M345A47 Finite Elements: Analysis and Implementation

Edition 2018.0

David A. Ham and Colin J. Cotter

Jan 10, 2019

CONTENTS

THE IMPLEMENTATION EXERCISE

The object of the implementation exercise is to gain an understanding of the finite element method by producing a working one and two dimensional finite element solver library. Along the way you will have the opportunity to pick up valuable scientific computing skills in coding, software engineering and rigorous testing.

There will be no conventional lectures for this part of the module. Instead, there will be twice weekly 1 hour computer lab sessions during the term. Some of this time will involve explanations at the board, but much of the time will be an opportunity to develop your finite element implementation and receive help on how to do so.

0.1 Formalities and marking scheme

The implementation exercise is due at the end of term. That is, by 1600 on Friday 22 March. You must submit your work uploading the git commit code on Blackboard. You can convenently find this code on the commits page for your repository on github. For the avoidance of doubt, the commit you submit must date from before the deadline!

The marking scheme will be as follows:

First/distinction (70-100) All parts of the implementation are correct and all tests pass. The code style is always very clear and the implementation of every exercise is transparent and elegant.

Upper second/merit (60-70) The implementation is correct but let down somewhat by poor coding style. Alternatively, submissions which are correct and well written up to and including solving the Helmholtz problem but which do not include a correct solution to boundary conditions will earn an upper second.

Lower second/pass (50-60) There are significant failings in the implementation resulting in many test failures, and/or the coding style is sufficiently poor that the code is hard to understand.

Fail (0-50) The implementation is substantially incomplete. Correct implementations may have been provided for some of the earlier exercises but the more advanced parts of the implementation exercise have not been attempted or do not work.

0.2 Extension (mastery) exercise

Fourth year and masters students must also complete the mastery exercise, which will be issued half way through the term. This will be worth 20% of the implementation exercise marks and will be marked on the same scheme as above.

0.3 Obtaining the skeleton code

This section assumes you've already done the *Git tutorial*.

0.3.1 Setting up your repository

We're using a tool called GitHub classroom to automate the creation of your copies of the repository. To create your repository, click here.

0.3.2 Cloning a local copy

At the command line on your working machine type:

```
git clone <url> finite-element-course
```

Substituting your git repository url for <url>. Your git repository url can be found by clicking on *clone or download* at the top right of your repository page on GitHub.

0.3.3 Setting up your venv

We're going to use a Python venv. This is a private Python environment in which we'll install the packages we need, including our own implementation exercise. This minimises interference between this project and anything else which might be using Python on the system. We can run a script from the git repository to make the venv:

```
./finite-element-course/scripts/fe_install_venv venv
```

This has to install several packages in the venv, so it might take a few minutes to run.

On Windows, the set of commands is somewhat different. In this case you would run:

```
./finite-element-course/scripts/fe_install_venv_win venv
```

0.3.4 Activating your venv

Every time you want to work on the implementation exercise, you need to activate the venv. On Linux or Mac do this with:

```
source venv/bin/activate
```

while on Windows the command is:

```
source venv/Scripts/activate
```

Obviously if you are typing this in a directory other than the one containing the venv, you need to modify the path accordingly.

0.3.5 Setting up an implementation branch

We'll keep the master branch of your repository in the original condition so we can compare to it later, and collect any updates which occur during the term. Instead, we'll create an implementation branch to actually work on:

```
cd finite-element-course
git checkout -b implementation
```

Your working directory is now a current checkout of your implementation branch. You'll also want to push this branch to GitHub:

```
git push --set-upstream origin implementation
```

0.3.6 Watching for updates and issues

You should make sure you are notified of all updates on the main repository and all issues anyone raises. For this, you should navigate to the main repository. On the top right there is an eye icon. Select the drop-down box and switch to `watching`.

0.3.7 Updating your fork

When you see that the main repository has been updated, you'll need to update your repository to incorporate those changes. *Just this once*, you need to tell your local git repo about the main repository:

```
git remote add upstream https://github.com/finite-element/finite-element-course.git
```

Now, *every time* you want to update you do the following:

1. Make sure you have commited all your local changes **and** pushed them to GitHub.
2. Execute the following commands:

```
git checkout master           # Switch to the master branch.
git pull upstream master      # Update from the main repository.
git push                      # Push the updated master branch to GitHub.
git checkout implementation   # Switch back to the implementation branch.
git merge master              # Merge the new changes from master into␣
↪implementation.
git push                      # Push the updated implementation branch to␣
↪GitHub.
```

0.4 Skeleton code documentation

There is web documentation for the complete fe_utils. There is also an alphabetical index and a search page.

0.5 How to do the implementation exercises

The implementation exercises build up a finite element library from its component parts. Quite a lot of the coding infrastructure you will need is provided already. Your task is to write the crucial mathematical operations at key points. The mathematical operations required are described on this website, interspersed with exercises which require you to implement and test parts of the mathematics.

The code on which you will build is in the `fe_utils` directory of your repository. The code has embedded documentation which is used to build the fe_utils web documentation.

As you do the exercises, **commit your code** to your repository. This will build up your finite element library. You should commit code early and often - small commits are easier to understand and debug than large ones. **Never** commit back to the `master` branch of your fork, that should always remain a clean copy of the main repository.

0.6 Pull requests for feedback

There will be a formal opportunity to recieve feedback on your code progress twice during the term. To take part, you should set up a pull request from your `implementation` branch to the `master` branch of your repository. This will enable the lecturer to write line by line comments on your code.

0.6.1 Creating your pull request

1. Click on the `New pull request` button at the top of your repository page on GitHub.
2. Make sure **left** dropdown box ("base") is set to `master`.
3. Make sure **right** dropdown box ("compare") is set to `implementation`.
4. Type a suitable title in the title box. For example `Request for feedback 30/1/19`.

5. If you have any comments you would like to pass on to the lecturer (for example questions about how you should have done a particular exercise) then type these in the `Description` box.
6. Click `Create pull request`.

0.7 Testing your work

As you complete the exercises, there will often be test scripts which exercise the code you have just written. These are located in the `test` directory and employ the pytest testing framework. You run the tests with:

```
py.test test_script.py
```

from the bash command line, replacing `test_script.py` with the appropriate test file name. The `-x` option to `py.test` will cause the test to stop at the first failure it finds, which is often the best place to start fixing a problem. For those familiar with debuggers, the `--pdb` option will drop you into the Python debugger at the first error.

You can also run all the tests by running `py.test` on the tests directory. This works particularly well with the -x option, resulting in the tests being run in course order and stopping at the first failing test:

```
py.test -x tests/
```

0.8 Coding style and commenting

Computer code is not just functional, it also conveys information to the reader. It is important to write clear, intelligible code. **The readability and clarity of your code will count for marks**.

The Python community has agreed standards for coding, which are documented in PEP8. There are programs and editor modes which can help you with this. The skeleton implementation follows PEP8 quite closely. You are encouraged, especially if you are a more experienced programmer, to follow PEP8 in your implementation. However nobody is going to lose marks for PEP8 failures.

0.9 Tips and tricks for the implementation exercise

Work from the documentation. The notes, and particularly the exercise specifications, contain important information about how and what to implement. If you just read the source code then you will miss out on important information.

Read the hints The pink sections in the notes starting with a lightbulb are hints. Usually they contain suggestions about how to go about writing your answer, or suggest Python functions which you might find useful.

Don't forget the 1D case Your finite element library needs to work in one and two dimensions.

Return a `numpy.array()` Many of the functions you have to write return arrays. Make sure you actually return an array and not a list (it's usually fine to build the answer as a list, but convert it to an array before you return it).

CHAPTER

ONE

NUMERICAL QUADRATURE

The core computational operation with which we are concerned in the finite element method is the integration of a function over a known reference element. It's no big surprise, therefore, that this operation will be at the heart of our finite element implementation.

The usual way to efficiently evaluate arbitrary integrals numerically is numerical quadrature. This basic idea will already be familiar to you from undergraduate maths (or maybe even high school calculus) as it's the generalisation of the trapezoidal rule and Simpson's rule for integration.

The core idea of quadrature is that the integral of a function $f(X)$ over an element e can be approximated as a weighted sum of function values evaluated at particular points:

$$\int_e f(X) = \sum_q f(X_q) w_q + O(h^n) \tag{1.1}$$

we term the set $\{X_q\}$ the set of *quadrature points* and the corresponding set $\{w_q\}$ the set of *quadrature weights*. A set of quadrature points and their corresponding quadrature weights together comprise a *quadrature rule* for e. For an arbitrary function f, quadrature is only an approximation to the integral. The global truncation error in this approximation is invariably of the form $O(h^n)$ where h is the diameter of the element.

If f is a polynomial in X with degree p such that $p \leq n - 2$ then it is easy to show that integration using a quadrature rule of degree n results in exactly zero error.

Definition 1 *The* degree of precision *of a quadrature rule is the largest p such that the quadrature rule integrates all polynomials of degree p without error.*

1.1 Exact and incomplete quadrature

In the finite element method, integrands are very frequently polynomial. If the quadrature rule employed for a particular interval has a sufficiently high degree of precision such that there is no quadrature error in the integration, we refer to the quadrature as *exact* or *complete*. In any other case we refer to the quadrature as *incomplete*.

Typically, higher degree quadrature rules have more quadrature points than lower degree rules. This results in a trade-off between the accuracy of the quadrature rule and the number of function evaluations, and hence the computational cost, of an integration using that rule. Complete quadrature results in lower errors, but if the error due to incomplete quadrature is small compared with other errors in the simulation, particularly compared with the discretisation error, then incomplete quadrature may be advantageous.

1.2 Examples in one dimension

We noted above that a few one dimensional quadrature rules are commonly taught in introductory integration courses. The first of these is the midpoint rule:

$$\int_0^h f(X)\mathrm{d}X = hf(0.5h) + O(h^3) \tag{1.2}$$

In other words, an approximation to the integral of f over an interval can be calculated by multiplying the value of f at the mid-point of the interval by the length of the interval. This amounts to approximating the function over the integral by a constant value.

If we improve our approximation of f to a straight line over the interval, then we arrive at the trapezoidal (or trapezium) rule:

$$\int_0^h f(X)\mathrm{d}X = \frac{h}{2}f(0) + \frac{h}{2}f(h) + O(h^4) \tag{1.3}$$

while if we employ a quadratic function then we arrive at Simpson's rule:

$$\int_0^h f(X)\mathrm{d}X = \frac{h}{6}f(0) + \frac{2h}{3}f\left(\frac{h}{2}\right) + \frac{h}{6}f(h) + O(h^5) \tag{1.4}$$

1.3 Reference elements

As a practical matter, we wish to write down quadrature rules as arrays of numbers, independent of h. In order to achieve this, we will write the quadrature rules for a single, *reference element*. When we wish to actually integrate a function over cell, we will change coordinates to the reference cell. We will return to the mechanics of this process later, but for now it means that we need only consider quadrature rules on the reference cells we choose.

