Numerical Methods for MFC CDT

Ian Hawke Hilary Weller

2024-11-20

Table of contents

Introduction

This book introduces the concepts needed for numerical modelling with climate applications in mind. These are the notes for the Numerical Methods course for the Mathematics for our Future Climate CDT.

For more detail, the most precise comparator would be (Durran 2010) which covers numerical methods for Geophysics. For high accuracy methods including spectral and finite element approaches the books of (Hesthaven 2017), (Hughes 2012), (Trefethen 1996) and (Boyd 2001) are all important and useful. For high speed flows where discontinuities matter the books of (LeVeque 2002), (LeVeque 1992) and (Toro 2013) are key texts.

An aside on computers

The numerical methods we use and the design of the computers that we run the simulations on are closely linked. Changes in computer design (increases in memory, the move to parallel computing) can change which algorithms are most efficient. Sometimes computer architecture is specifically designed around particular algorithmic problems (such as graphical processing units - GPUs). Therefore it is important to have a loose understanding of how different parts of the algorithm interact with the computer architecture.

Key steps

Ultimately we want the computer to do an arithmetic calculation, which is then repeated many (many!) times. A typical calculation would update a single value on a single point of a grid, where there may be multiple values on each of the many (many!) grid points. A computer will do a single calculation incredibly quickly, in one *cycle*, typically less than a nanosecond. However, in order to do the calculation the computer needs to know the

values of the terms going in to the calculation. These will be stored in computer memory. If they are in *cache* memory then the values are "close" to the core doing the calculation and will be retrieved quickly (maybe 5-50 cycles). If they are in *main* memory then they will be retrieved more slowly (maybe 1,000 cycles). Main memory is much larger than cache memory, so only small calculations can use just the cache.

If we want to do extremely large calculations the the data will not fit in main memory of a single core or node. In this case we use parallel computing where the calculation is spread between different parts of a larger computer. In splitting the data this way we introduce artificial boundaries into the calculation, whose values need providing from somewhere. Communicating this data over the network is even slower again, taking maybe 100,000 cycles. The more parallel computing is needed, or the more values used in updating a single grid point, the more the communication cost increases.

Finally, we may want to store data permanently on disk, or read complex data from a large table on disk. This is slower again, taking at least 1,000,000 cycles or more for large amounts of data. Careful choices of which data is stored and how can significantly improve the efficiency of a code.

More precise numbers on the efficiency and latency of computational operations can be found online. However, for numerical methods it is one of many concerns that need balancing. This course covers a range of numerical methods, from simple finite difference methods, more complex finite element methods designed to work on complex domains, and the high accuracy spectral methods. Whilst the low accuracy of finite differences compared to spectral methods may make them seem like a poor choice in theory, their simplicity can make it easier to use them efficiently, as the limited amount of data needed improves cache locality and reduces communication. Most production codes are the result of careful consideration of the trade-offs, and practical measurement of code efficiency.

Part I

Overview

1 Climate models

1.1 Navier-Stokes Equations

The standard models of gases and fluids rely on the *Navier-Stokes* equations. These express the conservation of energy, momentum, and particle number of the material coupled to gravity and allowing for sources of energy and diffusion. The form we will use can be written

$$\begin{aligned}
\text{The Lagrangian derivative} \qquad & \frac{D\Psi}{Dt} = \frac{\partial \Psi}{\partial t} + \mathbf{u} \cdot \nabla \Psi \\
\text{Momentum} \qquad & \frac{D\mathbf{u}}{Dt} = -2\boldsymbol{\Omega} \times \mathbf{u} - \frac{\nabla p}{\rho} + \mathbf{g} + \\
& \qquad \mu_u \left(\nabla^2 \mathbf{u} + \frac{1}{3} \nabla (\nabla \cdot \mathbf{u}) \right) \qquad (1.1) \\
\text{Continuity} \qquad & \frac{D\rho}{Dt} + \rho \nabla \cdot \mathbf{u} = 0 \\
\text{Energy} \qquad & \frac{D\theta}{Dt} = Q + \mu_\theta \nabla^2 \theta
\end{aligned}$$

To close the system we need to link the microphysical behaviour of the particles making up the material to the forces they impose, through an equation of state. An example would be the perfect gas law $p = \rho R T$.

The various symbols defined above are outlined in the following table.

Symbol	Meaning
$\mathbf{u}$	Wind vector
t	Time
$\boldsymbol{\Omega}$	Rotation rate of planet
ρ	Density of air

Symbol	Meaning
p	Atmospheric pressure
$\mathbf{g}$	Gravity vector (downwards)
θ	Potential temperature, $T(p_0/p)^\kappa$
κ	heat capacity ratio $\simeq 1.4$
Q	Source of heat
μ_u, μ_θ	Diffusion coefficients

1.2 Shallow Water Equations

The Navier-Stokes equations are complex and accurately captures physics (such as acoustic waves) that has minimal impact on climate models, and whose numerical solution would impose problematic constraints on the cost and accuracy of the approximation. One simpler model is the *shallow water equations* (SWE). The assumptions needed to derive the SWE are

- Horizontal length scale $\gg$ vertical length scale;
- Very small vertical velocities.

To get the SWE, take the Navier-Stokes equations over orography and depth integrate. This gives the system

$$\begin{aligned} \frac{D\mathbf{u}}{Dt} &= -\mathbf{\Omega} \times \mathbf{u} - g\nabla(h + h_0) + \mu_u \nabla^2 \mathbf{u}, \\ \frac{Dh}{Dt} + h\nabla \cdot \mathbf{u} &= 0. \end{aligned} \tag{1.2}$$

In the SWE the terms are

$\mathbf{u}$	Depth integrated wind vector
t	Time
$\mathbf{\Omega}$	Rotation rate of planet
h	Fluid depth
g	Acceleration due to gravity (in the direction of the depth integration)

∇	Gradients in the horizontal directions
h_0	height of the bottom topography with respect to a reference point
μ_u	Diffusion of momentum

Exercise 1.1. Considering the meaning of the terms in the momentum equation in (Equation 1.1), what are the meanings of the terms of the momentum equation of the SWE?

1.3 Linearised Shallow Water Equations

The Shallow Water Equations are simpler than the Navier-Stokes equations but are still nonlinear. Dropping the nonlinear terms is an over-simplification for real models, but can be useful when developing methods. To linearise the SWE we assume that

- $\mathbf{u} = (u, v, 0)^T$ is small;
- $2\boldsymbol{\Omega} = (0, 0, f)^T$;
- $h = H + h'$, where H is uniform in space and time and h' is small;
- the product of two small variables is ignored (even if one or both are inside a differential);
- h_0 and μ_u are ignored.

This gives the linearised equations for u, v, and h', expressed in terms of f (rather than $\boldsymbol{\Omega}$), as

$$\begin{aligned} \frac{\partial u}{\partial t} &= fv - g\frac{\partial h'}{\partial x}, \\ \frac{\partial v}{\partial t} &= -fu - g\frac{\partial h'}{\partial y}, \\ \frac{\partial h'}{\partial t} &= -H\left(\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y}\right). \end{aligned} \tag{1.3}$$

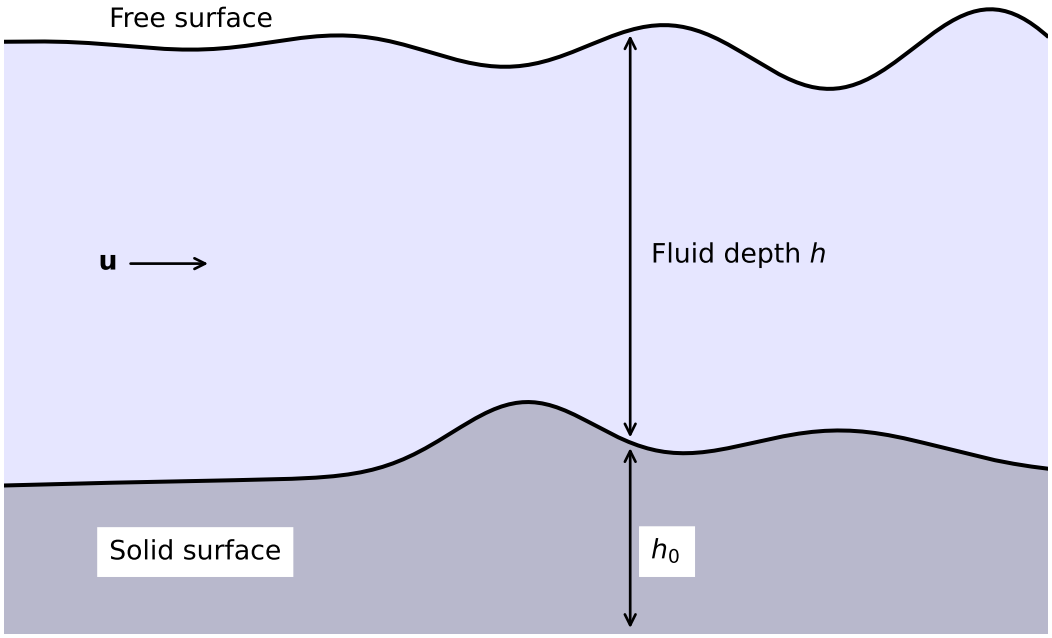

Figure 1.1: A sketch of the vertical fluid direction which is integrated out in the derivation of the shallow water equations.

1.4 Advection

If the wind vector **u** is known (through, for example, the solution of the Navier-Stokes or Shallow Water equations) then we can solve for the transport of a property or particle that is moved by the fluid material. A standard example in climate modelling is pollution, where the concentration of the pollutant is represented by the scalar field Ψ. The pollutant then obeys the *advection equation*

$$\underbrace{\frac{D\Psi}{Dt}}_{(1)} = \underbrace{\frac{\partial\Psi}{\partial t}}_{(2)} + \underbrace{\mathbf{u}\cdot\nabla\Psi}_{(3)} = \underbrace{S}_{(4)} + \underbrace{\mu_\Psi\nabla^2\Psi}_{(5)}\,. \tag{1.4}$$

The terms here are

1. the Lagrangian derivative of the pollution concentration,
2. the rate of change of the pollution concentration at a fixed point in space,
3. the advection of the pollution concentration by the wind velocity,
4. the source or sink of pollution concentration,
5. the diffusion of the pollution concentration.

Frequently when we refer to the linear advection equation we mean the case with no source or sink of pollution, nor any diffusion. In this case we have

$$\frac{D\Psi}{Dt} = \frac{\partial\Psi}{\partial t} + \mathbf{u}\cdot\nabla\Psi = 0. \tag{1.5}$$

We typically consider the wind velocity **u** to be a fixed function of time, and sometimes simplify further to make it spatially constant. Note now that if $\nabla\cdot\mathbf{u} = 0$ then the linear advection equation can be written in *flux form*

$$\frac{\partial\Psi}{\partial t} + \nabla\cdot(\Psi\mathbf{u}) = 0. \tag{1.6}$$

This is a special case of an equation in *conservation law form*,

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

for which special numerical methods can be constructed.

Part II

Spectral

2 Spectral methods

In previous chapters we have discussed the stability and accuracy of finite difference methods by looking at the impact in frequency space. This can be seen either as a Fourier transform, or as a function basis expansion as, for example, a complex Fourier series.

We could move from numerically solving the differential equation using finite differences to instead numerically solving the function basis expansion. This chapter outlines the key issues behind that approach, which is broadly called a *spectral method*.

2.1 Advection expansions

As usual we will start with the one dimensional linear advection equation (Equation 1.5) in the form

$$\frac{\partial \phi}{\partial t} + u\frac{\partial \phi}{\partial x} = 0. \tag{2.1}$$

We assume the boundaries are periodic. We then approximate the solution as a truncated complex Fourier series

$$\phi(x,t) = \sum_{k=-N}^{N} a_k(t)\,\mathrm{exp}(ikx)\,. \tag{2.2}$$

The time dependence is completely contains within the coefficients $a_n(t)$, which determine the solution. The number of modes used in the approximation, $2N+1$, controls the accuracy of the method, and can be thought to be like the number of grid points in a finite difference method.

The series expansion substituted into the advection equation (Equation 2.1) gives

$$\sum_{k=-N}^{N} \left\{ \frac{\mathrm{d}a_k}{\mathrm{d}t} + iku a_k \right\} \exp(ikx) = 0. \tag{2.3}$$

In the simplest case where u is a constant, the orthogonality of the complex exponentials decouples all the modes to leave the ordinary differential equations

$$\frac{\mathrm{d}a_k}{\mathrm{d}t} + iku a_k = 0. \tag{2.4}$$

This can be solved explicitly and exactly giving $a_k = a_k(0) \exp(-ikut)$. We therefore have the approximate solution

$$\phi(x,t) = \sum_{k=-N}^{N} a_k(0) \exp[ik(x-ut)]\,. \tag{2.5}$$

This solution has *no* dispersion error (all information propagates exactly at speed u) and its amplitude is constant. The error comes from projecting the exact initial data onto the basis functions, using a truncated series. This can have odd effects when N is small - strictly positive functions can appear to go negative - but for smooth data these problems converge very rapidly (as Fourier series approximations converge rapidly).

2.2 Non-uniform advection

With finite difference and finite volume methods moving to non-uniform and nonlinear problems is "straightforward", although their numerical analysis can be difficult. For example, if the advection velocity u is not constant so that $u = u(x)$, the standard FTBS method (as seen earlier) is directly written as

$$\phi_j^{n+1} = \phi_j^n - \frac{u_j \Delta t}{\Delta x} \left(\phi_j^n - \phi_{j-1}^n \right)\,. \tag{2.6}$$

The only change to the standard FTBS scheme is that the advection velocity is now evaluated at the update point, $u_j = u(x_j)$. The CFL limit constraining the timestep now needs to maximize over all points x_j, or equivalently over all advection velocities u_j.

The spectral method, however, now becomes markedly more complex. As u varies in space we need to represent it by Fourier series expansion as well, as (for example)

$$u(x) = \sum_{l=-N}^{N} u_l \exp(ilx)\,. \tag{2.7}$$

When we substitute the series expansions for both the solution and the advection velocity into the advection equation we get

$$\sum_{k=-N}^{N} \left\{ \frac{\mathrm{d}a_k}{\mathrm{d}t} + ika_k \sum_{l=-N}^{N} u_l \exp(ilx) \right\} \exp(ikx) = 0. \tag{2.8}$$

When we use orthogonality we find

$$\frac{\mathrm{d}a_k}{\mathrm{d}t} + i \sum_{\substack{k+l=N \\ |k|,|l|\leq N}} ka_k u_l = 0\,. \tag{2.9}$$

This couples different modes (different values of k). We can no longer solve this ordinary differential equation exactly; instead we have to use some time differencing method to compute the result.

The results of applying a spectral method to non-constant advection can be seen in Figure 2.1. The left panel shows the solution after "half a period", which is extremely well captured even with very few modes used (look at the solution near the right edge for the largest discrepancies). The convergence plot in the right panel is the key result, however. It shows how the error converges *exponentially*, which is far faster than any finite differencing scheme (which converges *polynomially*) can manage.

2.3 Problems and solutions

The fundamental lesson of spectral methods for complex, nonlinear systems is given by Equation 2.9. That is, the approximate solution is updated by coupling every mode of the solution at the current time. This leads to extremely high accuracy but is associated with

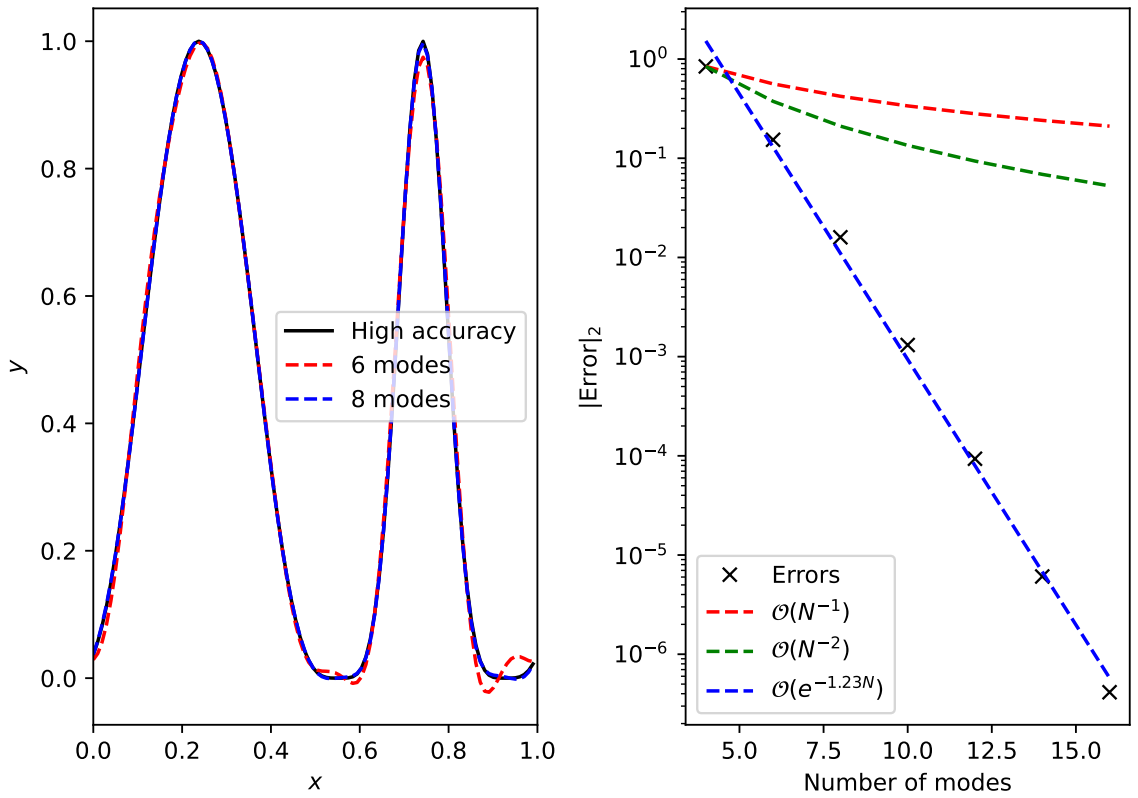

Figure 2.1: A spectral method advects the initial data $\sin^4(2\pi x)$ around the domain $x \in [0, 1]$ up to $t = 0.5$. Fourier modes $c_n = -N, \ldots, N$ are used. Spectral (exponential) convergence with the number of modes is seen. It is also clear that this is much faster than first or second order convergence as indicated on the right panel.

high costs. In particular, there are $\mathcal{O}(N)$ ordinary differential equations of the form (Equation 2.9) that need to be solved, and each requires $\mathcal{O}(N)$ terms to be evaluated within the sum, so each timestep costs $\mathcal{O}(N^2)$. For large numbers of coefficients this is much more expensive that a finite difference scheme.

