Femtools manual

February 13, 2013

Contents

Part I

The finite element method

Chapter 1

Finite approximations to functions

1.1 Functions on the unit interval

Suppose that we wish to represent functions on the unit interval. On this interval, there are two local coordinates $_1$ and $_2$ linked by the relation:

$$_1 + \ _2 = 1 \tag{1.1}$$

Note that there are two local variables for a one dimensional element. For this reason, one of the local coordinates may be regarded as an auxiliary variable expressed in terms of the other one.

In terms of $_2$, the specimen element is the interval $[0; 1]$. In $_1$ this becomes the reversed interval $[1; 0]$. Linear functions on this element may be expressed in the form:

$$F(\ ) = f_1 \ _{1,1} + f_2 \ _{1,2} \tag{1.2}$$

where the basis functions $_{1,1}$, $_{1,2}$ are given by:

$$_{1,1}(\ ) = \ _1 \tag{1.3}$$
$$_{1,2}(\ ) = \ _2 \ (= 1 - \ _1) \tag{1.4}$$

Of course we need not only consider linear functions. For example, we can express any quadratic function on the unit interval as the sum of three basis functions:

$$_{2,1}(\ ) = \ _1(2 \ _1 - 1) \tag{1.5}$$
$$_{2,2}(\ ) = 4 \ _1 \ _2 \tag{1.6}$$
$$_{3,3}(\ ) = \ _2(2 \ _2 - 1) \tag{1.7}$$

Figure 1.1 shows the basis functions for linear and quadratic functions on the unit interval.

More generally, these functions belong to the family of equispaced Lagrange elements known as P*n* where *n* is the degree of the polynomial basis function. The basis functions of these elements are defined by their value at a set of $n + 1$ equispaced nodes n_i. The th basis function for P*n* is defined to be the unique polynomial of degree *n* such that:

$$_{n,k} = \begin{cases} _{n,\alpha}(\mathsf{n}_\alpha) = 1 \\ _{n,\alpha}(\mathsf{n}_\beta) = 0 \qquad \neq \end{cases} \tag{1.8}$$

1.2 One dimensional meshes

As well as representing functions on an interval by higher degree polynomials, we may also subdivide the interval into a number of subintervals and represent a function by a polynomial on each

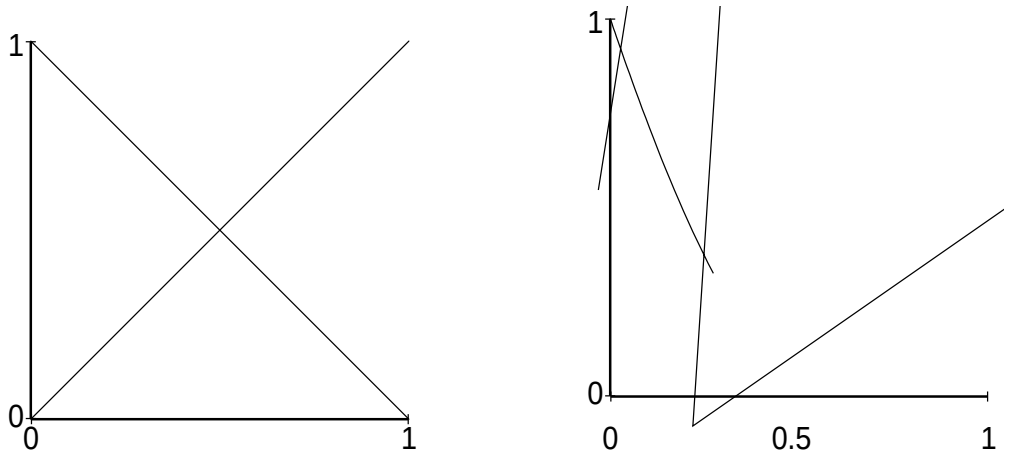


Figure 1.1: Basis functions for linear (left) and quadratic (right) functions on the interval $_1 = [0; 1]$

subdivision. To introduce terminology which will also be applicable in more dimensions, we refer to the whole interval as the *domain*, which we denote Ω and we refer to each sub-interval as an *element*. We will denote the current element E, possibly adding subscripts where required to avoid ambiguity.

For simplicity, let us choose to approximate the function $f(x)$ on the one dimensional domain (interval) Ω using piecewise linear functions. At this stage we shall also require that our piecewise linear function be continuous. We first partition Ω into N_E elements by choosing a set of nodes $\{x_\alpha | x_\alpha < x_{\alpha+1}; \quad = 1 : : : N_{\text{dof}}\}$ such that $E_\alpha = [x_\alpha; x_{\alpha+1}]$. On an (non-periodic) interval, with a continuous piecewise linear representation of the function $N_{\text{dof}} = N_E + 1$. On more complex domains, the relationship between the number of elements and number of nodes is far more complex. Figure 1.2 shows the piecewise linear approximation of a function over an interval subdivided into elements.

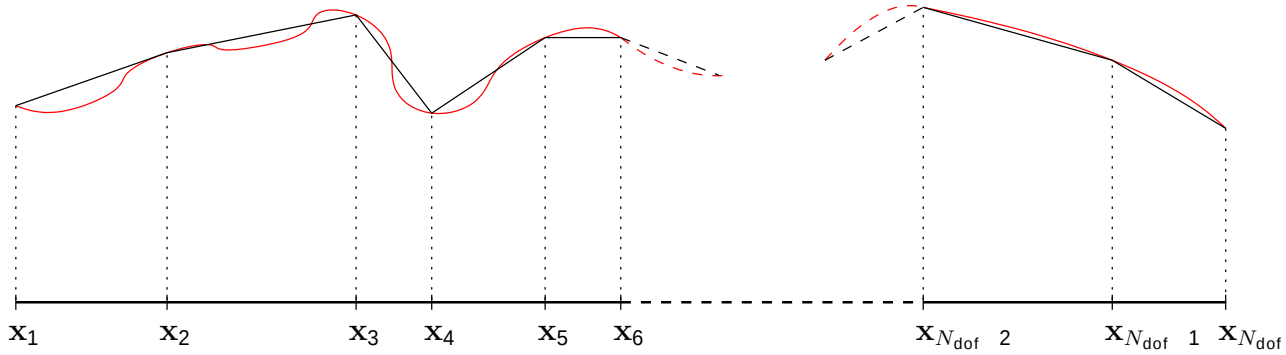


Figure 1.2: A piecewise linear approximation on a one-dimensional interval. The original function is shown in red and its piecewise linear interpolant over the set of nodes x_α is black.

Clearly the value of the piecewise linear approximation $F(x)$ is uniquely determined by the values of F at the nodes. From the previous section, it comes as no surprise that we may write our function:

$$F(x) = \sum_{\alpha=1}^{N_E+1} f_\alpha \ _\alpha(x) \tag{1.9}$$

where:

$$_\alpha(x) = \begin{cases} _{1,1}(\ ); \ _2(x) = \frac{x \ x_\alpha}{x_{\alpha+1} \ x_\alpha}; \ _1 = 1 - \ _2 & : x \in E_\alpha \\ _{1,2}(\ ); \ _2(x) = \frac{x \ x_{\alpha-1}}{x_\alpha \ x_{\alpha-1}}; \ _1 = 1 - \ _2 & : x \in E_{\alpha\ 1} \\ 0 & : \text{otherwise} \end{cases} \tag{1.10}$$

There are a number of features of this form of basis function which make it attractive. First, we write

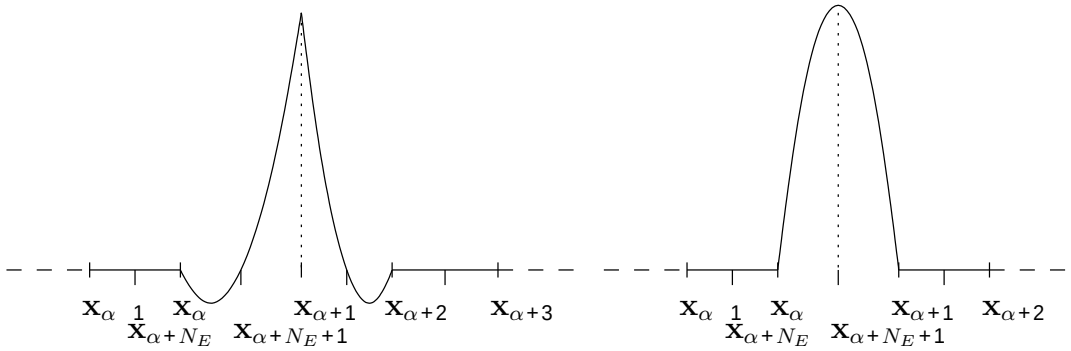


Figure 1.4: Basis functions for a one-dimensional continuous piecewise quadratic function. On the left, the basis function corresponding to a node on an element boundary. On the right, that corresponding to a node in the centre of an element.

In other words, the function maps from the local node numbering on each element to the global node numbering.

We shall also abbrevate the notation somewhat so that $_\beta$ refers to the th local basis function of the basis we are currently considering. We shall write Φ for the basis as a whole so that $\Phi = \{ _\beta | = 1 ::: N_{\text{loc}}\}$. These definitions now allow us to write a function F over a domain $\Omega = \{E_\alpha | = 1 ::: N_{\text{dof}}\}$ defined with respect to some basis Φ as:

$$F(\mathsf{x}) = \sum_{\beta=1}^{N_{\text{loc}}} f_{\gamma(\alpha(\mathbf{x}),\beta)} \ _\beta(\ ) \tag{1.15}$$

The pair $[\Omega; \Phi]$ defines a discrete function space containing all such functions F. It is important to remember that (1.15) is completely equivalent to (1.9) on the assumption (which is true for all commonly employed finite elements) that the basis functions have local support. From a computational perspective, the significant advantage of (1.15) is that it involves only a sum over the local nodes of element (x), the element containing the point at which the function is to be evaluated, rather than a sum over all of the basis functions in the mesh, most of which will be zero in element (x).

1.3 Function arithmetic

If we have two functions $F; G \in [\Omega; \Phi]$ then we can write their sum:

$$F(\mathsf{x}) + G(\mathsf{x}) = \sum_{\beta=1}^{N_{\text{loc}}} f_{\gamma(\alpha(\mathbf{x}),\beta)} \ _\beta(\ ) + \sum_{\beta'=1}^{N_{\text{loc}}} g_{\gamma(\alpha(\mathbf{x}),\beta')} \ _{\beta'}(\ ) \tag{1.16}$$

By simply gathering like terms, we can rewrite this as:

$$F(\mathsf{x}) + G(\mathsf{x}) = \sum_{\beta=1}^{N_{\text{loc}}} f_{\gamma(\alpha(\mathbf{x}),\beta)} + g_{\gamma(\alpha(\mathbf{x}),\beta)} \ _\beta(\ ) \tag{1.17}$$

In other words, we can add functions in the same function space simply by adding their corresponding basis function coefficients. Similarly, we can multiply functions by any constant scalar k:

$$kF(\mathsf{x}) = \sum_{\beta=1}^{N_{\text{loc}}} kf_{\gamma(\alpha(\mathbf{x}),\beta)} \ _\beta(\ ) \tag{1.18}$$

In the case of the product of two F and G, the situation is somewhat more complicated:

$$\begin{aligned} F(\mathbf{x})G(\mathbf{x}) &= \left(\sum_{\beta=1}^{N_{\mathrm{bc}}} f_{\gamma(\alpha(\mathbf{x}),\beta)}\ \phi_\beta(\ )\right)\left(\sum_{\beta'=1}^{N_{\mathrm{bc}}} g_{\gamma(\alpha(\mathbf{x}),\beta')}\ \phi_{\beta'}(\ )\right) \\ &= \sum_{\beta=1}^{N_{\mathrm{bc}}}\sum_{\beta'=1}^{N_{\mathrm{bc}}} f_{\gamma(\alpha(\mathbf{x}),\beta)} g_{\gamma(\alpha(\mathbf{x}),\beta')}\ \phi_\beta(\ )\ \phi_{\beta'}(\ ) \end{aligned} \tag{1.19}$$

Note that, unlike the cases of addition and multiplication by a scalar, multiplication of two functions is *not*

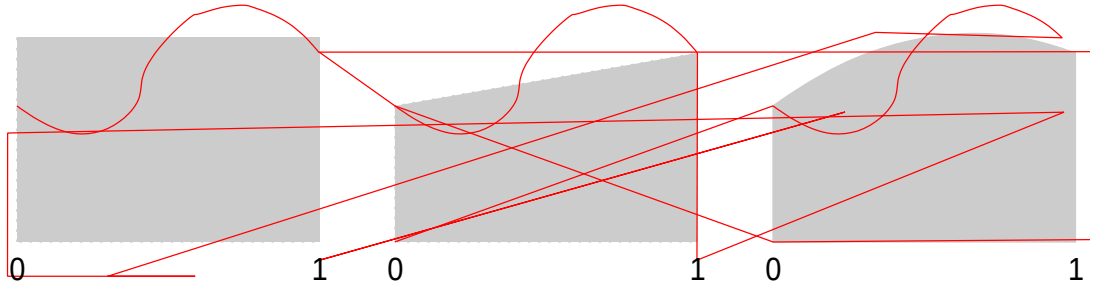


Figure 1.5: Numerical integration of a function on the interval $[0;1]$. From left to right, the midpoint rule, the trapezoidal rule and Simpson's rule. The points at which the function is evaluated are marked by black disks.

function but approximates the integrand by a quadratic function:

$$\begin{aligned}\int_0^1 F(\xi)\,\mathrm{d}\xi_1 &\approx \frac{1}{6}F([1;0]) + \frac{2}{3}F([0{:}5;0{:}5]) + \frac{1}{6}F([0;1]) \\ &= \sum_{\beta=1}^{N_{\mathrm{loc}}} f_\beta \left(\frac{1}{6}\,\phi_\beta([1;0]) + \frac{2}{3}\,\phi_\beta([0{:}5;0{:}5]) + \frac{1}{6}\,\phi_\beta([0;1])\right)\end{aligned} \tag{1.24}$$

These three methods are particular examples of the general form of numerical integration, otherwise known as *quadrature*. In general, a quadrature rule consists of a set of points $\{\xi_\delta | \delta = 1 \ldots N_{\mathrm{quad}}\}$ and a corresponding set of weights $\{w_\delta | \delta = 1 \ldots N_{\mathrm{quad}}\}$. The integral is approximated by:

$$\int_0^1 F(\xi)\,\mathrm{d}\xi_1 \approx \sum_{\beta=1}^{N_{\mathrm{loc}}} \sum_{\delta=1}^{N_{\mathrm{quad}}} f_\beta w_\delta \phi_\beta(\xi_\delta) \tag{1.25}$$

Many such rules exist and, as with the simple schemes above, it is generally the case that schemes which deliver lower error have larger sets of quadrature points and therefore require more shape function evaluations. If the basis functions Φ are piecewise polynomial then for sufficiently accurate quadrature (and accordingly sufficiently many quadrature points), the integral is exactly evaluated without error. This circumstance is referred to as complete quadrature. In the following sections we shall assume that the quadrature is complete so that the approximate equality ($\approx$) in (1.25) is exact ($=$).

1.4.2 Integration over a one dimensional domain

Having established a numerical integration algorithm in a single element, we will move to integration over a whole one-dimensional mesh. The full integral is simply the sum of the integrals on each element. The integral of a function $F \in [\Omega;\Phi]$ over the domain Ω is therefore:

$$\int_\Omega F(\mathsf{x})\mathrm{d}x = \sum_{\alpha=1}^{N_E} \int_{E_\alpha} F(\mathsf{x})\mathrm{d}x \tag{1.26}$$

In the previous section, we described integration over the specimen interval with respect to the local variable ξ_1. To evaluate (1.26) we therefore need to change coordinates on each element to the local variables:

$$\int_{E_\alpha} F(\mathsf{x})\mathrm{d}x = \int_0^1 F(\mathsf{x})\frac{\mathrm{d}x}{\mathrm{d}\xi_2}\mathrm{d}\xi_2 \tag{1.27}$$

where (1.12) is used to evaluate $F(\mathsf{x})$ in terms of ξ. Using the definition of ξ from page 13 we have:

$$\begin{aligned}\frac{\mathrm{d}\xi_2}{\mathrm{d}x} &= \frac{1}{\mathsf{x}_{\gamma(\alpha+1,2)} - \mathsf{x}_{\gamma(\alpha,1)}} \\ &= \frac{1}{\Delta\mathsf{x}_\alpha}\end{aligned} \tag{1.28}$$

where Δx_α is the width of element . So:

$$\frac{dx}{d\ _2} = \Delta x_\alpha \tag{1.29}$$

The choice to express the integral with respect to the second local coordinate is arbitrary. We could just as well use $_1$ however in this case it would be necessary to take into account the fact that $_1$ increases as x decreases so we would have:

$$\begin{aligned}\int_{E_\alpha} F(x)dx &= \int_1^0 F(x)\frac{dx}{d\ _1}d\ _1 \\ &= -\int_0^1 F(x)\frac{dx}{d\ _1}d\ _1\end{aligned} \tag{1.30}$$

Since $_1 = 1 - \ _2$,

$$\frac{dx}{d\ _1} = -\Delta x_\alpha \tag{1.31}$$

and so equation (1.30) reduces to equation (1.26).