A commonly employed one dimensional reference cell is the unit interval $[0, 1]$, and that is the one we shall adopt here (the other popular alternative is the interval $[-1, 1]$, which some prefer due to its symmetry about the origin).

In two dimensions, the cells employed most commonly are triangles and quadrilaterals. For simplicity, in this course we will only consider implementing the finite element method on triangles. The choice of a reference interval implies a natural choice of reference triangle. For the unit interval the natural correspondence is with the triangle with vertices $[(0, 0), (1, 0), (0, 1)]$, though different choices of vertex numbering are possible.

1.4 Python implementations of reference elements

The `ReferenceCell` class provides Python objects encoding the geometry and topology of the reference cell. At this stage, the relevant information is the dimension of the reference cell and the list of vertices. The topology will become important when we consider *meshes*. The reference cells we will require for this course are the `ReferenceInterval` and `ReferenceTriangle`.

1.5 Quadrature rules on reference elements

Having adopted a convention for the reference element, we can simply express quadrature rules as lists of quadrature points with corresponding quadrature weights. For example Simpson's rule becomes:

$$\begin{aligned} w &= \left[\frac{1}{6}, \frac{2}{3}, \frac{1}{6}\right] \\ X &= [(0), (0.5), (1)]\,. \end{aligned} \tag{1.5}$$

We choose to write the quadrature points as 1-tuples for consistency with the n-dimensional case, in which the points will be n-tuples.

The lowest order quadrature rule on the reference triangle is a single point quadrature:

$$\begin{aligned} w &= \left[\frac{1}{2}\right] \\ X &= \left[\left(\frac{1}{3}, \frac{1}{3}\right)\right] \end{aligned} \tag{1.6}$$

This rule has a degree of precision of 1.

Hint: The weights of a quadrature rule always sum to the volume of the reference element. Why is this?

1.6 Legendre-Gauß quadrature in one dimension

The finite element method will result in integrands of different polynomial degrees, so it is convenient if we have access to quadrature rules of arbitrary degree on demand. In one dimension the Legendre-Gauß quadrature rules are a family of rules of arbitrary precision which we can employ for this purpose. Helpfully, numpy provides an implementation which we are able to adopt. The Legendre-Gauß quadrature rules are usually defined for the interval $[-1, 1]$ so we need to change coordinates in order to arrive at a quadrature rule for our reference interval:

$$\begin{aligned} X_q &= \frac{X'_q + 1}{2} \\ w_q &= \frac{w'_q}{2} \end{aligned} \tag{1.7}$$

where $(\{X'_q\}, \{w'_q\})$ is the quadrature rule on the interval $[-1, 1]$ and $(\{X_q\}, \{w_q\})$ is the rule on the unit interval.

Legendre-Gauß quadrature on the interval is optimal in the sense that it uses the minimum possible number of points for each degree of precision.

1.7 Extending Legendre-Gauß quadrature to two dimensions

We can form a unit square by taking the Cartesian product of two unit intervals: $(0, 1) \otimes (0, 1)$. Similarly, we can form a quadrature rule on a unit square by taking the product of two interval quadrature rules:

$$\begin{aligned} X_{\text{sq}} &= \{(x_p, x_q) \mid x_p, x_q \in X\} \\ w_{\text{sq}} &= \{w_p w_q \mid w_p, w_q \in w\} \end{aligned} \tag{1.8}$$

where (X, w) is an interval quadrature rule. Furthermore, the degree of accuracy of $(X_{\text{sq}}, w_{\text{sq}})$ will be the same as that of the one-dimensional rule.

However, we need a quadrature rule for the unit triangle. We can achieve this by treating the triangle as a square with a zero length edge. The Duffy transform maps the unit square to the unit triangle:

$$(x_{\text{tri}},\ y_{\text{tri}}) = (x_{\text{sq}},\ y_{\text{sq}}(1 - x_{\text{sq}})) \tag{1.9}$$

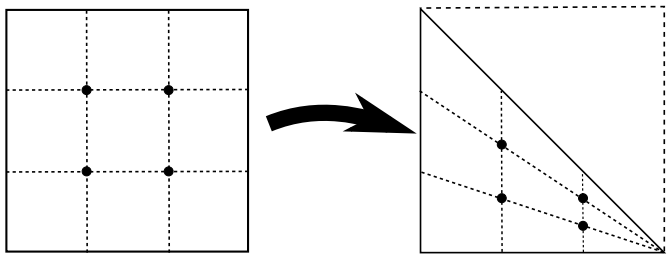

Fig. 1.1: The Duffy transform maps a square to a triangle by collapsing one side.

By composing the Duffy transform with (1.8) we can arrive at a quadrature rule for the triangle:

$$\begin{aligned} X_{\text{tri}} &= \{(x_p, x_q(1 - x_p)) \mid x_p \in X_h, x_q \in X_v\} \\ w_{\text{tri}} &= \{w_p w_q (1 - x_p) \mid w_p \in w_h, w_q \in w_v\} \end{aligned} \tag{1.10}$$

where (X_v, w_v) is a reference interval quadrature rule with degree of precision n and (X_h, w_h) is a reference interval quadrature rule with degree of precision $n+1$. The combined quadrature rule $(X_{\text{tri}}, w_{\text{tri}})$ will then be n. The additional degree of precision required for (X_h, w_h) is because the Duffy transform effectively increases the polynomial degree of the integrand by one.

1.8 Implementing quadrature rules in Python

The `fe_utils.quadrature` module provides the `QuadratureRule` class which records quadrature points and weights for a given `ReferenceCell`. The `gauss_quadrature()` function creates quadrature rules for a prescribed degree of precision and reference cell.

Exercise 2 *The `integrate()` method is left unimplemented. Using (1.1), implement this method.*

A test script for your method is provided in the `test` directory as `test_01_integrate.py`. Run this script to test your code:

```
py.test test/test_01_integrate.py
```

from the Bash command line. Make sure you commit your modifications and push them to your fork of the course repository.

Hint: You can implement `integrate()` in one line using a list comprehension and `numpy.dot()`.

Hint: Don't forget to activate your Python venv!

CHAPTER

TWO

CONSTRUCTING FINITE ELEMENTS

At the core of the finite element method is the representation of finite-dimensional function spaces over elements. This concept was formalised by [Cia02]:

Definition 3 *A* finite element *is a triple* (K, P, N) *in which* K *is a cell,* P *is a space of functions* $K \to \mathbb{R}^n$ *and* N*, the set of* nodes*, is a basis for* P^**, the dual space to* P*.*

Note that this definition includes a basis for P^*, but not a basis for P. It turns out to be most convenient to specify the set of nodes for an element, and then derive an appropriate basis for P from that. In particular:

Definition 4 *Let* $N = \{n_j\}$ *be a basis for* P^**. A* nodal basis*,* $\{\phi_i\}$ *for* P *is a basis for* P *with the property that* $n_j(\phi_i) = \delta_{ij}$*.*

2.1 A worked example

To illustrate the construction of a nodal basis, let's consider the linear polynomials on a triangle. We first need to define our reference cell. The obvious choice is the triangle with vertices $\{(0,0),(1,0),(0,1)\}$

Functions in this space have the form $a + bx + cy$. So the function space has three unknown parameters, and its basis (and dual basis) will therefore have three members. In order to ensure the correct continuity between elements, the dual basis we need to use is the evaluation of the function at each of the cell vertices. That is:

$$\begin{aligned} n_0(f) &= f\left((0,0)\right) \\ n_1(f) &= f\left((1,0)\right) \\ n_2(f) &= f\left((0,1)\right) \end{aligned} \tag{2.1}$$

We know that ϕ_i has the form $a + bx + cy$ so now we can use the definition of the nodal basis to determine the unknown coefficients:

$$\begin{pmatrix} n_0(\phi_i) \\ n_1(\phi_i) \\ n_2(\phi_i) \end{pmatrix} = \begin{pmatrix} \delta_{i,0} \\ \delta_{i,1} \\ \delta_{i,2} \end{pmatrix} \tag{2.2}$$

So for ϕ_0 we have:

$$\begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 1 & 0 & 1 \end{bmatrix} \begin{bmatrix} a \\ b \\ c \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} \tag{2.3}$$

Which has solution $\phi_0 = 1 - x - y$. By a similar process, we can establish that the full basis is given by:

$$\begin{aligned} \phi_0 &= 1 - x - y \\ \phi_1 &= x \\ \phi_2 &= y \end{aligned} \tag{2.4}$$

2.2 Types of node

We have just encountered nodes given by the evaluation of the function at a given point. Other forms of functional are also suitable for use as finite element nodes. Examples include the integral of the function over the cell or some sub-entity and the evaluation of the gradient of the function at some point. For some vector-valued function spaces, the nodes may be given by the evaluation of the components of the function normal or tangent to the boundary of the cell at some point.

In this course we will only consider point evaluation nodes. The implementation of several other forms of node are covered in [Kir04].

2.3 The Lagrange element nodes

The number of coefficients of a degree p polynomial in d dimensions is given by $\binom{p+d}{d}$. The simplest set of nodes which we can employ is simply to place these nodes in a regular grid over the reference cell. Given the classical relationship between binomial coefficients and Pascal's triangle (and between trinomial coefficients and Pascal's pyramid), it is unsurprising that this produces the correct number of nodes.

The set of equally spaced points of degree p on the triangle is:

$$\left\{ \left(\frac{i}{p}, \frac{j}{p} \right) \middle| 0 \leq i + j \leq p \right\} \tag{2.5}$$

The finite elements with this set of nodes are called the *equispaced Lagrange* elements and are the most commonly used elements for relatively low order computations.

Note: At higher order the equispaced Lagrange basis is poorly conditioned and creates unwanted oscillations in the solutions. However for this course Lagrange elements will be sufficient.

Exercise 5 *Use (2.5) to implement* `lagrange_points()`. *Make sure your algorithm also works for one-dimensional elements. Some basic tests for your code are to be found in* `test/test_02_lagrange_points.py`. *You can also test your lagrange points on the triangle by running:*

```
plot_lagrange_points degree
```

Where `degree` *is the degree of the points to plot.*

2.4 Solving for basis functions

The matrix in (2.3) is a *generalised Vandermonde*[1] matrix . Given a list of points $(x_i, y_i) \in \mathbb{R}^2, 0 \leq i < m$ the corresponding degree n generalised Vandermonde matrix is given by:

$$\mathrm{V} = \begin{bmatrix} 1 & x_0 & y_0 & x_0^2 & x_0 y_0 & y_0^2 & \dots & x_0^n & x_0^{n-1} y_0 & \dots & x_0 y_0^{n-1} & y_0^n \\ 1 & x_1 & y_1 & x_1^2 & x_1 y_1 & y_1^2 & \dots & x_1^n & x_1^{n-1} y_1 & \dots & x_1 y_1^{n-1} & y_1^n \\ \vdots & & & & & & & & & & & \\ 1 & x_m & y_m & x_m^2 & x_m y_m & y_m^2 & \dots & x_m^n & x_m^{n-1} y_m & \dots & x_m y_m^{n-1} & y_m^n \end{bmatrix} \tag{2.6}$$

If we construct the Vandermonde matrix for the nodes of a finite element, then the equation for the complete set of basis function polynomial coefficients is:

$$\mathrm{VC} = \mathrm{I} \tag{2.7}$$

[1] A Vandermonde matrix is the one-dimensional case of the generalised Vandermonde matrix.

where the j-th column of C contains the polynomial coefficients of the basis function corresponding to the j-th node. For (2.7) to be well-posed, there must be a number of nodes equal to the number of coefficients of a degree n polynomial. If this is the case, then it follows immediately that:

$$\mathrm{C} = \mathrm{V}^{-1} \tag{2.8}$$

The same process applies to the construction of basis functions for elements in one or three dimensions, except that the Vandermonde matrix must be modified to exclude powers of y (in one dimension) or to include powers of z.

