) on some interval containing the root then we can guarantee a solution.

Example 1.26. Solve $x^3 + 4x^2 - 10 = 0$ using different fixed point iterations. `FixedPoint.m, FixedPointErrors.m`.

In higher dimensions, given $\mathbf{F}(\mathbf{u}) = \mathbf{0}$, rewrite it as $\mathbf{u} = \mathbf{G}(\mathbf{u})$.

Then, if $\|\mathbf{G}(\mathbf{x}) - \mathbf{G}(\mathbf{y})\| \leq \|\mathbf{x} - \mathbf{y}\|$ for some vector norm in an interval containing the solution then we can guarantee convergence to the fixed point (in that norm).

Example 1.27. In our example above, we had the equation almost in fixed point form `A*u =b-(1-u.^2)`

suggesting the fixed point iteration

$$\mathbf{u}_{N+1} = \mathbf{A}^{-1}[\mathbf{b} - (1 - \mathbf{u}_N \odot \mathbf{u}_N)].$$

Or `u = A\(b-(1-u.^2))` in MATLAB

Remember that we almost never use the matrix inverse in MATLAB; instead we solve the linear system.

This method is easy to write and can converge but doesn't always. When it converges usually converges linearly.

Example 1.28. An example of using fixed point iteration to solve a nonlinear ODE using colocation. `p14.m`

1.5.3 Newton's Method

We obtain Newton's Method in 1D by locally approximating the function $f(x)$ by its linear approximation (tangent line).

So in 1D,

$$f(x_{n+1}) \approx f(x_n) - f'(x_n)(x_{n+1} - x_n),$$

or written as an update

$$x_{n+1} = x_n + \Delta x_n, \quad \text{where} \quad \Delta x_n = \frac{-f(x_n)}{f'(x_n)}.$$

We can get quadratic convergence with Newton's method, so $e_{n+1} \leq K e_n^2$, but it doesn't always converge. **When Newton's Method converges, it converges fast.**

Example 1.29. An example of Newton's method. `Lec5Newton.m, Newton.m.`

In 2D, we try and solve $F(u,v) = 0$, and $G(u,v) = 0$.

At (u_n, v_n) expand F, G in terms of Taylor series to obtain tangent planes. Then choose (u_{n+1}, v_{n+1}) where the tangent plane cuts the u, v plane.

So:

$$0 = F(u_{n+1}, v_{n+1}) \approx F(u_n, v_n) + \left.\frac{\partial F}{\partial u}\right|_n (u_{n+1} - u_n) + \left.\frac{\partial F}{\partial v}\right|_n (v_{n+1} - v_n),$$

$$0 = G(u_{n+1}, v_{n+1}) \approx G(u_n, v_n) + \left.\frac{\partial G}{\partial u}\right|_n \Delta u_n + \left.\frac{\partial G}{\partial v}\right|_n \Delta v_n,$$

Or in matrix form

$$\begin{pmatrix} \left.\frac{\partial F}{\partial u}\right|_n & \left.\frac{\partial F}{\partial v}\right|_n \\ \left.\frac{\partial G}{\partial u}\right|_n & \left.\frac{\partial G}{\partial v}\right|_n \end{pmatrix} \begin{pmatrix} \Delta u_n \\ \Delta v_n \end{pmatrix} = - \begin{pmatrix} F(u_n, v_n) \\ G(u_n, v_n) \end{pmatrix} = -\mathbf{F}(\mathbf{u_n}).$$

Where the first matrix is called the Jacobian, $J(\mathbf{u}_n)$.

So our system of equations is:

$$J(\mathbf{u_n}) \Delta \mathbf{u_n} = -\mathbf{F}(\mathbf{u_n}), \quad n = 0, 1, 2, \ldots,$$

where the update is

$$\mathbf{u_{n+1}} = \mathbf{u_n} + \Delta\mathbf{u_n}.$$

If it converges then a Newton's method typically converges quadratically.

The downside is that for an $N \times N$ system of equations we need to compute the $N \times N$ matrix J of partial derivatives and we need to solve a sequence of linear systems which generally takes $O(N^3)$ work (of course less if J is sparse). For small problems, this may still be manageable.

Example 1.30. Considering $u'' + 1 - u^2 = 0$ again,

$$F_i = \sum_j A_{ij}u_j + (1 - u_i^2) - b_i = 0,$$

$$J_{ij} = \frac{\partial F_i}{\partial u_j} = A_{ij} - 2u_j\delta_{ij}.$$

In MATLAB,

```
F =A*u+h^2*(1-u.^2)-b;
J =A-2 *diag(u);
Du =-J\F;
u=u+Du;
```
(Alternatively, `u=u-J\F;`)

In this case J was sparse (tridiagonal) so it's not too hard to compute or solve with. It's a good idea to declare it sparse or use `spdiags` to create it.

1.5.4 Improving on Newton

In theory Newton's method is almost optimal in terms of speed of convergence but there are three issues to deal with in when applying Newton's method to large systems:

1. Applicability: We need to know $J_{ij} = \frac{\partial F_i}{\partial u_j}$ which is often expensive because it has N^2 elements. It is also often hard to evaluate the Jacobian analytically.

   To remedy this we use a finite difference approximation for the partial derivative.
$$\frac{\partial F_i}{\partial u_j} \approx \frac{F_i(u_j + \delta) - F_i(u_j)}{\delta},$$
   where $\delta \sim C\sqrt{r}$ where C is a constant and r is the unit roundoff, this minimises the combined truncation and round off error as seen in 3rd year. This retains quadratic convergence down to $e_{n+1} \sim r$, which is the best the machine can do anyway.

   Example 1.31. Example code which calculates the jacobian, try it with your favourite function. `diffjac.m`.

   Alternatively, can use *automatic differentiation* (Ref: Andreas Griewank) which finds the action of the Jacobian on F.

2. Efficiency: To save work, we want to evaluate J, which has N^2 elements as seldom as possible whilst also solving $J\Delta u = -f$ ($O(N^3)$ operations) as seldom as possible.

   Recall that when we solve $J\Delta u = -f$ we factorise $J = LU$ into upper and lower triangular matrices then solve the triangular systems which takes $O(N^2)$ time. So we modify Newton's method to evaluate J only every so often instead of every step.

   - The simplest method is the *chord method.*
     In this method we start at $\mathbf{u_0}$ and only evaluate and factorise $J(\mathbf{u}_0)$ once and then solve
$$(LU)_0\Delta u_n = -F(\mathbf{u_n}), \quad u_{n+1} = u_n + \Delta u_n,$$
     repeatedly.
     This is Newton's method but only with the initial J; much cheaper but only has linear convergence.

- We can also re-evaluate J and refactorise every m steps (see Shamanski 1967, *A modification of Newton's method* Ukrain. Mat. Z. Vol. 19). ($m = 1$ is Newton's method and $m = \infty$ is the chord method).

  In each "step" between each factorisation $e_{n+1} \sim l^{m+1}$ but need m solves with current J so the effective order is $\frac{m+1}{m}$.
- Can also use a Hybrid method (Ref: Kelley 1995).

  This monitors $\sigma = \frac{\|F(x_{n+1})\|}{\|F(x_n)\|}$, the ratio of the residuals i.e. how fast the residual is decreasing. If $\sigma < \rho$, for some threshold $\rho < 1$ (e.g. $\rho = \frac{1}{2}$) then keep solving with the current J, otherwise re-evaluate J.

  For large problems this ends up doing the chord method once σ is small enough. These are called *modified Newton methods*

  Example 1.32. `Chandra.m, nsol.m`

  Try $c = 0.9, 0.99, 0.999, 0.9999$ with $N = 200 - 300$.
- The above methods all solve the system exactly (directly). We can also solve the systems approximately using an iterative solver – the idea being that since each system is an approximation anyway there might be no point solving it exactly. This leads to *inexact Newton methods* and *truncated Newton methods.*

3. Convergence: Newton's method only solves equations if $\mathbf{u}_0$ is close enough to the solution $\mathbf{u}^*$. For example (Ref:Kantorovich 1948) proved that there is convergence in a ball of radius r, i.e. Newton's method converges if $\|\mathbf{u}_0 - \mathbf{u}^*\| < r$. But we don't know $\mathbf{u}^*$, how do we find $\mathbf{u}_0$?

   - There are *homotopy/continuation methods* Ref: Allgower and Georg.

     Instead of solving the system $\mathbf{F}(\mathbf{u}) = \mathbf{0}$ for $\mathbf{u}_0$ we "creep up" on our solution using a parameter of the problem

     Example 1.33. It might be hard to slve the problem with the true parameter value $B = 3$. Instead solve with $B = 1$, then use that solution as initial guess for $B = 2$ etc.

     or by inserting a parameter:

     $$\mathbf{F}_\lambda(\mathbf{u}) = \lambda\mathbf{F}(\mathbf{u}) + (1 - \lambda)\mathbf{G}(\mathbf{u}) = 0,$$

     where we want to solve $\mathbf{F}(\mathbf{u}) = 0$ but $\mathbf{G}(\mathbf{u}) = 0$ is much easier to solve. We creep up on our solution by slowly varying λ from 0 to 1 using the last solution to solve for the next step.

- There are also *damped Newton methods* which seek to globalize Newton's method by increasing the basin of attraction, r.

  To do this, we observe that often the step size is too big causing us to over shoot the root, even though we have the direction right. To do this we replace the Newton step (or modified Newton step) by:

  $$\mathbf{u}_{n+1} = \mathbf{u}_n + \lambda \Delta u_n,$$

  where $\lambda \in (0,1)$ is the damping factor.

  We can modify this further by varying λ from step to step, always choosing λ_n such that $\|F\|$ falls by a "sufficient" amount (Ref: Armijo 1966).

  Example 1.34. For examples of Newton methods in action see `NewtonArmijoTest.m`.

Note: damped Newton iteration is used in MATLAB's `pdetool`.

1.5.5 Quasi-linearisation

Consider once again the boundary value problem:

$$u'' + q(u) = f(x), \quad u(a) = \alpha, u(b) = \beta.$$

Instead of discretizing then linearizing, we can linearize the operator to produce a sequence of linear BVPs. This is called Newton-Kantorovich iteration or Newton iteration in a function space or quasi-linearization.

We wish to approximate the operator $q(u)$ by a linear approximation about the function u_0.

$$q(u) \approx q(u_0) + q'(u_0)(u - u_0),$$

where $q'(u_0)$ denotes the *Fréchet* derivative.

We first guess a solution for u_0, then solve

$$\begin{aligned} u_1'' + q'(u_0)(u_1 - u_0) + q(u_0) &= f(x), \\ \implies u_1'' + q'(u_0)u_1 &= f(x) - q(u_0) + u_0 q'(u_0), \end{aligned}$$

subject to $u_1(a) = \alpha$ and $u_1(b) = \beta$.

Repeating this process we get

$$\begin{cases} u_{n+1}'' + q'(u_n)u_{n+1} = f(x) - q(u_n) + u_n q'(u_n), \\ u_{n+1}(a) = \alpha, u_{n+1}(b) = \beta, \end{cases}$$

until convergence, which is not guaranteed but fast if it happens.

Or we could solve for the Newton update at each iteration

$$\begin{cases} \Delta u_n'' + q'(u_n)\Delta u_n = f(x) - q(u_n) - u_n'', \\ u_{n+1} = u_n + \Delta u_n, \\ \Delta u_n(a) = 0, \Delta u_n(b) = 0. \end{cases}$$

The same technique can be used for nonlinear BCs. This method was developed by Richard Bellman.

Quasi-linearization is used by Chebfun's `solvebvp` with the derivatives being done by automatic differentiation.

2 2D Elliptic PDEs

2.0.1 2D PDEs

There are three classes of second order linear PDEs: elliptic, parabolic and hyperbolic. Classification is based on their characteristics, with elliptic PDEs having no real characteristics, parabolic having 1 and hyperbolic having 2. Elliptic equations require only boundary values to solve, while parabolic and hyperbolic PDEs also require initial conditions and are often called *evolution equations.*

Example 2.1.

$$\begin{aligned} -\nabla^2 u &= f, && \text{The Poisson equation, elliptic,} \\ \nabla^2 u &= u_t, && \text{The diffusion equation, parabolic,} \\ \nabla^2 u &= u_{tt}, && \text{The wave equation, hyperbolic.} \end{aligned}$$

We will first consider (a subclass of) the elliptic PDEs:

$$-\nabla \cdot (D(x,y)\nabla u) + q(x,y)u = f(x,y),$$

where $D > 0$ and $q \geq 0$ to satisfy ellipticity.

The simplest example satisfying these conditions, where $q = 0$ and D is constant, is the Poisson equation, seen above.

For simple domains (e.g. rectangles) we have many choices of discretisation: finite differences, finite element methods, spectral collocation (e.g. Chebyshev), spline collocation (e.g. piecewise cubics in C^1), 'fast Poisson solvers' (only for the Poisson equation), and multipole methods (dating to about 1988). But once the domain becomes difficult, finite element methods "reign supreme".

We will also find that as the size of the linear system gets bigger, we will need to consider alternatives to "\" in Matlab.

2.1 Elliptic PDEs: Finite Difference

Ref: Leveque Ch. 3

Consider the equation

$$-\nabla(D(x,y)\nabla u) = f(x,y), \quad x, y, \in \Omega.$$

If we have Dirichlet boundary conditions then:

$$u = g_D(x,y), \quad \text{on } \partial\Omega.$$

If we have Neumann boundary conditions then:

$$\mathbf{q} \cdot \hat{\mathbf{n}} = g_N(x,y), \quad \text{on } \partial\Omega,$$

where $\mathbf{q} = -D\nabla u$ is the flux at the boundary as $\frac{\partial u}{\partial n} \equiv \hat{\mathbf{n}} \cdot \nabla u$.

Finally Robin boundary conditions are:

$$\alpha u + \beta \mathbf{q} \cdot \hat{\mathbf{u}} = g_R, \quad \text{on } \partial\Omega.$$

2.1.1 Discretising the domain

First we need to define the mesh inside Ω. Immediately there are two questions to answer:

- How to handle the curved boundaries?
- How to define the mesh in 2D?

We will avoid these questions for the moment by considering a rectangle aligned with the axes. Then the use of a tensor product mesh $\{x_i\} \otimes \{y_i\}$, which gives a set of mesh points

$$\{x_i, y_i\}, \quad i = 0, \dots, m+1 \quad j = 0 \dots m+1,$$

with $\Delta x_i = x_{i+1} - x_i$, $\Delta y_j = y_{j+1} - y_j$ in general. If we have a (square) uniform mesh then $h_i = k_i = h = k$.

Let's simplify to the case $D(x, y) = 1$ and consider Dirichlet BCs.

Then there are $(m+2)^2$ mesh points, m^2 internal mesh values $u_{ij} = u(x_i, y_j)$ and $4m + 4$ boundary points (which are known if given Dirichlet boundary conditions). So we need m^2 equations for the unknowns u_{ij}.

Using central differences for $-\nabla^2 = -\delta_{xx} - \delta_{yy}$, we obtain a 5 point stencil.

$$\begin{aligned} -\nabla^2 u_{ij} &\approx -\frac{1}{\Delta x^2}(u_W - 2u_C + u_E) - \frac{1}{\Delta y^2}(u_N - 2u_C + u_S) \\ &\overset{\Delta x = \Delta y = h}{=} -\frac{1}{h^2}(u_W + u_E + u_S + u_N - 4u_C) + O(h^2) = f_C. \end{aligned}$$

Therefore for each internal mesh point (m^2 equations).

$$-\frac{1}{h^2}(u_{i-1,j} + u_{i+1,j} + u_{i,j+1} + u_{i,j-1} - 4u_{ij}) = f_{ij}, i, j = 1, \dots, m.$$

2.1.2 Ordering unknowns:

There are many possible orderings, each gives a different linear system which can affect the efficiency of the solution (when using direct methods).

We will use the natural ordering which defines a mapping from the 2D index set i, j to the 1D index p, $p = (j-1)m + i$.

Other possible orderings are possible, for example the red/black (checkerboard) ordering.

Given an ordering and a five point stencil, we obtain a linear system for mesh values u_{ij}.

Example 2.2. Consider a 6×6 mesh, so $i, j = 0, \dots, 5$. $m = 3$ so there are 9 internal mesh points and 16 boundary conditions.

On each internal mesh point $u_W + u_E + u_N + u_S - 4u_C = h^2 f_C$.

$$\begin{aligned}
p = 1, (1,1) :& \frac{1}{h^2}\left(-4u_1 + u_2 + u_4 + u_{0,1} + u_{1,0}\right) = f_{1,1} \\
p = 2, (2,1) :& \frac{1}{h^2}\left(u_1 - 4u_2 + u_3 + u_5 + u_{2,0}\right) = f_{2,1} \\
p = 3, (3,1) :& \frac{1}{h^2}\left(u_2 - 4u_3 + u_6 + u_{3,0} + u_{4,1}\right) = f_{3,1} \\
p = 4, (1,2) :& \frac{1}{h^2}\left(u_1 - 4u_4 + u_5 + u_7 + u_{0,2}\right) = f_{1,2} \\
p = 5, (2,2) :& \frac{1}{h^2}\left(u_2 + u_4 - 4u_5 + u_6 + u_8\right) = f_{2,2} \\
p = 6, (3,2) :& \frac{1}{h^2}\left(u_3 + u_5 - 4u_6 + u_9 + u_{4,2}\right) = f_{3,2} \\
p = 7, (1,3) :& \frac{1}{h^2}\left(u_4 - 4u_7 + u_8 + u_{0,3} + u_{1,4}\right) = f_{1,3} \\
p = 8, (2,3) :& \frac{1}{h^2}\left(u_5 + u_7 - 4u_8 + u_9 + u_{2,4}\right) = f_{2,3} \\
p = 9, (3,3) :& \frac{1}{h^2}\left(u_6 + u_8 - 4u_9 + u_{3,4} + u_{4,3}\right) = f_{3,3}.
\end{aligned}$$

where red denotes the known boundary conditions.