If F lies in the span of some basis then we can apply a suitable complete quadrature as given in (1.25) to write:

$$\int F(x)dx = \sum_{\alpha=1}^{N_E}\sum_{\beta=1}^{N_{loc}}\sum_{\delta=1}^{N_{quad}} f_{\beta\ \beta(\ \delta)}\,\Delta x_\alpha w_\delta \tag{1.32}$$

1.5 Numerical differentiation

Since we were able to integrate a discretised function by integrating its basis functions, we might try the corresponding approach to differentiate a function:

$$\begin{aligned}\frac{dF(x)}{dx} &= \frac{d}{dx}\sum_{\beta=1}^{N_{loc}} f_{\gamma(\alpha(x),\beta)\ \ \beta(\ )} \\ &= \sum_{\beta=1}^{N_{loc}} f_{\gamma(\alpha(x),\beta)}\frac{d\ _{\beta(\ )}}{dx}\end{aligned} \tag{1.33}$$

Once again we can employ the definition of in terms of x in (1.12) to change variables via the chain rule:

$$\begin{aligned}\frac{dF(x)}{dx} &= \sum_{\beta=1}^{N_{loc}} f_{\gamma(\alpha(x),\beta)}\frac{d\ _{\beta(\ )}}{dx} \\ &= \sum_{\beta=1}^{N_{loc}} f_{\gamma(\alpha(x),\beta)}\frac{d\ _{\beta(\ )}}{d\ _2}\frac{d\ _2}{dx} \\ &= \sum_{\beta=1}^{N_{loc}} f_{\gamma(\alpha(x),\beta)}\frac{1}{\Delta x_{\alpha(x)}}\frac{d\ _{\beta(\ )}}{d\ _2}\end{aligned} \tag{1.34}$$

This does indeed yield the exact derivative of F at every point in the domain. However, if $F \in [\Omega; \Phi]$ it will not generally be the case that the derivative of F, when calculated according to (1.34), will itself lie in $[\Omega; \Phi]$. As an example, suppose Ω is the interval $[0; 1]$ partitioned into the elements $[0; 1{=}3], [1{=}3; 2{=}3], [2{=}3; 1]$. If Φ is a basis for the space of continuous piecewise linear functions then the

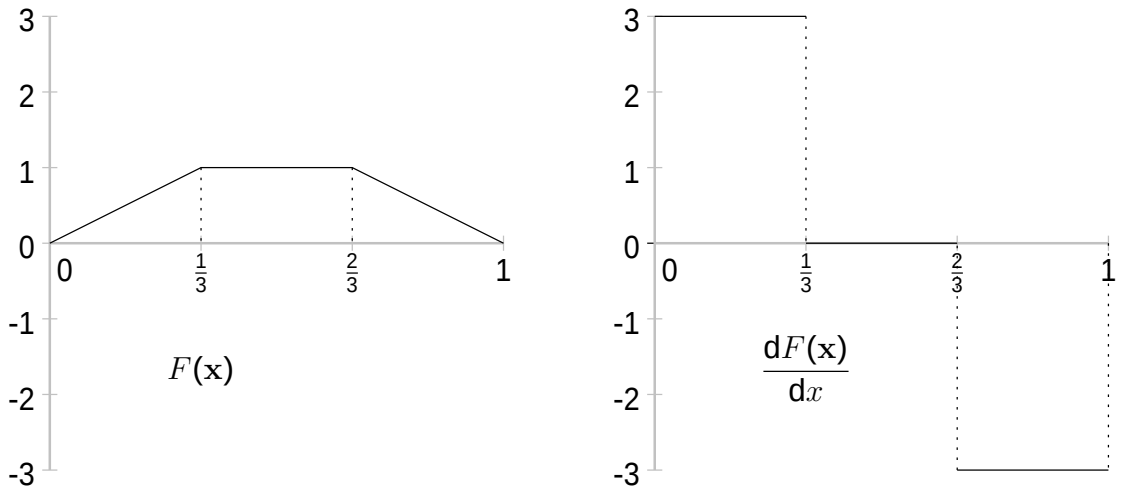


Figure 1.6: A continuous piecewise linear function and its derivative. The derivative is piecewise constant and discontinuous at element boundaries.

four nodes of the function space $[\Omega; \Phi]$ lie at 0, $1{=}3$, $2{=}3$ and 1. Let F be the function in $[\Omega; \Phi]$ given by the coefficient vector $[0; 1; 1; 0]$. Then the gradient of F is given by:

$$\frac{\mathrm{d}F(\mathsf{x})}{\mathrm{d}x} = \begin{cases} 3 & x \in [0; 1{=}3] \\ 0 & x \in [1{=}3; 2{=}3] \\ -3 & x \in [2{=}3; 1] \end{cases} \tag{1.35}$$

Figure 1.6 illustrates this function and its derivative. It is immediately apparent that the derivative is constant on each element and discontinuous at the boundaries between elements. On the element boundary itself, the derivative does not have a uniquely defined value. The derivative is therefore not a member of the space of piecewise linear continuous functions on Ω.

This result is general to a much wider class of function spaces. In particular, Pn, the space of continuous piecewise polynomial functions of degree n is not smooth at element boundaries and therefore has discontinuous derivatives.

However, for many purposes, such as the evaluation of expressions involving the derivative of a function, we will want to work in the same function space $[\Omega; \Phi]$ as the original function F. In other words, we will wish to solve the problem: for a given $F \in [\Omega; \Phi]$ find $D \in [\Omega; \Phi]$ such that

$$D(\mathsf{x}) = \frac{\mathrm{d}F(\mathsf{x})}{\mathrm{d}x} \tag{1.36}$$

Chapter 2

Weak forms and the Galerkin projection

2.1 Equality in the weak sense

We have seen above that if $[\Omega; \Phi]$ is not differentiable at a set of isolated points (the element boundaries) then (1.36) has no solutions. This suggests that we might look for solutions by weakening our test for equality so that the value at a set of isolated points can be neglected. To generalise the problem somewhat, suppose that f is a function defined over the domain Ω and we wish to find g in $[\Omega; \Phi]$ such that $g = f$.

Since the value of a function at an isolated set of points does not change its integral, we might ask if f and g are equal in the L^1 norm:

$$\int |g - f| \mathrm{d}x = 0 \tag{2.1}$$

An alternative is to ask the same question with respect to the L^2 norm:

$$\int (g - f)^2 \mathrm{d}x = 0 \tag{2.2}$$

An important generalisation is to say that f and g are equal if their integral when multiplied by any square-integrable function w is zero:

$$\int w(g - f) \mathrm{d}x = 0 \quad \forall w \in L^2 \tag{2.3}$$

The function w in this expression is termed a *test function*. In particular, if we choose the constant test function $w = 1$ then we recover (2.1) while if f and g also lie in L^2 (the space of functions with finite L_2 norm) then we can choose $w = g - f$ and (2.2) is recovered. It is therefore apparent that (2.3) is a stronger test for equality than either (2.1) or (2.2). It is also a definition which lends itself to discretisation. Suppose that we restrict w to lie in the discretised function space $[\Omega; \Phi]$, then we can rewrite (2.3) in terms of the *global* basis functions as:

$$\sum_{\gamma=1}^{N_{\text{dof}}} \int \gamma (g - f) \mathrm{d}x = 0 \quad \forall w \in L^2 \tag{2.4}$$

Since the global basis functions span $[\Omega; \Phi]$, it is sufficient to demonstrate this equality for *each* of the global basis functions:

$$\int \gamma (g - f) \mathrm{d}x = 0 \quad \forall \quad = 1 : : : N_{\text{dof}} \tag{2.5}$$

The next step is to rearrange the equation to place unknown terms on the left and known ones on the right. Since g also lies in $[\Omega; \Phi]$ then we can write the following system of equations:

$$\sum_{\hat{\gamma}=1}^{N_{\text{dof}}} \int \gamma f_{\hat{\gamma}} \, \hat{\gamma} \mathrm{d}x = \int \gamma g \mathrm{d}x \quad \forall \quad = 1 : : : N_{\text{dof}} \tag{2.6}$$

Equivalently, this can be written as a matrix equation:

$$\mathrm{M}\underline{g} = \underline{r} \tag{2.7}$$

where:

$$M(i;j) = \int \quad _i \quad _j \mathrm{d}x \tag{2.8}$$

$$\underline{g}(j) = g_j \tag{2.9}$$

$$\underline{r}(i) = \int \quad _i f \,\mathrm{d}x \tag{2.10}$$

with $i; j = 1 ::: N_{\mathrm{dof}}$. The matrix M is known in the finite element method as the *mass matrix*. It has the function of mapping between the test space and the space in which g is defined, which is termed the *trial space*.

2.1.1 Matrix assembly

The mass matrix in equation (2.8) is defined in terms of the global basis functions . However, as previously discussed, to efficiently evaluate the basis functions, we need to restate the problem in terms of the local basis functions on each element. Since each entry in M is an integral over the domain, M can also be expressed as a sum of integrals over each element. Remember that the basis functions have local support so the contribution from each element can be computed using only the local basis functions on each element. For each element we therefore have the following contribution to M:

$$\mathrm{M}_\alpha(i_{\mathrm{loc}}; j_{\mathrm{loc}}) = \int_{E_\alpha} \quad _{i_{\mathrm{loc}}} \quad _{j_{\mathrm{loc}}} \mathrm{d}x; \qquad i_{\mathrm{loc}}; j_{\mathrm{loc}} = 1 ::: N_{\mathrm{loc}} \tag{2.11}$$

To actually assemble M requires the use of the function , the map from local to global node numbers which we introduced on page 13. This produces the following algorithm:

```
do   =1, N_E
  do i_loc=1, N_loc
    do i_loc=1, N_loc
      M( ( ; i_loc); ( ; j_loc)) += ∫_{E_α}  _{i_loc}  _{j_loc} dx
    end
  end
end
```

The integral on the right-hand side is evaluated using numerical quadrature as shown in equation (1.25) although this time there are two shape functions:

$$\int_{E_\alpha} \quad _{i_{\mathrm{loc}}} \quad _{j_{\mathrm{loc}}} \mathrm{d}x = \sum_{q=1}^{N_{\mathrm{quad}}}$$

```
do   =1, N_E
  do i_loc=1, N_loc
    r( ( ;i_loc))+= ∫_{E_α} _{i_loc} g dx
  end
end
```

In this case the relevant quadrature rule is:

$$\int_{E_\alpha} {}_{i_{\text{loc}}} g \,\mathrm{d}x = \sum_{\delta=1}^{N_{\text{quad}}} {}_{i_{\text{loc}}}(\ {}_\delta)\, f(\ {}_\delta)\, \Delta \mathsf{x}_\alpha w_\delta; \qquad i_{\text{loc}} = 1 ::: N_{\text{loc}} \tag{2.14}$$

2.2 The weak form of numerical differentiation

Let us return to the original differentiation problem we set out to solve: given $F \in [\Omega; \Phi]$ find $D \in [\Omega; \Phi]$ such that

$$D(\mathsf{x}) = \frac{\mathrm{d}F(\mathsf{x})}{\mathrm{d}x} \tag{2.15}$$

Using equation (1.34) we can expand the derivative in terms of the basis functions and substitute for g into equation (2.14):

$$\int_{E_\alpha} {}_{i_{\text{loc}}} \frac{\mathrm{d}F(\mathsf{x})}{\mathrm{d}x} \mathrm{d}x = \sum_{\delta=1}^{N_{\text{quad}}} \sum_{\beta=1}^{N_{\text{loc}}} {}_{i_{\text{loc}}}(\ {}_\delta)\, f_{\gamma(\alpha,\beta)} \quad \frac{1}{\Delta \mathsf{x}_\alpha} \frac{\mathrm{d}\ {}_\beta(\ {}_\delta)}{\mathrm{d}\ {}_2} \quad \Delta \mathsf{x}_\alpha w_\delta; \qquad i_{\text{loc}} = 1 ::: N_{\text{loc}} \tag{2.16}$$

2.2.1 Numerical evaluation

Using this derivation and equations (2.16) and (2.12) in particular, we have defined the weak form of equation (2.15) only in terms of the following quantities:

- A set of quadrature points and corresponding quadrature weights on the specimen element;
- the values of the basis functions of the discrete space $[\Omega; \Phi]$ at the quadrature points of each element;
- the derivatives of those basis functions at the same quadrature points;
- the positions of the mesh nodes; and
- the values of the coefficient vector $\underline{f}$ defining the function F.

Let's use the Simpson's rule for quadrature. This rule uses three points per element so $N_{\text{quad}} = 3$ and we have the following quadrature points:

	1	2	w
$\delta=1$	1	0	1/6
$\delta=2$	0.5	0.5	2/3
$\delta=3$	0	1	1/6

Here ${}_{\delta=1}$ is the first quadrature point while in local coordinates while ${}_1$ the name of the first local coordinate.

Since we are using linear basis functions, their values and derivatives at these points are:

	1	2	$\frac{d\phi_1}{d\xi_1}$	$\frac{d\phi_1}{d\xi_2}$	$\frac{d\phi_2}{d\xi_1}$	$\frac{d\phi_2}{d\xi_2}$
$\delta=1$	1	0	1	-1	-1	1
$\delta=2$	0.5	0.5	1	-1	-1	1
$\delta=3$	0	1	1	-1	-1	1

Notice that the gradient of each basis function is constant across the element.

The mesh nodes are at $[0; 1=3; 2=3; 1]$ so that $N_E = 3$ and $\Delta x_\alpha = 1=3$ for any $\alpha \in \{1 : : : 3\}$. The coefficients of the function F are, as given before: $\underline{f} = [0; 1; 1; 0]$.