Note: The power series basis for polynomial spaces employed here becomes increasingly ill-conditioned at higher order, so it may be advantageous to employ a different basis in the construction of the Vandermonde matrix. See [Kir04] for an example.

Exercise 6 *Use (2.6) to implement* `vandermonde_matrix()`*. Think carefully about how to loop over each row to construct the correct powers of x and y. For the purposes of this exercise you should ignore the* `grad` *argument.*

Tests for this function are in `test/test_03_vandermonde_matrix.py`

Hint: You can use numpy array operations to construct whole columns of the matrix at once.

2.5 Implementing finite elements in Python

The *Ciarlet triple* (K, P, N) also provides a good abstraction for the implementation of software objects corresponding to finite elements. In our case K will be a `ReferenceCell`. In this course we will only implement finite element spaces consisting of complete polynomial spaces so we will specify P by providing the maximum degree of the polynomials in the space. Since we will only deal with point evaluation nodes, we can represent N by a series of points at which the evaluation should occur.

Exercise 7 *Implement the rest of the* `FiniteElement.__init__()` *method. You should construct a Vandermonde matrix for the nodes and invert it to create the basis function coefs. Store these as* `self.basis_coefs`*.*

Some basic tests of your implementation are in `test/test_04_init_finite_element.py`*.*

Hint: The `numpy.linalg.inv()` function may be used to invert the matrix.

2.6 Implementing the Lagrange Elements

The `FiniteElement` class implements a general finite element object assuming we have provided the cell, polynomial, degree and nodes. The `LagrangeElement` class is a subclass of `FiniteElement` which will implement the particular case of the equispaced Lagrange elements.

Exercise 8 *Implement the* `__init__()` *method of* `LagrangeElement`*. Use* `lagrange_points()` *to obtain the nodes. For the purpose of this exercise, you may ignore the* `entity_nodes` *argument.*

After *you have implemented* `tabulate()` *in the next exercise, you can use* `plot_lagrange_basis_functions` *to visualise your Lagrange basis functions.*

2.7 Tabulating basis functions

A core operation in the finite element method is integrating expressions involving functions in finite element spaces. This is usually accomplished using *numerical quadrature*. This means that we need to be able to evaluate

the basis functions at a set of quadrature points. The operation of evaluating a set of basis functions at a set of points is called *tabulation*.

Exercise 9 *Implement* `tabulate()`. *You can use a Vandermonde matrix to evaluate the polynomial terms and take the matrix product of this with the basis function coefficients. The method should have at most two executable lines. For the purposes of this exercise, ignore the* `grad` *argument.*

The test file `test/test_05_tabulate.py` *checks that tabulating the nodes of a finite element produces the identity matrix.*

2.8 Gradients of basis functions

A function f defined over a single finite element with basis $\{\phi_i\}$ is represented by a weighted sum of that basis:

$$f = \sum_i f_i \phi_i$$

In order to be able to represent and solve PDEs, we will naturally also have terms incorporating derivatives. Since the coefficients f_i are spatially constant, derivative operators pass through to apply to the basis functions:

$$\nabla f = \sum_i f_i \nabla \phi_i$$

This means that we will need to be able to evaluate the gradient of the basis functions at quadrature points.

Exercise 10 *Extend* `vandermonde_matrix()` *so that setting* `grad` *to* `True` *produces a rank 3 generalised Vandermonde tensor whose indices represent points, gradient component and basis function respectively. That is, each entry of V is replaced by a vector of the gradient of that polynomial term. For example, the entry x^2y^3 would be replaced by the vector $[2xy^3, 3x^2y^2]$.*

The `test/test_06_vandermonde_matrix_grad.py` *file has tests of this extension. You should also ensure that you still pass* `test/test_03_vandermonde_matrix.py`.

Hint: The `transpose()` method of numpy arrays enables generalised transposes swapping any dimensions.

Hint: At least one of the natural ways of implementing this function results in a whole load of `nan` values in the generalised Vandermonde matrix. In this case, you might find `numpy.nan_to_num()` useful.

Exercise 11 *Extend* `tabulate()` *to pass the* `grad` *argument through to* `vandermonde_matrix()`. *Then generalise the matrix product in* `tabulate()` *so that the result of this function (when* `grad` *is true) is a rank 3 tensor:*

$$\mathrm{T}_{ijk} = \nabla(\phi_j(X_i)) \cdot \mathbf{e}_k$$

where $\mathbf{e}_0 \ldots \mathbf{e}_{\dim -1}$ is the coordinate basis on the reference cell.

The `test/test_07_tabulate_grad.py` *script tests this extension. Once again, make sure you still pass* `test/test_05_tabulate.py`

Hint: The `numpy.einsum()` function implements generalised tensor contractions using Einstein summation notation. For example:

```
A = numpy.einsum("ijk,jl->ilk", T, C)
```

is equivalent to $A_{ilk} = \sum_j T_{ijk} C_{jl}$.

2.9 Interpolating functions to the finite element nodes

Recall once again that a function can be represented on a single finite element as:

$$f = \sum_i f_i \phi_i$$

Since $\{\phi_i\}$ is a nodal basis, it follows immediately that:

$$f_i = \phi_i^*(f)$$

where ϕ_i^* is the node associated with the basis function ϕ_i. Since we are only interested in nodes which are the point evaluation of their function input, we know that:

$$f_i = f(X_i)$$

where X_i is the point associated with the i-th node.

Exercise 12 *Implement* `interpolate()`.

Once you have done this, you can use the script provided to plot functions of your choice interpolated onto any of the finite elements you can make:

```
plot_interpolate_lagrange "sin(2*pi*x[0])" 2 5
```

Hint: You can find help on the arguments to this function with:

```
plot_interpolate_lagrange -h
```

CHAPTER

THREE

MESHES

When employing the finite element method, we represent the domain on which we wish to solve our PDE as a mesh. In order to work with meshes, we need to have a somewhat more formal mathematical notion of a mesh. The mesh concepts we will employ here are loosely based on those in [Log09], and are typical of mesh representations for the finite element method.

3.1 Mesh entities

A mesh is composed of *topological entities*, such as vertices, edges, polygons and polyhedra.

Definition 13 *The* (topological) dimension *of a mesh is the largest dimension among all of the topological entities in a mesh.*

In this course we will not consider meshes of manifolds immersed in higher dimensional spaces (for example the surface of a sphere immersed in $\mathbb{R}^3$) so the topological dimension of the mesh will always match the geometric dimension of space in which we are working, so we will simply refer to the *dimension* of the mesh.

Definition 14 *A topological entity of* codimension n *is a topological entity of dimension* $d - n$ *where* d *is the dimension of the mesh.*

Armed with these definitions we are able to define names for topological entities of various dimension and codimension:

entity name	dimension	codimension
vertex	0	
edge	1	
face	2	
facet		1
cell		0

The cells of a mesh can be polygons or polyhedra of any shape, however in this course we will restrict ourselves to meshes whose cells are intervals or triangles. The only other two-dimensional cells frequently employed are quadrilaterals.

The topological entities of each dimension will be given unique numbers in order that degrees of freedom can later be associated with them. We will identify topological entities by an index pair (d, i) where i is the index of the entity within the set of d-dimensional entities. For example, entity $(0, 10)$ is vertex number 10, and entity $(1, 10)$ is edge 10. Fig. 3.1 shows an example mesh with the topological entities labelled.

3.2 Reference cell entities

The reference cells similarly have locally numbered topological entities, these are shown in Fig. 3.2. The numbering is a matter of convention: that adopted here is that edges share the number of the opposite vertex. The orientation of the edges is also shown, this is always from the lower numbered vertex to the higher numbered one.

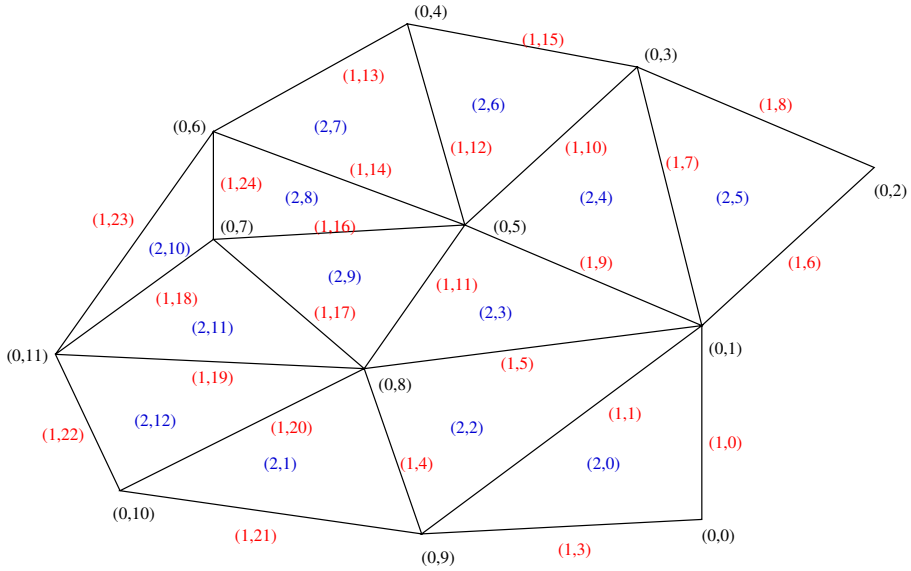

Fig. 3.1: A triangular mesh showing labelled topological entities: vertices (black), edges (red), and cells (blue).

The `ReferenceCell` class stores the local topology of the reference cell. Read the source and ensure that you understand the way in which this information is encoded.

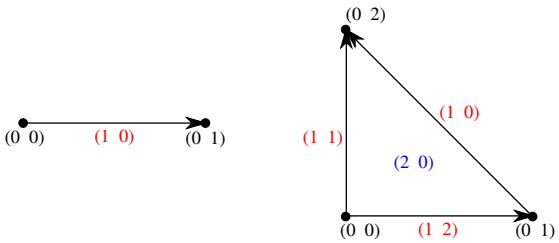

Fig. 3.2: Local numbering and orientation of the reference entities.