We now move the known Dirichlet data to the RHS to get

$$\begin{aligned}
p = 1, (1, 1) :& \frac{1}{h^2}(-4u_1 + u_2 + u_4) = f_{1,1} - \frac{u_{0,1} + u_{1,0}}{h^2}\\
p = 2, (2, 1) :& \frac{1}{h^2}(u_1 - 4u_2 + u_3 + u_5) = f_{2,1} - \frac{u_{2,0}}{h^2}\\
p = 3, (3, 1) :& \frac{1}{h^2}(u_2 - 4u_3 + u_6) = f_{31} - \frac{u_{30} + u_{4,1}}{h^2}\\
p = 4, (1, 2) :& \frac{1}{h^2}(u_1 - 4u_4 + u_5 + u_7) = f_{1,2} - \frac{u_{0,2}}{h^2}\\
p = 5, (2, 2) :& \frac{1}{h^2}(u_2 + u_4 - 4u_5 + u_6 + u_8) = f_{2,2}\\
p = 6, (3, 2) :& \frac{1}{h^2}(u_3 + u_5 - 4u_6 + u_9) = f_{3,2} - \frac{u_{4,2}}{h^2}\\
p = 7, (1, 3) :& \frac{1}{h^2}(u_4 - 4u_7 + u_8) = f_{1,3} - \frac{u_{0,3} + u_{1,4}}{h^2}\\
p = 8, (2, 3) :& \frac{1}{h^2}(u_5 + u_7 - 4u_8 + u_9) = f_{2,3} - \frac{u_{2,4}}{h^2}\\
p = 9, (3, 3) :& \frac{1}{h^2}(u_6 + u_8 - 4u_9) = f_{3,3} - \frac{u_{3,4} + u_{4,3}}{h^2}.
\end{aligned}$$

i.e. a linear system of the form $-\frac{1}{h^2}A\mathbf{u} = \frac{1}{h^2}\mathbf{b} + \mathbf{f}$ where A is a block tridiagonal matrix (also called A_5 due to the 5 point stencil)

$$A = -\frac{1}{h^2}\begin{pmatrix} T & I & 0 \\ I & T & I \\ 0 & I & T \end{pmatrix},$$

where I is the 3×3 identity matrix, and

$$T = \begin{pmatrix} -4 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & -4 \end{pmatrix},$$

is a 3×3 tridiagonal matrix.

$$\mathbf{b} = \begin{pmatrix} u_{0,1} + u_{1,0} \\ u_{2,0} \\ \vdots \\ \vdots \\ u_{3,4} + u_{4,3} \end{pmatrix},$$

and $\mathbf{f}$ is as given by the RHS of the PDE evaluated at the mesh points $f_p = f(x_i, y_j)$.

A is symmetric, block tridiagonal, banded with band-width m, and sparse. (There are 74 non zero entries in the 256 entries of the matrix.) In general the sparsity goes to $\frac{5}{N}$ as $N \to \infty$ where $N = m^2$.

So as we can see, we would like to use sparse matrix storage to only store the non-zero entries of the matrix. Otherwise if we have a 1000×1000 mesh, then $N = m^2 = 10^6$ and $A = (10^6)^2 = 10^{12}$ entries, which is about 8 terabytes!. Type in `help sparse` in MATLAB to see uses of sparse storage in `speye` and `spdiags`. For the moment we will assume we'll solve this system using \ which uses direct methods for sparse matrices, but later we'll discuss alternatives.

2.1.3 Error Analysis

As before, the local truncation error is the residual when the exact solution is put into the finite difference formula.

For a five point stencil, if we do the algebra (as in the 1D case) we get

$$\tau_{ij} = \frac{1}{12}h^2(u_{xxxx} + u_{yyyy}) + O(h^4),$$

so the method is consistent of order 2.

Again, as in 1 dimension, the error solves the equation $A\mathbf{E} = -\tau$ so if we can bound A^{-1} in some norm we have a stability result.

Example 2.3. Leveque §3.4 proves $\|A^{-1}\|_2 < \frac{1}{2\pi^2}$. He does this by knowing the eigenvalues of A since if A is symmetric $\|A\|_2 = \max|\lambda_i|$. This stability bound proves the method is stable, so the five point stencil method for the Poisson equation is convergent of order 2.

How convergence depends on the smoothness of f and the boundary conditions is ‘difficult’ and beyond the scope of the course (see Iserles for details).

2.1.4 2D Example

Example 2.4. Run the example `delsquare`. Note this is just a copy of an inbuilt MATLAB example, see `showdemo delsqdemo`. Experiment with different domain shapes, how does the node ordering affect the structure of the linear system?

This code solves the problem

$$\nabla^2 = -1,$$

on various domains with $u = 0$ on the boundaries. It uses the following commands: `numgrid`, `numgrid(R,n)`, `delsq`, `nnz`, `spy`. Look up what they all do.

Note: the commands

`A5 = gallery('poisson', n)` and

`A5 = delsq(numgrid('S',n))`

produce the same matrix (check this for yourselves!)

Recap of FD Methods

- Easy for simple geometry e.g. rectangles.
- Convergence theory can be tricky.
- Produces large sparse matrices, even in 2D.

2.1.5 Other topics

- There are many nice properties of the finite difference method that rely on the particular self-adjoint elliptic form (e.g. A is symmetric positive definite).

  Once we add the extra convection term, $\propto \nabla u$, we lose lots of nice properties and things become harder to prove, and it is also harder to converge fast.

- Curved Regions also make finite difference methods messy, but possible.

  See for example 'Immersed Interface Method' Ref: Li & Leveque 1994.

- Nonlinear elliptic PDEs can be tackled by quasi-linearisation or by Newton's method and its variants (these are used in the PDE toolbox).

- For easy problems ($-\nabla^2 u = f$) on a square there are special 'fast Poisson solvers' that don't generalize to the variable coefficient case, or other coordinate systems.

- Higher order methods:

  1. For $-\nabla^2 u = f$, there is a 4th order method (Ref: Collatz) that is not too difficult.
     At the internal mesh points apply a 9-pt stencil:

     $$\begin{aligned}\nabla_9^2 u_{ij} =& \frac{1}{6h^2}[4u_{i-1,j} + 4u_{i+1,j} + 4u_{i,j-1} + 4u_{i,j+1} + \\ & u_{i-1,j-1} + u_{i-1,j+1} + u_{i+1,j-1} + u_{i+1,j+1} - 20u_{i,j}].\end{aligned}$$

     Then (it can be shown that)

     $$\nabla_9^2 u(x_i, y_j) = \nabla^2 u + \underbrace{\frac{1}{12}h^2(u_{xxxx} + 2u_{xxyy} + u_{yyyy}) + O(h^4)}_{\tau_{ij}},$$

     which doesn't seem to help.
     But then

     $$\begin{aligned}\tau_{ij} =& \frac{1}{12}h^2\nabla^2(\nabla^2 u) + O(h^4) \\ =& -\frac{1}{12}h^2\nabla^2 f + O(h^4).\end{aligned}$$

     Therefore, if we know f analytically, we can evaluate the first term of τ_{ij}.

Example 2.5. For the special case $f = 0$ then error is $O(h^4)$, $\nabla_9^2 u = 0 \to A_9\mathbf{u} = \mathbf{b}$, $\tau = O(h^4)$.

Back to Poisson's equation: since

$$-\nabla_9^2 u = f + \frac{1}{12}h^2\nabla^2 f + O(h^4),$$

then if we solve the *different problem*:

$$-\nabla_9^2 u = f + \frac{1}{12}h^2\nabla^2 f = \tilde{f},$$

then

$$-\nabla^2 u + \frac{1}{12}h^2\nabla^2\tilde{f} + O(h^4) = \tilde{f},$$

therefore

$$-\nabla^2 u + \frac{1}{12}h^2\nabla^2 f + O(h^4) = f + \frac{1}{12}h^2\nabla^2 f,$$

so

$$-\nabla^2 u + O(h^4) = f,$$

i.e. we've solved $-\nabla^2 u = f$ to $O(h^4)$.

This also works even if we don't know f analytically, but just at grid points. Just form

$$\begin{aligned}\tilde{f}_{ij} =& f_{ij} + \frac{1}{12}h^2\nabla_5^2 f_{ij}\\ =& f_{ij} + \frac{1}{12}h^2(\nabla^2 f + O(h^2))\\ =& f_{ij} + \frac{1}{12}h^2\nabla^2 f_{ij} + O(h^4),\end{aligned}$$

so we still get a 4th order method.

2. **Method of deferred corrections**

This is a more general method that can be applied to more equations. (See Ref: Ascher, Mattheij, Russel)

Solve $A_5\hat{\mathbf{u}} = \mathbf{f}$ which gives $\mathbf{u}$ with an error $O(h^2)$. We know

$$\tau_{ij} = \frac{1}{12}h^2(u_{xxxx} + u_{yyyy}) + O(h^4).$$

Then, since $A_5\mathbf{u} - \mathbf{f} = \tau$ and $A_5\hat{\mathbf{u}} - \mathbf{f} = \mathbf{0}$,

$$A_5(\hat{\mathbf{u}} - \mathbf{u}) = -\tau,$$

so

$$A_5\mathbf{E} = -\tau.$$

We now attempt to use $\hat{\mathbf{u}}$ to estimate τ. For example we can use central differences to get u_{xxxx} and u_{yyyy} to $O(h^2)$. Then we obtain

$$\hat{\tau} = \tau + O(h^4),$$

and solve

$$A_5\mathbf{E} = -\hat{\tau},$$

then update

$$\tilde{\mathbf{u}} = \hat{\mathbf{u}} - \mathbf{E},$$

which has improved our estimate to $O(h^4)$.

2.2 Finite Element Methods in 2D

Consider the equation:

$$-\nabla \cdot (D(x,y)\nabla u(x,y)) + q(x,y)u(x,y) = f(x,y),$$

where $D > 0$, $q \geq 0$.

(If $q = 0$ then we have the Poisson equation, and if both $q = 0$ and $f = 0$ then we have the Laplace equation.)

How do we solve a PDE using Finite Element Methods?

1. Transform the PDE to its weak form.
2. Choose test/trial spaces (these depend on the boundary conditions).
3. Form the Galerkin equations.
4. Choose a mesh.
5. Choose finite element space on the mesh (which will generate a sparse linear system which is solved to get the solution).

2.2.1 Step 1: weak form

Multiply the equation by a test function v and integrate over the domain Ω, to get

$$\int_\Omega -\nabla\cdot(D(x,y)\nabla u)v\,dA + \int_\Omega quv\,dA = \int_\Omega fv\,dA.$$

Use Green's identity to lower the order of the derivative:

$$\implies \int_\Omega (\nabla\cdot\mathbf{u})v\,dA = -\int_\Omega \mathbf{u}\cdot\nabla v\,dA + \int_{\partial\Omega}(\mathbf{u}\cdot\mathbf{n})v\,ds, \qquad \text{(Green's Identity)}$$

where $\mathbf{n}$ is a unit normal pointing outwards from Ω.

Proof. Greens identity comes from the Divergence theorem which in 2D is

$$\int_\Omega \nabla\cdot\mathbf{q}\,dA = \int_{\partial\Omega}\mathbf{q}\cdot\mathbf{n}\,ds.$$

If we let $\mathbf{q} = v\mathbf{u}$, then $\nabla\cdot\mathbf{q} = \nabla v\cdot\mathbf{u} + v\nabla\cdot\mathbf{u}$,

$$\implies \int_\Omega \nabla\cdot\mathbf{u}v\,dA + \int_\Omega \mathbf{u}\cdot\nabla v\,dA = \int_{\partial\Omega} v(\mathbf{u}.\mathbf{n})\,ds.$$

□

So back to our original equation, we get

$$\int_\Omega D(x,y)\nabla u\cdot\nabla v\,dA + \int_\Omega quv\,dA = \int_\Omega fv\,dA + \int_{\partial\Omega} D(\underbrace{\nabla u\cdot n}_{=\frac{\partial u}{\partial n}})v\,ds.$$

where $u\in H^1$, (i.e. $u, \frac{\partial u}{\partial x}, \frac{\partial u}{\partial y}\in L^2$).

Note that if we have a pure Neumann problem, with $\frac{\partial u}{\partial n} = g$ on $\partial\Omega$ and $q = 0$, then we get

$$\underbrace{\int -\nabla\cdot(D\nabla u)\,dA}_{-\int_{\partial\Omega} D\nabla u\cdot n\,ds} = \int f\,dA,$$

$$\implies \int_\Omega f\,dA + \int_{\partial\Omega} Dg\,ds = 0.$$

This is a compatibility condition which puts restrictions on f and $\frac{\partial u}{\partial n}$ if a solution is to exist.

2.2.2 Step 2: trial/test spaces

Now we need to pick appropriate test spaces (i.e. spaces for u and v). These depend on the boundary conditions. There are the following options:

- Homogeneous Dirichlet boundary conditions;
- Homogeneous Neumann boundary conditions;
- Inhomogeneous Dirichlet boundary conditions;
- Inhomogeneous Neumann boundary conditions; or
- Mixed boundary conditions.

We will look at homogeneous Neumann boundary conditions and inhomogeneous Dirichlet boundary conditions. The others are relatively trivial extensions.

Homogeneous Neumann boundary conditions

$$\frac{\partial u}{\partial n} = 0, \qquad \text{on } \partial\Omega.$$

Here both u and v are in H^1 and no other restrictions are needed. The weak formulation reduces to

$$\int_\Omega (D\nabla u \cdot \nabla u + quv)\, dA = \int_\Omega fv\, dA, \qquad u, v \in H^1.$$

Inhomogeneous Dirichlet boundary conditions

$$u = g, \qquad \text{on } \partial\Omega,$$

As in the 1D case here we choose $v \in H_0^1$ (zero boundary conditions on $\partial\Omega$), "don't test where you know u". In addition we let $u = G + \bar{u}$, where $\bar{u} \in H_0^1$ and G is a function defined on Ω which satisfies the boundary, conditions $u = g$ on $\partial\Omega$.

The weak formulation becomes

$$\int (D\nabla\bar{u}\cdot\nabla v + q\bar{u}v)\, dA = \int fv\, dA + \underbrace{\int_{\partial\Omega} D\frac{\partial u}{\partial n} v\, ds}_{=0 (v=0 \text{ on } \partial\Omega)} - \int_\Omega (D\nabla G\cdot\nabla v + qGv)\, dA.$$

2.2.3 Step 3: Galerkin equations

We can now select a nodal basis $\{\phi_i\}$ for H^1 or H_0^1, which gives us the Galerkin equations

$$\sum_j u_j \int D\nabla\phi_j \cdot \nabla\phi_i + q\phi_i\phi_j \, dA = \int f\phi_i \, dA + \text{boundary terms}, \qquad \forall i.$$

So we obtain the system of linear equations $Au = F$, where, like the 1D case, we have $A = K + M$.

2.2.4 Step 4: mesh generation

In order to specify the nodal basis we need to discretise the domain Ω, to a mesh Ω_h. There are many options:

- Triangulation;
- Rectangles (rectangular domain); or
- Others like:
  - Triangles plus rectangles; or
  - Quadrilaterals;
- (In 3D: Tetrahedra or bricks.)

By using triangles, any polygonal domain can be covered. (Look up “triangle mesh” online.)

To treat curved boundaries we can either:

- Approximate the domain by a polygon. This gives $O(h)$ error in the energy norm, which is acceptable for P_1 (i.e. linear elements) but not for P_2 (i.e. quadratic)).
- Use a different kind of element (isoparametric), which has $O(h^2)$ error in the energy norm.

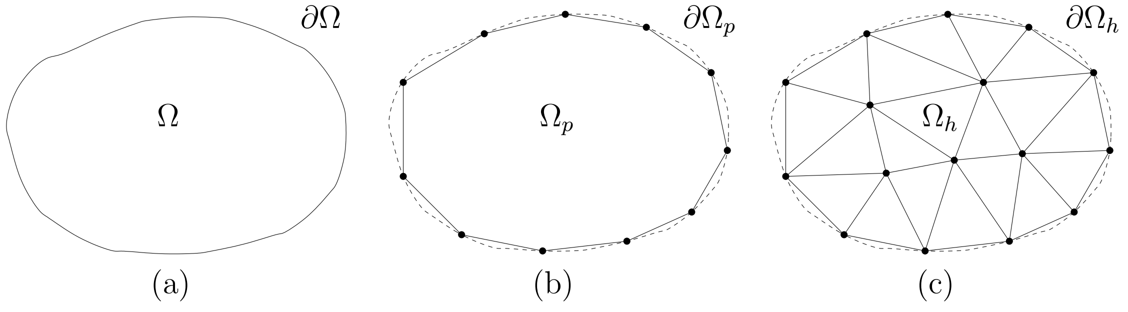

Figure 1: (a) Domain of solution, Ω, (b) approximation of Ω by a polygon, Ω_p, and (c) the mesh of the domain, Ω_h.