Some of these costs can be ameliorated by performing the nonlinear (or non-uniform) calculations in position space rather than frequency space. That is, the terms in the product $u\partial_x\phi$ can be Fourier transformed back to position space, multiplied, and then transformed back. This is cheaper, reducing the cost to $\mathcal{O}(N \log N)$. This idea is linked to *spectral collocation* methods.

A related problem caused by all modes coupling is that of accuracy. When a mode with wavenumber N couples to another mode with wavenumber N then the true result should be a non-trivial evolution of the mode with wavenumber $2N$. However, our truncated series approximation does not allow for this information to be captured. The information lost due to the nonlinear interactions at high wavenumber can have significant impacts, and particular techniques are needed to adjust for it. One standard method is to only use $2/3$ of the coefficients when calculating the update terms. This (additionally) truncated approximation avoids aliasing problems due to the interactions.

Another, sometimes critical, problem, is related to the use of a single function basis expansion over the entire domain. In the example above we used a Fourier series on a periodic domain to represent the function ϕ. However, when the function ϕ is discontinuous, it is well known that the Fourier series suffers from *Gibbs oscillations*. That is, the truncated Fourier series representation will oscillate (over- and under-shoot) around the discontinuity, and that the maximum error will not converge as the number of terms in the series increases. There is a lot of literature trying to make spectral methods work with discontinuous solutions, but in general it is not worth the effort: finite volume methods are more efficient for these problems.

Whilst Durran (2010) gives an excellent introduction to spectral-type methods specifically for climate modelling, Boyd (2001) is the indespensable reference for spectral methods in general.

Part III

Finite Elements

3 Finite Elements

We have seen that approximating a PDE using finite differences is straightforward (which both helps a human to implement it, and a computer to solve it very efficiently). However, we have also seen that the errors introduced can be both larger than we want and difficult to control, introducing unphysical effects (incorrect wave propagation, loss of monotonicity, loss of positivity, etc).

We also saw that high accuracy can be maintained by using spectral methods, linked to an approximation of the unknowns in terms of a series expansion of known functions (such as a Fourier series). However, these have problems near steep gradients (spectral ringing or Gibbs oscillations) which can again produce unphysical effects. They also couple every data point in the domain, which makes them less straightforward and more computationally expensive.

Finally, we saw through finite volume methods how general unstructured grids could be used, and how a suitable choice of how to represent the function some key features (conservation of mass, monotonicity) can be preserved. These schemes are more complex but still only couple a limited number of points. The methods introduced so far have relatively low orders of accuracy.

Our aim here is to discuss *finite element* methods. These have the flexibility of finite volume methods whilst (in principle) both allowing for high order (even spectral) accuracy and also allowing for key physical properties being maintained. They are necessarilly more complex, so we need to spend more time discussing the background theory of the mathematics and the software engineering of the implementation.

In a finite element method the domain is split into *elements* which are (in most ways) indistinguishable from finite volume cells. In finite element methods there is no expectation that the elements and their edges and nodes which bound the elements have any structure to them.

The notation in this section largely follows (Hughes 2012).

3.1 One dimension, time independent

Take the advection-diffusion equation describing the motion of pollution concentration with a source of pollution, (Equation 1.4),

$$\frac{D\Psi}{Dt} = \frac{\partial \Psi}{\partial t} + \mathbf{u} \cdot \nabla \Psi = S + \mu_\Psi \nabla^2 \Psi. \tag{3.1}$$

For now we will restrict to one spatial dimension and look for the *steady state* solution where the pollution generated by the source S is balanced by the diffusion term with coefficient $\mu_\Psi = \mu$, assuming that the wind velocity vanishes. We will also absorb the value of the diffusion coefficient μ into the source S. Therefore the equation to solve is

$$0 = S + \partial_{xx} \Psi. \tag{3.2}$$

To be concrete we will assume that the domain on which we are solving is $x \in [0, 1]$, that the amount of pollution at the left boundary is fixed, and that the flux of pollution at the right boundary is also fixed,

$$\Psi(0) = \alpha, \quad \partial_x \Psi|_{x=1} = \beta. \tag{3.3}$$

3.2 Boundary conditions

As finite element methods are designed to work on complex domains with complex boundaries, the boundary conditions are built in at the mathematical level. We need to consider the different types separately.

We remove the Dirichlet boundary condition (here at $x = 0$) by writing

$$\Psi(x) = \psi(x) + q(x), \tag{3.4}$$

where $q(x)$ is a *known* function chosen so that $q(0) = \alpha$. That means that $\psi(0) = 0$, and ψ satisfies homoegeneous boundary conditions. We will solve for ψ and then put the boundary condition back in later. This can either be done globally (by making q a constant,

or non-zero eveywhere), or locally (by making q non-zero only in a small region). The local approach is standard.

Neumann boundaries, however, are built into the way the method works.

3.3 Weak form

Now we want to remove the second derivatives from the problem, as it is easier to reason about first derivatives alone. When working with finite volumes we saw that we could remove derivatives by integrating over the domain. However, we then ended up working with volume averaged quantities. To keep our steps more general, we first multiply by an *arbitrary, smooth* function $w(x)$, and then integrate over the domain. As we already know the value of the solution at the left boundary due to the Dirichlet boundary condition, we can weight its value there to zero by enforcing $w(0) = 0$.

The function $w(x)$ is referred to as the *weighting* function. Using integration by parts the steady state advection diffusion equation becomes

$$\begin{aligned} 0 = [w(x)\partial_x \Psi(x)]_0^1 - \int_0^1 \partial_x \Psi(x) \partial_x w(x) \, \mathrm{d}x \\ + \int_0^1 w(x) S(x) \, \mathrm{d}x. \end{aligned} \tag{3.5}$$

We introduce the "inner product" notation

$$(f, g) = \int_0^1 f(x) g(x) \, \mathrm{d}x \tag{3.6}$$

and use the boundary conditions to give

$$(\partial_x \psi, \partial_x w) = w(1)\beta - (\partial_x q, \partial_x w) + (w, S). \tag{3.7}$$

This is the *weak form* of the equations. It is written in this fashion as the unknown term ($\psi(x)$) is on the left hand side, but all terms on the right are either known (β, q, S) or

arbitrary (w). It can be proved that solutions of the *strong form* in (Equation 3.2) are also solutions of (Equation 3.7).

3.4 Function representation

In finite volume methods the domain is split into cells, or volumes, within which Ψ is presented by a single number (its volume average). In finite element methods the domain is split into elements, which in many ways are indistinguishable from volumes, within which Ψ and any other function is represented in terms of a series expansion. For example, we could choose within each element to represent Ψ as a (truncated) Fourier series, or Taylor series.

However, for practical purposes, we want to link the representations in neighbouring elements, but decouple the representations in elements that are not neighbours. To do this, we introduce *shape* or *basis* functions which are associated with the nodes of the grid.

To make this concrete, take our domain $x \in [0, 1]$ and split it into two elements $I_0 = [0, \frac{1}{2}]$ and $I_1 = [\frac{1}{2}, 1]$. The boundaries of the elemnts give us the three nodes $\{x_A\} = \{0, \frac{1}{2}, 1\}$. Here $_A$ is a label - an integer labelling the nodes - which we count from 0 (so $A \in \{0, 1, 2\}$). We then write the function of interest, ψ, as

$$\psi(x) = \sum_A \psi_A N_A(x), \tag{3.8}$$

where $N_A(x)$ are the shape functions.

We choose $N_A(x)$ to take the value 1 at node x_A and take the value 0 at any other node. This immediately means that $\psi_A = \psi(x_A)$. Therefore the nodal values behave much like a finite difference representation.

We immediately note that our (approximate) solution process must compute, somehow, the values of ψ_A. Some are already known: the boundary condition at $x = 0$ in our case immediately implies that $\psi_0 = 0$. The other values must be fixed by the solution of (Equation 3.7).

There are now many choices we can make to fix the shape functions. The simplest is to choose the shape functions to be piecewise linear. This gives

$$
\begin{aligned}
N_0(x) &= \begin{cases} 1-2x & 0 \le x \le 1/2 \\ 0 & 1/2 \le x \le 1 \end{cases} \\
N_1(x) &= \begin{cases} 2x & 0 \le x \le 1/2 \\ 2-2x & 1/2 \le x \le 1 \end{cases} \\
N_2(x) &= \begin{cases} 0 & 0 \le x \le 1/2 \\ 2x-1 & 1/2 \le x \le 1 \end{cases}
\end{aligned}
\tag{3.9}
$$

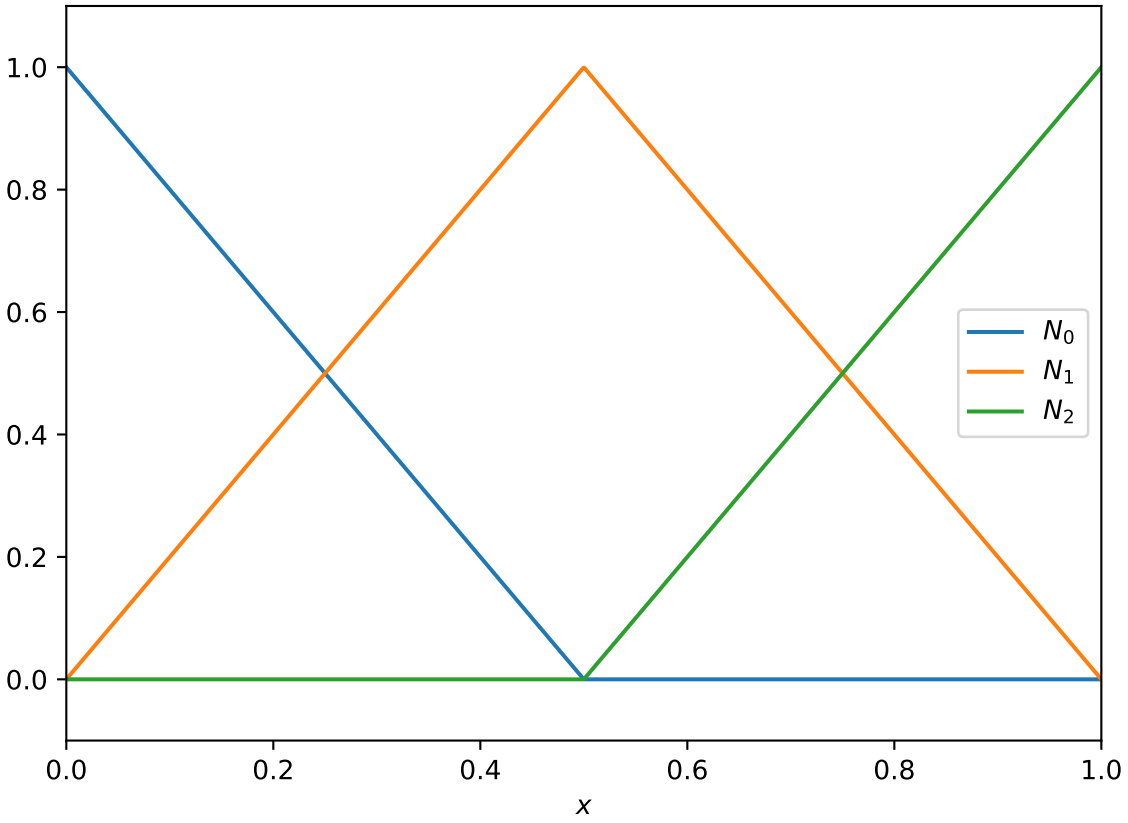

Figure 3.1: Shape functions for a domain with two elements (hence three nodes).

We now make the *Galerkin* assumption that the same function basis expansion is used for the unknown function ψ and also for the test function w. We write out w and q using the same shape functions. As noted above we enforce that q drops immediately to zero away from the boundary, which means we write

$$q(x) = \alpha N_0(x). \tag{3.10}$$

This means the weak form (Equation 3.7) becomes

$$\begin{aligned} \sum_B w_B \sum_A \psi_A(\partial_x N_A, \partial_x N_B) = w_{N_{\text{elements}}}\beta - \\ \alpha \sum_B w_B(\partial_x N_0, \partial_x N_B) + \sum_B w_B(N_B, S). \end{aligned} \tag{3.11}$$

This has to be true for any choice of weight function w, so for any choice of the coefficients w_B. We gather terms as

$$\sum_B w_B \left\{ \sum_A K_{AB}\psi_A - F_B \right\} = 0. \tag{3.12}$$

To hold for any choice of weight function the term in curly brackets must vanish. Here the *stiffness matrix* K and *force vector* **F** are independent of ψ. The steps here are very similar to those in the implicit finite difference methods such as BTCS. This gives

$$K\boldsymbol{\psi} = \mathbf{F}, \tag{3.13}$$

where the coefficients of the stiffness matrix K are given by

$$K_{AB} = \int_0^1 \partial_x N_A(x)\partial_x N_B(x)\,\mathrm{d}x \tag{3.14}$$

and the coefficients of the *force vector* **F** are given by

$$F_B = \beta\delta_B^{N_{\text{elements}}} + \int_0^1 N_B(x)S(x)\,\mathrm{d}x - \alpha\int_0^1 \partial_x N_0(x)\partial_x N_B(x)\,\mathrm{d}x. \tag{3.15}$$

Here δ_B^C is the Kronecker delta (zero except when $B \equiv C$, where it is one) and encodes the Neumann boundary value.

Note that the final term in the force vector (which results from the Dirichlet boundary condition) has a very similar form to the entries of the stiffness matrix. However, the shape

function N_0 is non-zero at the boundary of the domain, which can cause issues with directly using the stiffness matrix here. For that, and other reasons, it is best to use the element viewpoint below.

Exercise 3.1.

1. Compute (analytically) the coefficients of the stiffness matrix given the shape functions above in (Equation 3.9).
2. Compute (analytically) the coefficients of the force vector when $S(x) = 1 - x$.
3. Solve (numerically) for ψ_A and plot the resulting solution for $\psi(x)$. In the simplified case $\alpha = 0 = \beta$ we can choose $q(x) \equiv 0$, so that $\Psi = \psi$. Compare against the exact solution $\Psi(x) = x(x^2 - 3x + 3)/6$.

3.5 The element viewpoint

This works surprisingly well given the small number of elements and associated nodes. However, as the sources get more complex we will need to work with more elements to improve accuracy. We need to go about this more systematically.

First we note that in any element there are only two shape functions that are not zero. In the notation we have used so far element $I_A = [x_A, x_{A+1}]$ and the two non-zero shape functions have been N_A and N_{A+1}.

Second, we note that in general the elements will have different sizes. In higher dimensions this gets ever more complicated. However, by using a coordinate transformation, we can take any element from the interval $[x_A, x_{A+1}]$ to the interval $\xi \in [-1, 1]$. We have

$$\begin{aligned} \xi(x) &= \frac{2x - x_A - x_{A+1}}{x_{A+1} - x_A}, \\ x(\xi) &= \frac{(x_{A+1} - x_A)\xi + x_A + x_{A+1}}{2}. \end{aligned} \tag{3.16}$$

We can now write the two non-zero shape functions in terms of the *reference coordinates* ξ as

$$N_a(\xi) = \tfrac{1}{2}(1 + \xi_a \xi), \quad a = 1, 2. \tag{3.17}$$

The label a is labelling the shape functions within the reference element, written in terms of the reference coordinates. These labels can be linked back to the original shape functions in the end.

Now, we remember that the weak form is written in terms of the stiffness matrix and force vector, and these depend on integrals of the shape functions and their derivatives. Computing the derivatives in the reference coordinates is straightforward,

$$\partial_\xi N_a = \frac{(-1)^a}{2}. \tag{3.18}$$

To map this back to derivatives in the original coordinates we require a Jacobian, which needs the derivatives of the coordinate transformation. This needs

$$\begin{aligned} \partial_x \xi &= \frac{2}{x_{A+1} - x_A}, \\ \partial_\xi x &= \frac{x_{A+1} - x_A}{2}. \end{aligned} \tag{3.19}$$

We can now compute the contribution that one single element $e = I_A$ makes. This is

$$\begin{aligned} k^e_{ab} &= \int_{x_A}^{x_{A+1}} \partial_x N_a \partial_x N_b \,\mathrm{d}x \\ &= \int_{-1}^{1} \partial_\xi x \partial_x N_a \partial_x N_b \,\mathrm{d}\xi \\ &= \int_{-1}^{1} \left(\partial_\xi x\right)^{-1} \partial_\xi N_a \partial_\xi N_b \,\mathrm{d}\xi \\ &= \frac{(-1)^{(a+b)}}{x_{A+1} - x_A}. \end{aligned} \tag{3.20}$$

This is incredibly useful: there's no need to do any integrals at all. Note that this gives a 2×2 matrix corresponding to a single element: to get the complete stiffness matrix we need to "add all these up".