3.3 Adjacency

In order to implement the finite element method, we need to integrate functions over cells, which means knowing which basis functions are nonzero in a given cell. For the function spaces used in the finite element method, these basis functions will be the ones whose nodes lie on the topological entities adjacent to the cell. That is, the vertices, edges and (in 3D) the faces making up the cell, as well as the cell itself. One of the roles of the mesh is therefore to provide a lookup facility for the lower-dimensional mesh entities adjacent to a given cell.

Definition 15 *Given a mesh M, then for each $\dim(M) \geq d_1 > d_2 \geq 0$ the* adjacency *function $\mathrm{Adj}_{d_1,d_2} : \mathbb{N} \to \mathbb{N}^k$ is the function such that:*

$$\mathrm{Adj}_{d_1,d_2}(i) = (i_0, \ldots i_k)$$

where (d_1, i) is a topological entity and $(d_2, i_0), \ldots, (d_2, i_k)$ are the adjacent d_2-dimensional topological entities numbered in the corresponding reference cell order. If every cell in the mesh has the same topology then k will be fixed for each (d_1, d_2) pair. The correspondence between the orientation of the entity (d_1, i) and the reference cell of dimension d_1 is established by specifying that the vertices are numbered in ascending order[1]*. That is, for any*

[1] The numbering convention adopted here is very convenient, but only works for meshes composed of simplices (vertices, intervals, triangles and tetrahedra). A more complex convention would be required to support quadrilateral meshes.

entity (d_1, i)*:*

$$(i_0, \dots i_k) = \mathrm{Adj}_{d_1,0}(i) \quad \Longrightarrow \quad i_0 < \dots < i_k$$

A consequence of this convention is that the global orientation of all the entities making up a cell also matches their local orientation.

Example 16 *In the mesh shown in Fig. 3.1 we have:*

$$\mathrm{Adj}_{2,0}(3) = (1, 5, 8).$$

In other words, vertices 1, 5 and 8 are adjacent to cell 3. Similarly:

$$\mathrm{Adj}_{2,1}(3) = (11, 5, 9).$$

Edges 11, 5, and 9 are local edges 0, 1, and 2 of cell 3.

3.4 Mesh geometry

The features of meshes we have so far considered are purely topological: they deal with the adjacency relationships between topological entities, but do not describe the locations of those entities in space. Provided we restrict our attention to meshes in which the element edges are straight (ie not curved), we can represent the geometry of the mesh by simply recording the coordinates of the vertices. The positions of the higher dimensional entities then just interpolate the vertices of which they are composed. We will later observe that this is equivalent to representing the geometry in a vector-valued piecewise linear finite element space.

3.5 A mesh implementation in Python

The `Mesh` class provides an implementation of mesh objects in 1 and 2 dimensions. Given the list of vertices making up each cell, it constructs the rest of the adjacency function. It also records the coordinates of the vertices.

The `UnitSquareMesh` class creates a `Mesh` object corresponding to a regular triangular mesh of a unit square. Similarly, the `UnitIntervalMesh` class performs the corresponding (rather trivial) function for a unit one dimensional mesh.

You can observe the numbering of mesh entities in these meshes using the `plot_mesh` script. Run:

```
plot_mesh -h
```

for usage instructions.

CHAPTER

FOUR

FUNCTION SPACES: ASSOCIATING DATA WITH MESHES

A finite element space over a mesh is constructed by associating a finite element with each cell of the mesh. Will refer to the basis functions of this finite element space as *global* basis functions, while those of the finite element itself we will refer to as *local* basis functions. We can establish the relationship between the finite element and each cell of the mesh by associating the nodes (and therefore the local basis functions) of the finite element with the topological entities of the mesh. This is a two stage process. First, we associate the nodes of the finite element with the local topological entities of the reference cell. This is often referred to as *local numbering*. Then we associate the correct number of degrees of freedom with each global mesh entity. This is the *global numbering*.

4.1 Local numbering and continuity

Which nodes should be associated with which topological entities? The answer to this question depends on the degree of continuity required between adjacent cells. The nodes associated with topological entites on the boundaries of cells (the vertices in one dimension, the vertices and edges in two dimensions, and the vertices, edges and faces in three dimensions) are shared between cells. The basis functions associated with nodes on the cell boundary will therefore be continuous between the cells which share that boundary.

For the Lagrange element family, we require global C_0 continuity. This implies that the basis functions are continuous everywhere. This has the following implications for the association of basis functions with local topological entites:

vertices At the function vertices we can achieve continuity by requiring that there be a node associated with each mesh vertex. The basis function associated with that node will therefore be continuous. Since we have a nodal basis, all the other basis functions will vanish at the vertex so the global space will be continuous at this point.

edges Where the finite element space has at least 2 dimensions we need to ensure continuity along edges. The restriction of a degree p polynomial over a d-dimensional cell to an edge of that cell will be a one dimensional degree p polynomial. To fully specify this polynomial along an edge requires $p + 1$ nodes. However there will already be two nodes associated with the vertices of the edge, so $p - 1$ additional nodes will be associated with the edge.

faces For three-dimensional (tetrahedral) elements, the basis functions must also be continuous across faces. This requires that sufficient nodes lie on the face to fully specify a two dimensional degree p polynomial. However the vertices and edges of the face already have nodes associated with them, so the number of nodes required to be associated with the face itself is actually the number required to represent a degree $p - 2$ polynomial in two dimensions: $\binom{p-1}{2}$.

This pattern holds more generally: for a C_0 function space, the number of nodes which must be associated with a local topological entity of degree d is $\binom{p-1}{d}$.

Fig. 4.1 illustrates the association of nodes with reference entities for Lagrange elements on triangles. The numbering of nodes will depend on how `lagrange_points()` is implemented. The numbering used here is just one of the obvious choices.

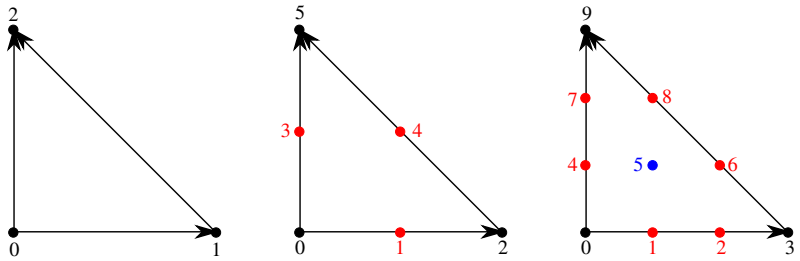

Fig. 4.1: Association of nodes with reference entities for the degree 1, 2, and 3 equispaced Lagrange elements on triangles. Black nodes are associated with vertices, red nodes with edges and blue nodes with the cell (face). The numbering of the nodes is arbitrary.

4.2 Implementing local numbering

Local numbering can be implemented by adding an additional data structure to the `FiniteElement` class. For each local entity this must record the local nodes associated with that entity. This can be achieved using a dictionary of dictionaries structure. For example employing the local numbering of nodes employed in Fig. 4.1, the `entity_node` dictionary for the degree three equispaced Lagrange element on a triangle is given by:

```
entity_node = {0: {0: [0],
                   1: [3],
                   2: [9]},
               1: {0: [6, 8],
                   1: [4, 7],
                   2: [1, 2]},
               2: {0: [5]}}
```

Note that the order of the nodes in each list is important: it must always consistently reflect the orientation of the relevant entity in order that all the cells which share that entity consistently interpret the nodes. In this case this has been achieved by listing the nodes in order given by the direction of the orientation of each edge.

Exercise 17 *Extend the `__init__()` method of `LagrangeElement` so that it passes the correct `entity_node` dictionary to the `FiniteElement` it creates.*

The `test/test_08_entity_nodes.py` script tests this functionality.

Hint: You can either work out the right algorithm to generate `entity_nodes` with the right node indices, or you can modify `lagrange_points()` so that it produces the nodes in entity order, thus making the construction of `entity_nodes` straightforward.

You may find the `point_in_entity()` method of the `ReferenceCell` class useful.

4.3 Global numbering

Given a mesh and a finite element, the global numbering task is to uniquely associate the appropriate number of global node numbers with each global entity. One such numbering[1] is to allocate global numbers in ascending entity dimension order, and within each dimension in order of the index of each global topological entity. The formula for the first global node associated with entity (d, i) is then:

$$G(d,i) = \left(\sum_{\delta<d} N_\delta E_\delta\right) + iN_d$$

[1] Many correct global numberings are possible, that presented here is simple and correct, but not optimal from the perspective of the memory layout of the resulting data.

where N_d is the number of nodes which this finite element associates with each entity of dimension d, and E_d is the number of dimension d entities in the mesh. The full list of nodes associated with entity (d, i) is therefore:

$$[G(d,i), \ldots, G(d,i) + N_d - 1] \tag{4.1}$$

4.4 The cell-node map

The primary use to which we wish to put the finite element spaces we are constructing is, naturally, the solution of finite element problems. The principle operation we will therefore need to support is integration over the mesh of mathematical expressions involving functions in finite element spaces. This will be accomplished by integrating over each cell in turn, and then summing over all cells. This means that a key operation we will need is to find the nodes associated with a given cell.

It is usual in finite element software to explicitly store the map from cells to adjacent nodes as a two-dimensional array with one row corresponding to each cell, and with columns corresponding to the local node numbers. The entries in this map will have the following values:

$$M[c, e(\delta,\epsilon)] = [G(\delta,i), \ldots, G(\delta,i) + N_\delta - 1] \qquad \forall 0 \le \delta \le \dim(c), \forall 0 \le \epsilon < \hat{E}_\delta \tag{4.2}$$

where:

$$i = \mathrm{Adj}_{\dim(c),\delta}[c,\epsilon], \tag{4.3}$$

$e(\delta,\epsilon)$ is the local entity-node list for this finite element for the (δ,ϵ) local entity, Adj has the meaning given under *Adjacency*, $\hat{E}_\delta$ is the number of dimension δ entities in each cell, and G and N have the meanings given above. This algorithm requires a trivial extension to adjacency:

$$\mathrm{Adj}_{\dim(c),\dim(c)}[c,0] = c \tag{4.4}$$

Hint: In (4.2), notice that for each value of δ and ϵ, $e(\delta,\epsilon)$ is a vector of indices, so the equation sets the value of zero, one, or more defined entries in row c of M for each δ and ϵ.

4.5 Implementing function spaces in Python

As noted above, a finite element space associates a mesh and a finite element, and contains (in some form) a global numbering of the nodes.

Exercise 18 *Implement the* `__init__()` *method of* `fe_utils.function_spaces.FunctionSpace`. *The key operation is to set* `cell_nodes` *using* (4.2).

You can plot the numbering you have created with the `plot_function_space_nodes` *script. As usual, run the script passing the* `-h` *option to discover the required arguments.*

Hint: Many of the terms in (4.2) are implemented in the objects in `fe_utils`. For example:

- $\mathrm{Adj}_{\dim(c),\delta}$ is implemented by the `adjacency()` method of the `Mesh`.
- You have $e(\delta,\epsilon)$ as `entity_nodes`. Note that in this case you need separate square brackets for each index:

```
element.entity_nodes[delta][epsilon]
```

Hint: `cell_nodes` needs to be integer-valued. If you choose to use `numpy.zeros()` to create a matrix which you then populate with values, you need to explicitly specify that you want a matrix of integers. This can

be achieved by passing the `dtype` argument to `numpy.zeros()`. For example `numpy.zeros((nrows, ncols), dtype=int)`.