Because we have chosen an equispaced mesh, the local mass matrix M_α is the same on each element. It is given by:

$$\begin{aligned}
\mathrm{M}_\alpha(1;1) &= \int_{E_\alpha} \phi_1 \phi_1 dx \\
&= \sum_{\delta=1}^{N_{quad}(=3)} \phi_1(\xi_\delta)\, \phi_1(\xi_\delta)\, \Delta x_\alpha w_\delta \\
&= \phi_1(\xi_1)\, \phi_1(\xi_1) \frac{1}{3} w_1 + \phi_1(\xi_2)\, \phi_1(\xi_2) \frac{1}{3} w_2 + \phi_1(\xi_3)\, \phi_1(\xi_3) \frac{1}{3} w_3 \\
&= 1 \times 1 \times \frac{1}{3} \times \frac{1}{6} + 0.5 \times 0.5 \times \frac{1}{3} \times \frac{2}{3} + 0 \times 0 \times \frac{1}{3} \times \frac{1}{6} \\
&= \frac{2}{18}
\end{aligned} \tag{2.17}$$

$$\begin{aligned}
\mathrm{M}_\alpha(1;2) &= \int_{E_\alpha} \phi_1 \phi_2 dx \\
&= \sum_{\delta=1}^{N_{quad}(=3)} \phi_1(\xi_\delta)\, \phi_2(\xi_\delta)\, \Delta x_\alpha w_\delta \\
&= \phi_1(\xi_1)\, \phi_2(\xi_1) \frac{1}{3} w_1 + \phi_1(\xi_2)\, \phi_2(\xi_2) \frac{1}{3} w_2 + \phi_1(\xi_3)\, \phi_2(\xi_3) \frac{1}{3} w_3 \\
&= 1 \times 0 \times \frac{1}{3} \times \frac{1}{6} + 0.5 \times 0.5 \times \frac{1}{3} \times \frac{2}{3} + 1 \times 0 \times \frac{1}{3} \times \frac{1}{6} \\
&= \frac{1}{18}
\end{aligned} \tag{2.18}$$

$$\begin{aligned}
\mathrm{M}_\alpha(2;1) &= \int_{E_\alpha} \phi_2 \phi_1 dx \\
&= \sum_{\delta=1}^{N_{quad}(=3)} \phi_2(\xi_\delta)\, \phi_1(\xi_\delta)\, \Delta x_\alpha w_\delta \\
&= \phi_2(\xi_1)\, \phi_1(\xi_1) \frac{1}{3} w_1 + \phi_2(\xi_2)\, \phi_1(\xi_2) \frac{1}{3} w_2 + \phi_2(\xi_3)\, \phi_1(\xi_3) \frac{1}{3} w_3 \\
&= 0 \times 1 \times \frac{1}{3} \times \frac{1}{6} + 0.5 \times 0.5 \times \frac{1}{3} \times \frac{2}{3} + 0 \times 1 \times \frac{1}{3} \times \frac{1}{6} \\
&= \frac{1}{18}
\end{aligned} \tag{2.19}$$

$$
\begin{aligned}
\mathrm{M}_\alpha(2;2) &= \int_{E_\alpha} {}_2\,{}_2 \mathrm{d}x \\
&= \sum_{\delta=1}^{N_{\text{quad}}(=3)} {}_2(\ _\delta)\ \ {}_2(\ _\delta)\,\Delta \mathsf{x}_\alpha w_\delta \\
&= \ {}_2(\ _1)\ \ {}_2(\ _1)\,\frac{1}{3}w_1 + \ {}_2(\ _2)\ \ {}_2(\ _2)\,\frac{1}{3}w_2 + \ {}_2(\ _3)\ \ {}_2(\ _3)\,\frac{1}{3}w_3 \\
&= 1\times 1\times\frac{1}{3}\times\frac{1}{6} + 0{:}5\times 0{:}5\times\frac{1}{3}\times\frac{2}{3} + 0\times 0\times\frac{1}{3}\times\frac{1}{6} \\
&= \frac{2}{18}
\end{aligned}
\tag{2.20}
$$

These can be combined to yield:

$$
\mathrm{M}_\alpha = \begin{bmatrix} \frac{2}{18} & \frac{1}{18} \\ \frac{1}{18} & \frac{2}{18} \end{bmatrix} \tag{2.21}
$$

To combine the M_α for $\alpha = 1:::3$ into the global mass matrix we need the local to global mapping function . For this simple example, we can easily write this out. Recall that the first argument of is the element number while the second is the local node number.

$$
\begin{aligned}
(1;1) &= 1 \\
(1;2) &= 2 \\
(2;1) &= 2 \\
(2;2) &= 3 \\
(2;1) &= 3 \\
(2;2) &= 4
\end{aligned}
$$

Using this function in the algorithm from section 2.1.1, we may assemble the global mass matrix:

$$
M = \begin{bmatrix} \frac{2}{18} & \frac{1}{18} & 0 & 0 \\ \frac{1}{18} & \frac{2}{9} & \frac{1}{18} & 0 \\ 0 & \frac{1}{18} & \frac{2}{9} & \frac{1}{18} \\ 0 & 0 & \frac{1}{18} & \frac{2}{18} \end{bmatrix} \tag{2.22}
$$

We now need to follow an essentially similar process to evaluate the right hand side vector. Evaluating equation (2.14) for the first element ($== 1$), we have:

$$
\begin{aligned}
r_{\alpha=1}(1) &= \sum_{\delta=1}^{N_{\text{quad}}=3} \sum_{\beta=1}^{N_{\text{lo}}=2} {}_1(\ _\delta)\, f_{\gamma(1,\beta)} \quad \frac{1}{\Delta \mathsf{x}_1}\frac{\mathrm{d}\ _\beta(\ _\delta)}{\mathrm{d}\ _2} \quad \Delta \mathsf{x}_1 w_\delta \\
&= 1\times 0\times \quad \frac{1}{1=3}\times -1 \quad \times\frac{1}{3}\times\frac{1}{6} \\
&\quad + 0{:}5\times 0\times \quad \frac{1}{1=3}\times -1 \quad \times\frac{1}{3}\times\frac{2}{3} \\
&\quad + 0\times 0\times \quad \frac{1}{1=3}\times -1 \quad \times\frac{1}{3}\times\frac{1}{6} \\
&\quad + 1\times 1\times \quad \frac{1}{1=3}\times 1 \quad \times\frac{1}{3}\times\frac{1}{6} \\
&\quad + 0{:}5\times 1\times \quad \frac{1}{1=3}\times 1 \quad \times\frac{1}{3}\times\frac{2}{3} \\
&\quad + 0\times 1\times \quad \frac{1}{1=3}\times 1 \quad \times\frac{1}{3}\times\frac{1}{6} \\
&= \frac{1}{3}
\end{aligned}
\tag{2.23}
$$

$$
\begin{aligned}
r_{\alpha=1}(2) &= \sum_{\delta=1}^{N_{\text{quad}}=3} \sum_{\beta=1}^{N_{\text{loc}}=2} {}_2(\ _\delta)\, f_{\gamma(1,\beta)} \quad \frac{1}{\Delta x_1} \frac{\mathrm{d}\ _\beta(\ _\delta)}{\mathrm{d}\ _2} \quad \Delta x_1 w_\delta \\
&= 0 \times 0 \times \quad \frac{1}{1=3} \times -1 \quad \times \frac{1}{3} \times \frac{1}{6} \\
&\quad + 0{:}5 \times 0 \times \quad \frac{1}{1=3} \times -1 \quad \times \frac{1}{3} \times \frac{2}{3} \\
&\quad + 1 \times 0 \times \quad \frac{1}{1=3} \times -1 \quad \times \frac{1}{3} \times \frac{1}{6} \\
&\quad + 0 \times 1 \times \quad \frac{1}{1=3} \times 1 \quad \times \frac{1}{3} \times \frac{1}{6} \\
&\quad + 0{:}5 \times 1 \times \quad \frac{1}{1=3} \times 1 \quad \times \frac{1}{3} \times \frac{2}{3} \\
&\quad + 1 \times 1 \times \quad \frac{1}{1=3} \times 1 \quad \times \frac{1}{3} \times \frac{1}{6} \\
&= \frac{1}{3}
\end{aligned}
\tag{2.24}
$$

In summary, the right hand side contribution from the first element is:

$$
r_{\alpha=1} = \begin{bmatrix} \frac{1}{3} \\ \frac{1}{3} \\ \frac{1}{3} \end{bmatrix} \tag{2.25}
$$

Skipping the arithmetic details, the contributions from the other two elements are:

$$
r_{\alpha=2} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \tag{2.26}
$$

$$
r_{\alpha=3} = \begin{bmatrix} -\frac{1}{3} \\ -\frac{1}{3} \\ -\frac{1}{3} \end{bmatrix} \tag{2.27}
$$

Notice that the contribution from each element is piecewise constant. This stems directly from the fact that the gradients of the basis functions are constant over each element. Using the global assembly algorithm from section 2.1.2 we obtain:

$$
\begin{aligned}
\underline{r} &= \begin{bmatrix} \frac{1}{3} \\ \frac{1}{3} \\ \frac{1}{3} \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ -\frac{1}{3} \end{bmatrix} \\
&= \begin{bmatrix} \frac{1}{3} \\ \frac{1}{3} \\ -\frac{1}{3} \end{bmatrix}
\end{aligned}
\tag{2.28}
$$

This allows us to form the equation for $\underline{g}$:

$$
\begin{bmatrix} \frac{2}{18} & \frac{1}{18} & 0 & 0 \\ \frac{1}{18} & \frac{2}{9} & \frac{1}{18} & 0 \\ 0 & \frac{1}{18} & \frac{2}{9} & \frac{1}{18} \\ 0 & 0 & \frac{1}{18} & \frac{2}{18} \end{bmatrix} \underline{g} = \begin{bmatrix} \frac{1}{3} \\ \frac{1}{3} \\ -\frac{1}{3} \\ -\frac{1}{3} \end{bmatrix} \tag{2.29}
$$

This system can be solved using a computer numerics package such as Numerical Python, Octave or Matlab or (for the very keen) by hand to produce:

$$
\underline{g} = \begin{bmatrix} 2{:}4 \\ 1{:}2 \\ -1{:}2 \\ -2{:}4 \end{bmatrix} \tag{2.30}
$$

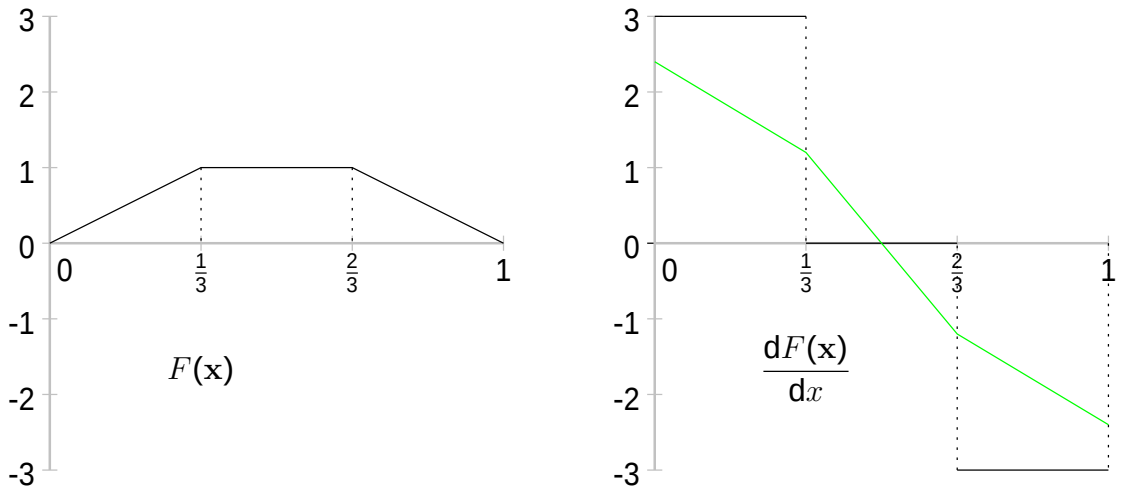


Figure 2.1: A continuous piecewise linear function and two diffent realisations of its derivative. The pointwise derivative (black) is piecewise constant and discontinuous at element boundaries. The green line shows a piecewise linear and continuous approximation to the derivative calculated using a Galerkin projection.

$\underline{g}$ is our continuous linear approximation to the gradient of F. Figure 2.1 shows $\underline{g}$ in green. The difference between the green line and the black pointwise gradient is the error in the approximation which is caused by projecting the discontinuous pointwise gradient into the space of continuous piecewise linear functions. This projection, which is performed by the inversion of the mass matrix M onto the right hand side vector r is known as the *Galerkin* projection and is the basis of the finite element method. The magnitude of the error in the projection is affected by the mesh resolution, the function spaces involved and the function being approximated. An exploration of these possibilities, however, depends on first getting a computer to do the assembly for us.

Chapter 3

Extension to more dimensions

3.1 Local coordinates on triangles and tets

In section 1.1 we introduced the local coordinate system on a one-dimensional unit element. To recapitulate, there are two local coordinates, $_1$ and $_2$ with the property that:

$$_1 + \ _2 = 1 \tag{3.1}$$

$_1$ takes the value 1 at the first end of the interval and 0 at the second end of the interval and varies linearly in between while $_2$ is 1 at the second end of the interval and 0 at the first. For two dimensional simplices (triangles) and three-dimensional simplices (tetrahedra), the local coordinate system on an element is defined in an analogous manner. On a triangle there are three local coordinates, $_1$, $_2$ and $_3$. Once again, these have the property that:

$$_1 + \ _2 + \ _3 = 1 \tag{3.2}$$

so that there are actually two independent coordinates. The vertices of the triangle are labelled 1, 2 and 3 and local coordinate $_n$ takes the value 1 at vertex *n* and 0 at each of the other two vertices. Figure 3.1 shows the value of $_1$ over an element. The value of each coordinate varies linearly along lines joining the corresponding vertex to the opposite edge of the triangle and is constant along lines parallel to that edge. The local coordinates of the point *x* in figure 3.1 are therefore given by:

$$= \begin{matrix} 2 & 3 \\ \frac{a}{a+b} \\ 4 \frac{c}{c+d} 5 \\ \frac{e}{e+f} \end{matrix} \tag{3.3}$$

where the symbols $a:::f$ denote the lengths of the intervals indicated in the figure.

The local coordinates on a tetrahedron are similar except that in this case there are four local coordinates. Once again, $_n$ is 1 at vertex *n* and in this case zero over the face opposite that vertex.

3.2 Integration over an element

Integration over an element in two or three dimensions is achieved using numerical quadrature in a manner similar to equation (1.25). That is to say, if we have a function:

$$F(\ ) = \sum_{i_{\text{loc}}=1}^{N_{\text{loc}}} f_i \ _i(\ ) \tag{3.4}$$

Then:

$$\int_E F(\ )\mathrm{d} \ _1 ::: \mathrm{d} \ _{N_{\text{dim}}} \approx \sum_{\beta=1}^{N_{\text{loc}}} \sum_{\delta=1}^{N_{\text{quad}}} f_\beta \ _\beta(\ _\delta) w_\delta \tag{3.5}$$

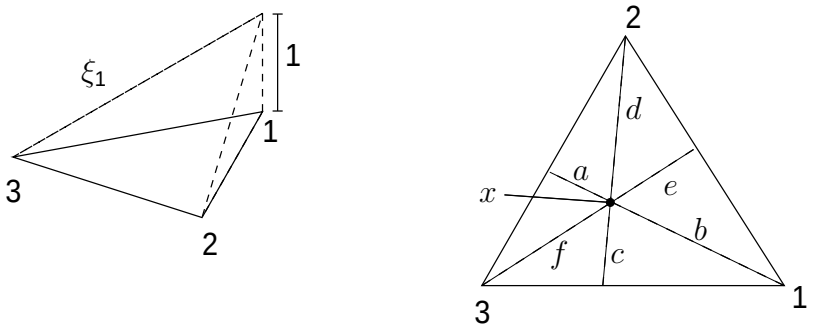


Figure 3.1: Local coordinates of the triangle. Left, the value of the local coordinate $_1$ over each point in the triangle. Right, the quantities required to calculate the local coordinates of the point x.

where $\{ _\delta| \ = 1 : : : N_{\text{quad}}\}$ is a set of quadrature points and $\{w_\delta| \ = 1 : : : N_{\text{quad}}\}$ are the corresponding weights. As previously, for complete quadrature, the approximation becomes exact and we will assume complete quadrature and write $=$ rather than $\approx$. We will also henceforth write a single integral sign for an integral in any number of dimensions.

In general, of course, we will wish to integrate over an element in a mesh, that is to say to integrate with respect to the physical coordinates x rather than the local coordinates . The change of coordinates function in more dimensions is:

$$\int_E F(\mathsf{x})\mathrm{d}\mathsf{x} = \int_E F(\ ) \ J \ ^1 \ \mathrm{d} \tag{3.6}$$

Where J is the Jacobian matrix of the coordinate transformation:

$$J = \begin{bmatrix} \frac{\partial \xi_1}{\partial x_1} & \cdots & \frac{\partial \xi_1}{\partial x_{N_{\text{dim}}}} \\ \vdots & \ddots & \vdots \\ \frac{\partial \xi_{N_{\text{dim}}}}{\partial x_1} & : : : & \frac{\partial \xi_{N_{\text{dim}}}}{\partial x_{N_{\text{dim}}}} \end{bmatrix} \tag{3.7}$$

and therefore $J \ ^1$ is the inverse Jacobian determinant. Note that in one dimension:

$$J \ ^1 = \frac{@x_1}{@ \ _1} \tag{3.8}$$

which reduces to the previous one-dimensional case.