The element force vector is, in general, more complex, as it involves an integral over the complex source S. However, we can approximate this by writing the source in terms of its values at the nodes as well, so

$$S(x) = \sum_a S_a N_a(x), \tag{3.21}$$

giving $S_a = S(x(\xi_a))$. We can then compute, for the simple shape functions we use here,

$$f_a^e = \frac{x_{A+1} - x_A}{6} \begin{pmatrix} 2S_1 + S_2 \\ S_1 + 2S_2 \end{pmatrix} . \tag{3.22}$$

Finally, we need to include the boundary condition terms in the force vector. To include the Neumann boundary condition at the right boundary we adjust the final entry,

$$F_{N_{\text{elements}}} \to F_{N_{\text{elements}}} + \beta. \tag{3.23}$$

To include the Dirichlet boundary condition at the left boundary we adjust the first entry, which needs adjusting using $\int \partial_x N_0 \partial_x N_B$. For the linear shape functions chosen here this is only non-zero within the first element, so we can use the *local* stiffness matrix to adjust the first entry,

$$F_0 \to F_0 - \alpha k_{12}^0. \tag{3.24}$$

(Note that the numbering here has the first element number $e = 0$ and the local labels $a \in \{1, 2\}$)

3.5.1 Neumann boundary conditions

The correction term for Neumann boundary conditions is the easier case to check. Assume that x_C is the node on the Neumann obundary (above $x_C = x_{N_e} = 1$), and that $\partial_x \psi(x_{N_e}) = \beta$. Then we have

$$\begin{aligned}\int_0^1 w\partial_{xx}\psi &= [w\partial_x\psi]_0^1 - \int_0^1 \partial_x w\partial_x\psi \\ &= w(1)\beta - (w\partial_x\psi)(0) - \int_0^1 \partial_x w\partial_x\psi \\ &= -\int_0^1 wS\,.\end{aligned}$$

Using the standard function basis expansion we have

$$\begin{aligned} & w_{N_e}\beta - w_0\psi_0(\partial_x N_0)(0) - \sum_B w_B \sum_A \psi_A \int_0^1 \partial_x N_B \partial_x N_A = -\sum_B w_B \int_0^1 N_B S \\ \implies \qquad & K_{AB}\psi A = F_B\,, \\ & F_{N_e} = \int_0^1 N_{N_e} S - \beta\,. \end{aligned}$$

That is, the standard force vector term on the Neumann boundary is corrected by the value of the Neumann boundary condition. We also note that the value at the left boundary is linked to w_0 and hence to ψ_0, which is not included in the vector to be solved (as it lies on a Dirichlet boundary).

3.5.2 Dirichlet boundary conditions

The correction term for Dirichlet boundary conditions may not be obvious. This is a bit more annoying to construct as the solution is not computed at the boundary node where the Dirichlet condition holds. That is because each node we have to compute increases the size of the linear system, making it harder and more expensive to solve. Instead we want to incorporate the effect of the boundary condition at a given node into the system through its impact on its neighbours.

We do this by a standard trick seen in the solution of PDEs with inhomogeneous boundary conditions: change variable to a variable that satisfies a similar PDE but with *homogeneous* boundary conditions. We can do this in many ways, but in finite elements, when using shape functions, there is one particularly neat choice.

Assume that x_C is the node on the Dirichlet boundary (above $x_C = x_0 = 0$), and that $\psi(x_C) = \psi_C = \alpha$. Choose $\phi(x) = \psi(x) - \alpha N_C(x)$. As N_C is an indicator function (so is one at the node x_C and zero at all other nodes) we have that $\phi(x_C) = \phi_C = 0$ and $\phi(x_A) = \psi(x_A)$ on all nodes not on that particular Dirichlet boundary. Therefore ϕ satisfies a homogeneous Dirichlet boundary condition, and matches the solution that we want at all interior points. It also does not interfere with the other boundary condition (unless we are working with a single element!).

We need to check the impact this has on the discrete equation. Focus on $x_C = x_0 = 0$; the argument is identical for other cases. We have

$$\begin{aligned}
\int_0^1 w\partial_{xx}\psi &= \int_0^1 w\partial_{xx}\phi + \alpha\int_0^1 w\partial_{xx}N_0 \\
&= \left[w\partial_x\phi\right]_0^1 - \int_0^1 \partial_x w\partial_x\phi + \\
&\quad \alpha\left[w\partial_x N_0\right]_0^1 - \alpha\int_0^1 \partial_x w\partial_x N_0 \\
&= -\int_0^1 wS\,.
\end{aligned}$$

Plugging in the usual basis function expansions and moving the terms to the conventional places we find

$$\begin{aligned}
\sum_B w_B \sum_A \phi_A K_{AB} = \sum_B w_B \Bigg\{ &\int_0^1 N_B S - \\
&- \left[N_B \sum_A \phi_A \partial_x N_A\right](0) + \alpha\left[N_B\partial_x N_0\right](0) - \\
&- \alpha\int_0^1 \partial_x N_B \partial_x N_0 \Bigg\}\,.
\end{aligned}$$

We see that the first line gives the standard $K\boldsymbol{\phi} = \mathbf{F}$ form. The last line is precisely the correction using the element K_{B0} of the stiffness matrix. In principle this correction is applied to every element of the force vector; however, only when N_B and N_0 overlap (that is, only when x_B is next to the Dirichlet boundary node x_0) is the overlap integral non-zero.

The second line cancels identically. This is because the terms are all evaluated at $x = 0$. The only non-zero contributions can come from the case $B = 0$ (as otherwise the undifferentiated indicator function N_B vanishes), which would modify a term that is not included in the force vector.

Hence we get the expected result: if element e contains a (global) node A that lies on a Dirichlet boundary, then the force vector at all other nodes B within the element must be corrected by $-\psi(x_A)K_{AB}$.

3.6 Linking elements to equations

Our goal is to construct a linear (matrix) equation to give us the solution ψ_A at all nodes A where it isn't enforced by the boundary conditions (which, in the example so far, is all nodes except the left-hand boundary). We note that each *interior* node is linked to two elements, so contributions from the element matrix will affect more than one equation.

To keep track of this, we construct the *location matrix* or *location array* LM which, given the node number $a \in \{1, 2\}$ and the element number e returns the associated equation number.

Any node that is not to be included (as its value is given by a Dirichlet boundary condition) has its associated equation number A set to -1. The first node that must be included is given value 0. We then go element-by-element: the left-hand node of element e is the same as the right-hand node of element $e-1$, so picks up the same equation number. The right-hand node of element e, if considered, then has equation number one higher than the left-hand node of that element. This translates directly into Python code:

```
N_elements = 4 # for example

LM = np.zeros((2, N_elements), dtype=np.int64)
for e in range(N_elements):
    if e==0:
        # Treat first element differently due to BC
        LM[0, e] = -1 # Left hand node of first element
                      # is not considered thanks to BC.
        LM[1, e] = 0 # the first equation
```

```
    else:
        # Left node of this element is
        # right node of previous element
        LM[0, e] = LM[1, e-1]
        LM[1, e] = LM[0, e] + 1
```

Now the *global* stiffness matrix and force vector can be assembled: for each element e we construct the element k^e_{ab} and f^e_b and add the appropriate components, as

$$\begin{aligned} K_{LM(a,e)\,LM(b,e)} &= K_{LM(a,e)\,LM(b,e)} + k^e_{ab} \quad a, b \in \{1, 2\}, \\ f_{LM(b,e)} &= f_{LM(b,e)} + f^e_b \qquad\qquad\qquad b \in \{1, 2\}. \end{aligned}$$

Note that we need one more structure to keep track of the boundary conditions. As noted above, if a node is on a boundary then the value of the force vector needs modifying, either by including its value directly (in the case of a Neumann boundary) or by using some appropriate multiple of the local stiffness matrix (in the case of a Dirichlet boundary). This structure must map the node number to the value in the boundary condition; the location matrix can be used to check the boundary condition type.

3.7 Algorithm

This gives our full algorithm:

1. Set the number of elements N_{elements}.
2. Set node locations x_A, where $A = 0, \ldots, N_{\text{elements}}$.
3. Set up the location matrix LM.
4. Set up a boundary value structure (in Python a dictionary would work).
5. Set up arrays, initially all zero, for the global stiffness matrix (size $N_{\text{elements}} \times N_{\text{elements}}$) and for vector (size N_{elements}).
6. For each element:
    1. Form the element stiffness matrix k^e_{ab}.
    2. Form the element force vector f^e_b.
    3. Add the contributions to the global stiffness matrix and force vector.
    4. Modify using the boundary values if needed.

7. Solve $K\boldsymbol{\psi} = \mathbf{F}$.

Exercise 3.2. Write a finite element solver for the problem above, as a function that takes as input the number of elements and the source function S, as well as the boundary conditions α, β. It should use a uniformly spaced grid, and return the nodes x_A and the solution at the nodes $\Psi_A = \psi_A + q_A$.

Check that the function returns the same result as above when used with two elements and $S(x) = 1 - x$.

The apply the solver to the case $S(x) = (1-x)^2$ with exact solution $\Psi(x) = x(4 - 6x + 4x^2 - x^3)/12$. Compute the 2-norm of the error and check how it converges with resolution.

Finally check that the solver works on the case

$$S(x) = \begin{cases} 1 & |x - \frac{1}{2}| < \frac{1}{4}, \\ 0 & \text{otherwise} \end{cases} \tag{3.25}$$

with boundary conditions

$$\alpha = \Psi(0) = 0.1, \quad \beta = \partial_x \Psi(1) = -0.2. \tag{3.26}$$

The exact solution in this case is

$$\Psi = \begin{cases} 0.3x + 0.1 & x < \frac{1}{4} \\ -\frac{1}{2}x^2 + 0.55x + \frac{11}{160} & \frac{1}{4} < x < \frac{3}{4} \\ -0.2x + 0.35 & x > \frac{3}{4} \end{cases}. \tag{3.27}$$

> **Tip**
>
> It will be useful for later purposes to write helper functions that compute the global coordinates from the reference coordinates, and compute the elements matrices and vectors from the nodes.

4 Two dimensions

We've looked at the problem of finding the steady state solution of pollution diffusing in one dimension. Now let's move on to finding the distribution in two dimensions. From this the generalisation to higher dimensions is "straightforward".

The steady state pollution distribution $\Psi(x, y)$ in cartesian coordinates satisfies

$$\nabla^2\Psi + S(\mathbf{x}) = \left(\partial_{xx} + \partial_{yy}\right)\Psi + S(\mathbf{x}) = 0. \tag{4.1}$$

We'll fix the distribution to be zero at the left edge, $\Psi(0, y) = 0$. We'll allow pollution to flow out of the other edges, giving the boundary conditions on all edges as

$$\begin{aligned} \Psi(0, y) = 0, \quad \partial_x\Psi(1, y) = 0, \\ \partial_y\Psi(x, 0) = 0, \quad \partial_y\Psi(x, 1) = 0. \end{aligned} \tag{4.2}$$

Once again we want to write down the weak form by integrating by parts. To do that we rely on the divergence theorem,

$$\int_\Omega \mathrm{d}\Omega\, \nabla_i\Psi = \int_\Gamma \mathrm{d}\Gamma\, \Psi n_i. \tag{4.3}$$

Here Ω is the domain (which in our problem is the square, $x, y \in [0, 1]$) and Γ its boundary (in our problem the four lines $x = 0, 1$ and $y = 0, 1$), whilst **n** is the (inward-pointing) normal vector to the boundary.

We then multiply the strong form of the steady state equation by a *weight function* $w(x, y)$ and integrate by parts, using the divergence theorem, to remove the second derivative. To enforce the boundary conditions effectively we again choose the weight function to vanish where the value of the temperature is explicitly given, i.e. $w(0, y) = 0$. That is, we split the boundary Γ into a piece Γ_D where the boundary conditions are in Dirichlet form (the

value Ψ is given) and a piece Γ_N where the boundary conditions are in Neumann form (the value of the normal derivative $n_i \nabla_i T$ is given). We then enforce that on Γ_D the weight function vanishes.

For our problem, this gives

$$\int_\Omega \mathrm{d}\Omega \, \nabla_i w \nabla_i \Psi = \int_\Omega \mathrm{d}\Omega \, wS. \tag{4.4}$$

Re-writing for our explicit domain and our Cartesian coordinates we get

$$\int_0^1 \mathrm{d}y \int_0^1 \mathrm{d}x \, \left(\partial_x w \partial_x \Psi + \partial_y w \partial_y \Psi \right) = \int_0^1 \mathrm{d}y \int_0^1 \mathrm{d}x \, w(x,y) S(x,y). \tag{4.5}$$

This should be compared to the one dimensional case

$$\int_0^1 \mathrm{d}x \, \partial_x w(x) \partial_x \Psi(x) = \int_0^1 \mathrm{d}x \, w(x) S(x). \tag{4.6}$$

We can now envisage using the same steps as the one dimensional case. Split the domain into elements, represent all functions in terms of known *shape functions* on each element, assemble the problems in each element to a single matrix problem, and then solve the matrix problem.

4.1 Elements

We now need to split the domain into subdomains - elements. Constructing a good grid for a general case is a *hard* problem for which there are many complex solvers available. In our case we are going to use one simple approach: triangulate the domain by using equal sized triangles.

What we're doing here is

1. Providing a list of nodes by their global coordinates.

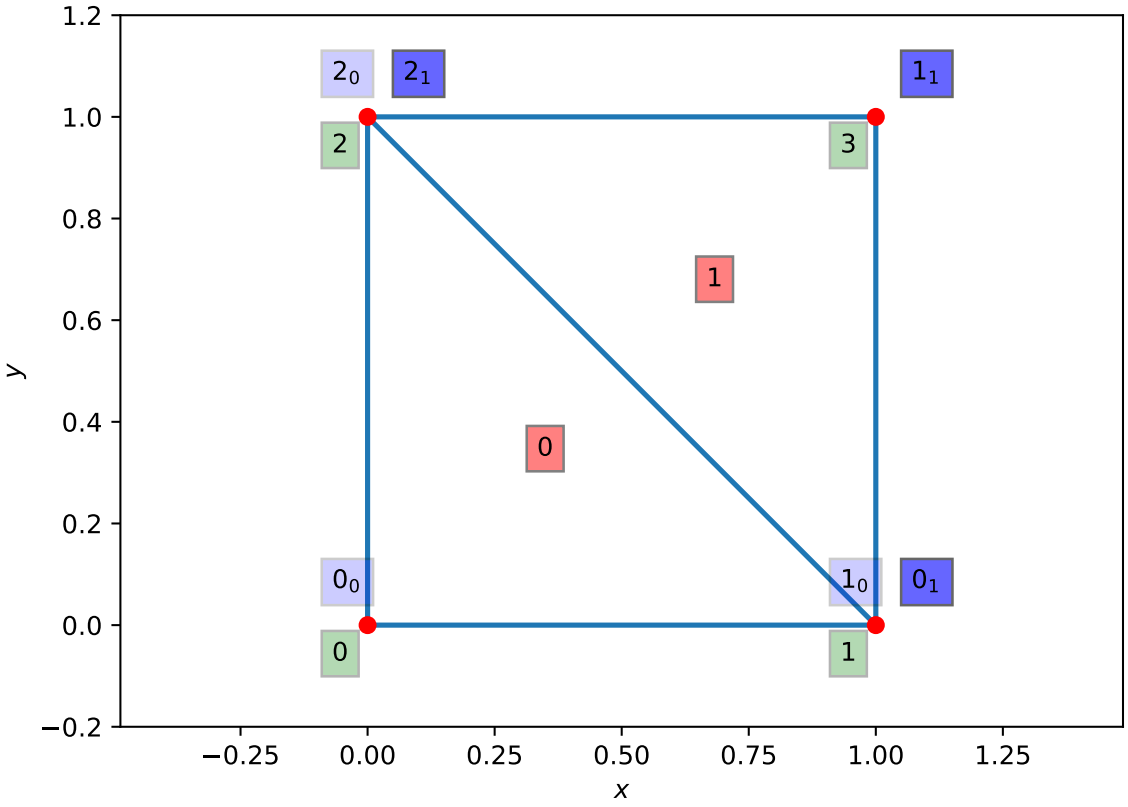

Figure 4.1: The square domain split into triangular elements. Red dots are the nodes. Red squares contain the element numbers. Green squares contain the *global* node numbers. Blue squares contain the *local* node numbers, with the associated element number as subscript.

2. Providing the (integer) *element node array* `IEN` which says how the elements are linked to the nodes.

We have that for element e and *local* node number $a = 0, 1, 2$ the global node number is $A = IEN(e, a)$. This notation is sufficiently conventional that `matplotlib` recognizes it with its `triplot`/`tripcolor`/`trisurf` functions. In this case we have

```
IEN = np.array([[0, 1, 2],
                [1, 3, 2]])
```

which says that the first element (element `0`) is made of the (global) nodes numbered `0`, `1`, and `2`, which the second element (element `1`) is made of the (global) nodes numbered `1`, `3`, and `2`. It is convention that the nodes are ordered in the anti-clockwise direction as the local number goes from `0` to `2`.

The plot shows the

- element numbers in the red boxes
- the *global* node numbers in the green boxes
- the *local* element numbers in the blue boxes (the subscript shows the element number).

We will need one final array, which is the ID or (integer) *destination* array. This links the *global* node number to the *global* equation number in the final linear system. As the order of the equations in a linear system doesn't matter, this essentially encodes whether a node should have any equation in the linear system. Any node on Γ_D, where the value of the temperature is given, should not have an equation. In the example above the left edge is fixed, so nodes `0` and `2` lie on Γ_D and should not have an equation. Thus in our case we have

```
ID = np.array([-1, 0, -1, 1])
```

In the one dimensional case we used the *location matrix* or LM array to link local node numbers in elements to equations. With the IED and ID arrays the LM matrix is strictly redundant, as $LM(a, e) = ID(IEN(e, a))$. However, it's still standard to construct it:

```
LM = np.zeros_like(IEN.T)
for e in range(IEN.shape[0]):
    for a in range(IEN.shape[1]):
        LM[a,e] = ID[IEN[e,a]]
LM
```

```
array([[-1,  0],
       [ 0,  1],
       [-1, -1]])
```

4.2 Function representation and shape functions

We're going to want to write our unknown functions Ψ, w in terms of shape functions. These are easiest to write down for a single reference element, in the same way as we did for the one dimensional case where our reference element used the coordinates ξ. In two dimensions we'll use the reference coordinates ξ_1, ξ_2, and the standard "unit" triangle shown in (Figure 4.2).

The shape functions on this triangle are

$$\begin{aligned} N_0(\xi_1, \xi_2) &= 1 - \xi_1 - \xi_2, \\ N_1(\xi_1, \xi_2) &= \xi_1, \\ N_2(\xi_1, \xi_2) &= \xi_2. \end{aligned} \tag{4.7}$$

The derivatives are all either 0 or ± 1.