CHAPTER

FIVE

FUNCTIONS IN FINITE ELEMENT SPACES

Recall that the general form of a function in a finite element space is:

$$f(x) = \sum_i f_i \phi_i(x) \tag{5.1}$$

Where the $\phi_i(x)$ are now the global basis functions achieved by stitching together the local basis functions defined by the *finite element*.

5.1 A python implementation of functions in finite element spaces

The `Function` class provides a simple implementation of function storage. The input is a `FunctionSpace` which defines the mesh and finite element to be employed, to which the `Function` adds an array of degree of freedom values, one for each node in the `FunctionSpace`.

5.2 Interpolating values into finite element spaces

Suppose we have a function $g(x) : \mathbb{R}^n \to \mathbb{R}$ which we wish to approximate as a function $f(x)$ in some finite element space V. In other words, we want to find the f_i such that:

$$\sum_i f_i \phi_i(x) \approx g(x) \tag{5.2}$$

The simplest way to do this is to *interpolate* $g(x)$ onto V. In other words, we evaluate:

$$f_i = n_i(g(x)) \tag{5.3}$$

where n_i is the node associated with ϕ_i. Since we are only concerned with point evaluation nodes, this is equivalent to:

$$f_i = g(x_i) \tag{5.4}$$

where x_i is the coordinate vector of the point defining the node n_i. This looks straightforward, however the x_i are the *global* node points, and so far we have only defined the node points in *local* coordinates on the reference element.

5.2.1 Changing coordinates between reference and physical space

We'll refer to coordinates on the global mesh as being in *physical space* while those on the reference element are in *local space*. We'll use case to distinguish local and global objects, so local coordinates will be written as X and global coordinates as x. The key observation is that within each cell, the global coordinates are the linear interpolation of the global coordinate values at the cell vertices. In other words, if $\{\Psi_j\}$ is the local basis for the

linear lagrange elements on the reference cell and $\hat{x}_j$ are the corresponding global vertex locations on a cell c then:

$$x = \sum_j \hat{x}_j \Psi_j(X) \quad \forall x \in c. \tag{5.5}$$

Remember that we know the location of the nodes in local coordinates, and we have the `tabulate()` method to evaluate all the basis functions of an element at a known set of points. So if we write:

$$A_{i,j} = \Psi_j(X_i) \tag{5.6}$$

where {X_i} are the node points of our finite element, then:

$$x = A \cdot \hat{x} \tag{5.7}$$

Where $\hat{x}$ is the $(\dim+1, \dim)$ array whose rows are the current element vertex coordinates, and x is the $(\text{nodes}, \dim)$ array whose rows are the global coordinates of the nodes in the current element. We can then apply $g()$ to each row of x in turn and record the result as the `Function` value for that node.

Hint: The observant reader will notice that this algorithm is inefficient because the function values at nodes on the boundaries of elements are evaluated more than once. This can be avoided with a little tedious bookkeeping but we will not concern ourselves with that here.

5.2.2 Looking up cell coordinates and values

In the previous section we used the vertex coordinates of a cell to find the node coordinates, and then we calculated `Function` values at those points. The coordinates are stored in a single long list associated with the `Mesh`, and the `Function` contains a single long list of values. We need to use *indirect addressing* to access these values. This is best illustrated using some Python code.

Suppose `f` is a `Function`. For brevity, we write `fs = f.function_space`, the `FunctionSpace` associated with `f`. Now, we first need a linear element and a corresponding `FunctionSpace`:

```
cg1 = fe_utils.LagrangeElement(fs.mesh.cell, 1)
cg1fs = fe_utils.FunctionSpace(fs.mesh, cg1)
```

Then the vertex indices of cell number `c` in the correct order for the linear Lagrange element are:

```
cg1fs.cell_nodes[c, :]
```

and therefore the set of coordinate vectors for the vertices of element `c` are:

```
fs.mesh.vertex_coords[cg1fs.cell_nodes[c, :], :]
```

That is, the `cg1fs.cell_nodes` array is used to look up the right vertex coordinates. By a similar process we can access the values associated with the nodes of element `c`:

```
f.values[fs.cell_nodes[c, :]]
```

5.2.3 A Python implementation of interpolation

Putting together the change of coordinates with the right indirect addressing, we can provide the `Function` class with a `interpolate()` method which interpolates a user-provided function onto the `Function`.

Exercise 19 *Read and understand the `interpolate()` method. Use `plot_sin_function` to investigate interpolating different functions onto finite element spaces at differering resolutions and polynomial degrees.*

Hint: There is no implementation work associated with this exercise, but the programming constructs used in `interpolate()` will be needed when you implement integration.

5.3 Integration

We now come to one of the fundamental operations in the finite element method: integrating a `Function` over the domain. The full finite element method actually requires the integration of expressions of unknown test and trial functions, but we will start with the more straightforward case of integrating a single, known, `Function` over a domain Ω:

$$\int_\Omega f \mathrm{d}x \quad f \in V \tag{5.8}$$

where $\mathrm{d}x$ should be understood as being the volume measure with the correct dimension for the domain and V is some finite element space over Ω. We can express this integral as a sum of integrals over individual cells:

$$\int_\Omega f \mathrm{d}x = \sum_{c \in \Omega} \int_c f \mathrm{d}x. \tag{5.9}$$

So we have in fact reduced the integration problem to the problem of integrating f over each cell. In *a previous part* of the module we implemented quadrature rules which enable us to integrate over specified reference cells. If we can express the integral over some arbitrary cell c as an integral over a reference cell c_0 then we are done. In fact this simply requires us to employ the change of variables formula for integration:

$$\int_c f(x) \mathrm{d}x = \int_{c_0} f(X) |J| \mathrm{d}X \tag{5.10}$$

where $|J|$ is the absolute value of the determinant of the Jacobian matrix. J is given by:

$$J_{\alpha\beta} = \frac{\partial x_\alpha}{\partial X_\beta}. \tag{5.11}$$

Hint: We will generally adopt the convention of using Greek letters to indicate indices in spatial dimensions, while we will use Roman letters in the sequence $i, j, \ldots$ for basis function indices. We will continue to use q for the index over the quadrature points.

Evaluating (5.11) depends on having an expression for x in terms of X. Fortunately, (5.5) is exactly this expression, and applying the usual rule for differentiating functions in finite element spaces produces:

$$J_{\alpha\beta} = \sum_j (\tilde{x}_j)_\alpha \nabla_\beta \Psi_j(X) \tag{5.12}$$

where $\{\Psi_j\}$ is once again the degree 1 Lagrange basis and $\{\tilde{x}_j\}$ are the coordinates of the corresponding vertices of cell c. The presence of X in (5.12) implies that the Jacobian varies spatially across the reference cell. However since $\{\Psi_j\}$ is the degree 1 Lagrange basis, the gradients of the basis functions are constant over the cell and so it does not matter at which point in the cell the Jacobian is evaluated. For example we might choose to evaluate the Jacobian at the cell origin $X = 0$.

Hint: When using simplices with curved sides, and on all but the simplest quadrilateral or hexahedral meshes, the change of coordinates will not be affine. In that case, to preserve full accuracy it will be necessary to compute the Jacobian at every quadrature point. However, non-affine coordinate transforms are beyond the scope of this course.

5.3.1 Expressing the function in the finite element basis

Let $\{\Phi_i(X)\}$ be a **local** basis for V on the reference element c_0. Then our integral becomes:

$$\int_c f(x) \mathrm{d}x = \int_{c_0} \sum_i F(M(c,i))\, \Phi_i(X)\, |J|\, \mathrm{d}X \tag{5.13}$$

where F is the vector of global coefficient values of f, and M is *the cell node map*.

5.3.2 Numerical quadrature

The actual evaluation of the integral will employ the quadrature rules we discussed in *a previous section*. Let $\{X_q\}, \{w_q\}$ be a quadrature rule of sufficient degree of precision that the quadrature is exact. Then:

$$\int_c f(x)\mathrm{d}x = \sum_q \sum_i F(M(c,i))\,\Phi_i(X_q)\,|J|\,w_q \tag{5.14}$$

5.3.3 Implementing integration

Exercise 20 *Use* (5.12) *to implement the* `jacobian()` *method of* `Mesh`. `test/test_09_jacobian.py` *is available for you to test your results.*

Hint: The $\nabla_\beta \Psi_j(X)$ factor in (5.12) is the same for every cell in the mesh. You could make your implementation more efficient by precalculating this term in the `__init__()` method of `Mesh`.

Exercise 21 *Use* (5.9) *and* (5.14) *to implement* `integrate()`. `test/test_10_integrate_function.py` *may be used to test your implementation.*

Hint: Your method will need to:

1. Construct a suitable `QuadratureRule`.
2. `tabulate()` the basis functions at each quadrature point.
3. Visit each cell in turn.
4. Construct the `jacobian()` for that cell and take the absolute value of its determinant (`numpy.absolute` and `numpy.linalg.det()` will be useful here).
5. Sum all of the arrays you have constructed over the correct indices to a contribution to the integral (`numpy.einsum()` may be useful for this).

Hint: You might choose to read ahead before implementing `integrate()`, since the `errornorm()` function is very similar and may provide a useful template for your work.

CHAPTER

SIX

ASSEMBLING AND SOLVING FINITE ELEMENT PROBLEMS

Having constructed functions in finite element spaces and integrated them over the domain, we now have the tools in place to actually assemble and solve a simple finite element problem. To avoid having to explicitly deal with boundary conditions, we choose in the first instance to solve a Helmholtz problem[1], find u in some finite element space V such that:

$$\begin{aligned} -\nabla^2 u + u &= f \\ \nabla u \cdot \mathbf{n} &= 0 \text{ on } \Gamma \end{aligned} \tag{6.1}$$

where Γ is the domain boundary and n is the outward pointing normal to that boundary. f is a known function which, for simplicity, we will assume lies in V. Next, we form the weak form of this equation by multiplying by a test function in V and integrating over the domain. We integrate the Laplacian term by parts. The problem becomes, find $u \in V$ such that:

$$\int_\Omega \nabla v \cdot \nabla u + vu \,\mathrm{d}x - \underbrace{\int_\Gamma v \nabla u \cdot \mathbf{n} \,\mathrm{d}s}_{=0} = \int_\Omega vf \,\mathrm{d}x \qquad \forall v \in V \tag{6.2}$$

If we write $\{\phi_i\}_{i=0}^{n-1}$ for our basis for V, and recall that it is sufficient to ensure that (6.2) is satisfied for each function in the basis then the problem is now, find coefficients u_i such that:

$$\int_\Omega \sum_j \left(\nabla \phi_i \cdot \nabla(u_j \phi_j) + \phi_i u_j \phi_j\right) \,\mathrm{d}x = \int_\Omega \phi_i \sum_k f_k \phi_k \,\mathrm{d}x \qquad \forall\, 0 \leq i < n \tag{6.3}$$

Since the left hand side is linear in the scalar coefficients u_j, we can move them out of the integral:

$$\sum_j \left(\int_\Omega \nabla \phi_i \cdot \nabla \phi_j + \phi_i \phi_j \,\mathrm{d}x \, u_j \right) = \int_\Omega \phi_i \sum_k f_k \phi_k \,\mathrm{d}x \qquad \forall\, 0 \leq i < n \tag{6.4}$$

We can write this as a matrix equation:

$$\mathrm{A}\mathbf{u} = \mathbf{f} \tag{6.5}$$

where:

$$\mathrm{A}_{ij} = \int_\Omega \nabla \phi_i \cdot \nabla \phi_j + \phi_i \phi_j \,\mathrm{d}x \tag{6.6}$$

$$\mathbf{u}_j = u_j \tag{6.7}$$

$$\mathbf{f}_i = \int_\Omega \phi_i \sum_k f_k \phi_k \,\mathrm{d}x \tag{6.8}$$

[1] Strictly speaking this is the positive definite Helmholtz problem. Changing the sign on u produces the indefinite Helmholtz problem, which is significantly harder to solve.