Part II

Key data structures

Chapter 4

Common data structure features

4.1 The Femtools data structure heirarchy

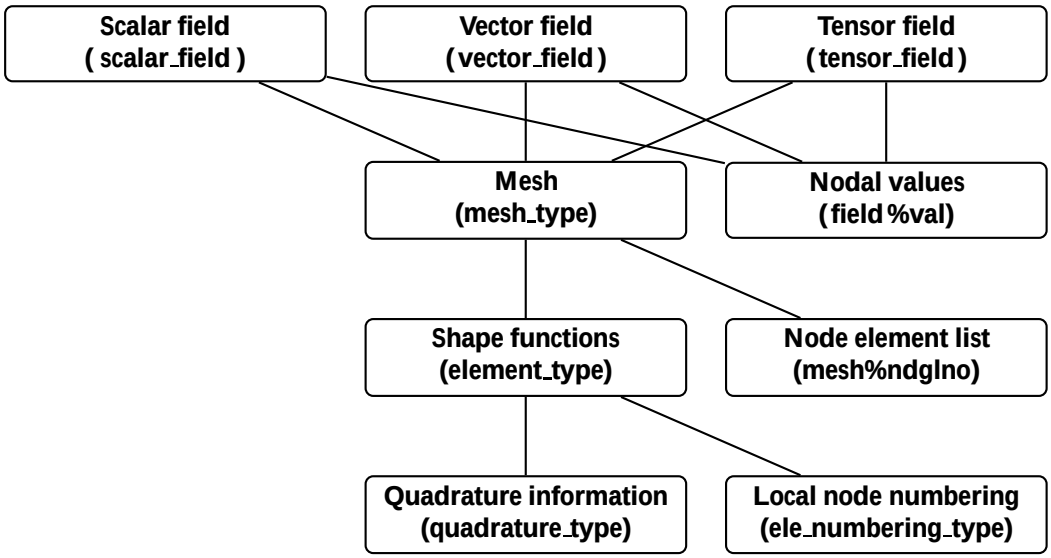


Figure 4.1: The dependency heirarchy of the field data structures in Femtools. From the top, field data types associciate a mesh (function space) with the nodal values of the field. In turn, the mesh has a specimen element and a list of the global node numbers associated with each element. In turn, the element shape functions depend on a quadrature rule and a local node numbering convention.

4.2 Object descriptors vs. data space

A Femtools data object encapsulates one or more (possibly large) arrays containing data with some descriptive information such as the object name and option path. We shall refer to the data arrays as the *data space* of the object. The objects are implemented as a Fortran derived type which contains components containing the descriptive information and components which are pointers to the data space. We refer to this derived type as the *object descriptor*. The significance of the distinction between the data space and the object descriptor flows from the semantics of assignment in fortran. If `a` and `b` are Femtools data objects of the same type then the assignment:

```
a=b
```

causes `a` to be a copy of the object descriptor of `b`. The data space common between `a` and `b`. Alternatively, if `a` is a pointer to a data object with the same type as `b` then:

```
a=>b
```

results in `a` and `b` refering to both the same descriptor and the same data space.

In most circumstances, it does not matter whether copies of field descriptors are made and to which copy reference is made. Most operations which modify data objects act on the data space which is always shared. Similarly, reference counting applies to the data space, not to the object descriptors and copies of object descriptors can typically be discarded when no longer needed without any impact on memory or data integrity.

However, when taking any action which changes the object descriptor, it is necessary to take into account that only *that* descriptor and not any copies will be modified. Operations which change the object descriptor include allocation and deallocation, changing the name or options path of an object and marking a field as aliased. This list is not exhaustive.

4.3 Option paths: integrating Femtools and Spud

Femtools is designed to be used with the Spud options system for scientific software. It is not necessary to use Spud in order to use Femtools, however the use of Spud is encouraged and is facilitated by the inclusion of `option_path` components in many object descriptors. The `option_path` component should be used to store the Spud option path pertaining to that object and below which options controlling that object are to be found. For further information on the use of Spud see http://amcg.ese.ic.ac.uk/spud.

Chapter 5

Quadrature

Linear and bilinear forms are integrated over elements using numerical quadrature as described in sections 1.4 and 3.2. The quadrature data type contains the information encoding a quadrature rule on a single element. A quadrature rule is defined by a set of quadrature points and the weights corresponding to those points. For example, there is a degree 3 quadrature rule for triangles with the following weights and positions:

1	2	3	w
0:33333333	0:33333333	0:33333333	$-0:28125$
0:2	0:2	0:6	0:26041666
0:2	0:6	0:2	0:26041666
0:6	0:2	0:2	0:26041666

Chapter 6

Cells

Finite element basis functions are defined over individual topological cells. Femtools currently supports interval cells in 1D, triangle and quadrialateral (quad) cells in 2D, and tetrahedral (tet) and hexahedral (hex) cells in 3D. For the purposes of numbering conventions and the like, these are split into the simplex family (intervals, triangles and tets) and the hypercube family (quads and hexes). The inclusion of the interval in the simplex family is arbitrary as the interval is both the 1-simplex and the 1-hypercube.

Figure 6.1: The reference interval.

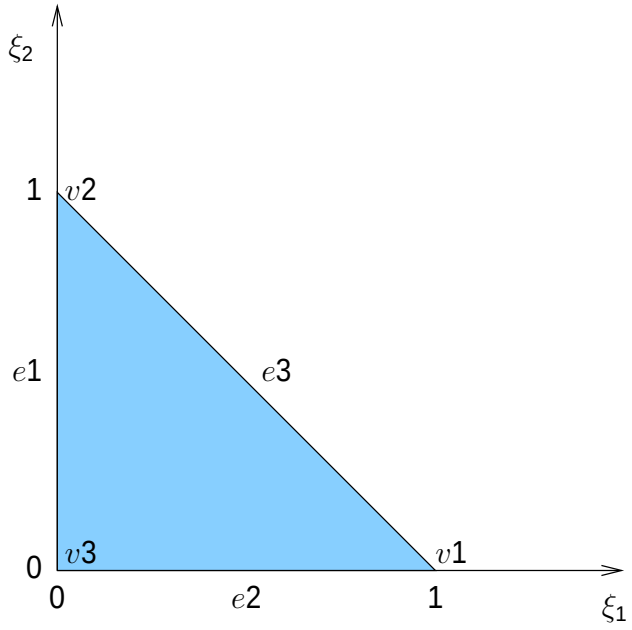


Figure 6.2: The reference triangle.

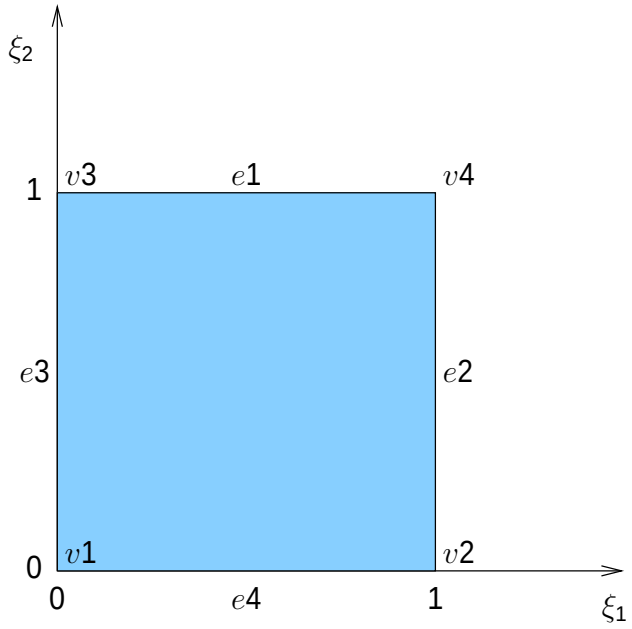
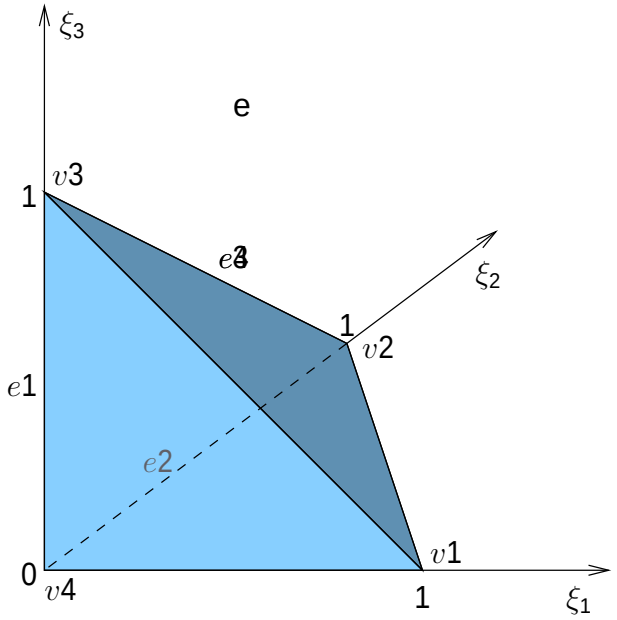


Figure 6.3: The reference quadrilateral.


ξ_3
e
$v3$
1
$e3$
ξ_2
1
$v2$
$e1$
$e2$
$v1$
0
$v4$
1
ξ_1

Chapter 7

Elements

The element_type data structure holds shape function information for a single specimen element. The element_type holds a reference to a particular quadrature_type object and stores the shape function information with respect to that quadrature object.

7.1 Shape functions at quadrature points

Finite element operations (that is, the evaluation of integrals over an element) only require the shape functions and their derivatives to be evaluated at the quadrature points. To enable the efficient evaluation of these integrals, the element objects directly store the value of each shape function on the element at each quadrature point on the element. For example, a quadratic triangle element has six basis functions. If the four-point quadrature rule shown in chapter 5 is employed then the element_type corresponding to this element will store the following matrix:

$$\mathrm{N} = \; :::: \tag{7.1}$$

7.2 Evaluating shape functions at arbitrary points

7.3 Local node numbering

7.4 Changing coordinates

Chapter 8

Meshes

A mesh is a collection of elements. The roles of the mesh as a data object in the finite element method include:

1. defining which nodes belong to which elements;
2. associating shape functions with elements;
3. associating faces with elements.

From the mesh information it is further possible to calculate adjacency lists for nodes and elements. A mesh defines a finite dimensional function space with respect to which fields may be defined.

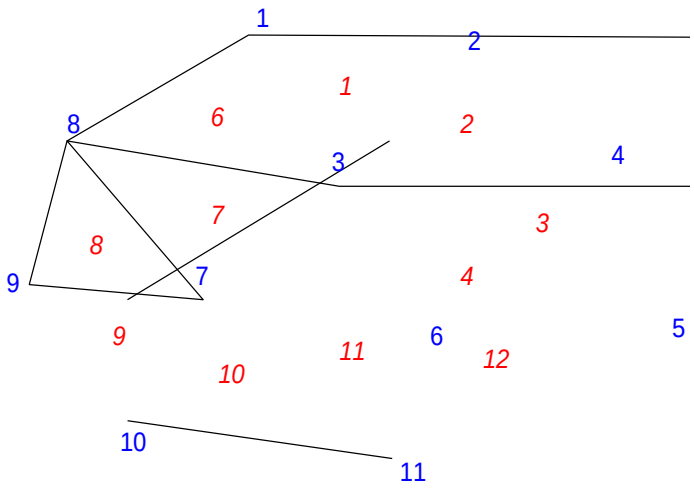
Mathematically, a finite element function space is defined by a tesselation of the domain into elements $\Omega = \{E_\alpha | \quad = 1 : : : N_E\}$ and a set of basis functions defined over Ω, $\Phi = \{ \ _\alpha | \quad = 1 : : : N_{\text{dof}}\}$. The role of the mesh_type is to provide a data structure equivalent to the pair $[\Omega; \Phi]$.

8.1 Node and element numbering

The association of nodes with elements is achieved by matching together the gobal node and element numbers. To understand the interaction between these two numberings it is useful to have an example. Figure 8.1 shows a mesh composed of linear triangles constrained to be continuous at element boundaries. For example, in this mesh, element 5 is composed of nodes 3, 6 and 7 and has, in common with all of the elements in this mesh, a basis composed of piecewise linear functions.

8.2 mesh_type

The derived data type mesh_type holds the information pertaining to a single mesh. In general, a simulation will have as many distinct mesh objects as distinct finite element spaces. For example, if a simulation has piecewise linear pressure and piecewise quadratic velocity, then there will need to be distinct mesh objects for the linear and quadratic spaces. Similarly, mesh types may differ in whether the function space they represent is continuous or permits discontinuities on element edges.


1
2
1
8
6
2
3
4
7
3
8
9
7
4
9
6
5
11
12
10
10
11

Chapter 9

Fields

A field is a data object which associates a value with each point in the domain. That is to say, a field is a function defined at every point in the domain. In principle the function may have values drawn from any space, however for computational mechanics purposes it is usually sufficient to consider fields whose values are scalars (ie real numbers), vectors and square rank 2 tensors (matrices). For example in fluid mechanics, temperature and pressure are scalar fields, velocity is a vector field and diffusivity is a tensor field.

In the finite element method, functions are restricted to a finite dimensional space spanned by the set of basis functions, $\quad_i$ such that a scalar field $F(\mathsf{x})$ may be written:

$$F(\mathsf{x}) = \sum_{\alpha=1}^{N_{\mathsf{dof}}} f_\alpha \quad_\alpha \tag{9.1}$$

In chapter 8, we introduced the mesh_type derived type to encapsulate the basis function and element pair $[\Omega; \Phi]$. If we write $\underline{f} \equiv \{f_\alpha | \quad = 1 ::: N_{\mathsf{dof}}\}$ then we can see fields as building on meshes thus:

$$F = \underline{f}; [\Omega; \Phi] \tag{9.2}$$

9.1 Vector and tensor fields

For vector and tensor fields, the situation is complicated by the the fact that there are multiple components. We assume that a vector field may be written as:

$$\mathsf{F}(\mathsf{x}) = \sum_{\alpha=1}^{N_{\mathsf{dof}}} \sum_{i=1}^{N_{\mathsf{dim}}} f_{\alpha,i} \quad_{\alpha,i} \tag{9.3}$$

Where $\quad_{\alpha,i}$ is a basis function for the space which is the Cartesian product of Φ. For example, if $N_{\mathsf{dim}} = 3$ then:

$$\quad_{\alpha,1} = [\quad_\alpha; 0; 0] \tag{9.4}$$
$$\quad_{\alpha,2} = [0; \quad_\alpha; 0] \tag{9.5}$$
$$\quad_{\alpha,3} = [0; 0; \quad_\alpha] \tag{9.6}$$

It is apparent that the basis for the vector function space can be trivially written in terms of that for the scalar space. It is therefore not necessary to have a separate mesh_type for the vector case. If we write $\underline{\mathsf{f}} \equiv \{f_{\alpha,i} | \quad = 1 ::: N_{\mathsf{dof}}; i = 1; ::: N_{\mathsf{dim}}\}$ then we have:

$$\mathsf{F} = [\underline{\mathsf{f}}; [\Omega; \Phi]] \tag{9.7}$$

A similar argument applies to tensor fields, in which case:

$$\overline{\overline{F}}(\mathbf{x}) = \sum_{\alpha=1}^{N_{\text{dof}}} \sum_{i=1}^{N_{\text{dim}}} \sum_{j=1}^{N_{\text{dim}}} f_{\alpha,i,j} = _{\alpha,i,j} \tag{9.8}$$

where, as an example, if $N_{\text{dim}} = 2$ then:

$$= _{\alpha,1,2} = \begin{matrix} 2 & & 3 \\ 0 & \alpha & 0 \\ 4 0 & 0 & 0 5 \\ 0 & 0 & 0 \end{matrix} \tag{9.9}$$

In this case we write $\underline{\overline{\overline{f}}} \equiv \{f_{\alpha,i,j} | \quad = 1 ::: N_{\text{dof}}; i = 1; ::: N_{\text{dim}}; j = 1; ::: N_{\text{dim}}\}$ and have:

$$\overline{\overline{F}} = \begin{matrix} \text{h} & & \text{i} \\ \underline{\overline{\overline{f}}}; [\Omega; \Phi] \end{matrix} \tag{9.10}$$

This representation of fields (9.2,9.7,9.10) means that scalar, vector and tensor fields can all be seen as the associating basis function coefficients with the same mesh object $[\Omega; \Phi]$. A drawback of this approach is that it does not naturally support vector basis functions, such as the Raviert-Thomas element, which are not a Cartesian product of a scalar basis.