As soon as we have the shape functions, our weak form becomes

$$\sum_A T_A \int_\Omega \mathrm{d}\Omega \, \left(\partial_x N_A(x,y)\partial_x N_B(x,y) + \partial_y N_A(x,y)\partial_y N_B(x,y)\right) = \int_\Omega \mathrm{d}\Omega \, N_B(x,y) S(x,y). \tag{4.8}$$

If we restrict to a single element the weak form becomes

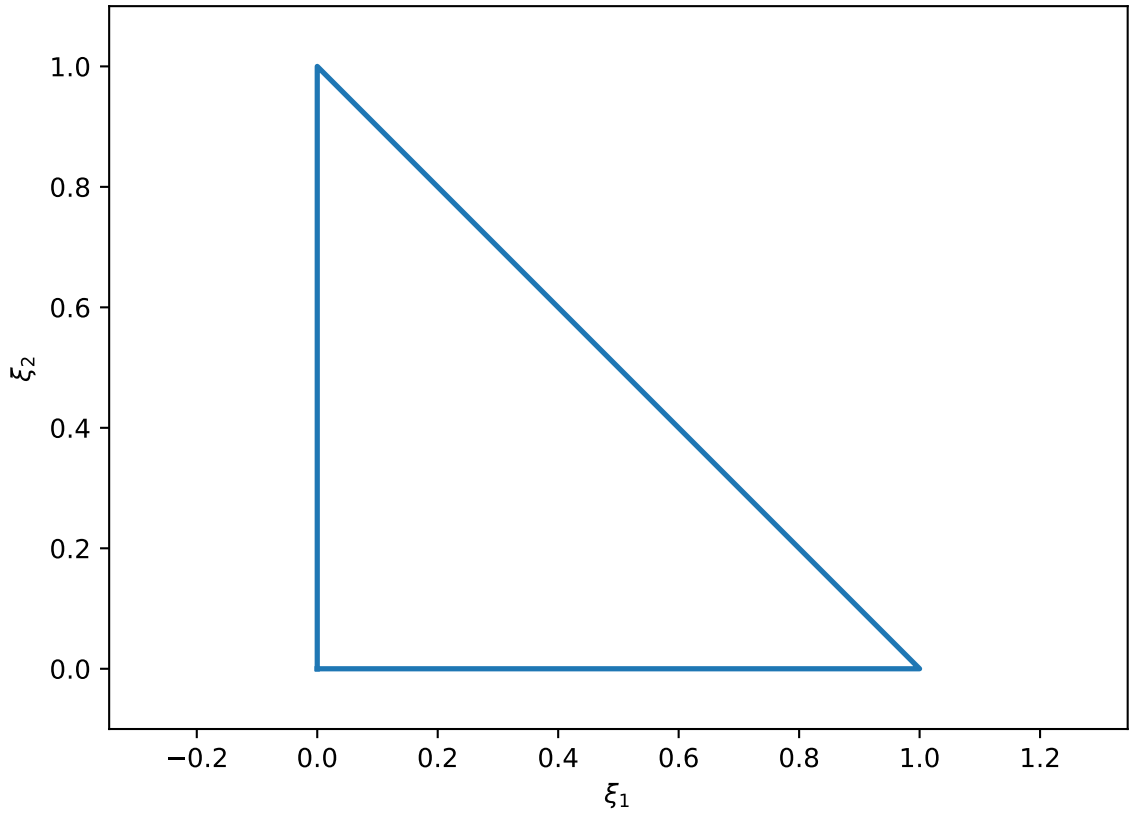

Figure 4.2: The standard reference triangle.

$$\sum_A T_A \int_{\triangle} \mathsf{d}\triangle \left(\partial_x N_A(x,y)\partial_x N_B(x,y) + \partial_y N_A(x,y)\partial_y N_B(x,y)\right) = \int_{\triangle} \mathsf{d}\triangle\, N_B(x,y)S(x,y). \tag{4.9}$$

We need to map the triangle and its $(x, y) = \mathbf{x}$ coordinates to the reference triangle and its $(\xi_1, \xi_2) = \boldsymbol{\xi}$ coordinates. We also need to work out the integrals that appear in the weak form. We need the transformation formula

$$\int_{\triangle} \mathsf{d}\triangle\, \phi(x,y) = \int_0^1 \mathsf{d}\xi_2 \int_0^{1-\xi_2} \mathsf{d}\xi_1\, \phi\left(x(\xi_1,\xi_2), y(\xi_1,\xi_2)\right) j(\xi_1,\xi_2), \tag{4.10}$$

where the *Jacobian matrix* J is

$$J = \left[\frac{\partial \mathbf{x}}{\partial \boldsymbol{\xi}}\right] = \begin{pmatrix} \partial_{\xi_1} x & \partial_{\xi_2} x \\ \partial_{\xi_1} y & \partial_{\xi_2} y \end{pmatrix} \tag{4.11}$$

and hence the *Jacobian determinant* j is

$$j = \det J = \det\left[\frac{\partial \mathbf{x}}{\partial \boldsymbol{\xi}}\right] = \det \begin{pmatrix} \partial_{\xi_1} x & \partial_{\xi_2} x \\ \partial_{\xi_1} y & \partial_{\xi_2} y \end{pmatrix}. \tag{4.12}$$

We will also need the Jacobian matrix when writing the derivatives of the shape functions in terms of the coordinates on the reference triangle, i.e.

$$\begin{pmatrix} \partial_x N_A & \partial_y N_A \end{pmatrix} = \begin{pmatrix} \partial_{\xi_1} N_A & \partial_{\xi_2} N_A \end{pmatrix} J^{-1}. \tag{4.13}$$

The integral over the reference triangle can be directly approximated using, for example, Gauss quadrature. To second order we have

$$\int_0^1 \mathsf{d}\xi_2 \int_0^{1-\xi_2} \mathsf{d}\xi_1 \, \psi\left(x(\xi_1,\xi_2), y(\xi_1,\xi_2)\right) \simeq \frac{1}{6}\sum_{j=1}^{3} \psi\left(x(\xi_1^{(j)},\xi_2^{(j)}), y(\xi_1^{(j)},\xi_2^{(j)})\right) \tag{4.14}$$

where

$$\begin{aligned} \xi_1^{(1)} &= \frac{1}{6}, \quad \xi_2^{(1)} = \frac{1}{6}, \\ \xi_1^{(2)} &= \frac{4}{6}, \quad \xi_2^{(2)} = \frac{1}{6}, \\ \xi_1^{(3)} &= \frac{1}{6}, \quad \xi_2^{(3)} = \frac{4}{6}. \end{aligned} \tag{4.15}$$

Finally, we need to map from the coordinates $\boldsymbol{\xi}$ to the coordinates **x**. This is straightforward if we think of writing each component (x, y) in terms of the shape functions. So for element e with node locations (x_a^e, y_a^e) for local node number $a = 0, 1, 2$ we have

$$\begin{aligned} x &= x_0^e N_0(\xi_1,\xi_2) + x_1^e N_1(\xi_1,\xi_2) + x_2^e N_2(\xi_1,\xi_2), \\ y &= y_0^e N_0(\xi_1,\xi_2) + y_1^e N_1(\xi_1,\xi_2) + y_2^e N_2(\xi_1,\xi_2). \end{aligned} \tag{4.16}$$

4.3 Algorithm

The steps needed to solve this case closely follow the algorithm in one dimension. The outline algorithm becomes

1. Set up the grid, including the mapping between elements and nodes (`IEN`) and between elements and equations (`ID`).
2. Set up a boundary value structure (in Python a dictionary would work).
3. Set up the location matrix LM.
4. Set up arrays, initially all zero, for the global stiffness matrix and for vector.
5. For each element:
    1. Form the element stiffness matrix k_{ab}^e.
    2. Form the element force vector f_b^e.

3. Add the contributions to the global stiffness matrix and force vector.
4. Modify using the boundary values if needed.

6. Solve $K\boldsymbol{\psi} = \mathbf{F}$.

The key difference is the complexity of mapping from a general element (triangle) to the reference element (triangle) on which all the coefficients are known. The key steps are:

1. Write a function that, given $\boldsymbol{\xi}$, returns that shape functions at that location.
2. Write a function that, given $\boldsymbol{\xi}$, returns the derivatives of the shape functions at that location.
3. Write a function that, given the (global) locations $\mathbf{x}$ of the nodes of a triangular element and the local coordinates $\boldsymbol{\xi}$ within the element returns the corresponding global coordinates.
4. Write a function that, given the (global) locations $\mathbf{x}$ of the nodes of a triangular element and the local coordinates $\boldsymbol{\xi}$, returns the Jacobian matrix at that location.
5. Write a function that, given the (global) locations $\mathbf{x}$ of the nodes of a triangular element and the local coordinates $\boldsymbol{\xi}$, returns the determinant of the Jacobian matrix at that location.
6. Write a function that, given the (global) locations $\mathbf{x}$ of the nodes of a triangular element and the local coordinates $\boldsymbol{\xi}$ within the element returns the derivatives $\partial_{\mathbf{x}} N_a = J^{-1}\partial_{\boldsymbol{\xi}} N_a$.
7. Write a function that, given a function $\psi(\boldsymbol{\xi})$, returns the quadrature of ψ over the reference triangle.
8. Write a function that, given the (global) locations of the nodes of a triangular element and a function $\phi(x, y)$, returns the quadrature of ϕ over the element.
9. Write a function to compute the coefficients of the stiffness matrix for a single element,

$$k^e_{ab} = \int_{\triangle^e} \mathsf{d}\triangle^e \left(\partial_x N_a(x,y)\partial_x N_b(x,y) + \partial_y N_a(x,y)\partial_y N_b(x,y)\right).$$

10. Write a function to compute the coefficients of the force vector for a single element,

$$f^e_b = \int_{\triangle^e} \mathsf{d}\triangle^e N_b(x,y)S(x,y). \tag{4.17}$$

4.3.1 Boundary conditions

In principle, the application of boundary conditions in more than one dimension precisely matches that seen in the one dimensional case. The practical implementation, as with all things in higher dimensions, can be more complex.

For Dirichlet boundaries, if an element has a node on a boundary, then every node within that element that is connected to the Dirichlet node needs updating using an appropriate multiple of the local stiffness matrix. For example, suppose that global node x_C is on the Dirichlet boundary Γ_D and the solution takes the value α there. Further suppose that x_C is mapped to *local* node 0 in the reference element. Finally, assume that the local node b within the same element is not on the Dirichlet boundary (so must be solved for), and is linked to global node x_B. Then the component of the force vector F_B must be updated by $-\alpha k^{(e)}_{0b}$.

For Neumann boundary conditions, the force vector is directly updated using the value of the boundary condition. *However*, in this case we have to note that the boundary contribution involves the surface integral over Γ_N (which will be a line integral in two dimensions). Therefore the update to F_B is not just the value $\beta = \partial_x \psi(x_B)$, but must also weight it by the length of the edge within the element that lies in the Neumann boundary. Suppose that x_B and x_C are both nodes of the same element, *and* that the edge connecting them lies within the Neumann boundary, *and* that this edge has length ℓ. Using a mid-point rule approximation to the line integral along the edge, *both* F_B and F_C should be updated by $\beta\ell/2$. Better approximations to the integral may be needed for accuracy reasons, and in higher dimensions than two.

4.4 Grid generation

The final, essential, topic that has not been covered is how to generate a grid. Good grid generators or meshers are generally hard (look at, for example, `gmesh` or `dmsh` for examples): here is a very simple one for this specific problem.

```
def generate_2d_grid(Nx):
    Nnodes = Nx+1
    x = np.linspace(0, 1, Nnodes)
    y = np.linspace(0, 1, Nnodes)
```

```
    X, Y = np.meshgrid(x,y)
    nodes = np.zeros((Nnodes**2,2))
    nodes[:,0] = X.ravel()
    nodes[:,1] = Y.ravel()
    ID = np.zeros(len(nodes), dtype=np.int64)
    boundaries = dict()  # Will hold the boundary values
    n_eq = 0
    for nID in range(len(nodes)):
        if np.allclose(nodes[nID, 0], 0):
            ID[nID] = -1
            boundaries[nID] = 0  # Dirichlet BC
        else:
            ID[nID] = n_eq
            n_eq += 1
            if ( (np.allclose(nodes[nID, 1], 0)) or
                 (np.allclose(nodes[nID, 0], 1)) or
                 (np.allclose(nodes[nID, 1], 1)) ):
                boundaries[nID] = 0 # Neumann BC
    IEN = np.zeros((2*Nx**2, 3), dtype=np.int64)
    for i in range(Nx):
        for j in range(Nx):
            IEN[2*i+2*j*Nx  , :] = (i+j*Nnodes,
                                    i+1+j*Nnodes,
                                    i+(j+1)*Nnodes)
            IEN[2*i+1+2*j*Nx, :] = (i+1+j*Nnodes,
                                    i+1+(j+1)*Nnodes,
                                    i+(j+1)*Nnodes)
    return nodes, IEN, ID, boundaries
```

The results of using a more complex mesh generator (in this case `dmsh`) on a more complex domain, but still solving the heat equation using exactly the functions outlined in the exercise below, is shown in (Figure 4.3).

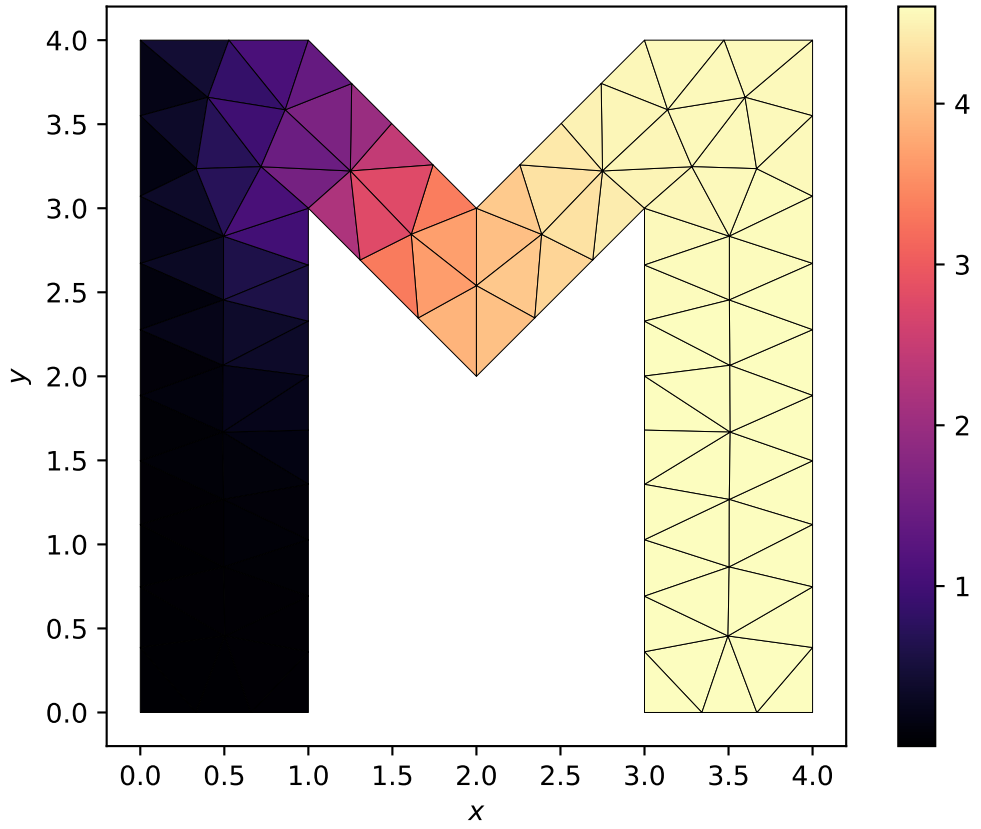

Figure 4.3: A more complex grid generated by `dmsh` for a more general polygonal domain. The heat equation is solved with a Gaussian source. The boundary at $x = 0$ is held fixed at $\Psi = 0$. All other boundaries use Neumann boundary conditions where the flux vanishes. The function that solves the finite element method here is identical to that on the simpler grids.

4.5 Exercise

Exercise 4.1.

1. Write a function that, given a list of nodes and the IEN and ID arrays, and also given the source function S, uses the finite element method to return $\mathbf{\Psi}$.
2. Test on the system $S(x, y) = 1$ with exact solution $\Psi = x(1 - x/2)$.
3. For a more complex case with the same boundary conditions try

$$S(x, y) = 2x(x - 2)(3y^2 - 3y + \tfrac{1}{2}) + y^2(y - 1)^2 \tag{4.18}$$

with exact solution

$$\Psi(x, y) = x(1 - \tfrac{x}{2})y^2(1 - y)^2. \tag{4.19}$$

5 Time evolution

We now want to solve the full time dependent problem, (Equation 1.4),

$$\frac{D\Psi}{Dt} = \frac{\partial \Psi}{\partial t} + \mathbf{u} \cdot \nabla \Psi = S + \mu_\Psi \nabla^2 \Psi. \tag{5.1}$$

Again we will start by restricting to one dimension, giving

$$\frac{D\Psi}{Dt} = \underbrace{\frac{\partial \Psi}{\partial t}}_{(1)} + \underbrace{u\frac{\partial \Psi}{\partial x}}_{(2)} = \underbrace{S}_{(3)} + \underbrace{\mu_\Psi \frac{\partial^2 \Psi}{\partial x^2}}_{(4)}. \tag{5.2}$$

We remember that the respective terms are

1. the time evolution of the pollution concentration,
2. the advection of the pollution concentration by the wind,
3. the source/sink of pollution,
4. the diffusion of pollution concentration.

We will again use the domain $x \in [0, 1]$ with the boundary conditions

$$\Psi(0) = \alpha, \quad \partial_x \Psi|_{x=1} = \beta. \tag{5.3}$$

5.1 Weak form

We will repeat the key steps from the static case: introducing the weak form by multiplying by a test function and integrating over the domain, and representing all functions in terms of shape, or basis, functions. The key difference now is that the functions depend on time.

The continuum weak form of (Equation 5.2) is

$$\underbrace{\int w\frac{\partial \Psi}{\partial t}}_{(1)} + \underbrace{uw(1)\Psi(1)}_{(2a)} - \underbrace{u\int \Psi\frac{\partial w}{\partial x}}_{(2b)} = \underbrace{\int wS}_{(3)} + \underbrace{\mu_\Psi w(1)\beta}_{(4a)} - \underbrace{\mu_\Psi \int \frac{\partial w}{\partial x}\frac{\partial \Psi}{\partial x}}_{(4b)}. \tag{5.4}$$

Note that we have used that the weighting function vanishes at the Dirichlet boundary, $w(0) = 0$, and assumed that the test function w is time independent. We have integrated by parts again to move derivatives onto the test function, with the notation linking the boundary terms to their associated integrals.

To include time dependence we take the static case of (Equation 3.8), which was

$$\Psi(x) = \sum_A \Psi_A N_A(x), \tag{5.5}$$

and generalise it so that $\Psi_A \equiv \Psi_A(t)$. Now the nodal values Ψ_A contain the time dependence. The shape functions N_A remain time independent.