We will examine triangulation further. If we use triangulation then the 'elements' are now triangles, T. Usually the triangulation is conforming $\equiv$ the triangles must meet only on a single common edge or vertex.

A virtue of triangulation is that you can refine uniformly or locally as required.

The process of generating 'good meshes' involves making the triangles as close to equilateral as possible and is a field in itself called *grid or mesh generation* including ideas of Delaunay and constrained Delaunay triangulations. REF: Strang and Persson 2001.

Example 2.6. Have a look at the distmesh functions `meshdemo2d` and `meshdemond`.

Example 2.7. Have a look online at the tools `triangle` (search for "triangle mesh") and `tetgen`.

2.2.5 Step 5: finite element space

Now we need to choose a piecewise polynomial space which is a finite dimensional subspace of our test space.

Simplest case: Lagrange linear elements, $P_1 \subset C_0$

Define shape functions on a reference triangle, the canonical triangle.

Any linear function in 2D has the form

$$p_1(x, y) = c_1 + c_2 x + c_3 y.$$

Therefore we need 3 basis functions per element. So we need 3 DOF per element, and for a nodal basis we need 3 nodes per element.

If we put the nodes on the vertices (simplest case) then we obtain Lagrange linear triangles (which is what the PDE toolbox in MATLAB uses).

We need 3 shape functions on on the reference triangle which satisfy $N_j(v_i) = \delta_{ij}$.

$$\begin{aligned} N_1(\xi, \eta) =& 1 - \xi - \eta, \\ N_2(\xi, \eta) =& \xi, \\ N_3(\xi, \eta) =& \eta. \end{aligned}$$

We map these shape functions from the reference triangle to a triangle, $T = \{(x_1, y_1), (x_2, y_2), (x_3, y_3)\}$, by an affine transformation:

$$\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} x_2 - x_1 & x_3 - x_1 \\ y_2 - y_1 & y_3 - y_1 \end{bmatrix} \begin{bmatrix} \xi \\ \eta \end{bmatrix} + \begin{bmatrix} x_1 \\ y_1 \end{bmatrix},$$

$$\implies \mathbf{x} = F_T(\mathbf{s}) = J_T \mathbf{s} + \mathbf{b}_T.$$

We can use this to calculate the linear system, e.g. for the element load vectors,

$$f_T = \begin{bmatrix} \int_T f \phi_i \, dA \\ \int_T f \phi_j \, dA \\ \int_T f \phi_k \, dA \end{bmatrix},$$

we can calculate the entries as

$$\begin{aligned}\int_T f\phi_i \, dA &= \int_T f(\mathbf{x})\phi_j(\mathbf{x}) \, dxdy \\ &= \int_{T_R} f(\mathbf{x}(\mathbf{s}))N_j(\mathbf{s})|\det(J_T)| \, d\xi d\eta.\end{aligned}$$

Similarly, for the stiffness matrix,

$$K_T = \begin{bmatrix} \int_T D\nabla\phi_i \cdot \nabla\phi_i \, dA & \int_T D\nabla\phi_i \cdot \nabla\phi_j \, dA & \int_T D\nabla\phi_i \cdot \nabla\phi_k \, dA \\ \int_T D\nabla\phi_j \cdot \nabla\phi_i \, dA & \int_T D\nabla\phi_j \cdot \nabla\phi_j \, dA & \int_T D\nabla\phi_j \cdot \nabla\phi_k \, dA \\ \int_T D\nabla\phi_k \cdot \nabla\phi_i \, dA & \int_T D\nabla\phi_k \cdot \nabla\phi_j \, dA & \int_T D\nabla\phi_k \cdot \nabla\phi_k \, dA \end{bmatrix},$$

we can calculate the entries as

$$\begin{aligned}\int_T D\nabla\phi_i \cdot \nabla\phi_j \, dA &= \int_T D(\mathbf{x})\nabla\phi_i(\mathbf{x}) \cdot \nabla\phi_j(\mathbf{x}) \, dxdy \\ &= \int_{T_R} D(\mathbf{x}(\mathbf{s})) \left(J_T^{-T}\nabla_{\mathbf{s}} N_i(\mathbf{s})\right) \cdot \left(J_T^{-T}\nabla_{\mathbf{s}} N_j(\mathbf{s})\right) |\det(J_T)| \, d\xi d\eta.\end{aligned}$$

Example 2.8 (Quadrature rules)**.** Some quadrature rules on reference triangles T_R:

- $\int_{T_R} g(\xi,\eta)d\xi d\eta = \frac{1}{2}g(1/3,1/3)$,
- $\int_{T_R} g(\xi,\eta)d\xi d\eta = \frac{1}{6}\left[g(1/2,0) + g(1/2,1/2) + g(0,1/2)\right]$,

So we have the process:

- Choose a piecewise polynomial space;
- Compose each element of $K, M, \mathbf{f}$;
- Assemble into global matrices;
- Impose boundary conditions;
- Solve linear system; then
- Plot the solution.

2.2.6 Other basis functions

While you can get a long way with linear basis functions in some cases you may want to use (or need to in some cases, i.e. locking in incompressible solids) higher order functions, or even different shaped elements.

Triangles: Lagrange quadratic elements, $P_2 \subset C_0$

Any function in $P_2 \subset C_0$ has the form:

$$p_2(x, y) = c_1 + c_2 x + c_3 y + c_4 x^2 + c_5 xy + c_6 y^2.$$

We have six degrees of freedom per element and therefore we need six nodes per element and the basis functions are extensions of in the 1D case.

Triangles: Lagrange cubic elements, $P_3 \subset C_0$

If we want cubic polynomials i.e. P_3, then we have 10 degrees of freedom, and need 10 nodes per element

Rectangles: Q_1**: bilinear element** If we have rectangular elements instead of triangles then our simplest function is

$$q_1(x, y) = (c_1 + c_2 x)(c_3 + c_4 y).$$

Which is linear in both x and y but not linear (it has an xy term which is not in P_1).

This has four degrees of freedom, so we need four nodes per element. On the canonical square (reference element) they are

$$\begin{aligned} N_1 =&(1 - \xi)(1 - \eta), \\ N_2 =&(1 - \xi)\eta, \\ N_3 =&\xi(1 - \eta), \\ N_4 =&\xi\eta. \end{aligned}$$

Example 2.9. Run the code `Shape_rect.m` to see examples of the basis functions.

Rectangles: Q_2: biquadratic element

Functions in Q_2 take the form

$$\begin{aligned} q_2(x,y) &= (c_1 + c_2 x + c_3 x^2)(c_4 + c_5 y + c_6 y^2) \\ &= c_1c_4 + c_2c_4x + c_1c_5y + c_3c_4x^2 + c_1c_6y^2 + c_2c_5xy + c_2c_6xy^2 + c_3c_5x^2y + c_3c_6x^2y^2, \end{aligned}$$

which has 9 degrees of freedom, which gives 4 corner shape functions, 4 edge shape functions and 1 bubble function.

2.2.7 Practice time

Example 2.10. Solving problems with `pdetool`

- using the GUI
- from the command line → writing scripts `Pde2p1.m`

Example 2.11. Now you can understand the code in the built in MATLAB demos (look at them and use `publish` to see them fully.

- `pdedemo1`: solves a Poisson equation on a disc.
- `pdedemo2`: solves a Helmholtz equation (spatial part of the solution of a wave equation).
- `pdedemo3`: solves a nonlinear elliptic equation (soap bubble equation) using `pdenonlin`, which uses damped Newton iterations.
- `pdedemo8`: solves a Poisson equation on a square and times `assempde` against the fast Poisson solver `poisolv`

Example 2.12. Try the codes in `fem50`: to run these double click the folder to put it at the top of your path and run `clear` and `close all` before each command. These are all based on the paper 'Remarks around 50 lines of Matlab: Short Finite Element Implementation'.

- `fem2d`: solves a 2d Poisson equation on a polygonal domain. Uses $P_1 Q_1$ elements.
- `fem2d_heat`: solves a 2d heat equation using implicit Euler time-stepping (see §3.2). At each time step you need to solve a Poisson equation.
- `fem2d_nonlin`: solves a nonlinear elliptic equation (Ginzburg-Landau) by using a Newton Raphson method.
- `fem3d`: solves a Poisson equation in a 3D piston. Uses tetrahedral elements.

Example 2.13. These ideas were later extended in 'P2Q2Iso2D = 2D Isoparametric FEM in MATLAB' to use P_2Q_2 isoparametric elements i.e. can approximate curved boundaries better. The codes are in `P2Q2`. The driver `P2Q2Iso2D` is set up to solve test problem 7. I could get problems 2,3,4 and 7 to run.

Example 2.14. Starting in the 1980s, the use of local error estimators in FEM codes allowed the development of adaptive refinement i.e. only locally refining the mesh where the local error is largest. This greatly improves the efficiency of FEM codes. Try the codes in `AFEM`.

Add the folder (and subfolders) `AFEM@matlab` to the path and run `demo`. Alternatively add the folder and subfolders to the path and look at the following codes. A Poisson problem in an L-shaped domain is solved in 3 codes:

- `simple` which refines the mesh uniformly
- `Lshape` which adaptively refines the mesh
- `Lbig` which generates the finest mesh of all - 117,000 triangles!!

Example 2.15. Many FEM solvers exist, commercial or open source.

An example I have had installed is called FreeFem++, from France.

Open an example code from Examples -> Chapter 3 -> membrane.edp

You can see how the problem is specified though a scripting language where you have to specify the weak form of the PDE, the FE space etc.

Other examples of FEM tools are `DealII`, `LibMesh` (both c++) and `FEniCS` (python). To find them just search for the names online.

2.3 Sparse solvers

For finite difference and finite element methods (but not spectral solvers) we get large sparse linear systems.

These can't be solved efficiently using standard methods for dense systems e.g. LU factorisation or Cholesky factorisation.

Example 2.16. For a finite difference grid, there are

- $m \times m$ unknowns, so $N = m^2$ in 2D
- $m \times m \times m$ unkowns, so $N = m^3$ in 3D

A dense Cholesky solver uses $\sim \frac{1}{6}N^3$ operations, so we need

- $O(m^6)$ operations in 2D
- $O(m^9)$ operations in 3D

But if we count the number of unzero elements in A we have

- $5m^2$ elements in 2D
- $7m^3$ elements in 3D

which suggests there is a lot of room to improve (our matrices are sparse!).

We have two choices:

- Sparse direct solvers, which are competitive in 2D
- Iterative solvers which are ultimately more efficient as we move up in dimensions

2.3.1 Direct Sparse methods

With direct sparse solvers we get exact answers (up to round-off) in principle, but only access the non-zero entries. Matlab uses these when using \ for sparse systems.

- if A is sparse and banded, \ invokes a sparse banded solver,
- if A is sparse symmetric, \ invokes sparse Cholesky factorisation.

The problem with using a traditional Cholesky directly is that row operations 'destroy zeros', i.e. the matrix loses sparsity during factorisation (this is called *fill in*). The aim is to try and use re-ordering of the equations to minimise fill in.

Example 2.17. Run the demo `sparsity` to see the effect of different orderings on the amount of fill-in using sparse Cholesky. On current desktop machines, the speedup from reordering is not apparent on the example matrix. See `MySparsity` for the same example with a different larger matrix, what is the best reordering now?

After much research into different orderings, Matlab uses `ammd` by default, but you can choose a different ordering if you want. For details, see Davis, SIAM, 2007.

Sparse Cholesky uses a re-ordering automatically and only handles non-zero elements, resulting in

- $O(m^3)$ operations in 2D (c.f. $O(m^6)$) and
- $O(m^6)$ operations in 3D (c.f. $O(m^9)$)

PDE toolbox in Matlab in fact uses \ to solve finite element equations. FreeFem++ uses iterative methods.

From personal experience we've found \ fastest in Matlab for 2D problems, because \ is compiled while iterative solvers are interpreted. In 3D, iterative solvers start to do better. In compiled languages (C++), iterative approaches are usually better off the bat.

2.3.2 Iterative solvers

There are three main classes of iterative solvers:

- Classical stationary (1950s)
- Krylov space methods: (1970s-1990s)
- Multigrid methods (1980s-)

There are two main reasons to look at iterative solvers for sparse systems:

- More Efficiency

  Each iteration typically involves a matrix-vector product Av, for finite difference with a 5 point stencil, A has 5 non-zero entries per row and A is $m^2 \times m^2$ so to form Av takes $5m^2$ operations, for finite element methods A has 9 non-zero entries per row so to form Av takes $9m^2$ operations.

  So in 2D as long as our solver converges in $O(m)$ iterations, we can achieve $O(m^3)$ complexity running time which matches (asymptotically) sparse Cholesky.

  In 3D we only have Lm^3 operations per iteration so we require $O(m^3)$ iterations for convergence to do equal to or better than sparse Cholesky.

- The matrix A is not exact.

  We already have a discretisation error $O(h^2)$ (local truncation error), so the residual is $O(h^2)$. Why use a solver whose relative residuals are $O(u)$ (unit round-off) when the backward error in A is $O(h^2)$? Instead, aim to solve $Au = F$ approximately till residual is $O(h^2)$ – this is not possible with direct methods.

We will look only at Krylov subspace methods. (Others covered in Student talks).

2.3.3 Krylov subspace methods

For A symmetric positive definite, e.g. FD/FEM discretisations of elliptic PDEs, the best iterative method is called *preconditioned conjugate gradient (PCG)*. This is one member of a large class of methods developed over 1971-1995 that have become current standard practice.

Refs: Trefethen & Bau, Elman et. al Chapter 2, Johnson Chapter 7, Braess Chapter 4, Gockenbach Chapter 11, Kelley, Iserles, Schewchuk.

We will start with gradient methods, move to conjugate gradient methods and then move onto preconditioned conjugate gradient methods.

Conjugate gradient methods started out as an optimisation algorithm (1952 Ref: Hestenes & Stiefel) and were later re-used as an iterative solver (1971).

1st Observation: find the solution of $A\mathbf{x} = \mathbf{b} \equiv$ find the minimum of $\phi(x)$, where $\phi(x) = \frac{1}{2}\mathbf{x}^T A\mathbf{x} - \mathbf{b}^T\mathbf{x}$.

If A is symmetric positive definite, $\phi(x)$ has a global minimum at $\mathbf{x}^* = A^{-1}\mathbf{b}$

Recall: A is positive definite if $\mathbf{x}^T A\mathbf{x} \geq 0$ and the equality holds if and only if $\mathbf{x} = \mathbf{0}$.

Aim: find this minimum as fast as possible (in as few iterations as possible).

2.3.4 Gradient method (steepest descent)

Tip from Newton's method: use $\nabla\phi$ to tell us a descent direction $\mathbf{p}$.

Since the directional derivative of ϕ in the direction $\mathbf{d}$ (a unit vector) is $\nabla\phi\cdot\mathbf{d}$, this is minimised if $\mathbf{d}$ is in the direction $-\nabla\phi$ which implies that $-\nabla\phi$ is the direction of steepest descent.

At current guess $\mathbf{x}_k$:

- Compute descent direction, $\mathbf{d}_k = -\nabla\phi\big|_{\mathbf{x}_k}$.
- Step along $\mathbf{d}_k$ in a line search a distance α_k, chosen to give the minimum possible value of ϕ.

$$\min_{\alpha\geq 0}\phi(\mathbf{x}_k+\alpha\mathbf{d}_k).$$

So

$$\begin{aligned}&\frac{d}{d\alpha}\phi(\mathbf{x}_k+\alpha\mathbf{d}_k)=0\\ \implies&\nabla\phi\big|_{\mathbf{x}_k+\alpha\mathbf{d}_k}\cdot\mathbf{d}_k=0\\ \implies&\nabla\phi\big|_{\mathbf{x}_{k+1}}\cdot\mathbf{d}_k=0 \implies \quad \text{so } \mathbf{d}_{k+1}\perp\mathbf{d}_k.\end{aligned}$$

This means we will follow a zig-zag path.

Since $\nabla\phi = Ax-b$, $\mathbf{d}_k = -\nabla\phi = b-A\mathbf{x}_k = \mathbf{r}_k$, the orthogonality of $\mathbf{d}_k$ and $\mathbf{d}_{k+1}$

$$\mathbf{d}_k\cdot\underbrace{[b-A(\mathbf{x}_k+\alpha_k\mathbf{d}_k)]}_{\mathbf{d}_{k+1}}=0,$$

give an equation for α_k:

$$\alpha_k=\frac{\mathbf{d}_k\cdot(\mathbf{b}-A\mathbf{x}_k)}{\mathbf{d}_k\cdot A\mathbf{d}_k}=\frac{\mathbf{r}_k^T\mathbf{r}_k}{\mathbf{r}_k^TA\mathbf{r}_k},$$

in terms of the *residual* $\mathbf{r}_k=\mathbf{b}-A\mathbf{x}_k$. Iterating these steps gives the **steepest descent algorithm**.

Steepest descent algorithm:

```
Choose x_0
repeat
     r_k = b - A x_k
     stopping criterion here
     alpha_k = r_k^T r_k/( r_k^T A r_k)
     x_k+1 = x_k + alpha_k r_k
end
```

which can be reorganised (Ref: Leveque p. 80) to a version which only uses 1 matrix-vector product per iteration:

```
Choose x_0
r_0 = b-A x_0
repeat k = 1,2, ...
     w_{k-1} = A r_{k-1}
     alpha_{k-1} = r_{k-1}^T r_{k-1}/(r_{k-1}^T w_{k-1})
     x_k = x_{k-1} + alpha_{k-1} r_{k-1}
     r_k = r_{k-1} - alpha_{k-1} w_{k-1}
     stopping criterion here
end
```

the last step using

$$\begin{aligned}\mathbf{r}_k =&\mathbf{b} - A\mathbf{x}_k\\ =&\mathbf{b} - A(\mathbf{x}_{k-1} + \alpha_{k-1}\mathbf{r}_{k-1})\\ =&\mathbf{r}_{k-1} - \alpha_{k-1}\mathbf{w}_{k-1}\end{aligned}$$

This always finds the minimum, but it may be very slow.