9.2 Field data types

There are separate data types for scalar, vector and tensor fields. In each case the field has a component %mesh which is the mesh_type upon which it builds and contain the list of basis function coefficients ($\underline{f}$, $\underline{\text{f}}$ and $\underline{\overline{\overline{f}}}$ for scalar, vector and tensor fields, respectively). Field types also include other components containing information such as boundary conditions. The three separate field data types are designed to be as identical as possible in interface and to the greatest extent possible, all routines which take fields as arguments are overloaded to take any type of field. For this reason, the documentation for these fields includes the metatype *anyfield* which is defined:

```
anyfield := scalar_field | vector_field | tensor_field
```

For most purposes, Fields are opaque data types which should be accessed via method routines. However there are some public components which are accessible to user routines:

```
type scalar_field
  !! The name of this field
  character(len=FIELD_NAME_LEN) :: name=""
  !! The Spud option path associated with this field.
  character(len=OPTION_PATH_LEN) :: option_path=""
  !! The mesh on which this field is built.
  type(mesh_type) :: mesh
end type scalar_field

type vector_field
  !! The name of this field
  character(len=FIELD_NAME_LEN) :: name=""
  !! The Spud option path associated with this field.
  character(len=OPTION_PATH_LEN) :: option_path=""
  !! The mesh on which this field is built.
  type(mesh_type) :: mesh
  !! The data dimension of this field
  integer :: dim
end type vector_field
```

```
type tensor_field
  !! The name of this field
  character(len=FIELD_NAME_LEN) :: name=""
  !! The Spud option path associated with this field.
  character(len=OPTION_PATH_LEN) :: option_path=""
  !! The mesh on which this field is built.
  type(mesh_type) :: mesh
  !! The data dimension of this field
  integer :: dim
end type tensor_field
```

9.3 Constant fields

A frequently occuring special case of a field is one which is known to be constant in space. Treating these fields specially can enable large memory savings since it is only necessary to store the value of the field at a single point rather than at every degree of freedom in the mesh. Constant fields are created by setting the optional `field_type` argument of the **`allocate`** call to `FIELD_TYPE_CONSTANT`.

Attempting to set the value of individual nodes of a constant field will result in an error. Instead, the whole field version of set should be used.

9.4 The coordinate field

The mesh_type object makes no mention of the geometric location of the nodes in the mesh. In fact, position is just another vector field: every point in the domain has a coordinate vector associated with it. In particular, the basis functions used for the coordinate field need not be linear: it is perfectly possible to have curving elements over which position is not a linear function of the local coordinates. Accordingly, there is no special data structure for position but rather a vector field, conventionally named “Coordinate” which records the position data.

9.5 Topological and data dimension

There is an important distinction between the topological dimension of the mesh and the data dimension of a vector or tensor field which is easy to confuse and which may lead to subtle bugs in software. The topological dimension of a mesh, as given by the %dim component of the mesh_type or by the mesh_dim function refers to the topological dimension of the elements of the mesh. For example, a mesh composed of triangular elements has a topological dimension of 2. On the other hand, the data dimension of a field, given by the %dim component of the field data type, refers to the extent of the vectors or tensors associated with each basis function. As a concrete example, we may choose to solve an equation on a two dimensional slice through a flow field which we imagine to be constant in the third dimension. In this case, the topological dimension of the mesh would be 2 but we might require all 3 components of the Velocity field in order to calculate the Coriolis term in the plane in which we are actually solving the equations.

9.6 Changing coordinates and bases

A number of routines have been implemented in `femtools/Coordinates.F90` that can be used for coordinate transformations and vector / tensor change of basis. The transformations available allow expression of tensor fields in three coordinate systems:

1. *Cartesian coordinates:* The coordinate-set used in Fluidity. In an ocean modelling context the origin of the axes is placed at the centre of the planet, the *x*–axis goes through the 0 East, 0 North point on the surface, the *y*–axis goes through the 90 East, 0 North point on the surface and the *z*–axis runs though the planet's poles (spherical planet assumed).

2. *Spherical-polar coordinates:* The position vector is specified in terms of $[r \quad]^T$, where *r* is the distance from the axes origin, is the polar angle and is the azimuthal angle, also shown in the left panel of figure 9.1. The vector basis is $[\hat{e}_r\ \hat{e}_\theta\ \hat{e}_\phi]^T$, also shown in figure 9.1.

3. *longitude–latitude coordinates:* The position vector is specified in terms of $[\quad r]^T$, where is the longitude is the latitude and *r* is the distance from the axes origin, as shown in the right panel of figure 9.1. The vector basis is $[\hat{e}_\lambda\ \hat{e}_\delta\ \hat{e}_r]^T$, also shown in figure 9.1.

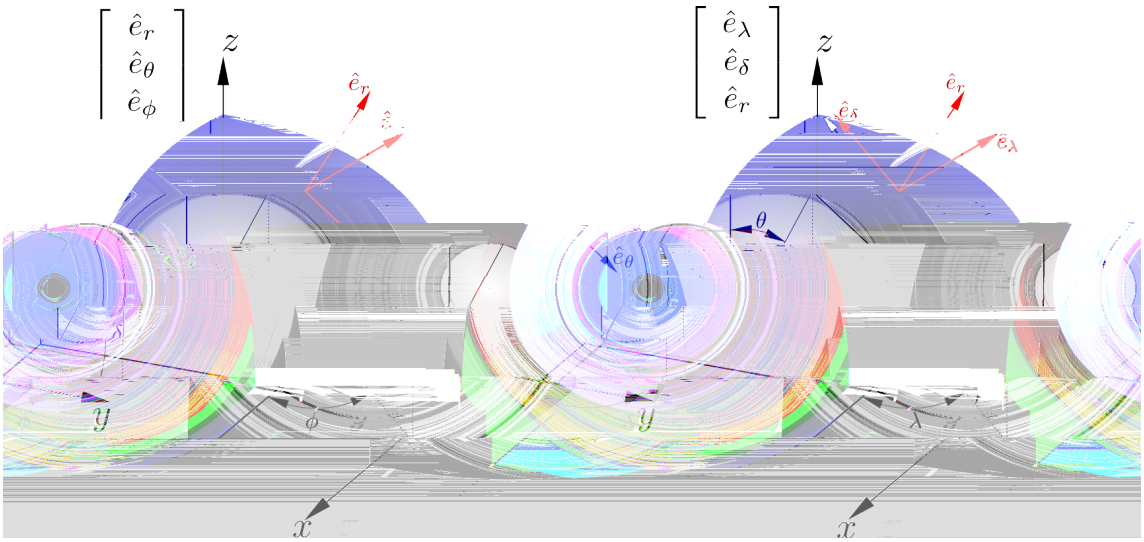


Figure 9.1: Polar coordinates. Left: Spherical-polar coordinates where is the azimuthal angle and is the polar angle. Right: Longitude-latitude coordinates where denotes longitude and denotes latitude.

The transformation routines also feature C-interoperable routines so that the tansformations can also be called from C / C++ code. The C function prototypes are located in **`include`**`/coordinates.h`.

The following lists the availabe transformations:

- **Spherical-polar to Cartesian**
  Fortran routine: `spherical_polar_2_cartesian`
  Fortran routine unit-test: `femtools/tests/test_spherical_polar_2_cartesian.F90`
  C-inter-operable wrapper: `spherical_polar_2_cartesian_c`
  C-inter-operable wrapper unit-test: `femtools/tests/test_spherical_polar_2_cartesian_c.F9`
  Interface: describe routine interface, state that angles are expected in radians
  Short description.

- **Cartesian to Spherical-polar**
  Fortran routine: `cartesian_2_spherical_polar`
  Fortran routine unit-test: `femtools/tests/test_cartesian_2_spherical_polar.F90`
  C-inter-operable wrapper: `cartesian_2_spherical_polar_c`
  C-inter-operable wrapper unit-test: `femtools/tests/test_cartesian_2_spherical_polar_c.F9`
  Interface: describe routine interface, state that angles are returned in radians
  Short description.

Write about the python version of the transformation routines. Module available as `python/GFD_basisChange`
Example of use in `tests/`**`data`**`/on_sphere_rotations/construct_fields.py`, where the dataset used in the unit-tests is constructed.

Chapter 10

State dictionaries

The state of a system of differential equations is determined by a number of fields together. State objects provide a mechanism for grouping together arbitrary collections of fields, meshes, matrices and sparsities which are in some way related. For many applications, there will be a single state object which encompasses all of the data in that simulation, however more complex simulations may require more. For example, a multiphase flow simulation might have one state object for each fluid phase. The velocity, pressure, density and other fields belonging to each phase would be listed in that phase.

10.1 Inserting and extracting objects in states

10.2 Aliased fields and listing the same field in multiple state dictionaries

Because a state dictionary simply holds a reference to a number of fields, it is perfectly possible to insert the same field in multiple state dictionaries. In the example of multiphase flow, this might be because there is a single pressure field applicable to all of the phases. Storing the same field in the state dictionary for each phase can be a simplyfying step which also allows for general code which may need to be aware of the multiphase nature of the simulation.

However, it is often desirable or necessary to have one state dictionary which is the defined owner of a particular field and which, for example, might be the only state via which the field is updated. For this purpose it is possible to designate a field descriptor as being an *alias* of the primary field. To specify that a field descriptor is an alias, set the `aliased` component of the descriptor to `.true.`. For example:

```
function make_aliased_field(field) result (alias)
  implicit none
  use fields
  type(scalar_field) :: alias
  type(scalar_field), intent(in) :: field

  alias=field
  alias%aliased=.true.

end function
```

This function returns a variable containing the same field as the input, but the field descriptor is a copy and the `aliased` component has been set to `.true.`. The aliased status of a field can be queried using the aliased routine.

Chapter 11

Reference counting

The key data types described here share complex relationships. For example, a single mesh might be used by any number of fields in a wide range of routines in a programme. This presents an immediate challenge for data management: how does the programme know when it is safe to deallocate the memory associated with the mesh? Of course that mesh in turn depends on an element type which must not be deallocated for as long as the mesh exists.

The solution to this problem which is adopted here is the reference counting. In this system, each data object records the number of other objects which are currently using it. When a new data object is created, the reference count associated with each object on which it depends is increased, and when an object is destroyed, the reference count of each on which it depends is decreased. When the reference count of an object falls to zero, the object itself is destroyed and the memory associated with it freed.

As a brief example, consider just the case of `element_type` objects and the `quadrature_type` objects on which they depend. Suppose we have the subroutine shown in figure 11.1 which produces two elements. In order to produce the elements, we first generate a quadadrature object by calling `make_quadrature`. This produces a new quadrature object and sets it reference count to 1. Next we call `make_element_shape` to produce the first element object, `element1`. Since `element1` uses quadrature, it increments the reference count of the quadrature object. `quadrature` now has a reference count of 2 and `element1` has a reference count of 1. We next call `make_element_shape` again to produce `element2`. At this stage `quadrature` has a reference count of 3: the original reference created with the object itself plus one each created by `element1` and `element2`. Finally, since we have finished using `quadrature` to generate new elements, we call **`deallocate`**`(quadrature)`. This destroys the original reference which we created and reduces the reference count of `quadrature` to 2. However, since the reference count of `quadrature` is still positive, the object itself is not destroyed and the memory associated with it is not freed. This is, in fact, the desired behaviour as the quadrature object must still be available for use by the elements we created.

Eventually, when the first element is no longer required, the program will call:

```
call deallocate(element1)
```

Assuming that there are no other objects which are using `element1`, this will reduce the reference count of `element1` to 0 and it will be destroyed and its memory freed. This will in turn cause the reference count of the quadrature object to be reduced to 1. At the point at which `element2` is deallocated, the reference count of the quadrature will drop to 0 and it will finally be actually destroyed and its memory freed.

```
subroutine generate_elements(element1, element2)
  use quadrature
  use shape_functions
  type(element_type), intent(out) :: element1, element2
  type(quadrature_type) :: quadrature

  quadrature=make_quadrature(...)
  ! The reference count of quadrature is 1.

  element1=make_element_shape(... degree=1, quad=quadrature ...)
  ! The reference count of quadrature is 2.

  element2=make_element_shape(... degree=2, quad=quadrature ...)
  ! The reference count of quadrature is 3.

  call deallocate(quadrature)
  ! The reference count of quadrature is 2.

end subroutine generate_elements
```

Figure 11.1: The reference count of a quadrature object as it is created and used to form elements. Finally, one of the references to quadrature is discarded by calling deallocate.

11.1 Creating and destroying references

11.1.1 Allocate

When the **allocate** subroutine is called for an object, this allocates a new data space for that object and creates a corresponding reference count for that object, which is set to 1.

In addition, the reference count of any object which the new object *directly* depends on is increased by 1. For example, when a field is allocated, the reference count of the mesh_typeon which that field is based on is increased by one. In other words the field *holds a reference* to the mesh.

The reference counts of objects which the new field only indirectly depends on are not affected by a call to allocate. For example, a mesh_type holds a reference to the element_type on which the mesh is based. When a field is allocated using that mesh_type object, the reference count of the corresponding element_type is unchanged.

It is important to note that these remarks apply only to the **allocate** subroutine called on a reference-counted object. The Fortran **allocate** statement has no effect on reference counts.

11.1.2 Deallocate

When **deallocate** is called for an object, the reference count of that object is decreased by 1. If, after this operation, the reference count is still positive, **deallocate** returns immediately and the object remains allocated. On the other hand, if the reference count has fallen to 0 then the data space associated with the object is deallocated thereby destroying the object.

In other words, routines call **deallocate** in order to release a reference which they hold. The object itself is only destroyed once there are no more references to it held.

For objects which hold references to other objects, for example a field holds a reference to the mesh_type on which it is based, the reference to the other object is released when the first object is destroyed. If this causes the reference count on the other object to fall to 0, then this will result in the deallocation

of the other object too. Naturally this in turn may cause further references to be released and any unused objects to be destroyed.

Calling **`deallocate`** on a state dictionary will cause the dictionary to delete the reference it holds to each of the objects in the dictionary.

As with **`allocate`** only the **`deallocate`** routines associated with reference-counted objects will destroy references. The Fortran **`deallocate`** object has no effect on reference counts.

11.1.3 Inserting into state

A state dictionary holds a reference to each object it contains. Inserting an object into a state therefore increments the reference count of that object. A direct result of this is that if an object is created and immediately inserted into a state, the reference count of that object will be 2. If it is desired that the object should be controlled by that state dictionary and destroyed when that state dictionary is deallocated, it will be necessary to call **`deallocate`** on the object after it is inserted into the state in order to reduce its reference count to 1.

11.1.4 Extracting from states

The usual model of use of objects in a state dictionary is that they are accessed but remain under the control of the dictionary. For example, a number of the fields in a state may be extracted and updated each timestep but the fields remain in the state at the end of the timestep.

To make this morel of use as natural as possible, extracting an object from a state dictionary does not increase the reference count of that object. Rather, it could be thought that the reference is borrowed from the state but still belongs to the state. This means that a routine which extracts an object from a state should not deallocate that object when it is finished with it. To do so would invalidly destroy the reference held by the state.

A consequence of this model is that if the state dictionary is itself deallocated before the extracting routine has finished with the object, the object may be destroyed and the pointer made invalid. Should it be necessary to preserve a reference to an object during the deallocation of the state object or after control passes from the routine extracting the object, this can be achieved by calling `incref` on the object. In this case, it will be necesary to deallocate the object in order to destroy this reference and avoid a memory leak.

11.1.5 Incref

In some circumstances, it may be necessary to manually create a new reference to an object. The usual example would be where an object is extracted from a state dictionary and the extracting routine wishes to ensure that that object is not destroyed when control passes to another routine. The subroutine `incref` is provided for this purpose. Calling `incref` on an object increases the reference count of that object by 1. It is essential that code which manually increments references ensures that those references are destroyed by an appropriate **`deallocate`** call. Failure to do so is likely to result in a memory leak.

11.2 Creating new reference counted data types

11.3 Memory accounting diagnostics

Femtools maintains some accounting information concerning the amount of memory allocated to reference counted objects. This is useful for diagnosing which data structures occupy the most memory,

Data structure	Accounting bin	Information recorded
mesh_type	MeshMemory	element-node, face-node and face-element lists. Boundary ID and surface node lists.
	MatrixSparsityMemory	element-element map.
	MatrixMemory	element-face map.
scalar_field	ScalarFieldMemory	nodal coefficient values.
vector_field	VectorFieldMemory	nodal coefficient values.
tensor_field	TensorFieldMemory	nodal coefficient values.
csr_sparsity	MatrixSparsityMemory	matrix nonzero locations.
csr_matrix block_csr_matrix	MatrixMemory	matrix nonzero values.