6.1 Assembling the right hand side

The assembly of these integrals exploits the same decomposition property we exploited previously to integrate functions in finite element spaces. For example, (6.8) can be rewritten as:

$$\mathbf{f}_i = \sum_c \int_c \phi_i \sum_k f_k \phi_k \,\mathrm{d}x \tag{6.9}$$

This has a practical impact once we realise that only a few basis functions are non-zero in each element. This enables us to write an efficient algorithm for right hand side assembly. Assume that at the start of our algorithm:

$$\mathbf{f}_i = 0. \tag{6.10}$$

Now for each cell c, we execute:

$$\mathbf{f}_{M(c,\hat{i})} \overset{+}{=} \int_c \Phi_{\hat{i}} \left(\sum_{\hat{k}} f_{M(c,\hat{k})} \, \Phi_{\hat{k}} \right) |J| \,\mathrm{d}X \qquad \forall 0 \leq \hat{i} < N \tag{6.11}$$

Where M is the cell-node map for the finite element space V, N is the number of nodes per element in V, and $\{\Phi_{\hat{i}}\}_{\hat{i}=0}^{N-1}$ are the local basis functions. In other words, we visit each cell and conduct the integral for each local basis function, and add that integral to the total for the corresponding global basis function.

By choosing a suitable quadrature rule, $\{X_q\}, \{w_q\}$, we can write this as:

$$\mathbf{f}_{M(c,\hat{i})} \overset{+}{=} \left(\sum_q \Phi(X_q)_{\hat{i}} \left(\sum_{\hat{k}} f_{M(c,\hat{k})} \, \Phi(X_q)_{\hat{k}} \right) w_q \right) |J| \qquad \forall 0 \leq \hat{i} < N, \forall c \tag{6.12}$$

6.2 Assembling the left hand side matrix

The left hand side matrix follows a similar pattern, however there are two new complications. First, we have two unbound indices (i and j), and second, the integral involves derivatives. We will address the question of derivatives first.

6.2.1 Pulling gradients back to the reference element

On element c, there is a straightforward relationship between the local and global bases:

$$\phi_{M(c,i)}(x) = \Phi_i(X) \tag{6.13}$$

We can also, as we showed in *Changing coordinates between reference and physical space*, express the global coordinate x in terms of the local coordinate X.

What about $\nabla\phi$? We can write the gradient operator in component form and apply (6.13):

$$\frac{\partial \phi_{M(c,i)}(x)}{\partial x_\alpha} = \frac{\partial \Phi_i(X)}{\partial x_\alpha} \quad \forall\, 0 \leq \alpha < \dim \tag{6.14}$$

However, the expression on the right involves the gradient of a local basis function with respect to the global coordinate variable x. We employ the chain rule to express this gradient with respect to the local coordinates, X:

$$\frac{\partial \phi_{M(c,i)}(x)}{\partial x_\alpha} = \sum_{\beta=0}^{\dim -1} \frac{\partial X_\beta}{\partial x_\alpha} \frac{\partial \Phi_i(X)}{\partial X_\beta} \quad \forall\, 0 \leq \alpha < \dim \tag{6.15}$$

Using the *definition of the Jacobian*, and using ∇_x and ∇_X to indicate the global and local gradient operators respectively, we can equivalently write this expression as:

$$\nabla_x \phi_{M(c,i)}(x) = J^{-\mathrm{T}} \nabla_X \Phi_i(X) \tag{6.16}$$

where $J^{-\mathrm{T}} = (J^{-1})^{\mathrm{T}}$ is the transpose of the inverse of the cell Jacobian matrix.

6.2.2 The assembly algorithm

We can start by pulling back (6.6) to local coordinates:

$$\mathrm{A}_{ij} = 0.$$
$$\mathrm{A}_{M(c,\hat{i}),M(c,\hat{j})} \stackrel{+}{=} \int_c \left(\left(J^{-T}\nabla_X \Phi_{\hat{i}}\right) \cdot \left(J^{-T}\nabla_X \Phi_{\hat{j}}\right) + \Phi_{\hat{i}}\Phi_{\hat{j}} \right) |J| \,\mathrm{d}X \quad \forall 0 \le \hat{i}, \hat{j} < N, \forall c \tag{6.17}$$

We now employ a suitable quadrature rule, $\{X_q\}, \{w_q\}$, to calculate the integral:

$$\mathrm{A}_{M(c,\hat{i}),M(c,\hat{j})} \stackrel{+}{=} \sum_q \left(\left(J^{-T}\nabla_X \Phi_{\hat{i}}(X_q)\right) \cdot \left(J^{-T}\nabla_X \Phi_{\hat{j}}(X_q)\right) + \Phi_{\hat{i}}(X_q)\Phi_{\hat{j}}(X_q) \right) |J| \, w_q \quad \forall 0 \le \hat{i}, \hat{j} < N, \forall c \tag{6.18}$$

Some readers may find this easier to read using index notation over the geometric dimensions:

$$\mathrm{A}_{M(c,\hat{i}),M(c,\hat{j})} \stackrel{+}{=} \sum_q \left(\sum_{\alpha\beta\gamma} J^{-1}_{\beta\alpha} \left(\nabla_X \Phi_{\hat{i}}(X_q)\right)_\beta \, J^{-1}_{\gamma\alpha} \left(\nabla_X \Phi_{\hat{j}}(X_q)\right)_\gamma + \Phi_{\hat{i}}(X_q)\Phi_{\hat{j}}(X_q) \right) |J| \, w_q \quad \forall 0 \le \hat{i}, \hat{j} < N, \forall c \tag{6.19}$$

6.2.3 A note on matrix insertion

For each cell c, the right hand sides of equations (6.18) and (6.19) have two free indices, $\hat{i}$ and $\hat{j}$. The equation therefore assembles a local $N \times N$ matrix corresponding to one integral for each test function, trial function pair on the current element. This is then added to the global matrix at the row and column pairs given by the cell node map $M(c, \hat{i})$ and $M(c, \hat{j})$.

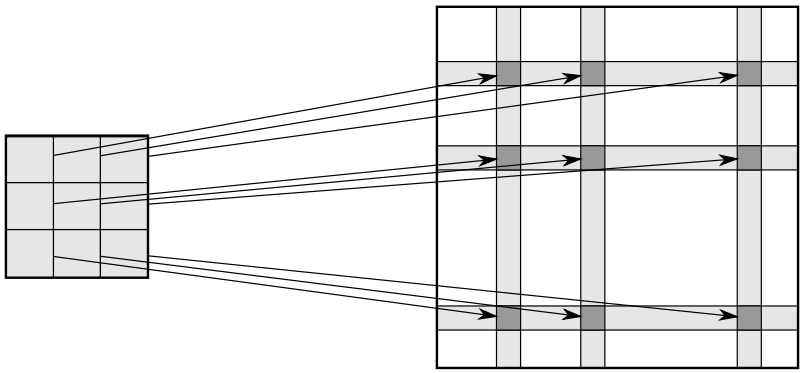

Fig. 6.1: Computing integrals for each local test and trial function produces a local dense (in this case, 3×3) matrix. The entries in this matrix are added to the corresponding global row and column positions in the global matrix.

Hint: One might naïvely expect that if `nodes` is the vector of global node numbers for the current cell, `m` is the matrix of local integral values and `A` is the global matrix, then the Python code might look like:

```
A[nodes, nodes] += m # DON'T DO THIS!
```

Unfortunately, `numpy` interprets this as an instruction to insert a vector into the diagonal of `A`, and will complain that the two-dimensional right hand side does not match the one-dimensional left hand side. Instead, one has to employ the `numpy.ix_()` function:

```
A[np.ix_(nodes, nodes)] += m # DO THIS!
```

No such problem exists for adding values into the global right hand side vector. If `l` is the global right hand side vector and `v` is the vector of local right hand integrals, then the following will work just fine:

```
l[nodes] += v
```

6.2.4 Sparse matrices

Each row of the global matrix corresponds to a single global basis function. The number of non-zeros in this row is equal to the number of other basis functions which are non-zero in the elements where the original basis function is non-zero. The maximum number of non-zeros on a row may vary from a handful for a low degree finite element to a few hundred for a fairly high degree element. The important point is that it is essentially independent of the size of the mesh. This means that as the number of cells in the mesh increases, the proportion of the matrix entries on each row which have the value zero increases.

For example, a degree 4 Lagrange finite element space defined on 64×64 unit square triangular mesh has about 66000 nodes. The full global matrix therefore has more that 4 billion entries and, at 8 bytes per matrix entry, will consume around 35 gigabytes of memory! However, there are actually only around 23 nonzeros per row, so more than 99.9% of the entries in the matrix are zeroes.

Instead of storing the complete matrix, sparse matrix formats store only those entries in the matrix which are nonzero. They also have to store some metadata to describe where in the matrix the non-zero entries are stored. There are various different sparse matrix formats available, which make different trade-offs between memory usage, insertion speed, and the speed of different matrix operations. However, if we make the (conservative) assumption that a sparse matrix takes 16 bytes to store each nonzero value, instead of 8 bytes, then we discover that in the example above, we would use less than 25 megabytes to store the matrix. The time taken to solving the matrix system will also be vastly reduced since operations on zeros are avoided.

Hint: The `scipy.sparse` package provides convenient interfaces which enable Python code to employ a variety of sparse matrix formats using essentially identical operations to the dense matrix case. The skeleton code already contains commands to construct empty sparse matrices and to solve the resulting linear system. You may, if you wish, experiment with choosing other sparse formats from `scipy.sparse`, but it is very strongly suggested that you do **not** switch to a dense numpy array; unless, that is, you particularly enjoy running out of memory on your computer!