Example 2.18. If $A = I$ it in fact finds the minimum in one step, but if

$$A = \begin{pmatrix}\lambda_1 & 0\\ 0 & \lambda_2\end{pmatrix},$$

and $\lambda_1 \gg \lambda_2$, then we get a 'zig-zagging' pattern and it takes a long time to converge.

Theorem 2.19. *If A is symmetric positive definite with condition number*

$$\kappa_2(A) = \frac{\lambda_{\max}(A)}{\lambda_{\min}(A)},$$

then the steepest descent algorithm converges according to

$$\|\mathbf{e}_k\|_A \leq \left(\frac{\kappa-1}{\kappa+1}\right)^k \|\mathbf{e}_0\|_A,$$

where we have used a new norm for the error

$$\|\mathbf{x}\|_A = (\mathbf{x}^T A\mathbf{x})^{\frac{1}{2}},$$

which has the properties of a norm only if A is spd.

Example 2.20. for $A = I$, $\|\mathbf{x}\|_A = \|\mathbf{x}\|_2$.

Example 2.21.

$$\text{If } A = \begin{pmatrix} \lambda_1 & 0 \\ 0 & \lambda_2 \end{pmatrix} \text{ then } \|\mathbf{x}\|_A = \sqrt{x_1^2\lambda_1 + x_2^2\lambda_2}$$

The theorem above implies that if $\kappa_2(A) \gg 1$, the convergence is slow since $\frac{\kappa-1}{\kappa+1} \to 1$. See Schewchuk, Fig.20.

Example 2.22. if $\lambda_1 \gg \lambda_2$, $\kappa_2 = \frac{\lambda_1}{\lambda_2} \gg 1$.

The method of steepest descent slows down for ill-condition matrices.

This is relevant to us:

- For a finite difference 5-point stencil, the matrix A that we obtain has $\kappa_2(A) \sim h^{-2}$
- For FEM + nodal basis ($P_m, Q_m, m \geq 1$) gives matrices with $\kappa_2(A) \leq Ch^{-2}$
- For a uniform mesh on a square we have $\kappa_2(A) = \frac{2}{\pi^2 h^2} + O(1)$

So as we refine our mesh to get smaller error, $\kappa_2(A)$ grows and steepest descent slows down. There are other FEM meshes (hierarchical meshes) that have smaller condition numbers.

We can estimate the number of iterations required to reduce the error below a specified tolerance ϵ.

$$\text{Want } \left(\frac{\kappa-1}{\kappa+1}\right)^n < \epsilon.$$

Where

$$\frac{\kappa-1}{\kappa+1} = 1 - \frac{2}{\kappa+1} \sim 1 - \frac{2}{\kappa}, \qquad \text{for large } \kappa.$$

So we have

$$\begin{aligned} & n \log\left(1 - \frac{2}{\kappa}\right) < \log(\epsilon), \\ \implies & \frac{-2n}{\kappa} < \log(\epsilon), && \text{by Taylor approximation (as } \tfrac{2}{\kappa} \text{ is small)} \\ \implies & n > \log\left(\frac{1}{\epsilon}\right)\frac{\kappa}{2}, \\ & \text{i.e. } n \propto \kappa_2(A). \end{aligned}$$

In 2D, $\kappa_2(A) \sim h^{-2} \sim m^2$ for an $m \times m$ grid, so

$$\begin{aligned} \text{Total work} \sim & \ \#\text{iterations} \times \ \text{ops/iteration} \\ =& m^2 \times 5m^2 \\ =& O(m^4). \end{aligned}$$

which is worse asymptotically than sparse Cholesky factorisation.

Aside. In 3D, $\kappa_2(A) \sim h^{-2}$ as well, so we use $O(m^5)$ operations to solve our system – this is already better than sparse Cholesky. But, for the 4th order biharmonic equation arising in elasticity, $\nabla^2\nabla^2 u = f$, $\kappa_2(A) \sim h^{-4}$ i.e. slowdown is worse.

BUT: we can do better than this with the Conjugate Gradient method.

2.3.5 Conjugate Gradient method

Idea: Choose different directions $\mathbf{p}_k$, constructed from steepest descent directions $\mathbf{r}_k$, which have very nice properties.

In particular, once we have searched along a particular direction, we never want to search in that direction again. We ensure this by:

$\mathbf{x}_1$ minimises ϕ when searching over $\{\mathbf{p}_1\}$ [as before]
$\mathbf{x}_2$ minimises ϕ when searching over span$\{\mathbf{p}_1, \mathbf{p}_2\}$ [not true of steepest descent]
$\mathbf{x}_3$ minimises ϕ when searching over span$\{\mathbf{p}_1, \mathbf{p}_2, \mathbf{p}_3\}$

$$\vdots$$

By making x_k minimise ϕ over nested subspaces, we never 'backtrack' or search along previous directions, so

$$\phi(\mathbf{x}_{k+1}) \leq \phi(\mathbf{x}_k),$$

i.e. ϕ decreases monotonically.

The hard part: making this cheap.

We want to construct $\{\mathbf{p}_j\}$ so that

$$\alpha_k = \min_{\alpha} \phi(\mathbf{x}_{k-1} + \alpha \mathbf{p}_k)$$

also minimises ϕ over $x \in$ span$\{\mathbf{p}_1, \ldots, \mathbf{p}_k\}$.

Let's try generating $\mathbf{p}_1$:

1. Guess $\mathbf{x}_0, \mathbf{p}_0$ [often $\mathbf{x}_0 = \mathbf{0}$, $\mathbf{p}_0 = \mathbf{r}_0 = \mathbf{b}$]
2. Find α_0 by a 1D minimisation of $\phi(\mathbf{x}_0 + \alpha \mathbf{p}_0)$

$$\begin{aligned}\mathbf{p}_1 \phi(\mathbf{x}_0 + \alpha \mathbf{p}_0) =& \frac{1}{2}(\mathbf{x}_0 + \alpha \mathbf{p}_0)^T A (\mathbf{x}_0 + \alpha \mathbf{p}_0) - (\mathbf{x}_0 + \alpha \mathbf{p}_0)^T \mathbf{b} \\ =& \frac{1}{2}\mathbf{x}_0^T A \mathbf{x}_0 - \mathbf{x}_0^T \mathbf{b} + \frac{1}{2}\alpha^2 \mathbf{p}_0^T A \mathbf{p}_0 + \alpha \mathbf{p}_0^T A \mathbf{x}_0 - \alpha \mathbf{p}_0^T \mathbf{b},\end{aligned}$$

$$\frac{d}{d\alpha}(\phi(\mathbf{x}_0 + \alpha \mathbf{p}_0) = \mathbf{0} \implies \alpha \mathbf{p}_0^T A \mathbf{p}_0 = \mathbf{p}_0^T (\mathbf{b} - A\mathbf{x}) = \mathbf{p}_0^T \mathbf{p}_0,$$

so that we get

$$\begin{aligned}\alpha_0 =& \frac{\mathbf{p}_0^T\mathbf{p}_0}{\mathbf{p}_0^T A\mathbf{p}_0},\\ \mathbf{x}_1 =& \mathbf{x}_0 + \alpha_0\mathbf{p}_0,\\ \mathbf{r}_1 =& \mathbf{r}_0 - \alpha_0 \underbrace{A\mathbf{p}_0}_{\mathbf{w}_0},\end{aligned}$$

as for the first step of steepest descent

3. Find $\mathbf{x}_2$ by 2D minimisation of

$$\begin{aligned}\phi(\mathbf{x}_0 + \alpha\mathbf{p}_0 + \beta\mathbf{p}_1) =& \phi(\mathbf{x}_1 + \beta\mathbf{p}_1)\\ =& \frac{1}{2}(\mathbf{x}_1 + \beta\mathbf{p}_1)^T A(\mathbf{x}_1 + \beta\mathbf{p}_1) - (\mathbf{x}_1 + \beta\mathbf{p}_1)^T\mathbf{b}\\ =& \underbrace{\frac{1}{2}\mathbf{x}_1^T A\mathbf{x}_1 - \mathbf{x}_1^T\mathbf{b}}_{\phi(\mathbf{x}_1)} + \frac{1}{2}\beta^2\mathbf{p}_1^T A\mathbf{p}_1 + \underbrace{\beta\mathbf{p}_1^T A\mathbf{x}_1 - \beta\mathbf{p}_1^T\mathbf{b}}_{-\beta\mathbf{p}_1^T\mathbf{r}_1}\\ =& \phi(\mathbf{x}_1) + \frac{1}{2}\beta^2\mathbf{p}_1^T A\mathbf{p}_1 - \beta\mathbf{p}_1^T(\mathbf{r}_0 - \alpha A\mathbf{p}_0),\end{aligned}$$

where we note that $\beta\alpha\mathbf{p}_1^T A\mathbf{p}_0$ is the only term that includes β and α. We can de-couple the 2D minimisation of ϕ into two 1D minimizations if we set this term to zero by choosing

$$\mathbf{p}_1^T A\mathbf{p}_0 = 0.$$

With this choice of search directions, we turn k-dimensional minimisation over nested subspaces into successive 1D minimisation problems, with no backtracking.

Conjugate directions

Two vectors that satisfy $\mathbf{u}^{\mathrm{T}}A\mathbf{v} = 0$ are called *A-conjugate*.

If A is symmetric we can define an inner product $(\mathbf{u}, \mathbf{v})_A \equiv \mathbf{u}^T A\mathbf{v}$. If A is also positive definite, this induces a norm $\|\mathbf{u}\|_A^2 \equiv <\mathbf{u}, \mathbf{u}>_A = \mathbf{u}^T A\mathbf{u}$. So A-conjugate vectors are orthogonal wrt the A-inner product.

We aim to produce a set of mutually A-conjugate vectors. To achieve this we do conjugate Gram-Schmidt to form $\{\mathbf{p}_0, \mathbf{p}_1, \ldots\}$, from $\{\mathbf{r}_0, \mathbf{r}_1, \ldots\}$.

1. Guess $\mathbf{p}_0 = \mathbf{r}_0$.
2. Let

$$\begin{aligned}\alpha_0 =& \frac{\mathbf{r}_0^T\mathbf{r}_0}{\mathbf{p}_0^T A\mathbf{p}_0},\\ \mathbf{x}_1 =& \mathbf{x}_0 + \alpha_0\mathbf{p}_0,\\ \mathbf{r}_1 =& \mathbf{r}_0 - \alpha_0 A\mathbf{p}_0,\end{aligned}$$

as before.

3. Now

$$\begin{aligned}\mathbf{p}_1 =& \mathbf{r}_1 - \text{component of } \mathbf{r}_1 \text{ not } A\text{-conjugate to } \mathbf{p}_0\\ =& \mathbf{r}_1 + \beta_0\mathbf{p}_0 \qquad (\text{since span } \{\mathbf{r}_0, \mathbf{r}_1\} = \text{span } \{\mathbf{p}_0, \mathbf{p}_1\})\end{aligned}$$

Determine β_0 by enforcing A-conjugacy

$$\mathbf{p}_1^T A\mathbf{p}_0 = 0 \implies (\mathbf{r}_1 + \beta_0\mathbf{p}_0)^T A\mathbf{p}_0 = 0,$$
$$\implies \beta_0 = \frac{-\mathbf{r}_1^T A\mathbf{p}_0}{\mathbf{p}_0^T A\mathbf{p}_0}.$$

Then since

$$\mathbf{r}_1 = \mathbf{r}_0 - \alpha_0 A\mathbf{p}_0, \qquad -A\mathbf{p}_0 = \frac{\mathbf{r}_1 - \mathbf{r}_0}{\alpha_0},$$

$$\begin{aligned}\beta_0 =& \frac{\mathbf{r}_1^T(\mathbf{r}_1 - \mathbf{r}_0)/\alpha_0}{\mathbf{p}_0^T A\mathbf{p}_0}\\ =& \frac{\mathbf{r}_1^T\mathbf{r}_1}{\alpha_0\mathbf{p}_0^T A\mathbf{p}_0}\\ =& \frac{\mathbf{r}_1^T\mathbf{r}_1}{\mathbf{r}_0^T\mathbf{r}_0}, \qquad (= \text{reduction in the square residual.})\end{aligned}$$

4. So given $\mathbf{p}_1$, $\mathbf{x}_1$, $\mathbf{r}_1$, we calculate $\mathbf{p}_2$:

$$\alpha_1 = \frac{\mathbf{r}_1^T\mathbf{r}_1}{\mathbf{p}_1^T A\mathbf{p}_1}, \qquad x_2 = \mathbf{x}_1 + \alpha_1\mathbf{p}_1, \qquad \mathbf{r}_2 = \mathbf{r}_1 - \alpha_1 A\mathbf{p}_1,$$
$$\beta_1 = \frac{\mathbf{r}_2^T\mathbf{r}_2}{\mathbf{r}_1^T\mathbf{r}_1}, \qquad \mathbf{p}_2 = \mathbf{r}_2 + \beta_1\mathbf{p}_1.$$

This all leaves $\mathbf{r}_2 \perp \mathbf{r}_0, \mathbf{r}_1$; $\mathbf{p}_2$ conjugate to $\mathbf{p}_0, \mathbf{p}_1$; and $\mathbf{r}_2 \perp \mathbf{p}_0, \mathbf{p}_1$ (but not $\mathbf{p}_2$.

After k iterations,

$$\mathbf{x}_k = \text{solution that minimizes } \phi \text{ over } x \in \mathbf{x}_0 + \text{span}\{\mathbf{p}_0, \mathbf{p}_1, \dots, \mathbf{p}_{k-1}\}$$

Minimising ϕ is same as minimising the error in the A-norm:
$\|\mathbf{e}\|_A = \|\mathbf{x} - \mathbf{x}^*\|_A$.

Proof.

$$\begin{aligned}\|\mathbf{e}\|_A^2 =& \|\mathbf{x} - \mathbf{x}^*\|_A^2 = (\mathbf{x} - \mathbf{x}^*)^T A(\mathbf{x} - \mathbf{x}^*) \\ =& \mathbf{x}^T A\mathbf{x} - 2\mathbf{x}^T A\mathbf{x}^* + \mathbf{x}^{*T} A\mathbf{x}^* \\ =& \underbrace{\mathbf{x}^T A\mathbf{x} - 2\mathbf{x}^T\mathbf{b}}_{2\phi(\mathbf{x})} + \underbrace{\mathbf{x}^{*T} A\mathbf{x}^*}_{\text{constant}}\end{aligned}$$

□

$\mathbf{x}_k$ minimises $\|\mathbf{e}\|_A$ over $\mathbf{x}_0 + \text{span}\{\mathbf{p}_0, \dots, \mathbf{p}_{k-1}\}$. But

$$\mathbf{r}_0 = \mathbf{p}_0, \mathbf{r}_1 = \mathbf{r}_0 - \alpha_0 A\mathbf{p}_0, \mathbf{r}_2 = \mathbf{r}_1 - \alpha_1 A\mathbf{p}_1 \dots$$

so

$$\text{span}\{\mathbf{p}_0, \mathbf{p}_1, \dots, \mathbf{p}_{k-1}\} = \underbrace{\text{span}\{\mathbf{r}_0, A\mathbf{r}_0, A^2, \dots, A^{k-1}\mathbf{r}_0\}}_{\text{Krylov Space}}.$$

Since $A\mathbf{x} = \mathbf{b} - \mathbf{r}$ and $A\mathbf{x}^* = \mathbf{b}$, $A\mathbf{e} = -\mathbf{r}$, so

$$\begin{aligned}\text{span}\{\mathbf{p}_0, \mathbf{p}_1, \dots, \mathbf{p}_{k-1}\} =& \text{span}\{\mathbf{r}_0, A\mathbf{r}_0, A^2, \dots, A^{k-1}\mathbf{r}_0\} \\ =& \text{span}\{A\mathbf{e}_0, A^2\mathbf{e}_0, \dots, A^k\mathbf{e}_0\}\end{aligned}$$

so

$$\mathbf{x}_k = \mathbf{x}_0 + \text{Linear combinations}\{A\mathbf{e}_0, \dots, A^k\mathbf{e}_0\}$$

If we subtract $\mathbf{x}^*$ from both sides then

$$\mathbf{e}_k = \mathbf{e}_0 + \text{Linear combinations}\{A\mathbf{e}_0, \dots, A^k\mathbf{e}_0\} = P_k(A)\mathbf{e}_0$$

where P_k is a polynomial of degree k with $p(0) = 1$ (*monic polynomial*).

Since conjugate-gradient minimises $\phi(x)$ and $\|\mathbf{e}\|_A$ over $\mathcal{K}_k$,

$$\|\mathbf{e}_k\|_A = \min_{p \in P_k} \|p(A)\mathbf{e}_0\|_A.$$

Also we have $\|\mathbf{e}_{k+1}\|_A \le \|\mathbf{e}_k\|_A$ since $\mathcal{K}_{k+1} \supset \mathcal{K}_k$ (or since $\phi(x_{k+1}) \le \phi(x_k)$). After N steps, (where A is $N \times N$, ($N = m^2$ in 2D)), we have $\mathbf{e}_N = \mathbf{0}$ (up to round-off), since $\mathcal{K}_N = \mathbb{R}^n$ or since A satisfies its characteristic polynomial $p_N(A) = 0$.