Table 11.1: Forms of data about which memory accounting information is stored. Note that since the mesh_type data structure contains some information which is stored as sparse matrices, some mesh data is recorded as sparse matrix data.

although it is important to note that the memory recorded here will not account for the total amount of memory allocated by a program using the femtools library: only the large arrays associated with the data objects listed here are stored.

Memory accounting data is recorded under a number of headings: MeshMemory, ScalarFieldMemory, VectorFieldMemory, TensorFieldMemory, MatrixSparsityMemory and MatrixMemory. There is also a TotalMemory heading which records all of the memory for which accounting data is available. Table 11.1 shows for each data object, which memory is logged and under which category. At this stage, not all memory associated with field boundary conditions and parallel halo data is logged.

For each memory heading, the current memory usage as well as the maximum and minimum ever used are stored. It is also possible to reset the maximum and minimum values, in which case the maximum and minumum values constitute the maximum and minimum values since the last reset. This enables timestepping simulations to log the minimum and maximum memory usage in each timestep.

11.3.1 Memory statistics in the .stat file

If the `diagnostic_variables` module is employed to produce a .stat file then the maximum and minimum values will be reset after the .stat file entry for each timestep is written. For each memory heading, the .stat file will contain fields recording the minimum, maximum and current memory usage.

Part III

Procedure reference

Chapter 12

General principles for procedures

Much of Femtools is designed in an essentially object-oriented manner and many of the procedures in the library should be understood as methods belonging to the data objects on which they act. In keeping with widespread practice, the first argument of a method is the data object on which it acts. This should be followed by any arguments which control which part of the object is affected. The next arguments are the input data, if any and finally any status variable.

12.1 Field, mesh and matrix interfaces

Fields, meshes and matrices have a number of common features which makes many of their methods amenable to consistent interfaces. To the extent to which it is applicable, methods for these data types should have interfaces of the following form:

```
procedure_name(object[, dims][, items][, values][, stat])
```

The items in square brackets ([]) are optional. All of the arguments other than the object itself may not appear in all interfaces however those which are present should appear in this order. For example, the version of the addto routine which sets a single component of a tensor field at one node has the argument sequence:

```
addto(field, dim1, dim2, node_number, val)
```

12.2 Status arguments

Routines which might not succeed typically have an optional integer output argument named **`stat`**. If **`stat`** is present and the procedure returns successfully, it is set to zero while if an error occurs it is set to a procedure-specific non-zero value. If **`stat`** is not present and an error occurs then execution will terminate with an error.

Chapter 13

Field and mesh methods

13.1 Global field and mesh enquiry routines

These routines return a single value for an entire mesh or field.

13.1.1 mesh_dim

```
pure function mesh_dim(mesh)
  integer :: mesh_dim
  type(mesh_type), intent(in) :: mesh

pure function mesh_dim(field)
  integer :: mesh_dim
  type(anyfield), intent(in) :: field
```

Return the topological dimension of the mesh or the mesh associated with the field provided. Note that in the case of a vector or tensor field, this may be different from the topological dimension of the mesh may be different from the dimension of the data in the field.

13.1.2 mesh_periodic

```
pure function mesh_periodic(mesh)
  logical :: mesh_periodic
  type(mesh_type), intent(in) :: mesh

pure function mesh_periodic(field)
  logical :: mesh_periodic
  type(anyfield), intent(in) :: field
```

Returns true if the mesh or the mesh associated with the field provided is periodic in any dimension.

13.1.3 node_count

```
pure function node_count(mesh)
  integer :: node_count
  type(mesh_type), intent(in) :: mesh

pure function node_count(field)
  logical :: node_count
  type(anyfield), intent(in) :: field
```

Return the number of nodes in the mesh or field provided.

13.1.4 element_count

```
pure function element_count(mesh)
  integer :: element_count
  type(mesh_type), intent(in) :: mesh

pure function element_count(field)
  logical :: element_count
  type(anyfield), intent(in) :: field
```

Return the number of elements in the mesh or field provided.

13.1.5 surface_element_count

```
pure function surface_element_count(mesh)
  integer :: surface_element_count
  type(mesh_type), intent(in) :: mesh

pure function surface_element_count(field)
  logical :: surface_element_count
  type(anyfield), intent(in) :: field
```

Return the number of surface elements in the mesh or field provided.

13.1.6 face_count

```
pure function face_count(mesh)
  integer :: face_count
  type(mesh_type), intent(in) :: mesh

pure function face_count(field)
  logical :: face_count
  type(anyfield), intent(in) :: field
```

Return the number of faces in the mesh or field provided. Note that this includes interior faces, not just surface faces (for which see surface_element_count above). Note also that on internal faces, there is a separate face for each of the two elements adjacent to the face.

13.1.7 aliased

Module: state_module

```
pure function aliased(field)
  type(anyfield), intent(in) :: field
  logical :: aliased
```

Return `.true.` if `field` is an alias. See section 10.2 for information on aliased fields.

13.2 Element enquiry routines

These routines return information about the properties of a single element in a field or mesh.

13.2.1 ele_loc

```
pure function ele_loc(mesh, ele_number)
  integer :: ele_loc
  type(mesh_type), intent(in) :: mesh
  integer, intent(in) :: ele_number

pure function ele_loc(field, ele_number)
  integer :: ele_loc
  type(anyfield), intent(in) :: field
  integer, intent(in) :: ele_number
```

Return the number of nodes in element `ele_number` of the field or mesh provided. Note that in the current implementation, this is always a constant value for all elements. This is expected to change in the future.

13.2.2 ele_and_faces_loc

```
pure function ele_and_faces_loc(mesh, ele_number)
  integer :: ele_and_faces_loc
  type(mesh_type), intent(in) :: mesh
  integer, intent(in) :: ele_number

pure function ele_and_faces_loc(field, ele_number)
  integer :: ele_and_faces_loc
  type(anyfield), intent(in) :: field
  integer, intent(in) :: ele_number
```

Return the number of nodes in element `ele_number` of the field or mesh provided plus the number of nodes in each of the adjacent faces. This is primarily useful for calculating the size of temporary matrices used in the local assembly of discontinuous Galerkin operators which include face integrals. Figure 13.1 illustrates the calculation performed by this function for a linear discontinuous triangular element.

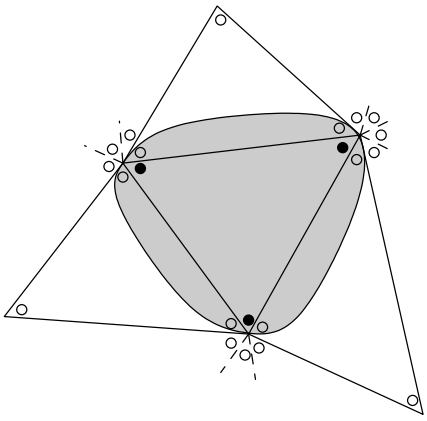


Figure 13.1: Illustration of the area included in the node count for ele_and_faces_loc for a linear triangular element. In this case the element has 3 nodes and each of the three neighbouring faces has 2 nodes so the return value is 9.

Note that in the current implementation, this is always a constant value for all elements. This is expected to change in the future.

13.2.3 ele_vertices

```
pure function ele_vertices(mesh, ele_number)
  integer :: ele_vertices
```

```
  type(mesh_type), intent(in) :: mesh
  integer, intent(in) :: ele_number

pure function ele_vertices(field, ele_number)
  integer :: ele_vertices
  type(anyfield), intent(in) :: field
  integer, intent(in) :: ele_number
```

Return the number of vertices in element `ele_number` of the field or mesh provided. Vertices are a purely geometric concept so a triangle always has three vertices regardless of the degree of polynomials used as basis functions. In contrast, the number of nodes in an element is a function of the degree of the polynomial basis functions. For example, a quadratic triangle has six nodes.

Note that in the current implementation, this is always a constant value for all elements. This is expected to change in the future.

13.2.4 ele_ngi

```
pure function ele_ngi(mesh, ele_number)
  integer :: ele_ngi
  type(mesh_type), intent(in) :: mesh
  integer, intent(in) :: ele_number

pure function ele_ngi(field, ele_number)
  integer :: ele_ngi
  type(anyfield), intent(in) :: field
  integer, intent(in) :: ele_number
```

Return the number of quadrature points in element `ele_number` of the field or mesh provided. Note that in the current implementation, this is always a constant value for all elements. This is expected to change in the future.

13.2.5 ele_nodes

```
function ele_nodes(mesh, ele_number)
  integer, dimension(:), pointer :: ele_nodes
  type(mesh_type),intent(in) :: mesh
  integer, intent(in) :: ele_number

function ele_nodes(field, ele_number)
  integer, dimension(:), pointer :: ele_nodes
  type(anyfield),intent(in) :: field
  integer, intent(in) :: ele_number
```

Return a pointer to a vector containing the global node numbers of element `ele_number` in the mesh provided or in the mesh associated with the field provided. For example, if `this_mesh` is the mesh shown in figure 8.1, then `ele_nodes(this_mesh, 5)` will return a pointer to a 3-vector containin the node numbers 3, 6 and 7. No guarantees are made about the order in which these node numbers will be arranged.

13.3 Face enquiry routines

These routines return information about the properties of a single element face in a field or mesh.

13.3.1 face_loc

```
pure function face_loc(mesh, face_number)
  integer :: face_loc
  type(mesh_type), intent(in) :: mesh
  integer, intent(in) :: face_number

pure function face_loc(field, face_number)
  integer :: face_loc
  type(anyfield), intent(in) :: field
  integer, intent(in) :: face_number
```

Return the number of nodes in face `face_number` of the field or mesh provided. Note that in the current implementation, this is always a constant value for all faces. This is expected to change in the future.

13.3.2 face_vertices

```
pure function face_vertices(mesh, face_number)
  integer :: face_vertices
  type(mesh_type), intent(in) :: mesh
  integer, intent(in) :: face_number

pure function face_vertices(field, face_number)
  integer :: face_vertices
  type(anyfield), intent(in) :: field
  integer, intent(in) :: face_number
```

Return the number of vertices in face `face_number` of the field or mesh provided. Vertices are a purely geometric concept so a triangle always has three vertices regardless of the degree of polynomials used as basis functions. In contrast, the number of nodes in an face is a function of the degree of the polynomial basis functions. For example, a quadratic triangle has six nodes.

Note that in the current implementation, this is always a constant value for all faces. This is expected to change in the future.

13.3.3 face_ngi

```
pure function face_ngi(mesh, face_number)
  integer :: face_ngi
  type(mesh_type), intent(in) :: mesh
  integer, intent(in) :: face_number

pure function face_ngi(field, face_number)
  integer :: face_ngi
  type(anyfield), intent(in) :: field
  integer, intent(in) :: face_number
```

Return the number of quadrature points in face `face_number` of the field or mesh provided. Note that in the current implementation, this is always a constant value for all faces. This is expected to change in the future.

13.3.4 face_local_nodes

```
function face_local_nodes(mesh, face_number)
  integer, dimension(:), pointer :: face_local_nodes
  type(mesh_type),intent(in) :: mesh
  integer, intent(in) :: face_number
```

```
function face_local_nodes(field, face_number)
  integer, dimension(:), pointer :: face_local_nodes
  type(anyfield),intent(in) :: field
  integer, intent(in) :: face_number
```

Return a pointer to a vector containing the *local* node numbers of face `face_number` in the mesh provided or in the mesh associated with the field provided. Note that these are the local numbers in the element containing face.

13.3.5 face_global_nodes

```
function face_global_nodes(mesh, face_number)
  integer, dimension(:), pointer :: face_global_nodes
  type(mesh_type),intent(in) :: mesh
  integer, intent(in) :: face_number

function face_global_nodes(field, face_number)
  integer, dimension(:), pointer :: face_global_nodes
  type(anyfield),intent(in) :: field
  integer, intent(in) :: face_number
```

Return a pointer to a vector containing the *global* node numbers of face `face_number` in the mesh provided or in the mesh associated with the field provided.

13.4 Data retrieval routines

13.4.1 ele_val

Return the value of a field at all of the nodes in an element. The precise shape of the function value depends on the rank and dimension of the field.

```
function ele_val(field, ele_number)
  type(scalar_field), intent(in) :: field
  integer, intent(in) :: ele_number
  real, dimension(ele_loc(field, ele_number)) :: ele_val
```

The scalar field version of this routine returns a vector with length equal to the number of nodes in this element.

```
function ele_val(field, ele_number)
  type(vector_field),intent(in) :: field
  integer, intent(in) :: ele_number
  real, dimension(field%dim, ele_loc(field, ele_number)) :: ele_val

function ele_val(field, ele_number)
  type(tensor_field),intent(in) :: field
  integer, intent(in) :: ele_number
  real, dimension(field%dim, field%dim, ele_loc(field, ele_number)) :: ele_val
```

Note that for these vector and tensor versions of the routine, the dim dimensions come first, followed by the number of element nodes.

```
function ele_val(field, ele_number, dim)
  type(vector_field), intent(in) :: field
  integer, intent(in) :: ele_number, dim
  real, dimension(ele_loc(field, ele_number)) :: ele_val
```

```
function ele_val(field, ele_number, dim1, dim2)
  type(tensor_field), intent(in) :: field
  integer, intent(in) :: ele_number, dim1, dim2
  real, dimension(ele_loc(field, ele_number)) :: ele_val
```

These versions of the routine return the value of a single component of the vector or tensor field at all of the nodes of the element.

13.4.2 ele_val_at_quad

```
function ele_val_at_quad_scalar(field, ele_number)
  type(scalar_field),intent(in) :: field
  integer, intent(in) :: ele_number
  real, dimension(ele_ngi(field, ele_number)) :: ele_val_at_quad
```

13.4.3 face_val

Return the value of a field at all of the nodes in an face. The precise shape of the function value depends on the rank and dimension of the field.

```
function face_val(field, face_number)
  type(scalar_field), intent(in) :: field
  integer, intent(in) :: face_number
  real, dimension(face_loc(field, face_number)) :: face_val
```

The scalar field version of this routine returns a vector with length equal to the number of nodes in this face.

```
function face_val(field, face_number)
  type(vector_field),intent(in) :: field
  integer, intent(in) :: face_number
  real, dimension(field%dim, face_loc(field, face_number)) :: face_val


function face_val(field, face_number)
  type(tensor_field),intent(in) :: field
  integer, intent(in) :: face_number
  real, dimension(field%dim, field%dim, face_loc(field, face_number)) :: face_val
```

Note that for these vector and tensor versions of the routine, the dim dimensions come first, followed by the number of face nodes.

```
function face_val(field, face_number, dim)
  type(vector_field), intent(in) :: field
  integer, intent(in) :: face_number, dim
  real, dimension(face_loc(field, face_number)) :: face_val


function face_val(field, face_number, dim1, dim2)
  type(tensor_field), intent(in) :: field
  integer, intent(in) :: face_number, dim1, dim2
  real, dimension(face_loc(field, face_number)) :: face_val
```

These versions of the routine return the value of a single component of the vector or tensor field at all of the nodes of the face.