The weak form will now give us

$$\begin{aligned}
\text{(1)} &\to \sum_B w_B \sum_A \frac{\partial \Psi_A}{\partial_t}\int N_A N_B,\\
\text{(2a)} &\to uw(1)\Psi(1),\\
\text{(2b)} &\to -u\sum_B w_B \sum_A \Psi_A \int N_A \frac{\partial N_B}{\partial x},\\
\text{(3)} &\to \sum_B w_B \int N_B S,\\
\text{(4a)} &\to \mu_\Psi w(1)\beta,\\
\text{(4b)} &\to -\mu_\Psi \sum_B w_B \sum_A \Psi_A \int \frac{\partial N_A}{\partial x}\frac{\partial N_B}{\partial x}.
\end{aligned} \tag{5.6}$$

As before, we gather together all of the terms with respect to the (*arbitrary*) coefficients of the test function w_B. We also introduce the function $q(x) = \alpha N_0(x)$ and write

$\Psi(x,t) = \psi(x,t) + q(x)$, to balance the Dirichlet boundary condition. This gives the matrix equation

$$M_{AB}\frac{\partial \psi_A}{\partial t} + K_{AB}\psi_A = F_B, \tag{5.7}$$

where M is the *mass* or *capacity* matrix, and K is the stiffness matrix and **F** the force vector as before. We read off the coefficients of the various terms using (Equation 5.6) as

$$\begin{aligned} M_{ab} &= \int N_a N_b, \\ K_{ab} &= \mu_\Psi \int \frac{\partial N_a}{\partial x}\frac{\partial N_b}{\partial x} - u\int N_a \frac{\partial N_b}{\partial x}, \\ F_b &= \int N_b S - u\Psi_0 \delta^0_{B(b)} + \mu_\Psi \delta^{N_{\text{elements}}}_{B(b)} - q\delta^0_{B(b)} k^0_{12}. \end{aligned} \tag{5.8}$$

The terms are given with respect to the *local* element number $\{a, b\}$ and are assembled into the *global* matrix in exactly the same way as before. The notation $B(b)$ indicates the global node number computed from the local number, and is needed to identify the boundary locations.

5.2 Time stepping

The matrix equation (Equation 5.7) solves for ψ via evolving its nodal values ψ_A in time. This equation is *semi-discrete*: it is continuous in time, but discrete in space.

Semi-discrete approaches to PDEs are more general than finite element methods. They have the huge advantage that we end up solving ODEs, for which there is a vast literature, considerable analysis, and many well implemented and tested codes. They have the disadvantage that they are typically less efficient than bespoke, fully discrete methods with the same order of accuracy.

Here we give two simple methods that are often used in solving the time evolution of a semi-discrete problem. We write the system to be solved as

$$\frac{\partial \mathbf{U}}{\partial t} = \mathcal{F}(\mathbf{U}) \tag{5.9}$$

where **U** is the *state vector*. In the case of (Equation 5.7) the state vector is the coefficients ψ_A and the *right-hand-side vector* $\mathcal{F}$ is

$$\mathcal{F}_A = M^{-1}_{AB} \left(F_B - K_{AB} \psi_A \right) . \tag{5.10}$$

Note that explicitly computing the matrix inverse is typically numerically inaccurate, and instead solving the linear system is preferred.

5.2.1 Euler

We denote the (known) state vector at time $t^{(n)}$ and $\mathbf{U}^n$. In Euler's method we update to the unknown time $t^{(n+1)} = t^{(n)} + \Delta t$ by

$$\mathbf{U}^{n+1} = \mathbf{U}^n + \Delta t \, \mathcal{F} \left(\mathbf{U}^n \right) . \tag{5.11}$$

This is exactly the result given by forward differencing in time, as seen (for example) in the derivation of FTCS. It is explicit, and gives first order accuracy in time.

5.2.2 RK2

Higher order methods can be constructed by updating in multiple *stages*. At each stage we compute the right hand side vector, and from that can approximate a solution at a time typically in $\left[t^{(n)}, t^{(n+1)} \right]$. By combining these stages appropriately a better approximation to the solution at the next timestep can be found. The *Runge-Kutta* family of methods is typical for this approach.

One particular subset of Runge-Kutta methods can ensure a form of total variation boundedness, linked to the TVD methods seen earlier. These *Strict Stability Preserving* (SSP) methods are preferred when dealing with semi-discrete PDE evolutions. The standard second order, explicit, SSP Runge-Kutta method (ESSPRK2) can be written

$$\begin{aligned} \mathbf{U}^{[1]} &= \mathbf{U}^n + \Delta t \, \mathcal{F} \left(\mathbf{U}^n \right) , \\ \mathbf{U}^{n+1} &= \tfrac{1}{2} \left\{ \mathbf{U}^n + \mathbf{U}^{[1]} + \Delta t \, \mathcal{F} \left(\mathbf{U}^{[1]} \right) \right\} . \end{aligned} \tag{5.12}$$

5.3 Evolving to steady state

Let us take the simplified problem from (Chapter 3), where

$$S = 1 - x, \quad \mu = 1. \tag{5.13}$$

We know the steady state solution is $x(x^2 - 3x + 3)/6$. However, we can start from the solution $x = 0$ and see how it evolves towards steady state. For this we will use Euler timestepping.

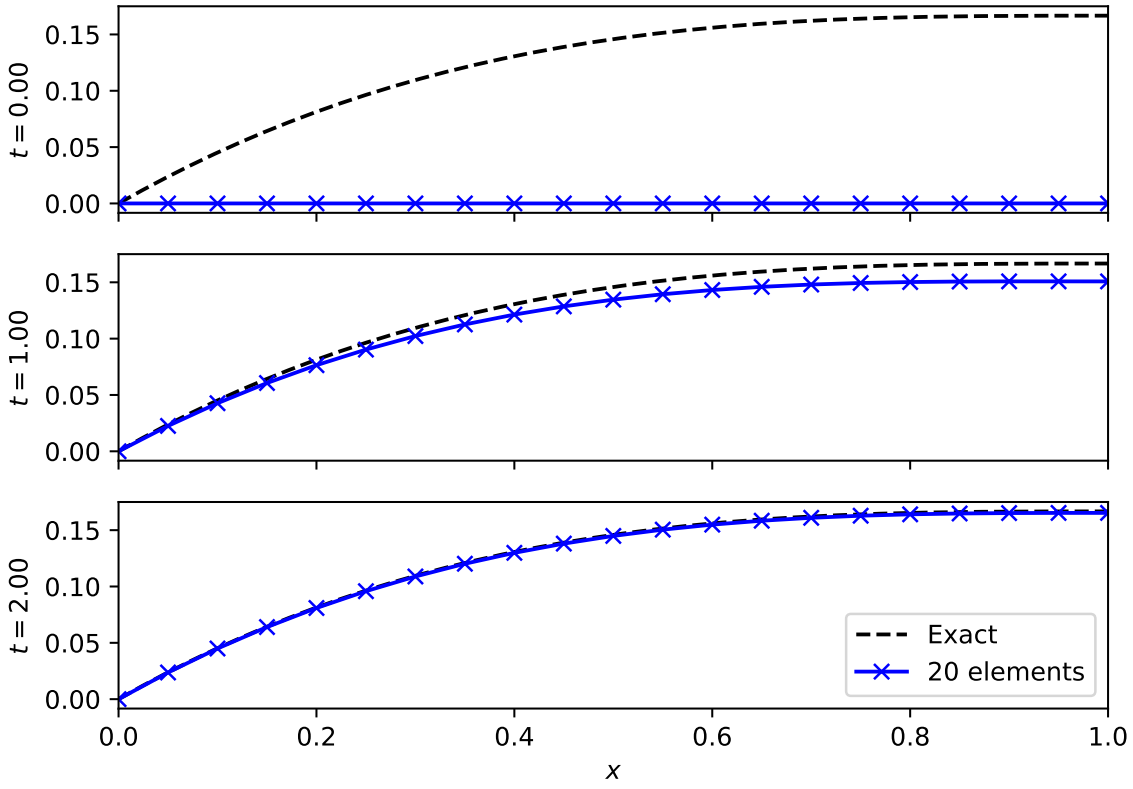

Figure 5.1: Evolving to the steady state solution to the advection diffusion equation with source $S = 1 - x$.

5.4 Advection dominated flow

Let us now look at the advection problem from setting $S(x) = 0 = \mu_\Psi$. The advection equation results and we have, from (Equation 5.8) that

$$\begin{aligned} M_{ab} &= \int N_a N_b, \\ K_{ab} &= -u \int N_a \frac{\partial N_b}{\partial x}, \\ F_b &= -u\Psi_0 \delta^0_{B(b)} - q\delta^0_{B(b)} k^0_{12}. \end{aligned} \tag{5.14}$$

The results are seen in (Figure 5.2) where the expected advective behaviour is seen. Note that the boundary conditions are built into the scheme here, through their appearance in the force vector. Changing boundary conditions will require modifying the scheme.

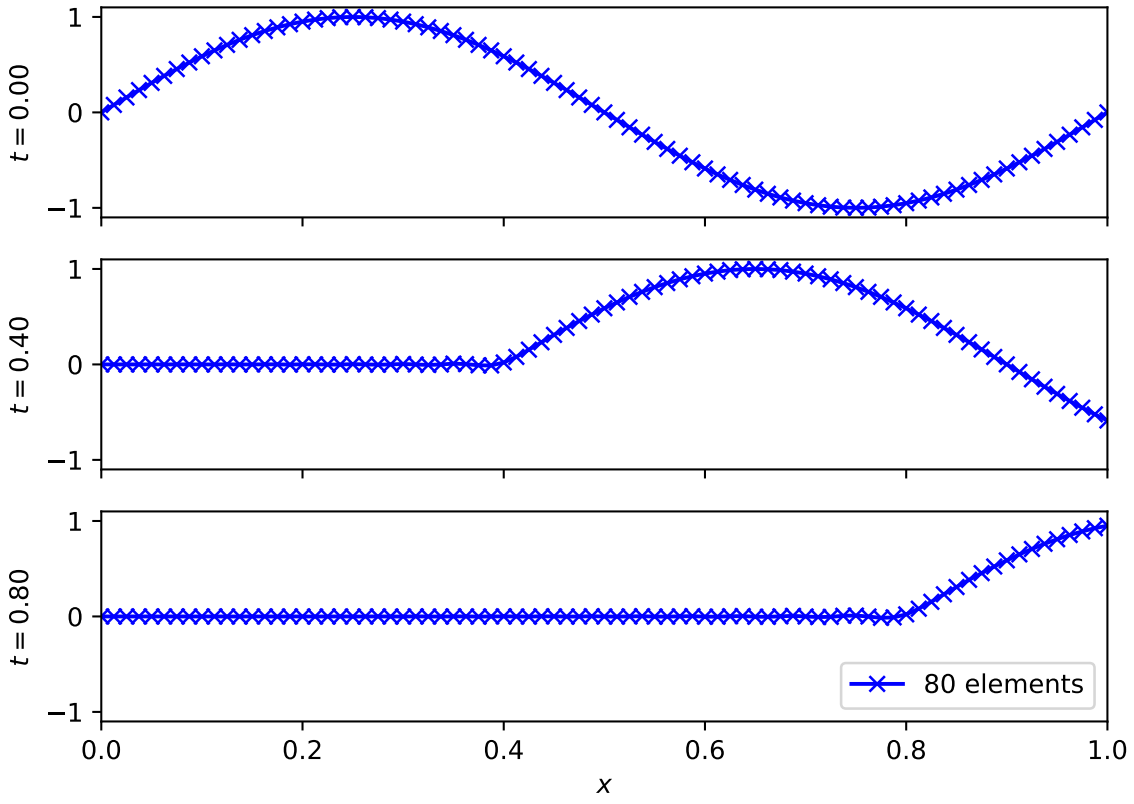

Figure 5.2: Advection of a sine wave. Note the standard boundary conditions (Dirichlet on the left, Neumann on the right) means the pulse leaves the domain.

5.5 Matrix structure

The properties of the finite element method is closely tied to the properties of the matrices involved.

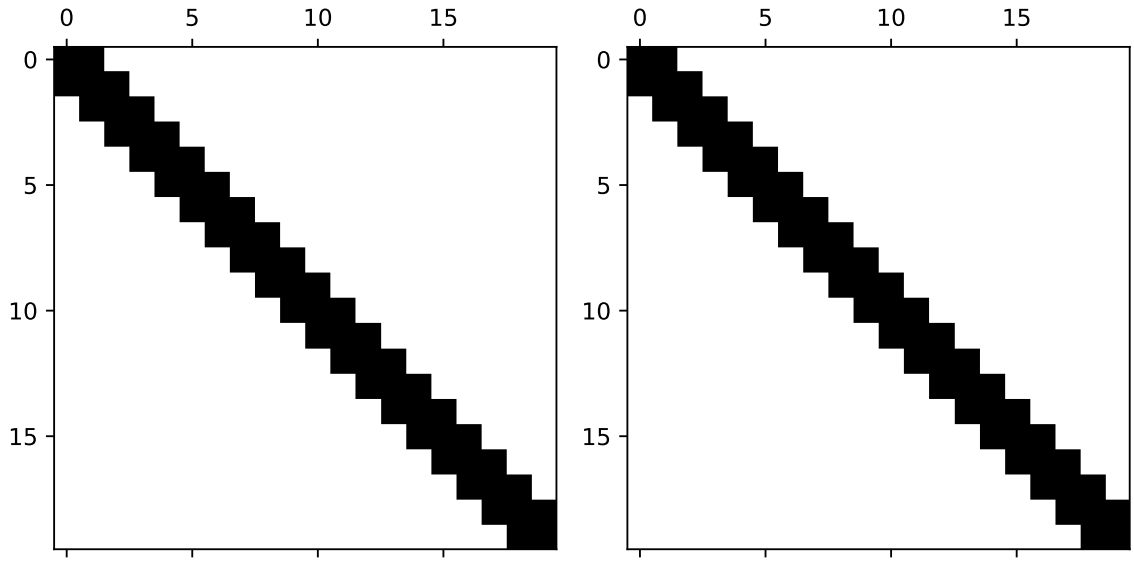

Figure 5.3: The structure of the mass (left) and stiffness (right) matrices for the advection diffusion problem with $u = 1, \mu = 0.1$. The symmetry and sparsity can both be used theoretically and practically.

In (Figure 5.3) we see the structure of the mass and stiffness matrices for the one dimensional advection-diffusion problem. The matrices are symmetric and sparse - in this case they are tridiagonal. In higher dimensions, or with more complex boundary conditions, or with more complex systems of equations, the matrices become messier. However, the key point of matrix sparsity will be retained.

The symmetry of the mass matrix is useful in proving invertibility (essential for the method to work). The symmetry of the stiffness matrix is useful for constructing the eigenvalues of the discrete system, which are linked to the amplification rates in a von Neumann stability analysis.

The sparsity of the matrices is crucial in implementing a method that scales to large numbers of elements. In the simplest one dimensional implementation the size of the matrices scales as N^2_{elements}. However, the tridiagonal nature means that the amount of *useful* (non-zero) information scales as N_{elements}. The computational memory and work saved

is substantial even with only tens of elements. In higher dimensions with large domains (millions of elements) the construction of the full matrix is impractical.

6 Flexibility with efficiency

In addition to the issues with higher order schemes noted above, there is one wasteful step that is worth noting. Each time a higher order scheme reconstructs the data it constructs a high-order piecewise polynomial representation of the data everywhere. Then, in the time update, most of this information is thrown away. At the beginning of each timestep we only know the value of the function (for finite difference approaches) or its integral average (for finite volume approaches).

Some alternative methods store the *moments* or *modes* of the solution, and update all of them. In these methods all of the information needed to evaluate the solution is available at all times and locations, and all the information is updated at every step. This should make the methods more efficient and more local (with a smaller stencil, and hence better on parallel machines). Their disadvantages will come in the timestep and with discontinuous data.

The presentation here closely follows (Hesthaven 2017) – check there for considerably more details, particularly on the theoretical results.

6.1 Function basis and weak form

Recall that in this chapter we are looking at the advection equation

$$\Psi_t + u\Psi_x = 0. \tag{6.1}$$

We want to be able to compute the value of the function $\Psi(t, x)$ at any point. We can do this by writing a in terms of a *function basis* $\phi_n(t, x)$ as

$$\Psi(t,x) = \sum_n \hat{\Psi}_n \phi_n(t,x). \tag{6.2}$$

Here the *modes* or *modal coefficients* $\hat{\Psi}_n$ are constants. An example of a function basis would be

$$\begin{aligned} &\phi_0(t,x) = 1, \quad &&\phi_1(t,x) = x, \quad &&\phi_2(t,x) = t, \\ &\phi_3(t,x) = \tfrac{1}{2}x^2, \quad &&\phi_4(t,x) = \tfrac{1}{2}t^2, \quad &&\phi_5(t,x) = xt. \end{aligned} \tag{6.3}$$

These six modes will perfectly describe any function that remains quadratic for all space and time.

Note that the function basis plays a very similar role to the shape functions discussed earlier for other finite element methods. The crucial distinctions here are that (a) the function basis is confined to a single element whilst a shape function is linked to a node and can be non-zero in multiple elements, and (b) shape functions are chosen so that the coefficients are directly linked to the values of the function at nodes, whilst basis functions are typically not normalized in that way.

It is often more convenient the explicitly separate space and time, as we saw using the semi-discrete approach in (Chapter 5). In this case we can represent the solution using a purely spatial function basis, as

$$\Psi(t,x) = \sum_n \hat{\Psi}_n(t) \phi_n(x). \tag{6.4}$$

Now the modes depend on time, and there will only be three basis functions needed to describe quadratic data.

Clearly we cannot store an infinite number of modes. By restricting our sum to the $m+1$ modes by writing

$$\Psi(t,x) = \sum_{n=0}^{m} \hat{\Psi}_n(t) \phi_n(x) \tag{6.5}$$

we are restricting out solution to live in a finite dimensional function space (denoted $\mathbb{V}$) with basis $\{\phi_n\},\ n = 0, \dots, m$. That means that, in general, any solution $\Psi(t,x)$ will

have an error when plugged into the advection equation. We can pick out *a* solution by insisting that this error is orthogonal to $\mathbb{V}$.