Conjugate gradient converges to the exact answer in N iterations (ignoring roundoff). Conjugate gradient is guaranteed to converge in N (which is m^2 or m^3) iterations, but its use depends on fast convergence in $k \ll N$ iterations.

Another way to characterise the convergence of CG:

$$\frac{\|\mathbf{e}_k\|_A}{\|\mathbf{e}_0\|_A} = \inf_{p \in P_k} \frac{\|p(A)\mathbf{e}_0\|_A}{\|\mathbf{e}_0\|_A}.$$

Expand $\mathbf{e}_0$ in eigenvectors of A: $\mathbf{e}_0 = \sum a_j \mathbf{v}_j$ (orthonormal) then $p(A)\mathbf{e}_0 = \sum a_j p(\lambda_j)\mathbf{v}_j$.

$$\begin{aligned}\implies \|\mathbf{e}_0\|_A^2 = \mathbf{e}_0^T A \mathbf{e}_0 &= \sum_k a_k \mathbf{v}_k^T \sum_j a_j \lambda_j \mathbf{v}_j \\ &= \sum_k a_k \delta_{kj} \sum a_j \lambda_j \\ &= \sum_j a_j^2 \lambda_j\end{aligned}$$

Similarly

$$\|p(A)\mathbf{e}_0\|_A^2 = \sum a_j^2 p(\lambda_j)^2 \lambda_j$$

So

$$\frac{\|p(A)\mathbf{e}_0\|_A}{\|\mathbf{e}_0\|_A} = \frac{\sum a_j^2 p(\lambda_j)^2 \lambda_j}{\sum a_j^2 \lambda_j} \le \max_{\lambda \in \Lambda(A)} p(\lambda)^2$$

$$\implies \frac{\|\mathbf{e}_k\|_A}{\|\mathbf{e}_0\|_A} \le \inf_{p \in P_k} \max_{\lambda \in \Lambda(A)} p(\lambda)^2$$

The convergence of CG is determined by how small a polynomial with $p(0) = 1$ can be at the eigenvalues of A, i.e. it is determined by the spectrum of A.

Example 2.23. If A has only $k < N$ different eigenvalues then conjugate gradient converges in k iterations.

Proof. The polynomial of degree k:

$$\prod_{i=1}^{k} \left(1 - \frac{x}{\lambda_i}\right),$$

has $p(0) = 1$, but $p(\lambda_i) = 0 \implies \mathbf{e}_k = \mathbf{0}$. □

Conjugate gradient converges fast if the eigenvalues are grouped in a few clusters:

Example 2.24. If we don't know the spectrum of A but only know $\lambda \in (\lambda_{\min}, \lambda_{\max})$, then the best result is:

$$\frac{\|\mathbf{e}_k\|_A}{\|\mathbf{e}_0\|} \leq 2\left(\frac{\sqrt{\kappa}-1}{\sqrt{\kappa}+1}\right)^k, \tag{2.1}$$

where $\kappa = \kappa_2(A) = \frac{\lambda_{\max}}{\lambda_{\min}}$.

So if κ is near 1, i.e. $\lambda_{\max} - \lambda_{\min} \ll \lambda_{\max} + \lambda_{\min}$, then convergence is rapid.

But this bound can be pessimistic, so if $\kappa \gg 1$, convergence *could* be slow but the previous result, $\inf \max |p(\lambda)|$, is a tighter bound. i.e. the clustering of eigenvalues determines convergence, not just their range.

Using the bound in Equation 2.1, we can estimate a (pessimistic) bound for the number of iterations to achieve a tolerance ϵ:

$$n > \frac{\sqrt{\kappa}}{2}\log\left(\frac{2}{\epsilon}\right).$$

So we need $\sim \sqrt{\kappa}$ iterations for CG to achieve convergence.

Example 2.25. For a 5-point FD stencil or P_1 or Q_1 FEM, $\kappa_2 \sim h^{-2} \sim m^2$. So if we require $\sim \sqrt{\kappa_2}$ iterations then the total work is $O(m^3)$ to find the solution – the same complexity as \ in 2D.

In 3D we have $\sim m$ iterations at $O(m^3)$ operations per iteration, so $O(m^4)$, compared to $O(m^6)$ for \.

CG is the 1st example we have seen where an iterative process depends on the spectrum of the matrix.

Stopping Criterion:

We want to know when to stop; we can't stop using $\|\mathbf{e}_k\|_A$ since we can't compute it. But $A\mathbf{e}_k = -\mathbf{r}_k$, $A\mathbf{e}_0 = -\mathbf{r}_0$, so $\mathbf{e}_k = -A^{-1}\mathbf{r}_k$, $\mathbf{e}_0 = -A^{-1}\mathbf{r}_0$. So we have

$$\begin{aligned}\|\mathbf{e}_k\|_A^2 =&\mathbf{e}_k^T A\mathbf{e}_k = \mathbf{r}_k^T A^{-T} A A^{-1}\mathbf{r}_k = \mathbf{r}_k^T A^{-1}\mathbf{r}_k, \qquad (\text{ since } A \text{ symmetric})\\ \|\mathbf{e}_0\|_0^2 =&\mathbf{r}_0^T A^{-1}\mathbf{r}_0,\end{aligned}$$

where $A^{-T}(= A^{-1})^T$.

$$\begin{aligned}\mathbf{r}_k^T A^{-1}\mathbf{r}_k \leq&\lambda_{\max}(A^{-1})\mathbf{r}_k^T\mathbf{r}_k,\\ \mathbf{r}_0^T A^{-1}\mathbf{r}_0 \geq&\lambda_{\min}(A^{-1})\mathbf{r}_0^T\mathbf{r}_0.\end{aligned}$$

So

$$\begin{aligned}\frac{\|\mathbf{e}_k\|_A^2}{\|\mathbf{e}_0\|_A^2} &\leq \underbrace{\frac{\lambda_{\max}(A^{-1})}{\lambda_{\min}(A^{-1})}}_{=\frac{\lambda_{\max}(A)}{\lambda_{\min}(A)}=\kappa_2(A)} \frac{\|\mathbf{r}_k\|^2}{\|\mathbf{r}_0\|^2}\\ \implies \frac{\|\mathbf{e}_k\|_A}{\|\mathbf{e}_0\|_A} &\leq \sqrt{\kappa_2(A)}\frac{\|\mathbf{r}_k\|}{\|\mathbf{r}_0\|}\end{aligned}$$

so we can use $\|\mathbf{r}_k\|$ or $\frac{\|\mathbf{r}_k\|}{\|\mathbf{r}_0\|}$ as a stopping criterion as a proxy for $\|\mathbf{e}_k\|_A$.

Since the local truncation error of discretisation (i.e. the residual $A\mathbf{x} - \mathbf{b}$) is $O(h^2) = O(m^{-2})$ anyway (for P_1/Q_1 etc.), often we use $\|\mathbf{r}_k\| \leq ch^2$ as a stopping criterion, rather than a fixed stopping tolerance ϵ i.e. the finer the discretisation, the tighter the stopping criterion, so we expect more iterations as h decreases, when compared to a fixed tolerance. But the effect is quite weak: if we use $\|\mathbf{r}_k\| \leq ch^2$ then $n > \sqrt{\kappa}|\log h|$ instead of $\sqrt{\kappa}$.

To see the effect of the spectrum on the convergence of CG, the following graph shows the convergence for 4 100×100 matrices, with different spectra.

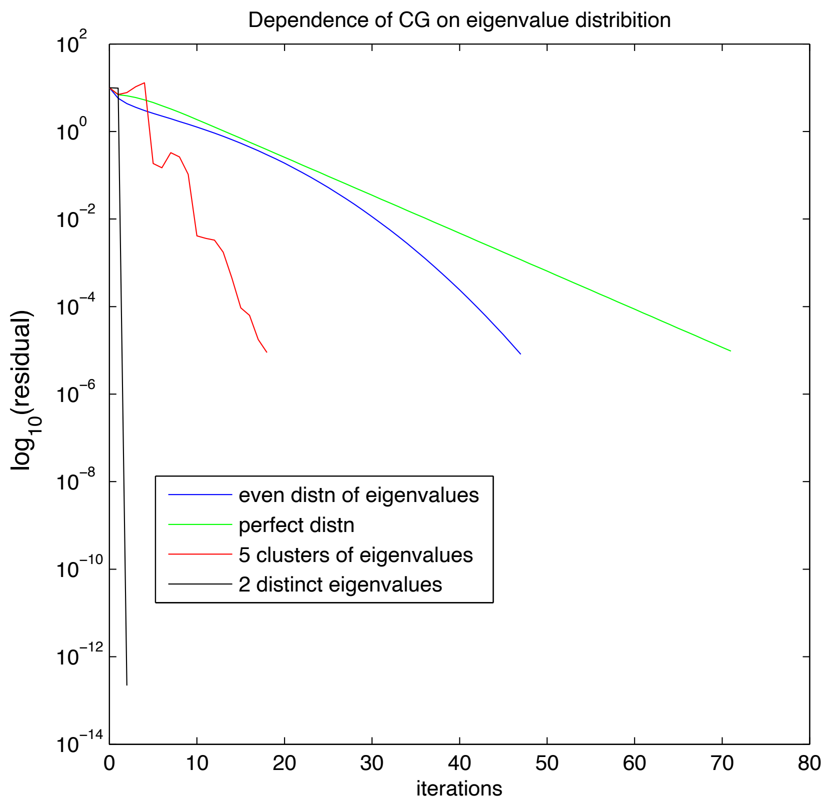

Figure 2: Convergence of CG for matrices of different spectra

Since CG converges faster if the spectrum is ‘nice’, we look for ways to improve the spectrum, i.e. **preconditioning**.

2.3.6 Preconditioned conjugate gradient (PCG)

Instead of solving $A\mathbf{x} = \mathbf{b}$, solve $M^{-1}A\mathbf{x} = M^{-1}\mathbf{b}$ such that:

- it must be cheap to solve $M\mathbf{z} = \mathbf{r}$ (which is the only place M enters in the PCG algorithm)
- $M^{-1}A$ must be sparse and symmetric positive definite (if A is)
- $M^{-1}A$ has a 'nicer' spectrum, e.g. $\kappa_2(M^{-1}A) \ll \kappa_2(A)$

These are conflicting requirements.

Example 2.26. If $M = I$ then $M\mathbf{z} = \mathbf{r}$ is easy to solve, but $\Lambda(M^{-1}A) = \Lambda(A)$ and we haven't done anything.

Example 2.27. If $M = A$ now $\Lambda(M^{-1}A) = \Lambda(I)$ which is perfect but $M\mathbf{z} = \mathbf{r}$ is $A\mathbf{z} = \mathbf{r}$ which is just our original problem.

We try and look for M that 'approximates' A so that the eigenvalues of $M^{-1}A$ are more clustered than A. We often look for a factorised $M = M_1M_2$ where M_1, M_2 are cheap to invert, e.g. Cholesky factors $M = LL^T = R^TR$

There are *no* magical preconditioners!

The best comes from a deep knowledge of A and where it came from. They often come from approximating A or solving a related simpler problem cheaply to get an approximation.

Some candidates that are symmetric positive definite and sparse:

- Jacobi (point), $M_J = D = \text{diag}(A)$.

  This has modest benefit unless the diagonal elements of A vary greatly. It has no effect on A_5 (since the diagonal elements are all the same).

- Incomplete factorisations

  For these we go through the process of (Cholesky) factorisation but prevent or reduce the amount of fill-in. So the matrix obtained is sparser than usual sparse Cholesky factorisation (less fill-in) but only gives an approximate factorisation.

Direct Cholesky (\)

$$A = R^T R \text{ or } LL^T.$$

Incomplete Cholesky

$$A = \bar{R}^T \bar{R} + \underbrace{\epsilon}_{\text{error}},$$

where $\bar{R}^T \bar{R} = M$ is a preconditioner, $M = M_1 M_2 = \bar{R}^T \bar{R}$ (or $M = \bar{L}\bar{L}^T$).

Matlab has (as at 2012):

- Incomplete Cholesky with no fill-in

  `L = ichol(A)`, which you then use in a call to `pcg`:

  `[ , , , ] = pcg(A,b,tol,maxits, L, L')`

  This only allows non-zeroes of L where A has non-zeroes i.e. no fill-in. To call CG in Matlab, use `pcg` with no preconditioner:

  `[ , , , ] = pcg(A,b,tol,maxits)`

- Modified incomplete Cholesky

  Where the matrix A has come from an elliptic PDE (as in our cases), it is better when you delete an entry that would have produced fill-in to add that entry to the diagonal entry on the same row, so that $A\mathbf{x} = \bar{L}\bar{L}^T\mathbf{x}$ when $\mathbf{x}$ is a constant vector. This is called Modified incomplete Cholesky (MIC) factorization. This can be called in Matlab by accessing the `options` structure in the call to `ichol`:

  ```
  options.type = 'nofill';
  options.michol = 'on';
  L = ichol(A,options)
  ```

  Supposedly, for A_5 this can reduce $\kappa_2(M^{-1}A)$ to $O(h^{-1})$.

  With this preconditioner, PCG has $\sqrt{\kappa} \sim m^{\frac{1}{2}}$ iterations, so the total work is $O(m^{2.5})$ in 2D, $O(m^{3.5})$ in 3D. This is called *optimal PCG* (for A_5) (not sure if this means it is proven to actually be the best possible algorithm).

- Incomplete Cholesky with a drop tolerance

  ```
  options.type = 'ict';
  options.droptol = 0.05;
  L = ichol(A,options)
  ```

  This throws away the fill-in entries smaller than a certain size determined by the drop tolerance. For large drop tolerance, this is the same as the no-fillin case (the default); as drop tolerance $\to 0$, $L \to$ same Cholesky factor as produced by \ (same fill-in as \).

There is another class of iterative solvers called multigrid iteration. This gives another preconditioner, *incomplete multigrid*, where the number of iterations is independent of problem size! This will be covered more in the final week presentations.

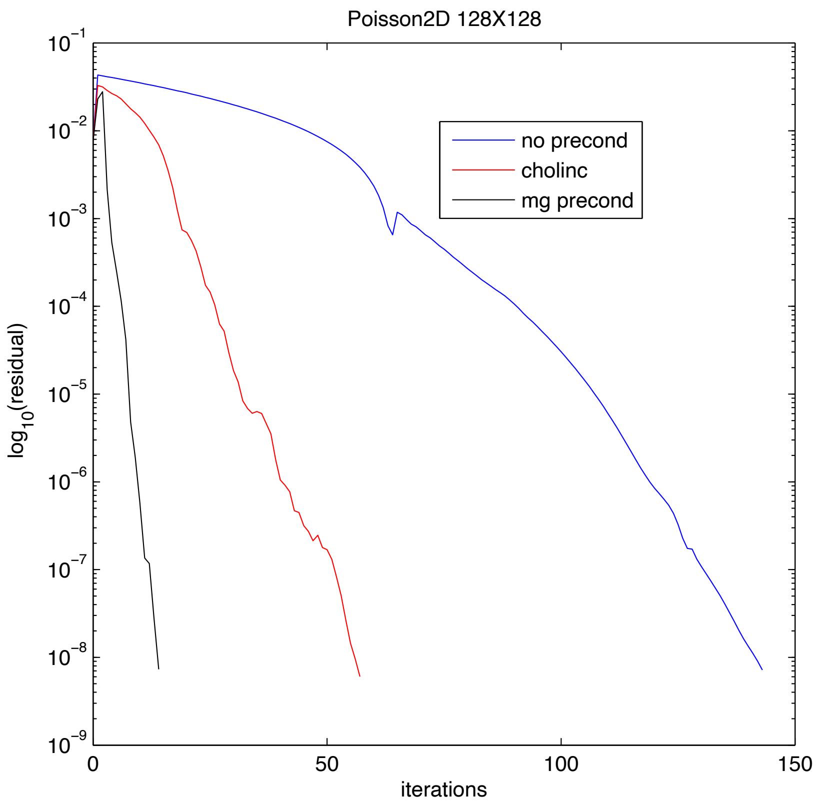

Figure 3: Convergence of PCG for 128×128 mesh : Poisson problem on L-shaped domain with Q_1 elements. The curves shown are: no preconditioner, incomplete Cholesky with no fill-in and incomplete multi grid.