13.4.4 node_val

Return the value of a field at one or more specified nodes. The shape of the value returned is given by the rank and dimension of the field and the number of nodes queried.

```
function node_val(field, node_number)
  type(scalar_field),intent(in) :: field
  integer, intent(in) :: node_number
  real :: node_val

function node_val(field, node_numbers)
  type(scalar_field),intent(in) :: field
  integer, dimension(:), intent(in) :: node_numbers
  real, dimension(size(node_numbers)) :: node_val
```

For a scalar field, there is clearly one real number per node.

```
function node_val(field, node_number)
  type(vector_field),intent(in) :: field
  integer, intent(in) :: node_number
  real, dimension(field%dim) :: node_val

function node_val(field, node_numbers)
  type(vector_field),intent(in) :: field
  integer, dimension(:), intent(in) :: node_numbers
  real, dimension(field%dim, size(node_numbers)) :: node_val

function node_val(field, node_number)
  type(tensor_field),intent(in) :: field
  integer, intent(in) :: node_number
  real, dimension(field%dim, field%dim) :: node_val

function node_val(field, node_numbers)
  type(tensor_field),intent(in) :: field
  integer, dimension(:), intent(in) :: node_numbers
  real, dimension(field%dim, field%dim, size(node_numbers)) :: node_val
```

In the result of the full vector and tensor versions of this routine, the dimension components come first followed by the number of nodes, for the multiple node version.

```
function node_val(field, node_number, dim)
  type(vector_field),intent(in) :: field
  integer, intent(in) :: node_number
  integer, intent(in) :: dim
  real :: node_val

function node_val(field, node_numbers, dim)
  type(vector_field),intent(in) :: field
  integer, dimension(:), intent(in) :: node_numbers
  integer, intent(in) :: dim
  real, dimension(size(node_numbers)) :: node_val

function node_val(field, dim1, dim2, node_number)
  type(tensor_field),intent(in) :: field
  integer, intent(in) :: node_number
  integer, intent(in) :: dim1, dim2
  real :: node_val
```

```
function node_val(field, dim1, dim2, node_numbers)
  type(tensor_field),intent(in) :: field
  integer, dimension(:), intent(in) :: node_numbers
  integer, intent(in) :: dim1, dim2
  real, dimension(size(node_numbers)) :: node_val
```

These versions of the the routine return the value of a single component of the field at one or more nodes.

13.5 Data setting routines

The routines in this section are typically the field versions of interfaces which are also available for structures such as sparse matrices.

13.5.1 addto

Adding to node values

Addto is the generic calling name for a number of subroutines which update their first argument by adding values to it. The first basic form of this routine adds a value to one or more nodes of a field. Its generic forms are:

```
subroutine addto(field, node_number, val)
  type(anyfield), intent(inout) :: field
  integer, intent(in) :: node_number
  real, dimension(valshape), intent(in) :: val

subroutine addto(field, node_numbers, val)
  type(anyfield), intent(inout) :: field
  integer, dimension(:), intent(in) :: node_numbers
  real, dimension(valshape, size(nodenumbers)), intent(in) :: val
```

The shape of the val argument varies according to the field type:

field type	*valshape*
scalar_field	
vector_field	`field%dim`
tensor_field	`field%dim, field%dim`

In the case of a scalar field, the value is clearly a scalar and so *valshape* does not contribute to the rank of `val` at all.

```
subroutine addto(field, dim1, dim2, node_number, val)
  type(vector_field), intent(inout) :: field
  integer, intent(in) :: dim1, dim2
  integer, intent(in) :: node_number
  real, intent(in) :: val

subroutine addto(field, dim1, dim2, node_numbers, val)
  type(vector_field), intent(inout) :: field
  integer, intent(in) :: dim1, dim2
  integer, dimension(:), intent(in) :: node_numbers
  real, dimension(size(nodenumbers)), intent(in) :: val
```

Adding fields to each other

Simple whole field addition operations are only mathematically defined where all of the fields share the same mesh. Where this is not the case, addto will call remap_field on field2 to produce a well-defined operation.

```
subroutine
```

It is also possible to scale any field by the entries of a scalar field on the same mesh:

```
subroutine scale(field, sfield)
  type(anyfield), intent(inout) :: field
  type(scalar_field), intent(in) :: sfield
```

However, care must be taken in using this routine since the product of the nodal values is not in general equal to the product of the discretised fields. Taking the example of two scalar fields F and G, this is because:

$$F(\mathbf{x})G(\mathbf{x}) \equiv \sum_{\alpha=1}^{N_{\mathrm{dof}}} \sum_{\beta=1}^{N_{\mathrm{dof}}} f_\alpha \, {}_\alpha g_\beta \, {}_\beta \neq \sum_{\alpha=1}^{N_{\mathrm{dof}}} f_\alpha g_\alpha \, {}_\alpha \tag{13.3}$$

Using the scale subroutine to multiply two fields amounts to the rightmost expression. Subject to the same caveat, it is also possible to scale a vector field by another vector field:

```
subroutine scale(field, vfield)
  type(anyfield), intent(inout) :: field
  type(vector_field), intent(in) :: vfield
```

13.5.3 set

As with addto, set is a family of routines which set the value of part or all of their first argument.

Setting node values

The first version of this routine sets the value of field at one or more nodes:

```
subroutine set(field, node_number, val)
  type(anyfield), intent(inout) :: field
  integer, intent(in) :: node_number
  real, dimension(valshape), intent(in) :: val

subroutine set(field, node_number, val)
  type(anyfield), intent(inout) :: field
  integer, dimension(:), intent(in) :: node_number
  real, dimension(valshape, size(nodenumbers)), intent(in) :: val
```

The shape of the val argument varies according to the field type:

field type	*valshape*
scalar_field	
vector_field	`field%dim`
tensor_field	`field%dim, field%dim`

In the case of a scalar field, the value is clearly a scalar and so *valshape* does not contribute to the rank of `val` at all.

Setting the whole field to a constant value

It is sometimes useful to set an entire field to a constant value. This is a chieved with:

```
subroutine set(field, val)
  type(anyfield), intent(inout) :: field
  real, dimension(valshape), intent(in) :: val
```

Whe all. Setshape val

Setting a field to the value of another field

Setting one field to the value of another field is currently only supported where the two fields are on the same mesh.

```
subroutine set(out_field, in_field )
  type(anyfield), intent(inout) :: out_field
  type(anyfield), intent(in) :: in_field
```

Clearly for this to be defined, in_field and out_field must have the same rank and dimension. Note that this does not allocate out_field, so it must already be allocated.

A related case which occurs frequently when implementing schemes for time-varying PDEs is that of assigning a linear combination of two other fields to a field. That is:

$$F(\mathbf{x}) = \quad F_{\text{new}}(\mathbf{x}) + (1 - \quad)F_{\text{old}}(\mathbf{x}) \tag{13.4}$$

This is achieved with the following form of set:

```
subroutine set(out_field, in_field_new, in_field_old, theta)
  type(anyfield), intent(inout) :: out_field
  type(anyfield), intent(in) :: in_field_new, in_field_old
  real, intent(in) :: theta
```

13.5.4 zero

```
subroutine zero(field)
  type(anyfield), intent(inout) :: field
```

This routine simply sets every entry in field to zero. There are also forms of this subroutine which zero single vector and tensor field components:

```
subroutine zero(field, dim)
  type(vector_field), intent(inout) :: field
  integer, intent(in) :: dim

subroutine zero(field, dim1, dim2)
  type(tensor_field), intent(inout) :: field
  integer, intent(in) :: dim1, dim2
```

Chapter 14

State dictionary methods

State objects provide a unified way of grouping a diverse range of femtools objects. The list of objects which can be contained in a state dictionary are listed in table 14.1

object	object_type
field	*anyfield*
mesh	mesh_type
halo	halo_type
matrix	csr_matrix
	block_csr_matrix
	petsc_csr_matrix

Table 14.1: List of object argument names and corresponding object types supported by the state_type and its various access methods.

14.1 Inserting objects in states

14.1.1 insert

Module: state_module

```
subroutine insert(state, object, name)
  type(state_type), intent(inout) :: state
  type(object_type), intent(in) :: object
  character(len=*), intent(in) :: name

subroutine insert(state, object, name)
  type(state_type), dimension(:), intent(inout) :: state
  type(object_type), intent(in) :: object
  character(len=*), intent(in) :: name
```

These routines insert an object into a state dictionary. The first form inserts an object into a single state dictionary while the second form inserts the object into the first state and an alias of the object into each subsequent state. The *object* argument can be any of those listed in table 14.1.

14.2 Extracting objects from states

Objects in states may be accessed by name or by index. The index of an object in a state is arbitrary and may change as objects are added to and deleted from the state so this latter option is mostly of use for iterating over all of the objects of a particular type in the state.

In each case, the routine returns a pointer to the object concerned. If the object is only to be read or if changes are only to be made to the data space then the pointer may be assigned to a non-pointer variable. On the other hand, if changes are made to the object descriptor then these should be made via the pointer in order to ensure that it is the descriptor in the state dictionary which is updated rather than a local copy.

14.2.1 Extracting objects by name

```
function extract_object_type(state, name, stat[, allocated])
  type{object_type}, pointer :: extract_object_type
  type(state_type), intent(in) :: state
  character(len=*), intent(in) :: name
  integer, intent(out), optional :: state
  logical, intent(out), optional :: allocated

function extract_object_type(state, name, stat[, allocated])
  type{object_type}, pointer :: extract_object_type
  type(state_type), dimension(:), intent(in) :: state
  character(len=*), intent(in) :: name
  integer, intent(out), optional :: state
  logical, intent(out), optional :: allocated
```

In each case, the *object_type* can be any of the types listed in table 14.1.

The first form of the routine takes a single state dictionary as an argument. If there is an object of the corresponding type with name matching **name** then a pointer to that object is returned. If **stat** is present then it is set to 0.

If there is no matching object with the correct name in the dictionary then if **stat** is present it is set to 1 and the function returns a pointer to a dummy object whose components should not be accessed. If **stat** is not present then execution will halt with an error.

The second form of the routine takes a vector of states and will return the first object it finds in any of the states matching the name provided. The return values of the **stat** argument and the handling of the case where there is no match is as before.

Reference counts of objects extracted from states

With the exception of the extraction of a scalar component from a vector field discussed below, the extraction of a pointer to an object from a state does not create a new reference to that object. Consequently the object returned should not be deallocated as this would result in the premature destruction of that object. See section 11.1.3 for a full discussion of the interaction between reference counts and state objects.

Extracting scalar components of vector fields

It is also possible to extract a single component of a vector field stored in a state dictionary. The syntax for this is to call `extract_scalar_field` and provide a logical variable in as the `allocated` argument. The name of the field should be specified as *name*%*n* where *name* is the name under which the vector field is stored in the state and *n* is the number of the component to be extracted. If the name and component number match then a new scalar field descriptor will be allocated and the corresponding data spaces will be associated with the correct component of the vector.

This form of `extract_scalar_field` extract scalar field can be an exception to the rule that extracting an object reference from a state dictionary does not create a new reference. If the `allocated` argument returns `.true.` then a new reference has been created and the resulting field will need to be deallocated in order for that reference to be destroyed.

It is possible to pass a vector of state dictionaries to `extract_scalar_field` in which case the scalar field returned will be extracted from the first matching vector field.

14.2.2 Extracting objects by index

It is frequently convenient to perform some action for each object of a particular type in a state. To facilitate this, the following alternative form of the extraction routines is provided:

```
function extract_object_type(state, index)
  type{object_type}, pointer :: extract_object_type
  type(state_type), intent(in) :: state
  integer, intent(in) :: index
```

This routine is available for any of the types in table 14.1. In each case `index` must not exceed the number of objects of that type in `state`. See object counts.

14.3 Auxiliary state routines

14.3.1 deallocate

```
subroutine deallocate_state(state)
  type(state_type), intent(inout) :: state
```

This routine removes all of the objects from `state` and calls **`deallocate`** on each of them to release the reference held by the state object.

14.3.2 remove object

```
subroutine remove_object_type(state, name, stat)
  type(state_type), intent(inout) :: state
  character(len=*), intent(in) :: name
  integer, optional, intent(out) :: stat
```

If there is an object of the specified type in `state` stored under the name **`name`**, then that object is removed fron `state` and **`deallocate`** is called for the object to release the reference which `state` is holding to it.

If **`stat`** is present then it will be set to 0 for success and nonzero for failure, which occurs if there was no matching object to remove. If the routine fails to remove an object and **`stat`** has not been specified, execution will cease with an error.

14.3.3 object counts

```
pure function object_type_count(state)
  integer :: object_type_count
  type(state_type), intent(in) :: state
```

This routine returns the number of objects of the given type stored in the `state`. *object_type* can be any of the types listed in table 14.1.

Chapter 15

Element methods

15.1 Quadrature methods

15.1.1 make_quadrature

Module: quadrature

```
function make_quadrature(vertices, dim, degree, ngi, family, stat)
  type(quadrature_type) :: make_quadrature
  integer, intent(in) :: vertices, dim
  integer, intent(in), optional :: degree, ngi
  integer, intent(in), optional :: family
  integer, intent(out), optional :: stat
```

This routine creates a new quadrature_type object. The element geometry for which the quadrature is produced is specified by the `vertices` and `dim` arguments. Valid combinations of these are:

vertices	dim	Element geometry
1	1	point
2	1	interval
3	2	triangle
4	2	quad
4	3	tet
8	3	hex

The accuracy of the quadrature rule is given by the `degree` argument which specifies the degree of polynomial which will be integrated exactly by this quadrature rule. If the specified quadrature degree is unavailable but a higher degree rule is supported then this will be substituted. Specifying a higher degree than is supported is an error. An alternative method of specifying the quadrature

If present, **stat** is set to 0 to indicate successful completion. Errors are indicated by the following values:

stat value	meaning
QUADRATURE_VERTEX_ERROR	Unsupported vertex count.
QUADRATURE_DEGREE_ERROR	Quadrature degree requested is not available.
QUADRATURE_DIMENSION_ERROR	Elements with this number of dimensions are not available.
QUADRATURE_NGI_ERROR	Unsupported number of quadrature points.
QUADRATURE_ARGUMENT_ERROR	Not enough arguments specified.

If the **stat** argument is not present then any error will cause execution to cease with an error message.

15.1.2 deallocate

Module: quadrature

```
subroutine deallocate(quad,stat)
  type(quadrature_type), intent(inout) :: quad
  integer, intent(out), optional :: stat
```

This routine releases one reference to quad. For a full discussion of deallocation of reference-counted data types see section 11.1.2.

15.2 Shape function methods

15.2.1 make_element_shape

Module: shape_functions

There are two forms of the make_element_shape function. The first is the basic form:

```
function make_element_shape(vertices, dim, degree, quad, type,&
       stat, quad_s)
  type(element_type) :: make_element_shape
  integer, intent(in) :: vertices, dim, degree
  type(quadrature_type), intent(in), target :: quad
  integer, intent(in), optional :: type
  integer, intent(out), optional :: stat
  type(quadrature_type), intent(in), optional, target :: quad_s
```

This routine formulates a basis for a discrete function space over a single element. The geometry of the element is defined by the arguments vertices and dim as follows:

vertices	dim	Element geometry
1	1	point
2	1	interval
3	2	triangle
4	2	quad
4	3	tet
8	3	hex

The integration rule is provided by the quadrature_type argument, quad. The geometry of quad must match that specified in this call.

The argument **type** selects the element family from which the shape functions should be drawn:

element type	description
ELEMENT_LAGRANGIAN	Equispaced Lagrangian elements.
ELEMENT_LAGRANGIAN	$P1_{NC}$ non-conforming elements on triangles.
ELEMENT_CONTROLVOLUME_SURFACE	A control volume surface interior to the domain.
ELEMENT_CONTROLVOLUMEBDY_SURFACE	A control volume surface on the boundary of the domain.

ELEMENT_LAGRANGIAN is the default.

As with other routines, the **stat** argument, if present, returns 0 for successful completion and a non-zero value otherwise. If **stat** is not present then unsuccessful completion of the routine results in the termination of execution and an error message.

The quad_s argument provides a mechanism for providing quadrature for the surfaces of the element. **Not sure when this is used**.

The second form of make_element_shape allows an element_type to be based on another variable of the same type:

```
function make_element_shape(model, vertices, dim, degree, quad, type,&
       stat, quad_s)
  type(element_type) :: make_element_shape
  type(element_type), intent(in) :: model
  integer, intent(in), optional :: vertices, dim, degree
  type(quadrature_type), intent(in), optional, target :: quad
  integer, intent(in), optional :: type
  integer, intent(out), optional :: stat
  type(quadrature_type), intent(in), optional, target :: quad_s
```

In this form of the subroutine, all the arguments are optional except for model. Where any argument is absent, the corresponding value will be taken from model.

15.2.2 deallocate

```
subroutine deallocate(element, stat)
  type(element_type), intent(inout) :: element
  integer, intent(out), optional :: stat
```

This routine releases one reference to element. If this reduces the reference count to zero, element is deallocated and the quadrature reference held by element will be released.

15.2.3 local_coords

```
function element_local_coords(n, element)
  integer, intent(in) :: n
  type(element_type), intent(in) :: element
  real, dimension(local_coord_count(element)) :: element_local_coords
```

This routine returns the local coordinates in element of the node with local number n.