6.3 The method of manufactured solutions

When the finite element method is employed to solve Helmholtz problems arising in science and engineering, the value forcing function f will come from the application data. However for the purpose of testing numerical methods and software, it is exceptionally useful to be able to find values of f such that an analytic solution to the partial differential equation is known. It turns out that there is a straightforward algorithm for this process. This algorithm is known as the *method of manufactured solutions*. It has but two steps:

1. Choose a function $\tilde{u}$ which satisfies the boundary conditions of the PDE.
2. Substitute $\tilde{u}$ into the left hand side of (6.1). Set f equal to the result of this calculation, and now $\tilde{u}$ is a solution to (6.1).

To illustrate this algorithm, suppose we wish to construct f such that:

$$\tilde{u} = \cos(4\pi x_0) x_1^2 (1 - x_1)^2 \tag{6.20}$$

is a solution to (6.1). It is simple to verify that $\tilde{u}$ satisfies the boundary conditions. We then note that:

$$-\nabla^2 \tilde{u} + \tilde{u} = \left((16\pi^2 + 1)(x_1 - 1)^2 x_1^2 - 12x_1^2 + 12x_1 - 2\right) \cos(4\pi x_0) \tag{6.21}$$

If we choose:

$$f = \left((16\pi^2 + 1)(x_1 - 1)^2 x_1^2 - 12x_1^2 + 12x_1 - 2\right)\cos(4\pi x_0) \tag{6.22}$$

then $\tilde{u}$ is a solution to (6.1).

6.4 Errors and convergence

6.4.1 The L^2 error

When studying finite element methods we are freqently concerned with convergence in the L^2 norm. That is to say, if V and W are finite element spaces defined over the same mesh, and $f \in V, g \in W$ then we need to calculate:

$$\sqrt{\int_\Omega (f-g)^2 \mathrm{d}x} = \sqrt{\sum_c \int_c \left(\left(\sum_i f_{M_V(c,i)}\Phi_i\right) - \left(\sum_j g_{M_W(c,j)}\Psi_j\right)\right)^2 |J|\mathrm{d}X} \tag{6.23}$$

where M_V is the cell-node map for the space V and M_W is the cell-node map for the space W. Likewise $\{\Phi_i\}$ is the local basis for V and $\{\Psi_j\}$ is the local basis for W.

A complete quadrature rule for this integral will, due to the square in the integrand, require a degree of precision equal to twice the greater of the polynomial degrees of V and W.

6.4.2 Numerically estimating convergence rates

Using the approximation results from the theory part of the course, we know that the error term in the finite element solution of the Helmholtz equation is expected to have the form $\mathcal{O}(h^{p+1})$ where h is the mesh spacing and p is the polynomial degree of the finite element space employed. That is to say if $\tilde{u}$ is the exact solution to our PDE and u_h is the solution to our finite element problem, then for sufficiently small h:

$$\|u_h - \tilde{u}\|_{L^2} < ch^{p+1} \tag{6.24}$$

for some $c > 0$ not dependent on h. Indeed, for sufficiently small h, there is a c such that we can write:

$$\|u_h - \tilde{u}\|_{L^2} \approx ch^{p+1} \tag{6.25}$$

Suppose we solve the finite element problem for two different (fine) mesh spacings, h_1 and h_2. Then we have:

$$\begin{aligned} \|u_{h_1} - \tilde{u}\|_{L^2} &\approx ch_1^{p+1} \\ \|u_{h_2} - \tilde{u}\|_{L^2} &\approx ch_2^{p+1} \end{aligned} \tag{6.26}$$

or equivalently:

$$\frac{\|u_{h_1} - \tilde{u}\|_{L^2}}{\|u_{h_2} - \tilde{u}\|_{L^2}} \approx \left(\frac{h_1}{h_2}\right)^{p+1} \tag{6.27}$$

By taking logarithms and rearranging this equation, we can produce a formula which, given the analytic solution and two numerical solutions, produces an estimate of the rate of convergence:

$$q = \frac{\ln\left(\dfrac{\|u_{h_1} - \tilde{u}\|_{L^2}}{\|u_{h_2} - \tilde{u}\|_{L^2}}\right)}{\ln\left(\dfrac{h_1}{h_2}\right)} \tag{6.28}$$

6.5 Implementing finite element problems

Exercise 22 *`fe_utils/solvers/helmholtz.py` contains a partial implementation of the finite element method to solve (6.2) with f chosen as in (6.22). Your task is to implement the `assemble()` function using (6.12), and (6.18) or (6.19). The comments in the `assemble()` function provide some guidance as to the steps involved. You may also wish to consult the `errornorm()` function as a guide to the structure of the code required.*

Run:

```
python fe_utils/solvers/helmholtz.py --help
```

for guidance on using the script to view the solution, the analytic solution and the error in your solution. In addition, `test/test_12_helmholtz_convergence.py` contains tests that the helmholtz solver converges at the correct rate for degree 1, 2 and 3 polynomials.

> ***Warning:*** *`test/test_12_helmholtz_convergence.py` may take many seconds or even a couple of minutes to run, as it has to solve on some rather fine meshes in order to check convergence.*

CHAPTER

SEVEN

DIRICHLET BOUNDARY CONDITIONS

The Helmholtz problem we solved in the previous part was chosen to have homogeneous Neumann or *natural* boundary conditions, which can be implemented simply by cancelling the zero surface integral. We can now instead consider the case of Dirichlet, or *essential* boundary conditions. Instead of the Helmholtz problem we solved before, let us now specify a Poisson problem with homogeneous Dirichlet conditions, find u in some finite element space V such that:

$$\begin{aligned} -\nabla^2 u &= f \\ u &= 0 \text{ on } \Gamma \end{aligned} \tag{7.1}$$

In order to implement the Dirichlet conditions, we need to decompose V into two parts:

$$V = V_0 \oplus V_\Gamma \tag{7.2}$$

where V_Γ is the space spanned by those functions in the basis of V which are non-zero on Γ, and V_0 is the space spanned by the remaining basis functions (i.e. those basis functions which vanish on Γ). It is a direct consequence of the nodal nature of the basis that the basis functions for V_Γ are those corresponding to the nodes on Γ while the basis for V_0 is composed of all the other functions.

We now write the weak form of (7.1), find $u = u_0 + u_\Gamma$ with $u_0 \in V_0$ and $u_\Gamma \in V_\Gamma$ such that:

$$\begin{aligned} \int_\Omega \nabla v_0 \cdot \nabla(u_0 + u_\Gamma)\,\mathrm{d}x - \underbrace{\int_\Gamma v_0 \nabla(u_0 + u_\Gamma)\cdot \mathbf{n}\,\mathrm{d}s}_{=0} &= \int_\Omega v_0 f\,\mathrm{d}x \qquad \forall v_0 \in V_0 \\ u_\Gamma &= 0 \qquad \text{on } \Gamma \end{aligned} \tag{7.3}$$

There are a number of features of this equation which require some explanation:

1. We only test with functions from V_0. This is because it is only necessary that the differential equation is satisfied on the interior of the domain: on the boundary of the domain we need only satisfy the boundary conditions.
2. The surface integral now cancels because v_0 is guaranteed to be zero everywhere on the boundary.
3. The u_Γ definition actually implies that $u_\Gamma = 0$ everywhere, since all of the nodes in V_Γ lie on the boundary.

This means that the weak form is actually:

$$\begin{aligned} \int_\Omega \nabla v_0 \cdot \nabla u\,\mathrm{d}x &= \int_\Omega v_0 f\,\mathrm{d}x \qquad \forall v_0 \in V_0 \\ u_\Gamma &= 0 \end{aligned} \tag{7.4}$$

7.1 An algorithm for homogeneous Dirichlet conditions

The implementation of homogeneous Dirichlet conditions is actually rather straightforward.

1. The system is assembled completely ignoring the Dirichlet conditions. This results in a global matrix and vector which are correct on the rows corresponding to test functions in V_0, but incorrect on the V_Γ rows.

2. The global vector rows corresponding to boundary nodes are set to 0.
3. The global matrix rows corresponding to boundary nodes are set to 0.
4. The diagonal entry on each matrix row corresponding to a boundary node is set to 1.

This has the effect of replacing the incorrect boundary rows of the system with the equation $u_i = 0$ for all boundary node numbers i.

Hint: This algorithm has the unfortunate side effect of making the global matrix non-symmetric. If a symmetric matrix is required (for example in order to use a symmetric solver), then forward substition can be used to zero the boundary columns in the matrix, but that is beyond the scope of this module.

7.2 Implementing boundary conditions

Let:

$$f = \left(16\pi^2(x_1-1)^2x_1^2 - 2(x_1-1)^2 - 8(x_1-1)x_1 - 2x_1^2\right)\sin(4\pi x_0)$$

With this definition, (7.4) has solution:

$$u = \sin(4\pi x_0)(x_1-1)^2x_1^2$$

Exercise 23 *`fe_utils/solvers/poisson.py` contains a partial implementation of this problem. You need to implement the `assemble()` function. You should base your implementation on your `fe_utils/solvers/helmholtz.py` but take into account the difference in the equation, and the boundary conditions. The `fe_utils.solvers.poisson.boundary_nodes()` function in `fe_utils/solvers/poisson.py` is likely to be helpful in implementing the boundary conditions. As before, run:*

```
python fe_utils/solvers/poisson.py --help
```

for instructions (they are the same as for `fe_utils/solvers/helmholtz.py`). Similarly, `test/test_13_poisson_convergence.py` contains convergence tests for this problem.

7.3 Inhomogeneous Dirichlet conditions

The algorithm described here can be extended to inhomogeneous systems by setting the entries in the global vector to the value of the boundary condition at the corresponding boundary node. This additional step is required for the mastery exercise, but will be explained in more detail in the next section.

CHAPTER

EIGHT

NONLINEAR PROBLEMS

The finite element method may also be employed to numerically solve *nonlinear* PDEs. In order to do this, we can apply the classical technique for solving nonlinear systems: we employ an iterative scheme such as Newton's method to create a sequence of linear problems whose solutions converge to the correct solution to the nonlinear problem.

Note: This section is the mastery exercise for this module. This exercise is explicitly intended to test whether you can bring together what has been learned in the rest of the module in order to go beyond what has been covered in lectures and labs.

This exercise is not a part of the third year version of this module.

8.1 A model problem

As a simple case of a non-linear PDE, we can consider a steady non-linear diffusion equation. This is similar to the Poisson problem, except that the diffusion rate now depends on the value of the solution:

$$\begin{aligned} -\nabla \cdot ((u+1)\nabla u) &= g \\ u &= b \text{ on } \Gamma \end{aligned} \tag{8.1}$$

where g and b are given functions defined over Ω and Γ respectively.

We can create the weak form of (8.1) by integrating by parts and taking the boundary conditions into account. The problem becomes, find $u \in V$ such that:

$$\begin{aligned} \int_\Omega \nabla v_0 \cdot (u+1)\nabla u \,\mathrm{d}x = \int_\Omega v_0 g \,\mathrm{d}x \qquad & \forall v_0 \in V_0 \\ u_\Gamma = b. & \end{aligned} \tag{8.2}$$

Once more, V_0 is the subspace of V spanned by basis functions which vanish on the boundary, $V = V_0 \oplus V_\Gamma$, and $u = u_0 + u_\Gamma$ with $u_0 \in V_0$ and $u_\Gamma \in V_\Gamma$. This is corresponds directly with the weak form of the Poisson equation we already met. However, (8.2) is still nonlinear in u so we cannot simply substitute $u = u_i \phi_i$ in order to obtain a linear matrix system to solve.