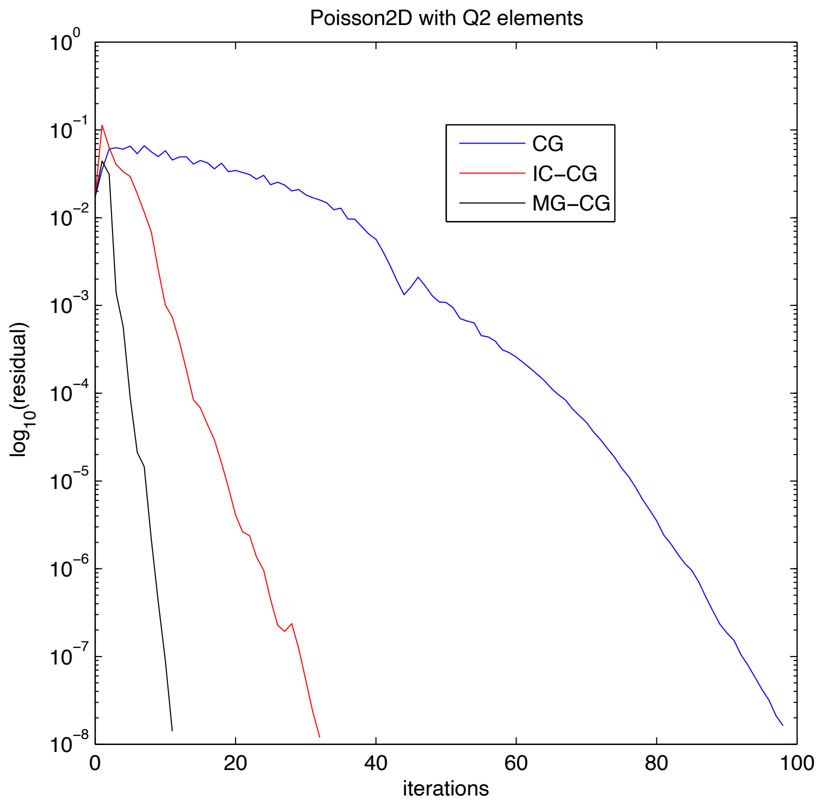

Figure 4: Convergence of PCG for 64×64 mesh : Poisson problem with Q_2 elements. The curves shown are: no preconditioner, incomplete Cholesky with no fill-in and incomplete multi grid.

3 Evolution PDEs (in 1+1 D)

Finally we add time to our equations!

Classical examples:

$$\begin{aligned} &u_t = u_{xx}, && \text{diffusion equation,} \\ &u_t + cu_x = 0, && \text{transport equation,} \\ &u_{tt} = c^2 u_{xx}, && \text{wave equation.} \end{aligned}$$

We will concentrate on 1+1 D (i.e. time + x) problems. This is because $1 + 2$ D and $1 + 3$ D problems combine the techniques of $1 + 1$ D with techniques for 2D and 3D elliptic problems that we have looked at before.

We can think of 1+1 D PDEs as adding time to a BVP or adding space to an IVP. To be well posed, typically we will need both boundary conditions (like a BVP) and initial conditions (like an IVP).

Example 3.1.
Consider

$$u_t - (D(x)u')' + q(x)u = r(x),$$

the reaction-diffusion equation with

$$\begin{aligned} \text{BCs} : u(a, t) &= \alpha(t), \\ u(b, t) &= \beta(t), \\ \text{IC} : u(x, 0) &= u_0(x). \end{aligned}$$

If $D = 1$, $q = 0$, $r = 0$ then this is the heat/diffusion equation.

3.1 Semi-discretisation

The simplest approach is *semi-discretisation* (the *Method of lines*). This discretises in space only, which creates a (large) system of ODEs which can be solved using IVP software. This was popularised by Schiesser in 1977. The method of lines relies on the existence of robust IVP software for solving stiff systems of ODEs (the first such software being due to Gear 1971).

Basic idea: discretise in space, e.g. FD, FEM, finite volume, spectral, collocation .etc. as before. This produces a system of ODEs that we solve with an IVP package e.g. `ode45` in MATLAB.

For larger problems, we need to discretise in time as well, e.g. if our IVP solver cannot handle the system of ODEs. This gives *fully discrete methods* which were invented in the 1950s and were actually the first methods developed.

But for modest problems the Method of Lines (MoL) is a good approach

Example 3.2.
The PDE toolbox, `pdetool`, solves 1+2 D parabolic/hyperbolic problems by the Method of lines (P_1 FEM) and `pdepe` solves 1+1 D parabolic-elliptic systems by the Method of lines (Petrov-Galerkin FEM).

3.1.1 Using Finite Difference to semi-discretize

Consider the problem

$$u_t - (D(x)u_x)_x = r(x,t).$$

As before, discrete in space on a (uniform) mesh of width h.

When we have an equation in flux form (or conservative form) e.g. from a conservation law, it's good practice to discretize each derivative separately, rather than expand derivatives and then discretize.

Example 3.3.
$(D(x)u_x)_x$ rather than $D'(x)u_x + D(x)u_{xx}$.

We will need to discretize this in stages, assuming we know u only at the mesh points.

$\rightarrow$ discretize using midpoints of mesh intervals.

Discretise $-(D(x)u_x)_x$ using central differences. We can use FD on $-D(x)u_x$ directly without expanding it. (We could expand and then apply FD too, similar to what we did earlier in the course, but this way guarantees certain nice properties for the resulting matrix.)

$$\begin{aligned}
D(x)u_x\big|_{x_j} &\approx \frac{D(x_j)(u_{j+1/2} - u_{j-1/2})}{h}, \\
(D(x)u_x)_x\big|_{x_j} &\approx \frac{D(x)u_x\big|_{x_{j+1/2}} - D(x)u_x\big|_{x_{j-1/2}}}{h} \\
&= \frac{1}{h}\left(\frac{D(x_{j+1/2})(u_{j+1} - u_j)}{h} - \frac{D(x_{j-1/2})(u_j - u_{j-1})}{h}\right) \\
&= \frac{1}{h^2}\left(D(x_{j+1/2})u_{j+1} - (D(x_{j+1/2}) + D(x_{j-1/2}))u_j + D(x_{j-1/2})u_{j-1}\right).
\end{aligned}$$

Now add Dirichlet boundary conditions:

$$u_1(t) = \alpha(t), \qquad u_{N+1}(t) = \beta(t).$$

Therefore at positions $j = 2, \ldots, N$ we have the following equations:

$$\dot{u}_j(t) - \frac{1}{h^2}\left(D(x_{j+1/2})u_{j+1} - (D(x_{j+1/2}) + D(x_{j-1/2}))u_j + D(x_{j-1/2})u_{j-1}\right) = r(x_j, t).$$

This is a complete system of $N-1$ ODEs. Use the BCs to modify the equations for $\dot{u}_2$ and $\dot{u}_N$. The initial condition becomes $u_j(0) = u_0(x_j)$.

Now plug this into an IVP/ODE solver. (They exist!)

Question: how big is the spatial discretisation error?
Answer: $O(h^2)$ because it's central differences

There is also a time discretisation error introduced by the IVP solver. Choose parameters so that time error is no worse than spatial error.

What type of IVP code should you use? There are lots out there.

3.1.2 Using FEM to semi-discretize

Now let's solve the same problem using finite elements. As we did before, we integrate by parts to get the weak formulation of the problem:

$$\int_a^b u_t\, v\, dx + \int_a^b D(x)\, u_x\, v_x\, dx = \int_a^b r\, v\, dx + \left.(D(x)\, u_x\, v)\right|_a^b .$$

Since we have Dirichlet BCs, choose our test space so that $D(x)\, u_x\, v = 0$, i.e. $v \in H_0^1$ (Sobolev space). Then:

$$\begin{aligned} u &= U_0(x) + \bar{u} \quad \in H_0^1 \times \mathbb{R}^+ \\ &= U_0(x) + \sum u_j(t) \underbrace{\phi_j(x)}_{\text{nodal basis}} \quad . \end{aligned}$$

Now we get the Galerkin equations. Replace v above with ϕ_k

$$\int_a^b \left(\sum_j \dot{u}_j \phi_j \right) \phi_k\, dx + \int_a^b D(x) \left[U_0' + \sum_j u_j(t) \phi_j' \right] \phi_k'\, dx = \int_a^b r\, \phi_k\, dx.$$

Swapping the order of the sum and the integral:

$$\sum_j \underbrace{\left(\int_a^b \phi_j \phi_k\, dx \right)}_{\mathbf{M}} \dot{u}_j + \sum_j \underbrace{\left(\int_a^b D(x) \phi_j' \phi_k'\, dx \right)}_{\mathbf{K}} u_j = \underbrace{\int_a^b r\phi_k\, dx - \int_a^b D(x) U_0' \phi_k'\, dx}_{\mathbf{f}} .$$

In matrix and vector notation:

$$\mathbf{M}\dot{\mathbf{u}} + \mathbf{K}\mathbf{u} = \mathbf{f}.$$

Again solve this using IVP solvers.

3.1.3 Recap of IVP solvers

a.k.a.
Things You Should Already Know About Solving ODEs
(But Have Probably Forgotten)

Consider the initial value problem,

$$\dot{\mathbf{y}} = \mathbf{f}(t, \mathbf{y}), \qquad \text{subject to } \mathbf{y}(0) = \mathbf{y}_0.$$

There are four classes of modern methods used to solve this:

1. **Explicit Runge-Kutta methods (1895–1905)**: Simple to code, one-step methods. Matlab's `ode23`, `ode45` are based on this.
2. **Explicit multi-step methods:** The most popular is Adams (1855). Very efficient, cheap higher-order methods (requiring one function evaluation per step, vs 3 or 6 for RK). Matlab: `ode113`.
3. **Implicit multi-step methods:** Most famous is Gear's Backward Differentiation Formula (1971). Matlab: `ode15s`.
4. **Implicit Runge-Kutta methods:** Have nice mathematical properties but are computationally expensive. Matlab: `ode23t`, `ode23tb`. There is also the TRBDF2 algorithm (1985) used for circuit simulation.

Explicit methods evaluate a simple formula to advance from $\mathbf{y}_n$ to $\mathbf{y}_{n+1}$. Implicit methods solve an equation or system of equations to go from $\mathbf{y}_n$ to $\mathbf{y}_{n+1}$. Usually this is more work.

One-step methods only need information from $[t_n, t_{n+1})$ to derive $\mathbf{y}_{n+1}$. Multi-step methods may need $\mathbf{y}_n, \mathbf{y}_{n-1}, \mathbf{y}_{n-2}, \ldots$ in order to compute $\mathbf{y}_{n+1}$, which can make them more efficient.

These days, most IVP solvers use a variable step size Δt, i.e. adaptively choose a time step to achieve an estimated local truncation error. Controlling global error is hard! If we have to write one by hand in this course we'll just use a fixed step size.

The user generally inputs a tolerance, either absolute, relative or both (c.f. `bvp4c`). The code tries to achieve this tolerance by varying the step size.

Spatial error is controlled by how you discretise space, time error is controlled by the parameters you give the IVP solver. These errors are approximately additive, so generally choose them to be of similar magnitude.

Some problems cannot be solved efficiently by explicit methods. These are called "stiff problems". Think of problems involving stiff springs. They arise a lot in discretised hyperbolic problems. They have have fast transients and a wide dynamic range (i.e., both slow and fast time scales present in the same problem). For these problems, we need a special class of IVP solvers called stiff solvers, e.g. `ode15s`.

There is no explicit formal measure of "stiffness".

3.1.4 Linear Stability Analysis

Consider the autonomous case (i.e., where f has no dependence on t):

$$\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y}).$$

We travel along a solution $\bar{\mathbf{u}}$, varying slowly. In each step, we make an error $\mathbf{z}$. How does this error propagate? Write $\mathbf{y}(t) = \bar{\mathbf{u}} + \mathbf{z}(t)$, so:

$$\begin{aligned} & \dot{\mathbf{y}} = \mathbf{f}(\mathbf{y}), \\ \implies \quad & \dot{\mathbf{z}} = \mathbf{f}(\bar{\mathbf{u}} + \mathbf{z}) \\ & \phantom{\dot{\mathbf{z}}} \approx \mathbf{f}(\bar{\mathbf{u}}) + \left.\frac{\partial \mathbf{f}}{\partial \bar{\mathbf{u}}}\right|_{\bar{\mathbf{u}}} \mathbf{z} + O(||\mathbf{z}||^2). \end{aligned}$$

So locally this behaves like $\dot{\mathbf{z}} = J|_{\bar{\mathbf{u}}}\mathbf{z} + \mathbf{b}$ where J is the Jacobian, which we assume to be roughly constant.

If J is diagonalisable (has a full set of linearly independent eigenvectors), we can write $J = T\Lambda T^{-1}$ (by the spectral theorem) where Λ is a diagonal matrix of eigenvalues and T is a matrix of eigenvectors. So:

$$\begin{aligned} \dot{\mathbf{z}} &= T\Lambda T^{-1}\mathbf{z} + \mathbf{b}, \\ \text{Define } \mathbf{w} &= T^{-1}\mathbf{z}, \qquad \mathbf{z} = T\mathbf{w}, \\ \text{So } \dot{\mathbf{w}} &= \Lambda \mathbf{w} + T^{-1}\mathbf{b}. \end{aligned}$$

This is a system of uncoupled scalar ODEs, $\dot{w}_i = \lambda_i w_i + b_i$. Since a real matrix can have complex eigenvalues, $\lambda \in \mathbb{C}$.

For z to be damped, the eigenvalues must have negative real part, i.e. we need $\mathfrak{Re}(\lambda_i) < 0$ in order to achieve asymptotic stability.

If this is true of the problem, we want the numerical method to have the same properties. This is less crucial if the Jacobian is positive as the problem is growing, so maybe relative error is stable anyway.

So we want the numerical errors to decay if the true errors decay. This is called **A-stability**. Test your method on $\dot{w} = \lambda w$ (for some $\lambda \in \mathbb{C}$). The method is A-stable if the numerical solution stays bounded as $n \to \infty$ whenever $\mathfrak{Re}(\lambda) < 0$.

Region of Absolute Stability (RAS): This is defined by

$$\{\lambda \Delta t \in \mathbb{C} : |w_n| \text{ stays bounded as } n \to \infty\}.$$

Ideally this region would be the whole left-hand side of the complex plane. For explicit methods (RK, Adams), the RAS is a bounded region.

Example 3.4.
See `RKRAS.m` for a demonstration on RAS for RK4.

For RK1 (Euler's method) on one problem we can derive the RAS easily:

$$\begin{aligned} y_{n+1} &= y_n + hf(t_n, y_n) \\ &= y_n + h\lambda y_n \\ &= y_n(1 + h\lambda), \\ |y_{n+1}| < \infty \text{ as } n \to \infty &\Rightarrow |1 + h\lambda| \leq 1, \\ &\Rightarrow 0 \leq h \leq 2/|\lambda|. \end{aligned}$$

This provides an upper bound on h to achieve numerical stability.

If $\lambda \approx -1$ stability is not an issue, but still need Δt smaller for accuracy reasons. But what if $\lambda \ll -1$? Then we need $\Delta t < \left|\frac{2}{\lambda_{\max}}\right| \ll 1$. Now stability requirements force a much smaller Δt than accuracy requirements.

This is a characteristic of stiff problems and cannot be solved efficiently by explicit methods except RKC (Runge-Kutta-Chebyshev) for moderately stiff problems.

A-stability: $|w_n|$ bounded whenever $\mathfrak{Re}(\lambda) < 0$.
L-stability: $\left|\frac{w_{n+1}}{w_n}\right| \to 0$ as $\lambda \to -\infty$.

Numerical errors are strongly damped whenever neighbouring solutions decay rapidly onto our solution, so need RAS to contain the negative real axis and $\left|\frac{w_{n+1}}{w_n}\right| \to 0$.

Example 3.5.
BDF methods order 1–6 (L-stable). MATLAB `ode15s`. Implicit Runge-Kutta Methods (these include TR-BDF4, `ode23tb`).

3.1.5 The Heat Equation

In summary the MoL gives rise to the following systems

$$\begin{aligned} \dot{\mathbf{u}} &= A\mathbf{u} + \mathbf{b}, \\ M\dot{\mathbf{u}} + K\mathbf{u} &= f, \end{aligned}$$

for Finite Difference and Finite Element formulations.

Consider the heat equation, $u_n = u_{xx}$. Method of Lines gives $\dot{\mathbf{u}} = A\mathbf{u}$.

Theorem 3.6.
(Iserles, Ch. 12) A (for finite difference) has eigenvalues $\frac{2}{h^2}(\cos(\pi kh) - 1)$*, for* $k = 1, \ldots, N-1$.

If $\lambda_i < 0$ we have a dissipative system of ODEs.

$$\begin{aligned} \lambda_1 &= \tfrac{2}{h^2}\left(\cos(\pi h) - 1\right) \\ &= \tfrac{2}{h^2}(1 - \tfrac{1}{2}\pi^2 h^2 + \cdots - 1) \\ &= -\pi^2 + O(h^2), \\ \lambda_{N-1} &= \tfrac{2}{h^2}\left(\cos(\pi(N-1)h) - 1\right) \\ &\approx \tfrac{2}{h^2}(-1 - 1) \\ &= -4/h^2. \end{aligned}$$

We can see that λ_1 is bounded away from zero, so $\|A^{-1}\|_2 \leq 1/\pi^2$; whereas $\lambda_{N-1} \to -\infty$ as $h \to 0$. Thus:

$$\kappa_2(A) = \left|\frac{\lambda_{\max}}{\lambda_{\min}}\right| \approx \frac{4}{\pi^2 h^2}.$$

Need both A-stability and L-stability?

3.1.6 The Heat Equation: Seperable Solutions

Take the heat equation with zero homogeneous boundary conditions

$$\begin{aligned} & u_t = u_{xx}, \qquad u(0,t) = 0, \qquad u(1,t) = 0, \\ \implies \quad & u = \sum_{n=1}^{\infty} a_n \sin(n\pi x) e^{-n^2\pi^2 t}. \end{aligned}$$

Where $\sin(n\pi x)$ is an eigenfunction of D^2, an operator subject to the boundary conditions:

$$\begin{aligned} \frac{d^2}{dx^2} sin(n\pi x) &= -n^2\pi^2 \sin(n\pi x), \\ \sin(n\pi x) &= 0 \text{ at } x = 0, 1. \end{aligned}$$

Thus the D^2 operator (second derivative in space) has an infinite set of eigenvalues: $-\pi^2, -4\pi^2, -9\pi^2, \ldots$

For a numerical methods: $\lambda_1 = -\pi^2 + O(h^2), \lambda_2, \ldots, \lambda_{N-1} \approx -\frac{4}{h^2}$. As $h \to 0$, we get more and more eigenvalues trying to match exact eigenvalues. Modes with wide and varying timescales are a sign of stiffness.