To see how this works, write the error term as $\epsilon(t, x)$. As our (infinite dimensional) function basis can describe any function, we expand the error in terms of the ϕ_n as well, as

$$\epsilon(t, x) = \sum_n \hat{\epsilon}_n(t)\phi_n(x). \tag{6.6}$$

Therefore our advection equation, including the error term, becomes

$$\sum_n \left[\left(\frac{\partial \hat{\Psi}_n}{\partial t} - \hat{\epsilon}_n \right) \phi_n(x) + u\hat{\Psi}_n \frac{\partial \phi_n}{\partial x}(x) \right] = 0. \tag{6.7}$$

As our solution is finite dimensional this can be written as

$$\sum_{n=0}^{m} \left[\frac{\partial \hat{\Psi}_n}{\partial t}\phi_n(x) + u\hat{\Psi}_n \frac{\partial \phi_n}{\partial x}(x) \right] = \sum_{n=m+1}^{\infty} \hat{\epsilon}_n \phi_n(x). \tag{6.8}$$

We have used here that the orthogonality of the error requires $\hat{\epsilon}_n$ does not contribute for $n = 0, \ldots, m$. Using standard linear algebra techniques (as ϕ_n is a *basis*), we can get individual equations by taking the inner product with another member of the basis. If we were dealing with vectors in $\mathbb{R}^n$ then the inner product would be a vector dot product. As we are dealing with functions the inner product requires multiplication and integration over the domain,

$$\langle f(x), \phi_l(x) \rangle = \int_V \mathsf{d}x\, f(x)\phi_l(x). \tag{6.9}$$

This will also write the conservation law in the integral, weak, form. This leads to, after integrating by parts,

$$\begin{aligned} \sum_{n=0}^{m} \Bigg[\frac{\partial \hat{\Psi}_n}{\partial t} \left(\int_V \mathsf{d}x\, \phi_n(x)\phi_l(x) \right) + \int_{\partial V} u\hat{\Psi}_n \phi_n(x)\phi_l(x) - \\ u\hat{\Psi}_n \int_V \mathsf{d}x\, \phi_n \frac{\partial \phi_l}{\partial x}(x) \Bigg] = \sum_{n=m+1}^{\infty} \hat{\epsilon}_n \int_V \mathsf{d}x\, \phi_n(x)\phi_l(x). \end{aligned} \tag{6.10}$$

Restricting ourselves to the first $m+1$ modes we see only the left hand side contributes.

We can write this result as a matrix equation. Define the state vector

$$\hat{\mathbf{\Psi}} = (\hat{\Psi}_0, \dots, \hat{\Psi}_N)^T. \tag{6.11}$$

For now, restrict to one dimension and set $V = [-1, 1]$: we can use a coordinate transformation to convert to other domains. Then define the matrices

$$\begin{aligned} \hat{M}_{ln} &= \int_{-1}^{1} \phi_l(x)\phi_n(x), \\ \hat{S}_{ln} &= \int_{-1}^{1} \phi_l(x)\frac{\partial \phi_n}{\partial x}(x), \end{aligned} \tag{6.12}$$

which can be pre-calculated and stored for repeated use. These are typically referred to (building on finite *element* work) as the *mass* matrix ($\hat{M}$) and the *stiffness* matrix ($\hat{S}$). We therefore finally have

$$\hat{M}\frac{\partial \hat{\mathbf{\Psi}}}{\partial t} + \hat{S}^T\left(u\hat{\mathbf{\Psi}}\right) = -\left[\boldsymbol{\phi} F\right]_{-1}^{1}. \tag{6.13}$$

The right hand side term is the *boundary flux* and requires coupling to neighbouring cells, or boundary conditions. It requires evaluating a product of basis functions $\phi_l(x)\phi_n(x)$ at the boundary of the domain.

We see that, once we have evaluated the mass and stiffness matrices, we can then update *all* modes $\hat{\mathbf{\Psi}}$ by evaluating the boundary flux term on the right hand side and solving a linear system. This illustrates the small stencil of discontinuous Galerkin schemes: the only coupling to the other cells is through that boundary integral, which only couples to direct neighbours. However, if the flux terms couple different modes (as evaluating them requires evaluating a product of basis functions $\phi_l(x)\phi_n(x)$), then the amount of information communicated may still be large. Therefore the communication cost of the scheme is linked to the properties of the basis functions at the domain boundary.

We also see that the behaviour of the scheme will crucially depend on the mass matrix $\hat{M}$. If it is singular the scheme cannot work. If it is poorly conditioned then the scheme will rapidly lose accuracy. Crucially, with the monomial basis of (Equation 6.3), the condition

number of the mass matrix grows very rapidly, and the scheme loses accuracy for moderate m.

The choice of whether to prioritize the behaviour of the mass matrix or the flux terms leads to two different schemes.

6.2 Modal Discontinuous Galerkin

If we prioritize the behaviour of the mass matrix as the most important starting point for our scheme we are led to the *modal* Discontinuous Galerkin approach. We noted above that the choice of a monomial basis led to a poorly conditioned mass matrix. Instead, it is sensible to pick as a function basis something from the class of *orthogonal polynomials*, where

$$\int_V w(x)\phi_l(x)\phi_n(x) \propto \delta_{ln}. \tag{6.14}$$

The Kronecker delta δ_{ln} ensures that the mass matrix is diagonal, and hence always easy to invert. When the *weight function* $w(x)$ is identically 1, as needed for the mass matrix in (Equation 6.12), this suggests we should use the *Legendre polynomials* $\phi_n(x) = P_n(x)$, which obey

$$\int_{-1}^{1} P_l(x)P_n(x) = \frac{2}{2n+1}\delta_{ln}. \tag{6.15}$$

A further simplification comes from choosing the normalized Legendre polynomials

$$\tilde{P}_n(x) = \sqrt{\frac{2n+1}{2}}P_n(x) \tag{6.16}$$

which ensures that the mass matrix $\widehat{M}$ is the identity matrix.

Now that we have fixed a choice of basis functions we can evaluate the mass matrix (which will be the identity here) and the stiffness matrix $\widehat{S}$. We still need to evaluate the boundary flux. If we explicitly write out equation (Equation 6.13) in index form (using Einstein summation convention over n) we have

$$\hat{M}_{ln}\frac{\partial\hat{\Psi}_n}{\partial t}+\hat{S}^T_{ln}u\hat{\Psi}_n=-\left[uP_l(x)P_n(x)\hat{\Psi}_n\right]^1_{-1}. \tag{6.17}$$

We can now directly use that $P_n(1)=1$ and $P_n(-1)=(-1)^n$ to get the boundary flux term as

$$-\left[uP_l(x)P_n(x)\hat{\Psi}_n\right]^1_{-1}=u\left\{(-1)^{l+n}\Psi_n(-1)-\Psi_n(1)\right\}. \tag{6.18}$$

Here $\Psi_n(1)$, for example, is the n^{th} mode of the solution at the boundary. As there are two solutions at the boundary of the element - the solution at $x=1_-$ from the interior of the element, and the solution at $x=1_+$ from the exterior (either another element, or from the boundary conditions), we need a Riemann solver to give us a single solution at $x=1$. In the case of linear advection, as here, we can use the upwind solver for the modes as well as for the solution, so

$$\Psi_n(x;\Psi^-_n,\Psi^+_n)=\begin{cases}\Psi^-_n & \text{if } u\geq 0\\ \Psi^+_n & \text{otherwise.}\end{cases} \tag{6.19}$$

Two points should be immediately noted about this discontinuous Galerkin method. First, if we restrict to only one mode ($N=0$), then the only basis function we have is $\tilde{P}_0(x)=1/\sqrt{2}$, the mass matrix $\hat{M}=1$, the stiffness matrix vanishes, and the boundary flux term reduces to the standard finite volume update. In general, the zero mode corresponds to the integral average over the cell or element.

Second, we note that the boundary flux term always couples different modes (when including more than just one), and only in the linear case will it be simple to give a flux formula that works for all modes. As the boundary flux term is crucial in many cases, we need to change approach to simplify the calculation of this term using (possibly approximate) solutions to the Riemann problem.

6.3 Nodal Discontinuous Galerkin

A problem with the modal form used above is with the boundary flux term. The solution (for nonlinear equations) of the flux for higher order modes is complex. The mode cou-

pling at the boundary also means the amount of information communicated could be large, meaning the scheme is not as efficient as it could be. Instead we note that in standard finite volume schemes we need the value of the function either side of the interface. This suggests that, rather than using a modal expansion as above, we should use a *nodal* expansion where the values of the functions are known at particular points. If two of those points are at the boundaries of the cell then those values can be used to compute the flux.

Let us denote these nodal locations by ξ_i, and the values of the solution at these locations by Ψ_i. We therefore have our solution in the form

$$\Psi(t,x) = \sum_{i=0}^{m} \Psi_i(t)\ell_i(x) \tag{6.20}$$

where the $\ell_i(x)$ are the standard indicator interpolating polynomials that obey

$$\ell_i(\xi_j) = \delta_{ij}. \tag{6.21}$$

This directly matches the *modal* form of the solution from (Equation 6.5),

$$\Psi(t,x) = \sum_{n} \hat{\Psi}_n(t)\phi_n(x), \tag{6.22}$$

with the basis functions ϕ_n being the indicator polynomials ℓ_n. We immediately see that the boundary flux term will simplify hugely, as the only term that is non-zero at $x = -1$ comes from the product of $\ell_0(-1)\ell_0(-1)$, using the convention that $\xi_0 = -1$, as $\ell_n(-1) = 0$ for $n \neq 0$. Similarly the only term that is non-zero at $x = +1$ comes from the product of $\ell_m(-1)\ell_m(-1)$. Therefore, for any number of modes m, we only need to communicate one piece of information from the neighbouring element in order to solve the Riemann problem, and this is the value of the solution at that interface.

However, by choosing as a basis the indicator polynomials $\ell_n(x)$, the resulting mass matrix will not be the identity, as the indicator polynomials are not orthogonal. The properties of the mass matrix will now crucially depend on how we choose the locations of the nodes, ξ_i. This is most easily done by linking the nodal form of (Equation 6.20) to the modal form (Equation 6.5), where here we are thinking of ϕ_n as being a different basis ($\phi_n \neq \ell_n$) which is known to be well behaved. This implicitly allows us to restrict ξ_j.

By evaluating both forms at a node ξ_j we get

$$\Psi_j = \sum_n \phi_n(\xi_j)\hat{\Psi}_n. \tag{6.23}$$

By defining a (generalized) *Vandermonde* matrix $\hat{V}$ as

$$\hat{V}_{jn} = \phi_n(\xi_j) \tag{6.24}$$

we see that we can translate from the modal state vector $\hat{\mathbf{\Psi}} = (\hat{\Psi}_0, \dots, \hat{\Psi}_N)^T$ to the nodal state vector $\mathbf{\Psi} = (\Psi_0, \dots, \Psi_N)^T$ via the matrix equation

$$\hat{V}\hat{\mathbf{\Psi}} = \mathbf{\Psi}. \tag{6.25}$$

We can also connect the basis functions ϕ_n to the interpolating polynomials ℓ_i via the Vandermonde matrix. Note that

$$\begin{aligned}
\Psi(t,x) &= \sum_n \hat{\Psi}_n \phi_n(x) \\
&= \sum_i \Psi_i \ell_i(x) \\
&= \sum_i \sum_n \hat{V}_{in} \hat{\Psi}_n \ell_i(x) \\
&= \sum_n \sum_i \hat{V}_{in} \hat{\Psi}_n \ell_i(x) \\
\implies \qquad 0 &= \sum_n \hat{\Psi}_n \left(\sum_i \left[\hat{V}_{in}\ell_i(x) - \phi_n(x) \right] \right).
\end{aligned} \tag{6.26}$$

This immediately gives

$$\hat{V}_{in}\ell_i(x) = \phi_n(x) \tag{6.27}$$

or, by thinking of the basis functions and interpolating polynomials as vectors,

$$\hat{V}^T \ell(x) = \boldsymbol{\phi}(x). \tag{6.28}$$

This allows us to convert the modal approach to deriving a scheme to a nodal approach directly through the Vandermonde matrix.

To construct the nodal scheme we need to fix the location of the nodal points. We have constructed the modal scheme to be well conditioned by looking at the mass matrix. This suggests that to make the nodal scheme well behaved we should ensure good conditioning of the Vandermonde matrix. This requires carefully choosing the nodes ξ_i. We also want to ensure that two of the nodes are at $x = \pm 1$, and that the accuracy of the scheme is as good as possible. All these conditions combine to suggest that the nodes ξ_i should be given by the *Legendre-Gauss-Lobatto* points, which are the zeros of $P'_N(x)$ combined with ± 1.

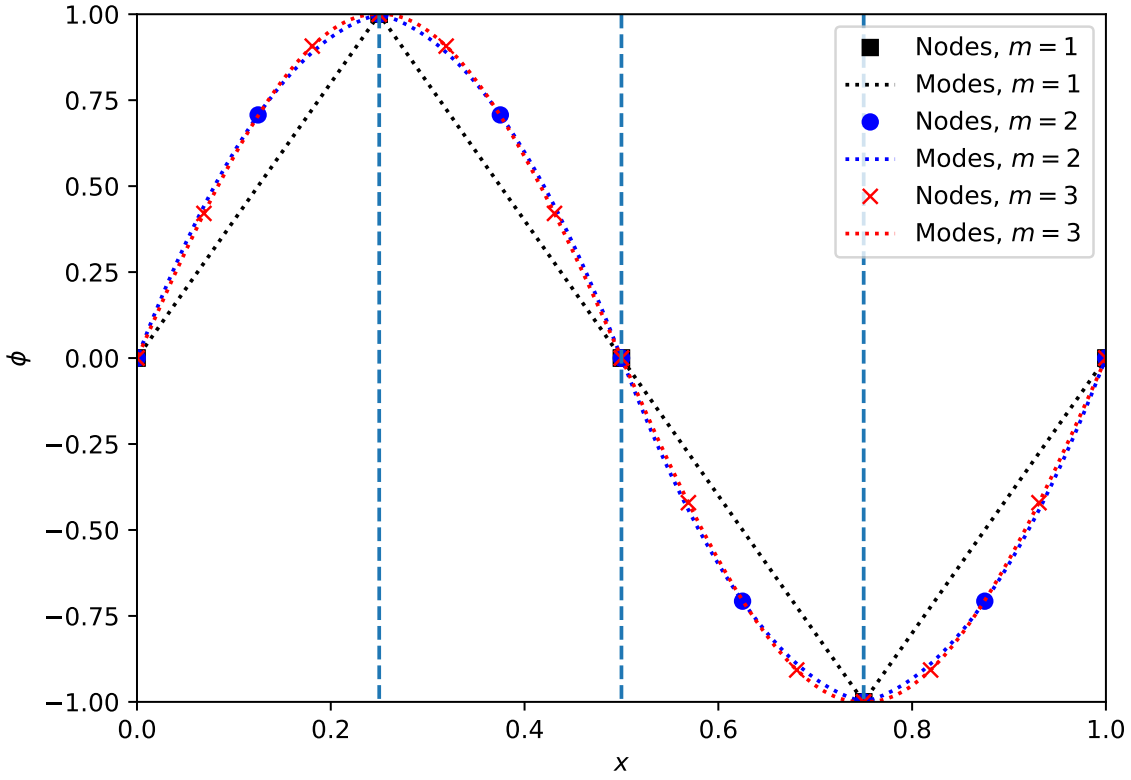

Figure 6.1: The grid for a Discontinuous Galerkin method is split into cells or elements as indicated by the vertical dashed lines – here there are only 4 cells. Within each cell the solution is represented by an m^{extth} order polynoial, as shown by the dashed lines. This representation is central to the modal DG method. Equivalent information can be stored at specific nodes, as shown by the markers. Note how the number and location of the nodes varies with m.

Figure (Figure 6.1) shows the nodes and modes for a sine wave represented by a Discontin-

uous Galerkin method on a grid with only 4 cells. We see how rapidly the representation appears to converge to the smooth sine wave with increasing m. Note also how the locations of the nodes varies with m, as the optimal nodes changes with the order of the method. However, in all cases there are nodes at the boundaries of each cell.

Exercise 6.1. Construct the Vandermonde matrix converting modal coefficients, based on orthonormal Legendre polynomials, to nodal coefficients, based on Gauss-Lobatto nodal points, on the interval $x \in [-1, 1]$. For example, for $m = 2$ the result is, to 4 significant figures,

$$V = \begin{pmatrix} 0.7071 & -1.225 & 1.581 \\ 0.7071 & 0 & -0.7906 \\ 0.7071 & 1.225 & 1.581 \end{pmatrix}. \tag{6.29}$$

Using the Vandermonde matrix and its inverse, check that you can convert from nodes to modes and vice versa. Check that the condition number grows slowly with m (roughly as $m^{1/2}$ for large m).

With these restrictions, we can now construct the nodal scheme. As noted above, this scheme remains a modal scheme as generally introduced in (Section 6.1), but the basis functions are the indicator polynomials $\ell_n(x)$. Thus the scheme can be written in the mass matrix form as in (Equation 6.13) of

$$\hat{M}\frac{\partial \hat{\mathbf{\Psi}}}{\partial t} + \hat{S}^T u \hat{\mathbf{\Psi}} = -\left[\boldsymbol{\phi}\mathbf{F}\right]_{-1}^{1}, \tag{6.30}$$

but now the two matrices are given by

$$\begin{aligned} \hat{M}_{ln} &= \int_{-1}^{1} \ell_l(x)\ell_n(x), \\ \hat{S}_{ln} &= \int_{-1}^{1} \ell_l(x)\frac{\partial \ell_n}{\partial x}(x). \end{aligned} \tag{6.31}$$

By using the Vandermonde matrix to link the nodal basis to an orthogonal basis such as the Legendre polynomials we can simplify the mass matrix to

$$\hat{M} = \left(\hat{V}\hat{V}^T\right)^{-1}. \tag{6.32}$$

The stiffness matrix can also be simplified, by re-writing $\frac{\partial \ell_n}{\partial x}(x)$ as an expansion in terms of $\ell_n(x)$. Defining the *differentiation matrix* $\hat{D}$ as

$$\hat{D}_{ln} = \left.\frac{\partial \ell_n}{\partial x}(x)\right|_{x=\xi_l} \tag{6.33}$$

we have $\frac{\partial \ell_n}{\partial x}(x) = \sum_k \hat{D}_{kn}\ell_k(x)$

$$\begin{aligned}
\hat{S}_{ln} &= \int_{-1}^{1} \ell_l(x)\frac{\partial \ell_n}{\partial x}(x) \\
&= \int_{-1}^{1} \ell_l(x)\sum_k \hat{D}_{kn}\ell_k(x) \\
&= \sum_k \left(\ell_l(x)\ell_k(x)\right)\hat{D}_{kn} \\
&= \hat{M}_{lk}\hat{D}_{kn}.
\end{aligned} \tag{6.34}$$

This shows that the stiffness matrix simplifies to

$$\hat{S} = \hat{M}\hat{D}. \tag{6.35}$$

Finally, using similar methods to the steps above, we can link the differentiation matrix back to the Vandermonde matrix, via

$$\hat{D} = \left(\frac{\partial \hat{V}}{\partial x}\right)\hat{V}^{-1}. \tag{6.36}$$

This is primarily useful when the modal function basis is a standard library function such as the (normalized) Legendre polynomials. This means that the basis functions and their derivatives, and hence the Vandermonde matrix and its derivatives, can be written solely in terms of library functions. For example, in Python the `numpy` package contains (in `numpy.polynomial.legendre`) the functions `legval` (which evaluates the Legendre polynomials), legder (which links the derivatives of the

`Legendre polynomials back to the Legendre polynomials themselves)`, and`legvander'` (which evaluates the Vandermonde matrix directly, but in un-normalized form).