8.2 Residual form

The general weak form of a non-linear problem is, find $u \in V$ such that:

$$f(u; v) = 0 \qquad \forall v \in V \tag{8.3}$$

The use of a semicolon is a common convention to indicate that f is assumed to be linear in the arguments after the semicolon, but might be nonlinear in the arguments before the semicolon. In this case, we observe that f may be nonlinear in u but is (by construction) linear in v.

The function f is called the *residual* of the nonlinear system. In essence, $f(u;v) = 0\ \forall v \in V$ if and only if u is a weak solution to the PDE. Since the residual is linear in v, it suffices to define the residual for each ϕ_i in the basis of V. For $\phi_i \in V_0$, the residual is just the weak form of the equation, but what do we do for the boundary? The simple answer is that we need a linear functional which is zero if the boundary condition is satisfied at this test function, and nonzero otherwise. The simplest example of such a functional is:

$$f(u;\phi_i) = n_i(u) - n_i(b) \tag{8.4}$$

where n_i is the node associated with basis function ϕ_i. For point evaluation nodes, $n_i(u)$ is the value of the proposed solution at node point i and $n_i(b)$ is just the boundary condition evaluated at that same point.

So for our model problem, we now have a full statement of the residual in terms of a basis function ϕ_i:

$$f(u;\phi_i) = \begin{cases} \displaystyle\int_\Omega \nabla\phi_i \cdot ((u+1)\nabla u) - \phi_i g \,\mathrm{d}x & \phi_i \in V_0 \\ n_i(u) - n_i(b) & \phi_i \in V_\Gamma \end{cases} \tag{8.5}$$

Hint: Evaluating the residual requires that the boundary condition be evaluated at the boundary nodes. A simple (if slightly inefficient) way to achieve this is to interpolate the boundary condition onto a function $\hat{b} \in V$.

8.3 Linearisation and Gâteaux Derivatives

Having stated our PDE in residual form, we now need to linearise the problem and thereby employ a technique such as Newton's method. In order to linearise the residual, we need to differentiate it with respect to u. Since u is not a scalar real variable, but is instead a function in V, the appropriate form of differentiation is the Gâteaux Derivative, given by:

$$J(u;v,\hat{u}) = \lim_{\epsilon\to 0} \frac{f(u+\epsilon\hat{u};v) - f(u;v)}{\epsilon}. \tag{8.6}$$

Here, the new argument $\hat{u} \in V$ indicates the "direction" in which the derivative is to be taken. Let's work through the Gâteaux Derivative for the residual of our model problem. Assume first that $v \in V_0$. Then:

$$\begin{aligned} J(u;v,\hat{u}) &= \lim_{\epsilon\to 0} \frac{\displaystyle\int_\Omega \nabla v \cdot ((u+\epsilon\hat{u}+1)\nabla(u+\epsilon\hat{u})) - vg\,\mathrm{d}x - \int_\Omega \nabla v\cdot((u+1)\nabla u) - vg\,\mathrm{d}x}{\epsilon} \\ &= \lim_{\epsilon\to 0} \frac{\displaystyle\int_\Omega \nabla v \cdot (\epsilon\hat{u}\nabla u + (u+1)\nabla(\epsilon\hat{u}) + \epsilon\hat{u}\nabla(\epsilon\hat{u}))\ \mathrm{d}x}{\epsilon} \\ &= \int_\Omega \nabla v \cdot (\hat{u}\nabla u + (u+1)\nabla\hat{u})\ \mathrm{d}x. \end{aligned} \tag{8.7}$$

Note that, as expected, J is linear in $\hat{u}$.

Next, we can work out the boundary case by assuming $v = \phi_i$, one of the basis functions of V_Γ:

$$\begin{aligned} J(u;\phi_i,\hat{u}) &= \lim_{\epsilon\to 0} \frac{n_i(u+\epsilon\hat{u}) - n_i(b) - (n_i(u) - n_i(b))}{\epsilon} \\ &= n_i(\hat{u}) \qquad \text{since } n_i(\cdot) \text{ is linear.} \end{aligned} \tag{8.8}$$

Once again, we can observe that J is linear in $\hat{u}$. Indeed, if we choose $\hat{u} = \phi_j$ for some ϕ_j in the basis if V then the definition of a nodal basis gives us:

$$J(u;\phi_i,\phi_j) = \delta_{ij} \tag{8.9}$$

8.4 A Taylor expansion and Newton's method

Since we now have the derivative of the residual with respect to a perturbation to the prospective solution u, we can write the first terms of a Taylor series approximation for the value of the residual at a perturbed solution $u+\hat{u}$:

$$f(u+\hat{u};v) = f(u;v) + J(u;v,\hat{u}) + \dots \qquad \forall v \in V. \tag{8.10}$$

Now, just as in the scalar case, Newton's method consists of approximating the function (the residual) by the first two terms and solving for the update that will set these terms to zero. In other words:

$$u^{n+1} = u^n + \hat{u} \tag{8.11}$$

where $\hat{u} \in V$ is the solution to:

$$J(u^n;v,\hat{u}) = -f(u^n;v) \qquad \forall v \in V. \tag{8.12}$$

In fact, (8.12) is simply a linear finite element problem! To make this explicit, we can expand v and $\hat{u}$ in terms of basis functions:

$$J(u^n;\phi_i,\phi_j)\hat{u}_j = -f(u^n;\phi_i). \tag{8.13}$$

For our nonlinear diffusion problem, the matrix J is given by:

$$J(u^n;\phi_i,\phi_j) = \begin{cases} \displaystyle\int_\Omega \nabla\phi_i \cdot (\phi_j \nabla u^n + (u^n+1)\nabla\phi_j)\ \mathrm{d}x & \phi_i \in V_0 \\ \delta_{ij} & \phi_i \in V_\Gamma, \end{cases} \tag{8.14}$$

and the right hand side vector f is given by (8.5). This matrix, J, is termed the *Jacobian matrix* of f.

8.4.1 Stopping criteria for Newton's method

Since Newton's method is an iterative algorithm, it creates a (hopefully convergent) sequence of approximations to the correct solution to the original nonlinear problem. How do we know when to accept the solution and terminate the algorithm?

The answer is that the update, $\hat{u}$ which is calculated at each step of Newton's method is itself an approximation to the error in the solution. It is therefore appropriate to stop Newton's method when this error estimate becomes sufficiently small in the L^2 norm.

The observant reader will observe that $\hat{u}$ is in fact an estimate of the error in the *previous* step. This is indeed true: the Newton step is both an estimate of the previous error and a correction to that error. However, having calculated the error estimate, it is utterly unreasonable to not apply the corresponding correction.

Note: Another commonly employed stopping mechanism is to consider the size of the residual f. However, the residual is not actually a function in V, but is actually a linear operator in V^*. Common practice would be to identify f with a function in V by simply taking the function whose coefficients match those of f. The L^2 or l^2 norm is then taken of this function and this value is used to determine when convergence has occured.

This approach effectively assumes that the Riesz map on V is the trivial operator which identifies the basis function coefficients. This would be legitimate were the inner product on V the l^2 dot product. However, since the inner product on V is defined by an integral, the mesh resolution is effectively encoded into f. This means that this approach produces convergence rates which depend on the level of mesh refinement.

Avoiding this mesh dependency requires the evaluation of an operator norm or, equivalently, the solution of a linear system in order to find the Riesz representer of f in V. However, since the error-estimator approach given above is both an actual estimate of the error in the solution, and requires no additional linear solves, it should be regarded as a preferable approach. For a full treatment of Newton methods, see [Deu11].

8.4.2 Stopping threshold values

What, then, qualifies as a sufficiently small value of our error estimate? There are two usual approaches:

relative tolerance Convergence is deemed to occur when the estimate becomes sufficiently small compared with the first error estimate calculated. This is generally the more defensible approach since it takes into account the overall scale of the solution. 10^{-6} would be a reasonably common relative tolerance.

absolute tolerance Computers employ finite precision arithmetic, so there is a limit to the accuracy which can ever be achieved. This is a difficult value to estimate, since it depends on the number and nature of operations undertaken in the algorithm. A common approach is to set this to a very small value (e.g. 10^{-50}) initially, in order to attempt to ensure that the relative tolerance threshold is hit. Only if it becomes apparent that the problem being solved is in a regime for which machine precision is a problem is a higher absolute tolerance set.

It is important to realise that both of these criteria involve making essentially arbitrary judgements about the scale of error which is tolerable. There is also a clear trade-off between the level of error tolerated and the cost of performing a large number of Newton steps. For realistic problems, it is therefore frequently expedient and/or necessary to tune the convergence criteria to the particular case.

In making these judgements, it is also important to remember that the error in the Newton solver is just one of the many sources of error in a calculation. It is pointless to expend computational effort in an attempt to drive the level of error in this component of the solver to a level which will be swamped by a larger error occurring somewhere else in the process.

8.4.3 Failure modes

Just as with the Newton method for scalar problems, Newton iteration is not guaranteed to converge for all nonlinear problems or for all initial guesses. If Newton's method fails to converge, then the algorithm presented so far constitutes an infinite loop. It is therefore necessary to define some circumstances in which the algorithm should terminate having failed to find a solution. Two such circumstances are commonly employed:

maximum iterations It is a reasonable heuristic that Newton's method has failed if it takes a very large number of iterations. What constitutes "too many" is once again a somewhat arbitrary judgement, although if the approach takes many tens of iterations this should always be cause for reconsideration!

diverged error estimate Newton's method is not guaranteed to produce a sequence of iterations which monotonically decrease the error, however if the error estimate has increased to, say, hundreds or thousands of times its initial value, this would once again be grounds for the algorithm to fail.

Note that these failure modes are heuristic: having the algorithm terminate for these reasons is really an instruction to the user to think again about the problem, the solver, and the initial guess.

8.5 Implementing a nonlinear problem

This problem will be released in the middle of the term.

BIBLIOGRAPHY

[Cia02] Philippe G Ciarlet. *The finite element method for elliptic problems*. Elsevier, 2002. doi:10.1137/1.9780898719208.

[Deu11] Peter Deuflhard. *Newton Methods for Nonlinear Problems*. Springer, 2011. doi:10.1007/978-3-642-23899-4.

[Kir04] R.C. Kirby. Algorithm 839: fiat, a new paradigm for computing finite element basis functions. *ACM Transactions on Mathematical Software (TOMS)*, 30(4):502–516, 2004. doi:10.1145/1039813.1039820.

[Log09] A. Logg. Efficient representation of computational meshes. *International Journal of Computational Science and Engineering*, 4(4):283 – 295, 2009. doi:10.1504/IJCSE.2009.029164.