Rule of thumb: parabolic problems with method of lines yield a stiff IVP problem requiring a stiff solver.

What about infinite space boundary conditions? Apply Fourier transforms with respect to x:

$$\begin{aligned} u_t &= u_{xx}, \\ \hat{u} &= \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} u \, e^{-ikx} \, dx, \\ \hat{u}_x &= -ik\hat{u}, \\ \implies \quad \hat{u}_k(t) &= \hat{u}_k(0) e^{-k^2 t}. \end{aligned}$$

The mode with wave number k is damped by factor $e^{-k^2 t}$.

3.2 Fully discrete: Parabolic PDEs

If we also discretise the time derivative we get a set of fully discrete methods. With these methods we can afford to be a little bit outside the Region of Absolute Stability, but not much, to keep solutions bounded.

These were the first methods invented due to much lower available computing power at the time. For big problems, even now, nobody uses Method of Lines because the systems of ODEs get too big and becomes to computationally demanding.

When solving PDEs you need to adapt these methods to the specific problem at hand, or even use different methods for different parts of a problem.

First of all, we will consider parabolic problems, e.g. $u_t = u_{xx}$. These are numerically easier or more forgiving: the DE itself tends to smooth out errors. But they can still be tricky. The first simplest method we will see is very inefficient.

3.2.1 FTCS (Forward Time, Central Space)

First, discretise u_{xx} using central differences to get:

$$\dot{\mathbf{u}} = A\mathbf{u} + \mathbf{b}.$$

Where the matrix A is

$$A = \frac{1}{h^2}\begin{bmatrix} -2 & 1 & 0 & \dots & & & \\ 1 & -2 & 1 & 0 & \dots & & \\ & & & \ddots & & & \\ & & \dots & 0 & 1 & -2 & 1 \\ & & & \dots & 0 & 1 & -2 \end{bmatrix}.$$

Now we look at time. Consider an even simpler problem, $\dot{y} = f(t, y)$. The simplest possible method we could use is Euler's method (a.k.a. Runge-Kutta 1). This gives us

$$y_{n+1} = y_n + k\, f(t_n, y_n),$$

where k is the time-step (equivalent to h in space). Using superscripts to represent discrete time indices and subscripts for space indices, we can rearrange this to give:

$$\dot{y}|_{t_n} = \frac{y^{n+1} - y^n}{k}.$$

Combine this with our equation for u to get our first fully-discrete method: FTCS.

$$u_j^{n+1} = u_j^n + \frac{k}{h^2}\left(u_{j-1}^n - 2u_j^n + u_{j+1}^n\right) + b_j^n.$$

Or, in vector form:

$$\mathbf{u}^{n+1} = \mathbf{u}^n + \underbrace{\frac{k}{h^2}}_{=\,r} h^2 A\mathbf{u}^n + \mathbf{b}.$$

This is an explicit method.

Our discretisation error is $O(k + h^2)$, i.e. the method is first order in time and second order in space. The fact that these are not the same order is often an issue. As you refine the space-time mesh with k/h fixed, we would like to have the time and space errors balanced. FTCS doesn't really allow this as time error will be much bigger than space error, i.e. $O(k) \gg O(h^2)$ if $k \approx h \ll 1$.

The other reason comes from stability. Recall that the eigenvalues of A lie in $[-4/h^2, \pi^2]$ so $|\lambda_i|_{\max} \approx 4/h^2$. For stability we need $k\lambda_i$ to lie within the RAS for the time-stepping scheme. Euler's method has RAS $|1 + k\lambda| < 1$.

(Sketch of proof: apply $y_{n+1} = y_n + kf(t_n, y_n)$ to $\dot{y} = \lambda y$ to get $y_{n+1} = (1 + k\lambda)y_n$. We want $|y_{n+1}/y_n|$ bounded as $n \to \infty$.)

So substitute in our value for λ to get the stability of FTCS:

$$\begin{aligned}
\left|1 + k\left(\tfrac{-4}{h^2}\right)\right| &< 1,\\
\left|1 - \tfrac{4k}{h^2}\right| &< 1,\\
-1 < 1 - \tfrac{4k}{h^2} &< 1,\\
\implies \quad -\tfrac{4k}{h^2} &< 0, \qquad \text{tells us nothing,}\\
\text{and} \quad \tfrac{k}{h^2} &< 1/2, \qquad \text{i.e. } r < 1/2.
\end{aligned}$$

Thus FTCS is **conditionally stable**. This is a severe restriction on the time step. If $h = 0.01$ and diffusion constant $= 1$, $k < 5 \times 10^{-5}$. That is a very small time step! So FTCS is quick to evaluate but takes a lot of time steps to get anything (similar to `ode45`, which is a non-stiff solver, used with MoL).

This is inefficient, so people don't generally use it for parabolic problems.

3.2.2 BTCS (Backwards Time, Central Space)

Let's try using Euler's method backwards:

$$\begin{aligned} \dot{y}\big|_{t_n} &= \frac{y_n - y_{n-1}}{k}, \\ \implies \quad y_n &= y_{n-1} + k\, f(t_n, y_n). \end{aligned}$$

Note that y_n is present on both sides of the equation, so this is an implicit method. This is called Backward Euler or Implicit Euler method (also same as first order BDF).

Adding in space gives us the BTCS method. We must solve an algebraic system for each time step, so it is more work per time step than FTCS.

What is the RAS for BTCS?

$$\begin{aligned} \dot{y} = \lambda y \quad \implies \quad y_{n+1} &= y_n + k\lambda y_{n+1}, \\ y_{n+1}(1 - k\lambda) &= y_n, \\ \left|\tfrac{y_{n+1}}{y_n}\right| \text{ bounded if } \left|\tfrac{1}{1-k\lambda}\right| &< 1, \\ \text{i.e. } |1 - k\lambda| &> 1. \end{aligned}$$

This method is **A-stable** and **L-stable** (as $\lambda \to \infty$, $\left|\frac{y_{n+1}}{y_n}\right| = \left|\frac{1}{1-k\lambda}\right| \to 0$).

But once again, our discretisation error is $O(k+h^2)$. Our system of equations for u:

$$u_j^{n+1} = u_j^n + \frac{k}{h^2}\left(u_{j-1}^{n+1} - 2u_j^{n+1} + u_{j+1}^{n+1}\right) + b_j.$$

In vector notation, using $\bar{A} = h^2 A$ (i.e. remove the $1/h^2$ factor from out the front):

$$\begin{aligned} \mathbf{u}^{n+1} &= \mathbf{u}^n + r\bar{A}\mathbf{u}^{n+1} + \mathbf{b}^n, \\ (I - r\bar{A})\mathbf{u}^{n+1} &= \mathbf{u}^n + \mathbf{b}^n. \end{aligned}$$

The matrix $I - r\bar{A}$ is tridiagonal so solving is cheap: just use \ inMATLAB. (In 1D, anyway. In 2D it's a bit harder.)

This method is unconditionally stable. Choose k and h to achieve the desired accuracy without having to worry about stability (but may still need $k \ll h$ because first-order error in k but second-order in h).

Sometimes people use FTCS anyway if the problem is convection-dominated.

3.2.3 Crank-Nicholson Method (Implicit Trapezoid Rule)

This is the method that solves all of the above problems.

Recall the trapezoid rule for solving an ODE:

$$y_{n+1} = y_n + \tfrac{1}{2}k(f(t_n, y_n) + f(t_{n+1}, y_{n+1})).$$

This has $O(k^2)$ time step error, i.e. is second order.

The RAS includes the entire left complex plane, so it is A-stable but not L-stable.

$$\left|\frac{2+k\lambda}{2-k\lambda}\right| < 1$$

This method results in the matrix equations:

$$\begin{aligned} \mathbf{u}^{n+1} &= \mathbf{u}^n + \tfrac{k}{2h^2}\left(\bar{A}\mathbf{u}^n + \bar{A}\mathbf{u}^{n+1}\right) + \mathbf{b}, \\ \implies \quad \left(I - \tfrac{r}{2}\bar{A}\right)\mathbf{u}^{n+1} &= \left(I + \tfrac{r}{2}\bar{A}\right)\mathbf{u}^n + \mathbf{b}, \qquad \text{(recall } r = k/h^2\text{)}. \end{aligned}$$

This is another implicit method, but again it's easy to solve: use \.

This is known as the Crank-Nicholson method (1947). It's quite cheap in 1D. Truncation error is $O(k^2 + h^2)$. It is unconditionally stable for diffusion equations so is the first method people try for these problems.

3.2.4 Von Neumann Stability Analysis

Recall that for BVPs:

$$\begin{array}{ccccc} \text{consistency} & + & \text{stability} & \to & \text{convergence.} \\ \tau = O(h^p) & & \|A^{-1}\| \le c & & E = O(h^p) \\ \text{(local truncation error)} & & & & \end{array}$$

Now we have linear evolution PDEs:

$$\begin{array}{ccccc} \text{consistency} & + & \text{stability} & \to & \text{convergence.} \\ \tau = O(k+h^2) & & \text{eigenvalues of } A & & E = O(k+h^2) \\ \text{(local truncation error,} & & & & \\ \text{FTCS/BTCS)} & & & & \end{array}$$

Next: look at Von Neumann stability analysis, a.k.a. Fourier stability analysis. We will consider only interior nodes, periodic BCs, and problems on $\mathbb{R}$ but these ideas can be extended.

Use a Fourier transform to solve a linear PDE on $\mathbb{R}$:

$$\frac{\partial}{\partial x} e^{iqx} = iqe^{iqx}.$$

For any difference operator (FD, BD, CD), we get a grid function $W_j = e^{iqx_j} = e^{iqjh}$ which is an eigenfunction of the difference operator.

E.g. with forward differences:

$$\begin{aligned} \frac{W_{j+1} - W_j}{h} = \frac{e^{iq(j+1)h} - e^{iqjh}}{h} &= \frac{1}{h} e^{iqjh}(e^{iqh} - 1) \\ &= e^{iqjh} \underbrace{\left(\frac{e^{iqh} - 1}{h} \right)}_{\text{eigenvalue}}, \end{aligned}$$

$$W_j = \frac{1}{\sqrt{2\pi}} \int_{-\pi/h}^{\pi/h} \hat{W}(q)\, e^{iqhj}\, dq,$$

where

$$\hat{W}(q) = h \sum_{j=-\infty}^{\infty} e^{-iqhj}\, W_j.$$

We now have a whole bunch of different Fourier transform-ish methods:

	continuous x	discrete x_j
continuous q	Fourier transform	"semi-discrete" FT
discrete q_j	Fourier series	discrete FT

On a grid with spacing h, we can't resolve frequencies over $2\pi/h$, so $q \in [0, 2\pi/h)$ or $q \in [-\pi/h, \pi/h)$.

Parseval's Relation: Using the grid function 2-norm,

$$\|u\|_{\ell^2} = \left(h \sum_j |u_j|^2 \right)^{1/2} = \|\hat{U}\|_2 = \left(\int_{-\pi/h}^{\pi/h} |\hat{U}(q)|^2 \, dq \right)^{1/2}.$$

For example, consider FTCS. Look for $\hat{U}^{n+1}(q) = g(q)\hat{U}^n(q)$, where $g(q)$ is an "amplification factor". To find $g(q)$ we use the eigenfunction e^{iqjh}:

$$\begin{aligned} u_j^{n+1} &= u_j^n + r(u_{j+1}^n - 2u_j^n + u_{j-1}^n), \\ u_j^{n+1} &= e^{iqhj} + r(e^{iqh(j+1)} - 2e^{iqhj} + e^{iqh(j-1)}) \\ &= e^{iqjh}(1 + re^{iqh} - 2 + e^{-iqh}) \\ &= u_j^n \underbrace{(1 + 2r(\cos(qh) - 1))}_{= \, g(qh)}. \end{aligned}$$

For strong stability, we want $|g(qh)| < 1$ for $qh \in [-\pi, \pi]$. So we need:

$$\begin{aligned} 1 - 4r < g = 1 + 2r(\cos(qh) - 1) &< 1, \\ 1 - 4r &> -1, \\ r &< 1/2. \end{aligned}$$

Note that this gives us a necessary condition, not a sufficient condition. Boundary conditions can also cause instability.

3.2.5 1+2D parabolic problem

The following material mostly comes from Leveque Ch. 10. Suppose we have a 1+2D parabolic problem:

$$u_t = u_{xx} + u_{yy}.$$

Using Crank-Nicholson plus a five-point stencil, we end up with equations like:

$$\begin{aligned} u_{ij}^{n+1} &= u_{ij}^n + \tfrac{k}{2}\left(\nabla_h^2 u_{ij}^n + \nabla_h^2 u_{ij}^{n+1}\right), \\ \implies \quad \underbrace{(I - \tfrac{k}{2}\nabla_h^2)}_{=\,A} u_{ij}^{n+1} &= (I + \tfrac{k}{2}\nabla_h^2) u_{ij}^n. \end{aligned}$$

Note that the matrix for ∇_h^2 will be block tridiagonal, and A will be a fixed $m^2 \times m^2$ matrix. The condition number $\kappa_2(A) = O(k/h^2)$.

We have a good initial guess for conjugate gradient methods—it doesn't vary much with each time step.

Alternative: split the method and solve each direction separately. This only works on easy domains like rectangles (not butterflies!).

3.2.6 LOD: Locally One-Dimensional

Use the fact that $\nabla_h^2 = D_x^2 + D_y^2$ to break up the computation: solve in x direction, then in y.

Start with the Laplacian:

$$\begin{aligned}
u_{ij}^{n+1} &= u_{ij}^n + \tfrac{k}{2}(D_x^2 u_{ij}^n + D_x^2 u_{ij}^{n+1} + D_y^2 u_{ij}^n + D_y^2 u_{ij}^{n+1}),\\
\implies \quad u_{ij}^* &= u_{ij}^n + \tfrac{k}{2}(D_x^2 u_{ij}^n + D_x^2 u_{ij}^*),\\
u_{ij}^{n+1} &= u_{ij}^* + \tfrac{k}{2}(D_y^2 u_{ij}^* + D_y^2 u_{ij}^{n+1}),\\
\implies \quad (I - \tfrac{k}{2}D_x^2)u^* &= (I + \tfrac{k}{2}D_x^2)u^n, \qquad \text{C–N in } x, \qquad (1)\\
(I - \tfrac{k}{2}D_y^2)u^{n+1} &= (I + \tfrac{k}{2}D_y^2)u^*, \qquad \text{C–N in } y. \qquad (2)
\end{aligned}$$

In (1), u^* is compiled over rows through D_x^2. Each equation for j is independent, so there are $j = 0, \dots, m+1$ systems, and each system has a tridiagonal matrix for u_{ij}^*. So it's $O(m^2)$ operations to solve for u^*.

In (2), u^{n+1} is compiles over columns through D_y^2, giving m tridiagonal systems. Total work is $2m + 2$ tridiagonal systems of size m, for $O(m^2)$ operations.

See Leveque §9.8 for more information.

3.2.7 ADI: Alternative Direction Implicit

Also $O(k^2 + h^2)$ error, unconditionally stable, $O(m^2)$ operations per time step. In the first step, the y diffusion is explicit, x diffusion is implicit. In the second step, it is reversed.

$$\begin{aligned} u_{ij}^* &= \tfrac{k}{2}(D_y^2 u_{ij}^n + D_x^2 u_{ij}^*), \\ u_{ij}^{n+1} &= u_{ij}^* + \tfrac{k}{2}(D_x^2 u_{ij}^* + D_y^2 u_{ij}^{n+1}). \end{aligned}$$

See Leveque §9.8.2 for more information.

3.3 Fully discrete: Hyperbolic PDEs

Parabolic equations are easy to solve numerically. The solution is always smooth, so the local truncation errors are okay. Need implicit methods for stability. Hyperbolic equations are typically solved by explicit methods so the solution can develop and propagate discontinuities or shocks. Numerically tougher.

Most of the assumptions in finite difference methods break down when looking at Hyperbolic problems: **Finite Volume Methods**.

We cover "classical methods". References: Leveque, CLAWPACK. LINPACK (high-res methods), EISPACK (eigenvalues), FISHPACK (Poisson), CRAY-FISHPACK (Poisson). MATLAB: PDEPE.

3.3.1 The Advection Equation

The wave equation $u_{tt} = c^2 u_{xx}$ can be treated as a system of first-order problems.

Define $v = u_t, w = -cu_x$ then

$$v_t = u_{tt} = c^2 u_{xx} = -cw_x,$$

and

$$v_t + cw_x = 0, \qquad w_t + cv_x = 0.$$

So if we let

$$\mathbf{u} = \begin{bmatrix} v \\ w \end{bmatrix},$$

$$\mathbf{u}_t + \begin{bmatrix} 0 & c \\ c & 0 \end{bmatrix} \mathbf{u}_x = 0,$$

$$\mathbf{u}_t + A\mathbf{u}_x = 0, \qquad \text{(both linear)}.$$

The system is hyperbolic if A is diagonalisable with real eigenvalues, $\det(A - I\lambda) = 0$. e.g.:

$$A = \begin{bmatrix} 0 & c \\ c & 0 \end{bmatrix}, \quad \Longrightarrow \quad \lambda = \pm c.$$

The simplest hyperbolic PDE is $u_t + au_x = 0$. This is the advection equation, or one-way wave equation.