There is one final step needed to construct the full scheme. So far, the method has been built assuming a single element with the coordinates $x \in [-1, 1]$. For most cases we will want to use a "small" number of modes, say $m \leq 5$, and split the domain into N elements, like the cells in a finite volume scheme. If we assume a general element has coordinates $x \in [x_{j-1/2}, x_{j+1/2}]$ with width Δx, then the *form* of the scheme remains the same:

$$M\frac{\partial \hat{\mathbf{\Psi}}}{\partial t} + S^T u \hat{\mathbf{\Psi}} = -\left[u\boldsymbol{\phi}\right]_{x_{j-1/2}}^{x_{j+1/2}}. \tag{6.37}$$

However, the change of coordinates needs to be factored in. We can see how this works by looking at the integral definitions, such as (Equation 6.31). We see that the mass matrix transforms as

$$M = \frac{\Delta x}{2}\hat{M}, \tag{6.38}$$

but that the stiffness matrix is unchanged.

Exercise 6.2. From the Vandermonde matrices constructed above, build the mass, differentiation and stiffness matrices $\hat{M}, \hat{D}, \hat{S}$, on the interval $x \in [-1, 1]$. For example, for $m = 2$ the results are, to 4 significant figures,

$$\begin{aligned}
\hat{M} &= \begin{pmatrix} 0.2667 & 0.1333 & -0.0667 \\ 0.1333 & 1.067 & 0.1333 \\ -0.06667 & 0.1333 & 0.2667 \end{pmatrix}, \\
\hat{D} &= \begin{pmatrix} -1.5 & 2 & -0.5 \\ -0.5 & 0 & 0.5 \\ 0.5 & -2 & 1.5 \end{pmatrix}, \\
\hat{S} &= \begin{pmatrix} -0.5 & 0.6667 & -0.1667 \\ -0.6667 & 0 & 0.6667 \\ 0.1667 & -0.6667 & 0.5 \end{pmatrix}.
\end{aligned} \tag{6.39}$$

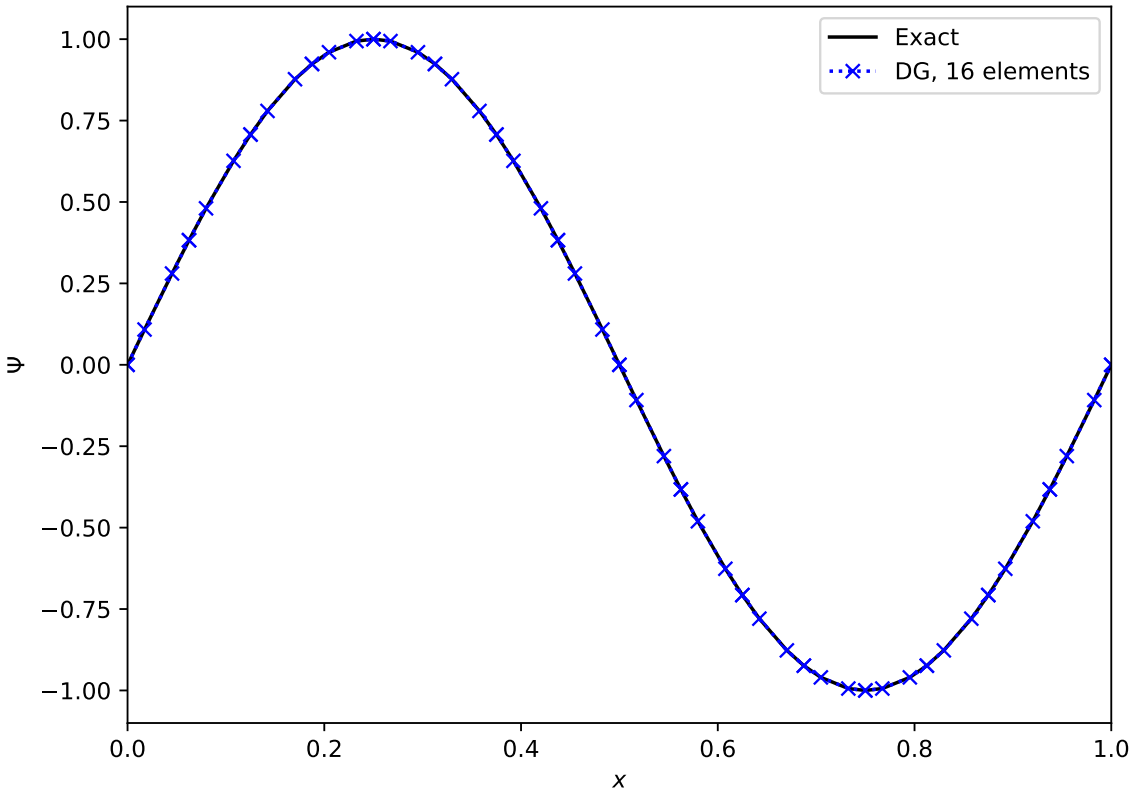

Figure 6.2: A Discontinuous Galerkin method with $m = 3$ and 16 elements applied to the advection equation, where a sine wave is advected once around the domain. Even at this low resolution the result is visually exact. The solutions are plotted at the nodal values, which are not evenly spaced.

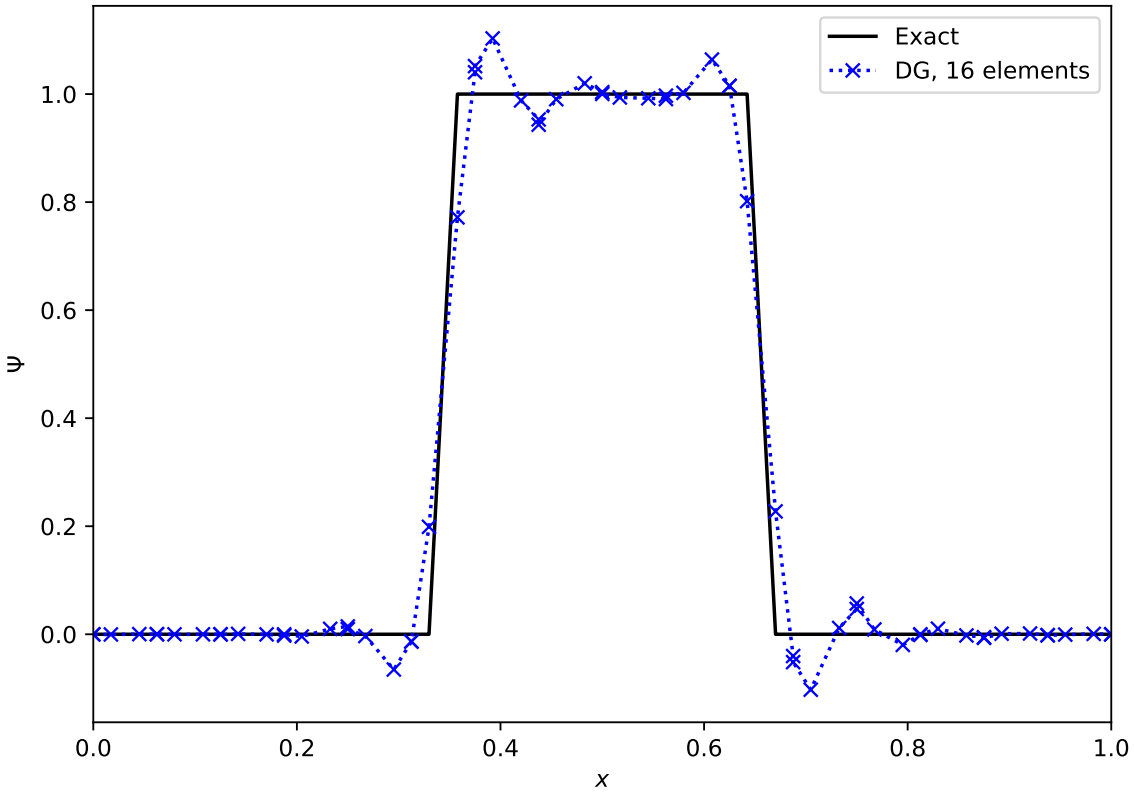

Figure 6.3: A Discontinuous Galerkin method with $m = 3$ and 16 elements applied to the advection equation, where a discontinuous top hat function is advected once around the domain. The expected Gibbs oscillations are seen.

By combining the nodal DG update described above with a time integrator we can look at the performance of the scheme. We need to take care in choosing the timestep. From the nodal point of view we can see that the width of the cell, Δx, is not going to be the limiting factor. Instead, the smallest distance between the (unequally spaced!) nodes is going to be crucial. General results (see e.g. (Hesthaven 2017)) suggest that reducing the timestep by a factor of $2m + 1$ is sufficient to ensure stability, but it does increase computational cost.

Figures (Figure 6.2) and (Figure 6.3) show the advection of two initial profiles one period around a periodic domain. In (Figure 6.2) we see the excellent performance when applied to a smooth profile. The method is essentially indistinguishable from the exact solution. However, in (Figure 6.3), we see that when the method is applied to a discontinuous initial profile then Gibbs oscillations result. The only "nice" feature of the Discontinuous Galerkin method here is that these oscillations are confined to the elements next to the discontinuities, and do not spread to cover the entire grid.

As with finite difference schemes, there are a range of modifications that can be made to limit or eliminate these oscillations. In Discontinuous Galerkin methods it is typical to do this in two steps: first, identify which elements need limiting, and second, modify the data in the required cells. The identification step can be done using the nodal values: construct limited slopes from cell average values and compare the predicted values at cell boundaries to the nodal values actually stored. The modification step can be done in many ways. A number are outlined in (Hesthaven 2017).

With smooth solutions we can check the convergence rate of the method. In (Figure 6.4) the smooth sine profile is again advected once around a periodic domain, using mode numbers $m = 1, \ldots, 4$, and checking convergence with the number of elements. This using a third order Runge Kutta method in time, and eventually the time integration error dominates over the spatial error.

```
/Users/ih3/opt/anaconda3/envs/mfc/lib/python3.12/site-packages/scipy/integrate/
  self._y, self.t = mth(self.f, self.jac or (lambda: None),
```

In (Figure 6.5) an 8^{th} order time integrator is used. This reduces the time integrator error far below what is needed, and we now see that every scheme converges at the expected rate. In more complex systems in multiple dimensions the error from the spatial terms will be much larger, and so lower order methods can be used without compromising the accuracy.

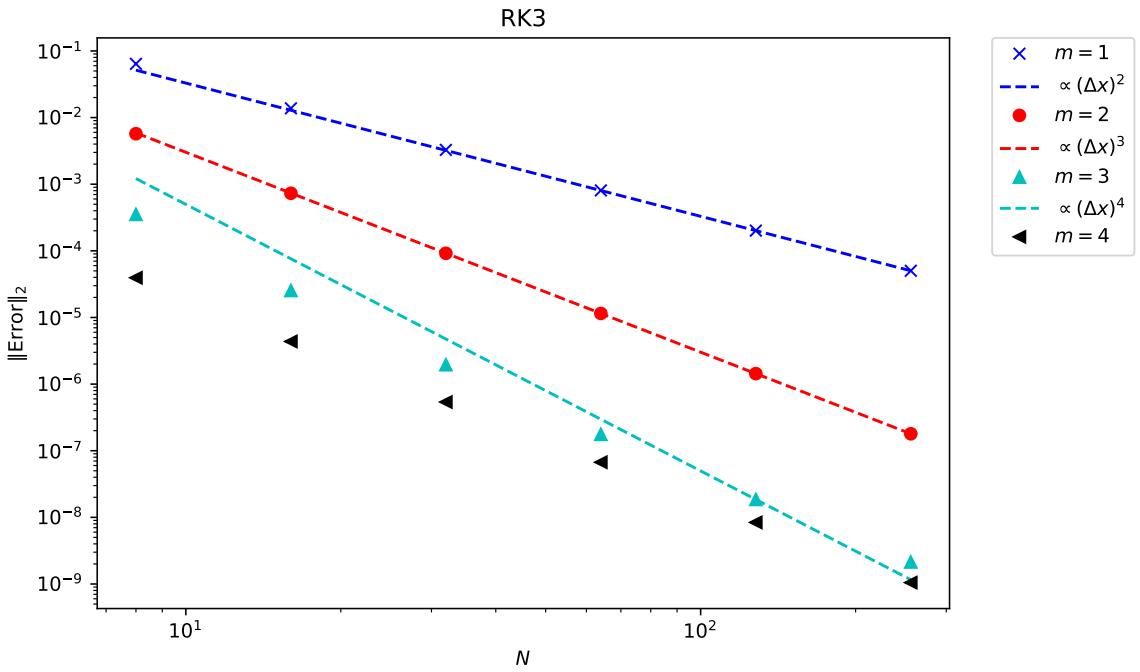

Figure 6.4: The convergence rate of the DG method applied to a sine wave. An explicit SSP third order Runge-Kutta time integrator is used. The expected convergence rate $(m + 1)$ is seen until the limits of the time integrator are reached.

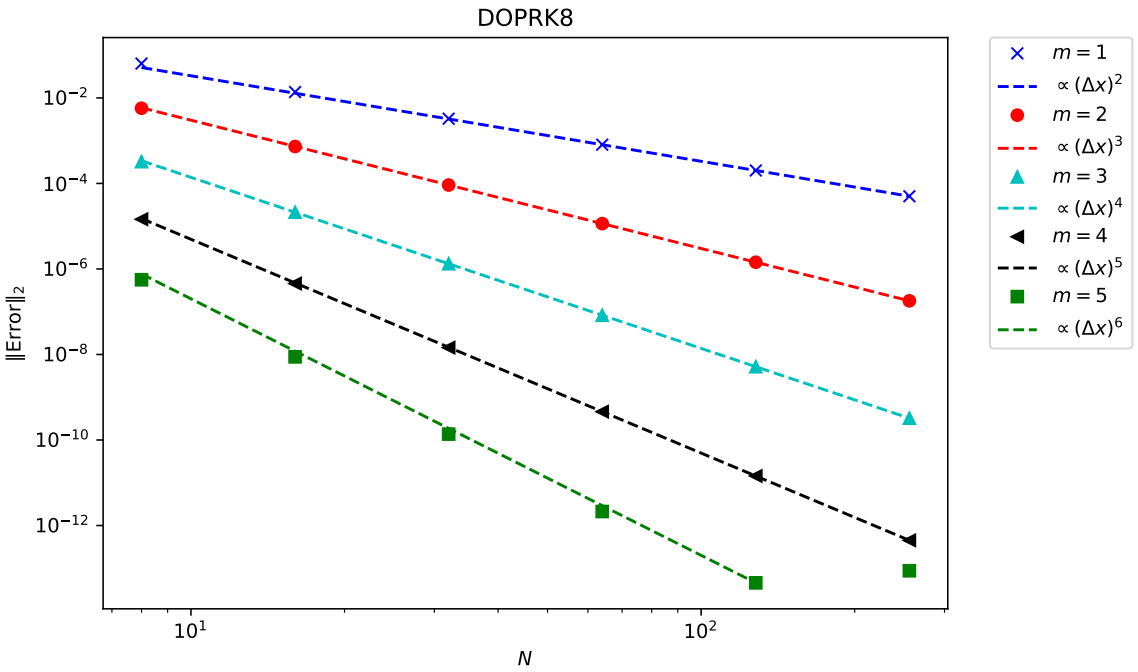

Figure 6.5: The convergence rate of the DG method applied to a sine wave. An eighth order Runge-Kutta time integrator is used. The expected convergence rate ($m + 1$) is seen even at high orders.

Exercise 6.3. Write a Python code to advect a profile around $x \in [0, 1]$ using the Discontinuous Galerkin method. A variety of Python packages can be used to facilitate this: `numpy` will solve linear systems and construct the matrices linked to Legendre polynomials, `scipy.integrate` will solve the ODE in time, and `quadpy` will construct the Gauss-Lobatto integration points.

Check that the solution converges as expected.

6.4 Discussion

In some ways Discontinuous Galerkin type methods seem a half-way-house between spectral methods and finite difference or finite volume methods. In principle the number of modes used within each element can be increased arbitrarily, giving the extremely rapid convergence of a spectral method. However, each element is linked to its neighbour, so there is still the communication with neighbouring points as in, for example, finite difference methods.

The key advantage of Discontinuous Galerkin methods comes with the latest computing hardware. These "Exascale" High Performance machines will rely on codes using very large numbers of relatively cheap, energy efficient individual computing cores (nearly always GPUs). This means the calculation must be performed in parallel across millions (or more) different compute cores. In this situation the limiting factor will be the communication with neighbouring points. This makes pure spectral methods totally impractical, and high accuracy finite difference methods (that have to communicate with many neighbours) will also not reach the performance expectations. As Discontinuous Galerkin methods only have to compute with *one* neighbouring element on each side, they minimise communication whilst giving high accuracy.

However, the stiffness and mass matrices involved in the update grow rapidly with the number of spatial dimensions and with the size of the system to solve. In addition, simple Discontinuous Galerkin methods struggle with steep gradients and discontinuities. The complexity and cost of making these methods practical means that they are - as yet - rarely used. Future computing hardware considerations may make them increasingly important.

References

Boyd, John P (2001). *Chebyshev and Fourier spectral methods*. Dover Books in Mathematics.