Initial profile u_0 is advected with velocity a, no dissipation.

We want methods that will generalise to: systems $\mathbf{u}_t + A\mathbf{u}_x = \mathbf{0}$; and conservation laws $u_t + (f(u))_x = 0$ (where $f(u)$ is the flux), e.g. Burger's equation $u_t + (u^2)_x = 0$.

3.3.2 Method of Lines

We are trying to solve $u_t + cu_x = 0$ on periodic BCs, $u(0,t) = u(1,t)$. Use central differences for u_x. Assume we have N mesh intervals, so $k = 1/N$. Then our BCs are $u_{N+1} = u_1$, and we need to solve for $u_2, \dots, u_{N+1}$. Then using our MoL equations from before:

$$\begin{aligned}
\dot{u}_j(t) &= -\tfrac{c}{2h}(u_{j+1} - u_{j-1}), \\
\dot{u}_2(t) &= -\tfrac{c}{2h}(u_3 - u_1) = -\tfrac{c}{2h}(u_3 - u_{N+1}), && \text{(not solving for } u_1\text{)}, \\
\dot{u}_{N+1}(t) &= -\tfrac{c}{2h}(u_{N+2} - u_N) = -\tfrac{c}{2h}(u_2 - u_N), && \text{(periodic BCs)}.
\end{aligned}$$

Therefore

$$\dot{\mathbf{u}} = A\mathbf{u},$$

where the matrix A is given by:

$$A = -\frac{c}{2h}\begin{bmatrix} 0 & 1 & \cdots & & & -1 \\ -1 & 0 & 1 & \cdots & & \\ \vdots & -1 & 0 & 1 & \cdots & \\ \vdots & & & & & \\ 1 & -1 & 0 & 1 & \cdots & \end{bmatrix}.$$

This time A is skew-symmetric: $A^T = -A$! Its eigenvalues are imaginary.

$$\lambda_p = -\frac{ic}{h}\sin(2\pi ph), \qquad p = 1, \dots, N.$$

Starts small, goes up and down. $\lambda_p \in [-ic/h, ic/h]$.

For stability, we need RAS for time stepper $\supset \Lambda(A)$.

So we try FTCS. But its RAS never includes λ_p. FTCS is *never stable for the advection equation.* So try CTCS and BTCS.

3.3.3 Leapfrog Methods

This is like central differences in time (CT).

$$\dot{u} = \frac{u^{n+1} - u^{n-1}}{2k}.$$

Its RAS is $|\mathfrak{Im}(z)| \leq 1$. Leapfrog will be stable provided

$$\lambda_k \in [-ick/h, ick/h] \subset [-i, i],$$

i.e. $\left|\frac{ck}{h}\right| < 1$.

We write $\frac{ck}{h} = \nu$ and call this the Courant Number. It is our condition for stability.

Another algorithm is the fully discrete leapfrog method (CTCS).

$$u_j^{n+1} = u_j^{n-1} - \tfrac{ck}{h}(u_{j+1}^n - u_{j-1}^n).$$

This is an explicit method, $O(h^2 + k^2)$. Three time levels means we need a starting procedure and more memory.

3.3.4 Lax-Friedrichs Method (LxF)

This is our starting procedure. For FTCS we had:

$$u_j^{n+1} = u_j^n - \tfrac{\nu}{2}(u_{j+1}^n - u_{j-1}^n).$$

If we replace u_j^n with the average $\frac{1}{2}(u_{j+1}^n + u_{j-1}^n)$ we get LxF:

$$u_j^{n+1} = \tfrac{1}{2}(u_{j+1}^n + u_{j-1}^n) - \tfrac{\nu}{2}(u_{j+1}^n - u_{j-1}^n).$$

and CLAW (for nonlinear conservation laws):

$$u_j^{n+1} = \tfrac{1}{2}(u_{j+1}^n + u_{j-1}^n) - \tfrac{\nu}{2}(f(u_{j+1}^n) - f(u_{j-1}^n)).$$

Because eigenvalues are on i axis, they are always marginally stable, so it is dangerous. So change space disc so that eigenvalues are on lefthand plane, so errors will be damped.

Note that $\frac{1}{2}(u_{j+1}^n + u_{j-1}^n)$ can be rewritten as $u_j^n + \frac{1}{2}(u_{j+1}^n - 2u_j^n + u_{j-1}^n)$. So you've added dissipation (using central differences of u_{xx}). Adding numerical diffusion to stabilise the solution.

Approximation of time derivative at j:

$$\frac{u_j^{n+1} - u_j^n}{k} = -\tfrac{c}{2h}A_-\mathbf{u} + \tfrac{1}{2k}A_+\mathbf{u}.$$

Splitting A into skew-symmetric and symmetric parts. Note: as $k \to 0$, the $1/2k$ term will blow up.

$$\dot{u}_j = -\tfrac{c}{2h}A_-\mathbf{u} + \tfrac{1}{2k}A_+\mathbf{u} = B\mathbf{u}.$$

We write down the eigenvalues of B: $-\frac{ic}{h}\sin(2\pi ph) - \frac{1}{k}(1 - cos(2\pi ph)) = \mu$

So at time step k, $k\mu = -i\nu\sin(2\pi ph) - (1 - cos(2\pi ph)$ which lie on an ellipse. If p is a parameter then

$$k\mu = -\underbrace{i\nu\sin(2\pi ph)}_{\text{imaginary}} - \underbrace{(1 - cos(2\pi ph))}_{\text{real}}.$$

So $-\nu < \mathfrak{Im}(k\mu) < \nu$ and $-2 \leq \mathfrak{Re}(k\mu) \leq 0$.

LxF is stable if FT RAS $\subset k\mu$ of B, which happens iff $|\nu| < 1$. In other words, the distance the wave moves in a time step must be smaller than the spatial grid.

Can also calculate local truncation error:

$$\frac{1}{2}k(c^2 - h^2/k^2)u_{xx} + O(\cdots)$$

This is an easy to code two-step method, but always need to satisfy stability condition.

3.3.5 Lax-Wendroff Method (LxW)

We return to the advection equation, $u_t + cu_x = 0$. Now try the LxF method with a Taylor expansion:

$$\begin{aligned}
&& u(x,t+k) &= u(x,t) + ku_t(x,t) + \tfrac{1}{2}k^2 u_t(x,t) + O(k^3),\\
&& u_t = -cu_x, &\qquad u_{tt} = c^2 u_{xx},\\
\Longrightarrow && u(x,t+k) &= u(x,t) - kc\underbrace{u_x}_{\text{CD}} + kc^2\underbrace{u_{xx}}_{\text{CD}} + O(k^3),\\
\Longrightarrow && u_j^{n+1} &= u_j^n - \tfrac{ck}{2h}(u_{j+1}^n - u_{j-1}^n) + \tfrac{c^2k^2}{2h^2}(u_{j+1}^n - 2u_j^n + u_{j-1}^n),
\end{aligned}$$

This is called the Lax-Wendroff method (1960s), a.k.a. LxW. It is $O(k^2+h^2)$.

$\dot{\mathbf{u}} = B\mathbf{u}$ evolves the system and:

$$k\mu = -i\left(\tfrac{ck}{h}\right)\sin(\pi ph) + \left(\tfrac{ck}{h}\right)^2(\cos(\pi ph) - 1), \qquad p = 1,\dots,N.$$

Therefore, the RAS is $|\nu| \le 1$.

Proof. Von Neumann Error Analysis:

$$\begin{aligned}
u_j^n e^{iqhj} &= e^{iqhj} - \tfrac{1}{2}\nu(e^{iqh(j+1)} - e^{iqh(j-1)}) + \tfrac{1}{2}\nu^2(e^{iqh(j+1)} - 2e^{iqhj} + e^{iqh(j-1)})\\
&= e^{iqhj}(1 - \tfrac{1}{2}\nu 2i\sin(qh) + \tfrac{1}{2}\nu^2(2\cos(qh) - 2))\\
&= e^{iqhj}(1 - \tfrac{1}{2}\nu 2i\sin(qh) + \nu^2(\cos(qh) - 1)),\\
g(qh) &= 1 - i\nu 2\sin(qh/2)\cos(qh/2) - \nu^2 2\sin^2(qh/2),\\
|g|^2 &= (1 - 2\nu^2\sin^2(qh/2))^2 + 4\nu^2\sin^2(qh/2)\cos^2(qh/2)\\
&= 1 - 4\nu^2\sin^2(qh/2) + 4\nu^4\sin^4(qh/2) + 4\nu^2\sin^2(qh/2)\cos^2(qh/2)\\
&= 1 - 4\nu^2\sin^2(qh/2)(1 - \cos^2(qh/2)) + 4\nu^4\sin^4(qh/2)\\
&= 1 - 4\nu^2(1-\nu^2)\underbrace{\sin^4(qh/2)}_{\in\,[0,1]}.
\end{aligned}$$

$$\Longrightarrow \quad |g|^2 \le 1, \quad \text{if } |\nu| \le 1.$$

$\square$

3.3.6 Upwind Methods

The advection equation has a direction, depending on the sign of the wave speed c. If $c > 0$, information comes from the left. If $c < 0$, information comes from the right. So: use 1-sided FD, depending on sign of c. This gives us "upwind methods": use FTBS for $c > 0$ and use FTFS for $c < 0$.

Both of these are first-order methods, $O(k+h)$. This seems like a backwards step, but it respects the properties of the exact solution better.

$$\begin{aligned} u_j^{n+1} &= \begin{cases} u_j^n - \nu(u_j^n - u_{j-1}^n), & \text{if } c > 0 \\ u_j^n - \nu(u_{j+1}^n - u_j^n), & \text{if } c < 0 \end{cases} \\ &= u_j^n - \tfrac{\nu}{2}(u_{j+1}^n - u_{j-1}^n) + \tfrac{\nu}{2}(\operatorname{sign} c)(u_{j+1}^n - 2u_j^n + u_{j-1}^n). \end{aligned}$$

Von Neumann analysis: FTBS stable for $0 < \nu < 1$ ($c > 0$ only). FTFS stable for $-1 < \nu < 0$ ($c < 0$ only).

3.3.7 Upwind Methods: Alternative derivation

The exact solution satisfies $u(x_j, t_n + k) = u(x_j - ck, t_n)$. If $0 < \frac{ck}{h} < 1$, then $ck > h$ so $x_j - ck \in (x_{j-1}, x_j)$. If we choose $\frac{ck}{h} = \nu = 1$, then $x_j - ck = x_j - h = x_{j-1}$. Thus $u_j^{n+1} = u_{j-1}^n$ exactly.

But this is unstable! (True for all methods we've seen: FTCS, LxF, LxW, UW.)

If $0 < \nu < 1$ then $x_j - ck$ lies in (x_{j-1}, x_j). We have numerical values for $u(x_j) \approx u_j^n$ and $u(x_{j-1}) \approx u_{j-1}^n$. So we can evaluate $u(x_j - ct, t_n)$ by interpolating numerical data!

At t_n we know u_j, u_{j-1} so we form the linear interpolant p_1:

$$\begin{aligned}
p_1(x) &= u_j^n + (x - x_j)\tfrac{1}{h}(u_j^n - u_{j-1}^n),\\
p_1(x_j - ck) &= u_j^n + (-ck)\tfrac{1}{h}(u_j^n - u_{j-1}^n),\\
&= u_j^n - \nu(u_j^n - u_{j-1}^n).
\end{aligned}$$

This is just the upwind method, for $c > 0$. Similarly if $c < 0$ we get $p_1(x_j - ck) = u_j^n - \nu(u_{j+1}^n - u_j^n)$ which is the UW method for $c < 0$.

If $\frac{ck}{h} > 1$ we would be extrapolating in a region outside (x_{j-1}, x_j). This would be bad.

If we use a quadratic interpolant p_2 based on $u_{j-1}^n, u_j^n, u_{j+1}^n$. This results in the LxW method.

Using p_2 based interpolation on $u_{j-2}^n, u_{j-1}^n, u_j^n$ gives the Beam-Warming equation (See: Assignment 3). This is a second order upwind method.

For a method in which u_j^{n+1} depends on $u_{j+p}^n, \dots, u_{j+q}^n$ $(p < q)$, we must have

$$\begin{aligned}
&x_j + ph = x_{j+p} \le x_j - ck \le x_{j+q} = x_j + qh,\\
&\qquad \implies \quad -q \le \underbrace{\tfrac{ck}{h}}_{=\,\nu} \le -p, \qquad \text{in order to be interpolating, not extrapolating.}
\end{aligned}$$

For example, LxW has $p = -1, q = 1$ so $-1 \le \nu \le 1$.

3.3.8 Upwind Methods: Domain of Dependence

For the exact solution: the domain of dependence of (X,T) is the point $X-cT$.

$$\begin{aligned} u_{tt} &= c^2 u_{xx} : \{X-ct, X+ct\}, \\ u_t &= Du_{xx} : \mathbb{R}. \end{aligned}$$

Numerical method also has a domain of dependence:

$$u_j^n \text{ depends on} \begin{cases} u_{j+p}^{n-1} & \cdots & u_{j+q}^{n-1}, \\ \vdots & & \vdots \\ u_{j+np}^{0} & \cdots & u_{j+nq}^{0}, \end{cases}$$

Define $T = nk$, then $nh = T\frac{h}{r}$ and $n = \frac{T}{r}$ (mesh ratio), where $r = k/h$. So the domain of dependence for u_j^n will be

$$(u((j+np)h,0), u((j+nq)h,0)) = \left(u^0\left(X+p\frac{T}{r}\right), u^0\left(X+q\frac{T}{r}\right)\right).$$

The numerical domain of dependence will fill the interval

$$\left(X+p\frac{T}{r}, X+q\frac{T}{r}\right).$$

For example LxW has $p=-1, q=1$ so domain of dependence is

$$\left(X-\frac{T}{r}, X+\frac{T}{r}\right).$$

UW $(c>0)$ has $p=-1, q=0$ so domain of dependence is

$$\left(X-\frac{T}{r}, X\right).$$

For the numerical solution to converge to exact solution as $k, h \to 0$, must have the domain of dependence $X-cT \subseteq$ numerical domain of dependence $[X+\frac{pT}{r}, X+\frac{qT}{r}]$. Otherwise, can change $u^0(X-cT)$ which should change $u(X,T)$ but that change doesn't affect numerical solution.

3.3.9 CFL Condition (Courant–Friedrichs–Lewy, 1928)

For the domain of dependence for UW we have:

$$\begin{aligned} & X - cT \subseteq [X + \tfrac{pT}{r}, X + \tfrac{qT}{r}], \\ \implies & \tfrac{pT}{r} \leq -cT \leq \tfrac{qT}{r}, \\ & -q \leq \underbrace{cr}_{=\,\nu} \leq -p, \qquad \text{as before!} \end{aligned}$$

This is a necessary (but not sufficient) condition for convergence, and hence stability. In summary:

$$\begin{array}{ccc} \text{LxF} & p=-1, q=1, & -1 \leq \nu \leq 1 \quad \text{i.e.} \quad |\nu| \leq 1, \\ \text{LxW} & \text{the same,} & \\ \text{UW}, c>0 & p=-1, q=0, & 0 \leq \nu \leq 1, \\ \text{UW}, c<0 & p=0, q=1, & -1 \leq \nu \leq 0. \end{array}$$

FTCS has $|\nu| \leq 1$ with this relation but is not stable for any ν.

For the heat equation on $[a,b] \subset \mathbb{R}$, the domain of dependence is $[a,b]$. For FTCS, u_j^{n+1} depends on $u_{j+1}^n, u_j^n, u_{j-1}^n$. Don't refine at constant $r = k/h$ but at constant $k/h^2 = \mu$.

u_j^n depends on $u_{j-n}^0, \ldots, u_{j+n}^0 \approx u(jh-nh, jh+nh) = u(X-Th/k, X+Th/k)$.

Refine at constant $\mu = \frac{k}{h^2} \to \frac{h}{k} = \frac{1}{\mu h}$. So the numerical domain of dependence is

$$\left(X - \frac{T}{\mu h}, X + \frac{T}{\mu h}\right).$$

BTCS/CN? Implicit methods so much harder...

$$u^{n+1} = (I - rA)^{-1} u^n \ldots = \left(I - \frac{1}{2} rA\right)^{-1} u^n \ldots$$

4 Tips for Giving a Good Talk

Handout: Top Ten Ways to Lose an Audience
SIAM News, Volume 44, Number 3, April 2001.

But what about positive things that you *should* do?

4.1 Content

- Motivation: what application was the method invented for? History.
- Why is it useful.
- Show how it works.
- Give examples (even if you don't code it up).
- Comparisons with other methods.
- Cite references, books.

4.2 Presentation

- Look over your slides: not too many words, no whole sentences, not too much info. The audience should be listening, not reading.
- Readability: think about your font size readable (large) and colour choices.
- Graphs: label your axes. Use big, thick lines.
- Don't learn a script, just use slides as a prompt.
- Don't look at computer or slides, look at your audience!

4.3 Time

- Don't talk too fast.
- Prepare!!

- Give one talk, not three separate talks. Keep style of slides consistent.
- Practice!!
- Make sure you run on time—not too quick, definitely not too long.
- Think in advance about what you might get rid of if you're running out of time, and/or have additional slides at the end for if you finish early.