Durran, Dale R (2010). *Numerical methods for fluid dynamics: With applications to geophysics*. Vol. 32. Springer Science & Business Media.

Ferziger, Joel H and Milovan Perić (2002). *Computational methods for fluid dynamics*. Springer.

Ferziger, Joel H, Milovan Perić, and Robert L Street (2019). *Computational methods for fluid dynamics*. Springer.

Hesthaven, Jan S (2017). *Numerical methods for conservation laws: From analysis to algorithms*. SIAM.

Hughes, T.J.R. (2012). *The Finite Element Method: Linear Static and Dynamic Finite Element Analysis*. Dover Civil and Mechanical Engineering. Dover Publications. ISBN: 9780486135021.

LeVeque, Randall J (1992). *Numerical methods for conservation laws*. Vol. 214. Springer.

LeVeque, Randall J (2002). *Finite volume methods for hyperbolic problems*. Vol. 31. Cambridge University Press.

Ortega, James M and William G Poole (1981). *An introduction to numerical methods for differential equations*. Pitman London.

Sweby, Peter K (1984). "High resolution schemes using flux limiters for hyperbolic conservation laws". In: *SIAM journal on numerical analysis* 21.5, pp. 995–1011.

Toro, Eleuterio F (2013). *Riemann solvers and numerical methods for fluid dynamics: a practical introduction*. Springer Science & Business Media.

Trefethen, Lloyd Nicholas (1996). "Finite difference and spectral methods for ordinary and partial differential equations". In: URL: https://people.maths.ox.ac.uk/trefethen/pdetext.html.

A Background material

A series of standard results used without justification in these notes.

A.1 Taylor expansions

Given a function $f(x)$ with sufficient derivatives at a point X, the function can be represented as a polynomial using a *Taylor series* about the point X. There are multiple forms of interest.

The general form with remainder term obscured is

$$f(x) = f(X) + (X - x)f'(X) + \mathcal{O}\left((X - x)^2\right) . \tag{A.1}$$

The series expanded as a general polynomial is

$$f(x) = \sum_{k=0} \frac{(X - x)^k}{k!} f^{(k)}(X) \tag{A.2}$$

where the notation $f^{(k)}(X)$ corresponds to the k^{th} derivative of f evaluated at X.

For finite differencing it is useful to restrict to an evenly spaced grid $x_j = x_0 + j\,\Delta x$. Then, expanding about x_i, we have

$$f(x_j) = \sum_{k=0} \frac{(j - i)^k\,\Delta x^k}{k!} f^{(k)}(x_i) . \tag{A.3}$$

The most useful results, written out explicitly to low orders, are

$$f(x_{i\pm1}) = f(x_i) \pm \Delta x\, f'(x_i) + \frac{\Delta x^2}{2} f''(x_i) \pm \frac{\Delta x^3}{6} f'''(x_i) + \mathcal{O}(\Delta x^4)\,. \quad \text{(A.4)}$$

A.2 Series expansions

A.2.1 Fourier Series

An L_2 square integrable *periodic* function of one variable x defined on $[-\pi, \pi]$ can be represented by the *complex Fourier series*

$$f(x) \sim \sum_{n=-\infty}^{\infty} a_n \exp(inx) \quad \text{(A.5)}$$

where the (complex) coefficients a_n are given by

$$a_n = \frac{1}{2\pi} \int_{-\pi}^{\pi} f(x) \exp(-inx)\, \mathrm{d}x\,. \quad \text{(A.6)}$$

When the function depends on more than one variable the series expansion can be performed separately for each, or the dependency on the additional variables can be retained in the coefficient. Typically spatial dependence is expanded in a series and time dependence is retained in the coefficients, for example as

$$f(x, y, t) \sim \sum_{n_x=-\infty}^{\infty} \sum_{n_y=-\infty}^{\infty} a_n(t) \exp(in_x x) \exp(in_y y)\,. \quad \text{(A.7)}$$

A.2.2 Eigenfunctions

One key use for series expansions is in differential equations. Each individual mode $f_n(x) = \exp(inx)$ is an eigenfunction of both the first and second derivative operators,

$$\begin{aligned} \partial_x f_n(x) &= in \exp(inx) \\ &= in f_n(x)\,, \\ \partial_{xx} f_n(x) &= -n^2 \exp(inx) \\ &= -n^2 f_n(x)\,. \end{aligned} \tag{A.8}$$

When problems are linear the action of the differential operator can be studied as an algebraic operation on the individual modes.

This generalizes to more complex problems. Sturm-Liouville theory looks at problems where the spatial differential operator has the form

$$\mathcal{L}(y) = \frac{\mathrm{d}}{\mathrm{d}x}\left[p(x)\frac{\mathrm{d}y}{\mathrm{d}x}\right] + q(x)y\,. \tag{A.9}$$

Here p, q are known functions. The eigenfunctions of this operator obey

$$\mathcal{L}(y_n) = -\lambda_n w(x) y_n \tag{A.10}$$

where w is a known *weighting* function. With these eigenfunctions, an arbitrary function can be represented as

$$f(x) \sim \sum_{n=1}^{\infty} a_n y_n(x) \tag{A.11}$$

where the coefficients a_n can be explicitly computed as

$$a_n = \int_{-\pi}^{\pi} f(x) y_n(x) w(x)\,\mathrm{d}x\,. \tag{A.12}$$

This is an expansion in terms of *orthogonal functions*, using that

$$\int_{-\pi}^{\pi} y_m(x) y_n(x) w(x)\,\mathrm{d}x = \delta_{mn}\,. \tag{A.13}$$

A.2.3 Key examples

In the Fourier Series case $p(x) = 1$, $q(x) = 0$, and $w(x) = 2\pi$. The eigenvalues are $\lambda_n = n^2$.

In the Legendre equation case (where the domain is conventionally $x \in [-1, 1]$) $p(x) = 1 - x^2$, $q(x) = 0$, and $w(x) = (2n + 1)/2$. The eigenvalues are $\lambda_n = n(n + 1)$.

Spherical harmonics $Y_\ell^m(\theta, \varphi)$ are eigenfunctions of the covariant second derivative operator on a spherical shell with radius r, so that

$$r^2 \nabla^2 Y_\ell^m = -\ell(\ell + 1) Y_\ell^m \,. \tag{A.14}$$

Spherical harmonics look like a Fourier mode in φ multiplied by an eigenfunction of the (associated) Legendre equation in θ.

B Order of accuracy

We here give precise definitions of the order of accuracy of a numerical methods, and a sketch proof of the link between the local error and the global error.

The notation here will be slightly different (and annoyingly more pedantic) than other sections.

B.1 Notation

Assume that we are given a PDE with appropriate initial boundary conditions. We denote the solution to the PDE, using initial data at $t = 0$ of $y_0(x)$, as

$$y\left(x, t | y_0(x; t = 0)\right) . \tag{B.1}$$

We will assume that the PDE is well posed, in the sense that a small perturbation in the initial conditions is bounded. That is, we assume that for all small numbers ϵ and for all perturbations δ there exists a K independent of time such that

$$\|y(x, t | y_0(x; t = 0) + \delta) - y(x, t | y_0(x; t = 0))\| \leq \delta e^{Kt} . \tag{B.2}$$

Next we assume that we are constructing a numerical approximation to the solution $\mathsf{y}(x, t)$. It is possible that this solution is only constructed at a finite number of points (for example, a finite difference method on a grid), or it might be constructed at finitely many time intervals but be computable everywhere in space (as in a spectral or finite element method). We denote the numerical method's update scheme by

$$\mathsf{y}\left(x, t^{n+1} | \left\{\mathsf{y}(x, t^n)\right\}\right) = \mathcal{F}\left(\left\{\mathsf{y}(x, t^n)\right\}\right) . \tag{B.3}$$

Even an implicit method can (in principle) be written in this form, but it is simplest to think of this as an explicit method.

B.2 Local truncation error

The easiest analysis of the accuracy of a method looks at the error introduced over a single step, assuming that the input data is *exact*. This is the *Local Truncation Error* ϵ_{n+1}, given by

$$\epsilon_{n+1} = \left\| y(x, t^{n+1} | y_0(x; t=0)) - \mathsf{y}\left(x, t^{n+1} | \left\{ y(x, t^n | y_0(x; t=0)) \right\}\right) \right\| . \quad \text{(B.4)}$$

When computing the local truncation error it is usual to use a series expansion, such as a Taylor series expansion, in terms of small quantities controllable by the numerical method. Typical examples would be the grid spacings $\Delta t,\ \Delta x$ in finite difference methods, and the inverse of the number of degrees of freedom N^{-1} in spectral or finite element methods. We then assume that only the first term in the expansion survives, giving (for example)

$$\epsilon_{n+1} \propto \Delta t^{p+1} . \quad \text{(B.5)}$$

This makes the *local order of accuracy* be $p+1$. We typically assume that p is independent of time (and hence independent of n), and write the local truncation error as ϵ by maximising over all time.

B.3 Global truncation error

Local truncation error is the easiest thing to analyse, but not what we want to compute. We want to relate the error at the *end* of the simulation, when it has taken many steps, to the parameters (such as grid spacing). This is the *Global Truncation Error*

$$\mathcal{E}_T = \left\| y(x, T | y_0(x; t=0)) - \mathsf{y}\left(x, T | \left\{ \mathsf{y}(x, T - \Delta t) \right\}\right) \right\| . \quad \text{(B.6)}$$

This cannot be directly computed using the local truncation error, as the numerical solution at time T is not computed from the exact solution at the previous timestep, but from the approximate numerical solution at that point. However, we can add and subtract the exact solution with *different* initial conditions, at the time $T - \Delta t$. Thus

$$\begin{aligned}
\mathcal{E}_T &= \|y(x,T|y_0(x;t=0)) - y(x,T|\mathsf{y}(x;T-\Delta t)) + \\
&\quad y(x,T|\mathsf{y}(x;T-\Delta t)) - \mathsf{y}\,(x,T|\,\{\mathsf{y}(x,T-\Delta t)\})\| \\
&\leq \|y(x,T|y_0(x;t=0)) - y(x,T|\mathsf{y}(x;T-\Delta t))\| + \\
&\quad \|y(x,T|\mathsf{y}(x;T-\Delta t)) - \mathsf{y}\,(x,T|\,\{\mathsf{y}(x,T-\Delta t)\})\| \\
&\leq \|y(x,T|y_0(x;t=0)) - y(x,T|\mathsf{y}(x;T-\Delta t))\| + \epsilon \\
&\leq \|y(x,T|y(x;t=T-\Delta t)) - y(x,T|\mathsf{y}(x;T-\Delta t))\| + \epsilon \\
&\leq \mathcal{E}_{T-\Delta t} e^{K\,\Delta t} + \epsilon\,.
\end{aligned} \tag{B.7}$$

Here we have first used the definition of the local truncation error, then the definition of well-posedness.

Using this recursion relation, combined with the fact that the global truncation error after a single step is precisely the local truncation error, we find that

$$\mathcal{E}_T \propto \frac{\epsilon}{\Delta t}\,. \tag{B.8}$$

This result says that if the local truncation error has order of accuracy $p+1$, then the global truncation error has order of accuracy p. Loosely, this can be understood as the finite time T requires $\sim \Delta t^{-1}$ timesteps to reach, and each of those steps introduces an error $\sim \epsilon$.

C Lax Equivalence Theorem

The Lax Equivalence Theorem is usually phrased roughly as

> A numerical scheme converges to the true solution if, and only if, it is consistent and stable.

There are a number of different versions of this theorem. The main result is the Lax-Richtmyer case which holds for *linear* numerical methods applied to *well-posed*, *linear* partial differential equations with periodic boundaries (or infinite domains where the data has compact support). There are extensions (for example, the Lax-Wendroff case) which at least partially lift these restrictions, but well-posedness remains essential and results in the nonlinear case are limited.

We sketch the Lax-Richtmyer case.

As a motivating example, think of the advection equation

$$\partial_t \phi + v \partial_x \phi = 0 \tag{C.1}$$

with $v > 0$ constant, approximated by the standard FTBS scheme

$$\phi_i^{n+1} = \phi_i^n + \frac{v\,\Delta t}{\Delta x}\left(\phi_i^n - \phi_{i-1}^n\right) = \mathcal{L}_{\Delta t}\left(\{\phi_i^n\}\right)_i = \mathcal{L}_{\Delta t}\left(\boldsymbol{\phi}^n\right)_i \,. \tag{C.2}$$

We have introduced the operator notation $\mathcal{L}_{\Delta t}$. The discrete solution at time $t^n = n\,\Delta t$ is given by the vector $\boldsymbol{\phi}^n$. The discrete solution at the next timestep, $\boldsymbol{\phi}^{n+1}$, is given by applying the operator to the solution at the current timestep,

$$\boldsymbol{\phi}^{n+1} = \mathcal{L}_{\Delta t}\boldsymbol{\phi}^n. \tag{C.3}$$

This is therefore a *linear map* on a vector space, whose vectors are (potential) discrete solutions to the PDE. Note that the map depends only on the timestep Δx, as it is assumed that the grid (here Δx, but it extends to arbitrary dimensions and unstructured grids) depends directly, if implicitly, on Δx.

C.1 Banach spaces

A *Banach space* is a complete normed vector space. For our purposes, this will be the space of discrete approximations to the PDE. The norm will be some measure of the "size" of the solution, such as the infinity norm

$$\|\boldsymbol{\phi}\|_\infty = \max_i \phi_i \,, \tag{C.4}$$

or the 2-norm

$$\|\boldsymbol{\phi}\|_2 = \sum_i \sqrt{(\phi_i)^2 \, \Delta x} \,. \tag{C.5}$$

This allows us to formally, but abstractly, state our two key conditions and our goal.

C.2 Consistency

A numerical approximation is *consistent* if it correctly represents the PDE in the continuum limit. Formally, let ϕ be the exact solution to the PDE, sampled onto the discrete grid as appropriate. Also let $\mathcal{L}$ be the formal operator that evolves the exact solution forward in time. Therefore

$$\mathcal{L}(T)\boldsymbol{\phi}(0) = \boldsymbol{\phi}(T) \,, \tag{C.6}$$

and

$$\mathcal{L}(\Delta t)\boldsymbol{\phi}(T) = \boldsymbol{\phi}(T + \Delta t) \,. \tag{C.7}$$

Then the numerical scheme is consistent if

$$\lim_{\Delta t \to 0} \|(\mathcal{L}(\Delta t) - \mathcal{L}_{\Delta t})\, \boldsymbol{\phi}(T)\| \to 0\,. \tag{C.8}$$

For later purposes we write that the scheme is consistent if there exists a constant $K \in [0, \infty)$ such that

$$\|(\mathcal{L}(\Delta t) - \mathcal{L}_{\Delta t})\, \boldsymbol{\phi}(T)\| \leq K_C\, \Delta t \tag{C.9}$$

and K_C is independent of Δt.

C.3 Stability

A numerical approximation is *stable* if it does not blow up "too fast". This needs some care, as it is possible that the exact solution to the PDE grows quickly, even exponentially or instantaneously in special cases. Therefore the most general stability criteria that we can work with is to state that the scheme is stable at time $T = N\, \Delta t$ if

$$\left\|(\mathcal{L}_{\Delta t})^N\, \boldsymbol{\phi}(0)\right\| \leq K_T\, \|\boldsymbol{\phi}(0)\| + D\, \Delta t\,. \tag{C.10}$$

Again, the constants $K_T, D \in [0, \infty)$ and must be independent of Δt (although they can depend on the finite time T). The term involving K_T ensures that the numerical approximation does not grow too fast compared to the true solution, but does allow for *bounded* growth. The term involving D allows for purely numerical growth even if the true solution is bounded, but ensures that any such growth converges to zero in the continuum limit.

C.4 Convergence

Our goal, via the Lax Equivalence Theorem, is to show that the continuum limit of the numerical scheme is the true solution. This means the numerical scheme converges. The scheme is said to be *convergent* at time $T = N\, \Delta t$ if

$$\lim_{\Delta t \to 0} \left\| \left(\mathcal{L}(T) - \left(\mathcal{L}_{\Delta t}\right)^N \right) \boldsymbol{\phi}(0) \right\| \to 0 \,. \tag{C.11}$$

More precisely, the scheme converges if there exists a constant $K \in [0, \infty)$ such that

$$\left\| \left(\mathcal{L}(T) - \left(\mathcal{L}_{\Delta t}\right)^N \right) \boldsymbol{\phi}(0) \right\| \leq K \, \Delta t. \tag{C.12}$$

Again, the constant K must be independent of Δt.

The key point that distinguishes convergence from consistency is the number of steps taken by the discrete scheme. To be consistent the error needs to converge over a single discrete step. For convergence the error needs to converge after N steps at fixed T, and in the continuum limit $\Delta t \to 0$ this means $N \to \infty$.

C.5 Lax Equivalence Theorem

With the formal machinery set up, the Lax theorem follows.

Theorem C.1 (Lax Equivalence Theorem)**.** *Given a well-posed, linear, initial value problem, and a consistent numerical scheme to that problem, stability is necessary and sufficient for convergence.*

Proof. First note that

$$\begin{aligned} \left\| \left(\mathcal{L}(T) - \left(\mathcal{L}_{\Delta t}\right)^N \right) \boldsymbol{\phi}(0) \right\| &= \| (\mathcal{L}(\Delta t) - \mathcal{L}_{\Delta t}) \, \boldsymbol{\phi}(T - \Delta t) + \\ &\qquad \mathcal{L}_{\Delta t} \boldsymbol{\phi}(T - \Delta t) - \left(\mathcal{L}_{\Delta t}\right)^N \boldsymbol{\phi}(0) \| \\ &\leq K_C \, \Delta t + \left\| \mathcal{L}_{\Delta t} \left(\mathcal{L}(T - \Delta t) - \left(\mathcal{L}_{\Delta t}\right)^{N-1} \right) \boldsymbol{\phi}(0) \right\| \\ &\leq K_C \, \Delta t + K_T \left\| \left(\mathcal{L}(T - \Delta t) - \left(\mathcal{L}_{\Delta t}\right)^{N-1} \right) \boldsymbol{\phi}(0) \right\| + \\ &\quad D \, \Delta t. \end{aligned} \tag{C.13}$$

By induction, and using that the initial error is zero, this allows us to write all terms as $\propto \Delta t$ and hence say that consistency plus stability imply convergence.

In the other direction, it is immediate that convergence implies stability. □