15.2.4 local_coord_count

```
function element_local_coord_count(element)
  integer :: element_local_coord_count
  type(element_type), intent(in) :: element
```

This routine returns the number of local coordinates associated with element. The results of the function are as follows:

element geometry	number of local coordinates
point	1
interval	2
triangle	3
quadrilateral	2
tet	4
hex	3

15.2.5 eval_shape

```
pure function eval_shape(shape, node, l)
    real :: eval_shape
    integer, intent(in) :: node
    type(element_type), intent(in) :: shape
    real, dimension(element_local_coord_count(element)), intent(in) :: l

pure function eval_shape(shape, l) result(eval_shape)
  type(element_type), intent(in) :: shape
  real, dimension(element_local_coord_count(element)), intent(in) :: l
  real, dimension(shape%loc) :: eval_shape
```

This routine evaluates one or more of the shape functions associated with the element `shape` at the local coordinates specified by `l`. In the first form of the routine, `node` is the local number of the node associated with the shape function to be evaluated. In the second form, all of the shape functions associated with the element are evaluated and the results returned as a vector.

Chapter 16

Functions implementing integrals over elements

16.1 Bilinear forms

The routines in this section form the local contributions pertaining to a bilinear form evaluated over a single element. As such they are the building blocks for the assembly of equations. Many of the arguments are essentially common across the routines. These are shown in table 16.1.

Argument	Shape	Mathematical notation	Meaning
`shape1`		$\{\ _i : i = 1 :: N_{\text{loc}}(\ )\}$	The set of local basis functions for the *test* space on the current element.
`shape2`		$\hat{}_j : j = 1 :: N_{\text{loc}}(\hat{})$	The set of local basis functions for the *trial* space on the current element.
`dshape1`	$N_{\text{loc}} \times N_{\text{quad}} \times N_{\text{dim}}$	$\{\nabla\ _i : i = 1 :: N_{\text{loc}}(\ )\}$	The set of gradients of the local basis functions for the *test* space on the current element.
`dshape2`	$N_{\text{loc}} \times N_{\text{quad}} \times N_{\text{dim}}$	$\nabla\hat{}_j : j = 1 :: N_{\text{loc}}(\hat{})$	The set of gradients of the local basis functions for the *trial* space on the current element.
`detwei`	N_{quad}	$w_{gi}\ J_{gi}^{-1} : gi = 1 :: N_{\text{quad}}$	The quadrature weights transformed by the change of coordinates from the current element to the reference element.

`vector`
`vector1`
`vector2`
	$N_{\text{dim}} \times N_{\text{quad}}$	$\{\mathbf{v}_\alpha : \ = 1 :: N_{\text{dim}}(\mathbf{v})\}$	A vector quantity evaluated at each quadrature point. This might be a value returned by ele_val_at_quad.

`tensor`	$N_{\text{dim}} \times N_{\text{dim}} \times N_{\text{quad}}$	$\{\bar{}_{\alpha\beta} : \ ; \ = 1 :: N_{\text{dim}}(\bar{})\}$	A tensor quantity evaluated at each quadrature point. This might be a value returned by ele_val_at_quad.

Table 16.1: Common arguments used in linear and bilinear form routines.

16.1.1 shape_shape

Module: fetools

```
function shape_shape(shape1, shape2, detwei)
  type(element_type), intent(in) :: shape1, shape2
  real, dimension(shape1%ngi), intent(in) :: detwei

  real, dimension(shape1%loc,shape2%loc) :: shape_shape
```

This routine constructs the following matrix:

$$\mathrm{M}(i;j) = \int_E \hat{}_i \hat{}_j \mathrm{d}V \quad i = 1:::N_{\mathrm{loc}}(\ ); \quad j = 1:::N_{\mathrm{loc}}(\hat{}) \tag{16.1}$$

Where the arguments are as given in table 16.1.

16.1.2 shape_shape_vector

Module: fetools

```
function shape_shape_vector(shape1, shape2, detwei, vector)
  type(element_type), intent(in) :: shape1, shape2
  real, dimension(shape1%ngi), intent(in) :: detwei
  real, dimension(:,:), intent(in) :: vector

  real, dimension(vector_dim(vector),shape1%loc,shape2%loc) :: &
       & shape_shape_vector
```

This routine constructs the following tensor:

$$\mathrm{M}(\ ;i;j) = \int_E \hat{}_i \hat{}_j \mathbf{v}_\alpha \mathrm{d}V \qquad = 1:::N_{\mathrm{dim}}(\mathbf{v}); i = 1:::N_{\mathrm{loc}}(\ ); \quad j = 1:::N_{\mathrm{loc}}(\hat{}) \tag{16.2}$$

Where the arguments are as given in table 16.1.

16.1.3 shape_shape_tensor

Module: fetools

```
function shape_shape_tensor(shape1, shape2, detwei, tensor)
  type(element_type), intent(in) :: shape1, shape2
  real, dimension(shape1%ngi), intent(in) :: detwei
  real, dimension(:,:,:), intent(in) :: tensor

  real, dimension(tensor_dim(tensor,1),tensor_dim(tensor,2),shape1%loc,shape2%loc)
       & shape_shape_tensor
```

This routine constructs the following tensor:

$$\begin{aligned}\mathrm{M}(\ ;\ ;i;j) = \int_E \hat{}_i \hat{}_j \bar{=}_{\alpha\beta} \mathrm{d}V \quad ; \quad &= 1:::N_{\mathrm{dim}}(\bar{=}); \\ &i = 1:::N_{\mathrm{loc}}(\ ); \quad j = 1:::N_{\mathrm{loc}}(\hat{})\end{aligned} \tag{16.3}$$

Where the arguments are as given in table 16.1.

16.1.4 shape_shape_vector_outer_vector

Module: fetools

```
function shape_shape_vector_outer_vector(shape1, shape2, detwei, &
       & vector1, vector2)
  type(element_type), intent(in) :: shape1, shape2
  real, dimension(shape1%ngi), intent(in) :: detwei
  real, dimension(:,:), intent(in) :: vector1
  real, dimension(:,:), intent(in) :: vector2

  real, dimension(vector_dim(vector1),vector_dim(vector2), &
       & shape1%loc,shape2%loc) :: shape_shape_vector_outer_vector
```

This routine constructs the following tensor:

$$\mathrm{M}(\ ;\ ;i;j) = \int_E {}_i\hat{}_j \mathsf{u}_\alpha \mathsf{v}_\beta \mathrm{d}V \qquad \begin{array}{l} = 1:::N_{\mathsf{dim}}(\mathsf{u}); \qquad = 1:::N_{\mathsf{dim}}(\mathsf{v}); \\ i = 1:::N_{\mathsf{loc}}(\ ); \quad j = 1:::N_{\mathsf{loc}}(\hat{}) \end{array} \tag{16.4}$$

Where the arguments are as given in table 16.1.

16.1.5 shape_dshape

Module: fetools

```
function shape_dshape(shape, dshape, detwei)
  type(element_type), intent(in) :: shape
  real, dimension(:,:,:), intent(in) :: dshape
  real, dimension(shape%ngi), intent(in) :: detwei

  real, dimension(dshape_dim(dshape), shape%loc, dshape_loc(dshape)) :: shape_dsha
```

This routine constructs the following tensor:

$$\begin{aligned} \mathrm{M}(\ ;i;j) &= \int_E {}_i(\nabla\hat{}_j)_\alpha \mathrm{d}V \\ &= \int_E {}_i\frac{\mathrm{d}\hat{}_j}{\mathrm{d}\mathsf{x}_\alpha}\mathrm{d}V \qquad = 1:::N_{\mathsf{dim}}(\hat{}); \quad i = 1:::N_{\mathsf{loc}}(\ ); \quad j = 1:::N_{\mathsf{loc}}(\hat{}) \end{aligned} \tag{16.5}$$

Where the arguments are as given in table 16.1.

16.1.6 dshape_shape

Module: fetools

```
function dshape_shape(dshape, shape, detwei)
  real, dimension(:,:,:), intent(in) :: dshape
  type(element_type), intent(in) :: shape
  real, dimension(shape%ngi), intent(in) :: detwei

  real, dimension(dshape_dim(dshape), shape%loc, dshape_loc(dshape)) :: shape_dsha
```

This routine constructs the following tensor:

$$\begin{aligned} \mathrm{M}(\ ;i;j) &= \int_E (\nabla\ _i)_\alpha\ \hat{}_j \mathrm{d}V \\ &= \int_E \frac{\mathrm{d}\ _i}{\mathrm{d}x_\alpha}\ \hat{}_j \mathrm{d}V \qquad = 1:::N_{\mathrm{dim}}(\ );\quad i = 1:::N_{\mathrm{loc}}(\ );\quad j = 1:::N_{\mathrm{loc}}(\ \hat{}\ ) \end{aligned} \tag{16.6}$$

Where the arguments are as given in table 16.1.

16.1.7 shape_vector_dot_dshape

Module: fetools

```
function shape_vector_dot_dshape(shape, vector, dshape, detwei)
  type(element_type), intent(in) :: shape
  real, dimension(:,:), intent(in) :: vector
  real, dimension(:,:,:), intent(in) :: dshape
  real, dimension(shape%ngi) :: detwei

  real, dimension(shape%loc,dshape_loc(dshape)) :: shape_vector_dot_dshape
```

This routine constructs the following matrix:

$$\begin{aligned} \mathrm{M}(i;j) &= \int_E\ _i\mathbf{v}\cdot\nabla\ \hat{}_j \mathrm{d}V \\ &= \int_E\ _i\mathbf{v}_\alpha \frac{\mathrm{d}\ \hat{}_j}{\mathrm{d}x_\alpha}\mathrm{d}V \qquad = 1:::N_{\mathrm{dim}}(\mathbf{v})\quad i = 1:::N_{\mathrm{loc}}(\ );\quad j = 1:::N_{\mathrm{loc}}(\ \hat{}\ ) \end{aligned} \tag{16.7}$$

Where the arguments are as given in table 16.1. Summation is implied over the repeated index with the effect that this operator is only defined if $N_{\mathrm{dim}}(\mathbf{v}) = N_{\mathrm{dim}}(\ \hat{}\ )$.

16.1.8 dshape_dot_vector_shape

Module: fetools

```
function dshape_dot_vector_shape(dshape, vector, shape, detwei)
  real, dimension(:,:,:), intent(in) :: dshape
  real, dimension(:,:), intent(in) :: vector
  type(element_type), intent(in) :: shape
  real, dimension(shape%ngi) :: detwei

  real,           ) 0Td[(dshape_dotor_dot_dshape)]T 0g0G0g0G/F3 10.9091Tf 111.2   2
```

16.1.9 dshape_dot_tensor_shape

Module: fetools

```
function dshape_dot_tensor_shape(dshape, tensor, shape, detwei)
  real, dimension(:,:,:), intent(in) :: dshape
  real, dimension(:,:,:), intent(in) :: tensor
  type(element_type), intent(in) :: shape
  real, dimension(dshape_ngi(dshape)) :: detwei

  real, dimension(tensor_dim(tensor,2),dshape_loc(dshape),shape%loc) :: &
       & dshape_dot_tensor_shape
```

This routine constructs the following matrix:

$$\begin{aligned} \mathrm{M}(\ ;i;j) &= \int_E \nabla\ _i \cdot \bar{\ } \hat{\ }_j \mathrm{d}V \\ &= \int_E \frac{\mathrm{d}\ _i}{\mathrm{d}x_\beta} \bar{\ }_{\alpha\beta} \hat{\ }_j \mathrm{d}V \quad ; \quad = 1:::N_{\mathrm{dim}}(\bar{\ }) \quad i = 1:::N_{\mathrm{loc}}(\ ); \quad j = 1:::N_{\mathrm{loc}}(\hat{\ }) \end{aligned} \tag{16.9}$$

Where the arguments are as given in table 16.1. Summation is implied over the repeated index with the effect that this operator is only defined if $N_{\mathrm{dim}}(\bar{\ }) = N_{\mathrm{dim}}(\ )$.

16.1.10 shape_vector_outer_dshape

Module: fetools

```
function shape_vector_outer_dshape(shape, vector, dshape, detwei)
  type(element_type), intent(in) :: shape
  real, dimension(:,:), intent(in) :: vector
  real, dimension(:,:,:), intent(in) :: dshape
  real, dimension(shape%ngi) :: detwei

  real, dimension(vector_dim(vector), dshape_dim(dshape), &
       & shape%loc, dshape_loc(dshape)) :: shape_vector_outer_dshape
```

This routine constructs the following tensor:

$$\begin{aligned} \mathrm{M}(\ ;\ ;i;j) &= \int_E {}_i\mathbf{v}_\alpha (\nabla \hat{\ }_j)_\beta \mathrm{d}V \\ &= \int_E {}_i\mathbf{v}_\alpha \frac{\mathrm{d}\hat{\ }_j}{\mathrm{d}x_\beta} \mathrm{d}V \quad = 1:::N_{\mathrm{dim}}(\mathbf{v}); \quad = 1:::N_{\mathrm{dim}}(\hat{\ }); \\ & \qquad\qquad i = 1:::N_{\mathrm{loc}}(\ ); \quad j = 1:::N_{\mathrm{loc}}(\hat{\ }) \end{aligned} \tag{16.10}$$

Where the arguments are as given in table 16.1.

16.1.11 dshape_outer_vector_shape

Module: fetools

```
function dshape_outer_vector_shape(dshape, vector, shape, detwei)
  type(element_type), intent(in) :: shape
  real, dimension(:,:), intent(in) :: vector
  real, dimension(:,:,:), intent(in) :: dshape
  real, dimension(dshape_ngi(dshape)) :: detwei
```

```
  real, dimension(dshape_dim(dshape), vector_dim(vector), &
       & dshape_loc(dshape,1), shape%loc) :: dshape_outer_vector_shape
```

This routine constructs the following tensor:

$$\begin{aligned} \mathrm{M}(\ ;\ ;i;j) &= \int_E (\nabla\ _i)_\alpha \mathsf{v}_\beta\ \hat{}_j \mathrm{d}V \\ &= \int_E \frac{\mathrm{d}\ _i}{\mathrm{d}\mathsf{x}_\alpha} \mathsf{v}_\beta\ \hat{}_j \mathrm{d}V \qquad = 1:::N_{\mathsf{dim}}(\ ); \qquad = 1:::N_{\mathsf{dim}}(\mathsf{v}); \\ & \qquad\qquad\qquad\qquad\qquad\qquad i = 1:::N_{\mathsf{loc}}(\ ); \quad j = 1:::N_{\mathsf{loc}}(\hat{}) \end{aligned} \tag{16.11}$$

Where the arguments are as given in table 16.1.

16.1.12 dshape_dot_dshape

Module: fetools

```
function dshape_dot_dshape(dshape1, dshape2, detwei)
  real, dimension(:,:,:), intent(in) :: dshape1, dshape2
  real, dimension(shape1%ngi), intent(in) :: detwei

  real, dimension(dshape_loc(dshape1),dshape_loc(dshape2)) :: dshape_dot_dshape
```

This routine constructs the following matrix:

$$\mathrm{M}(i;j) = \int_E \nabla\ _i \cdot \nabla\ \hat{}_j \mathrm{d}V \quad i = 1:::N_{\mathsf{loc}}(\ ); \quad j = 1:::N_{\mathsf{loc}}(\hat{}) \tag{16.12}$$

Where the arguments are as given in table 16.1.

16.1.13 dshape_tensor_dshape

Module: fetools

```
function dshape_tensor_dshape(dshape1, tensor, dshape2, detwei)
  real, dimension(:,:,:), intent(in) :: dshape1, dshape2
  real, dimension(:,:,:), intent(in) :: tensor
  real, dimension(shape1%ngi), intent(in) :: detwei

  real, dimension(dshape_loc(dshape1),dshape_loc(dshape2)) :: dshape_tensor_dshape
```

This routine constructs the following matrix:

$$\begin{aligned} \mathrm{M}(i;j) &= \int_E \nabla\ _i \cdot {}^{=} \cdot \nabla\ \hat{}_j \mathrm{d}V \\ &= \int_E (\nabla\ _i)_\alpha {}^{=}{}_{\alpha\beta} (\nabla\ \hat{}_j)_\beta \mathrm{d}V \qquad = 1:::N_{\mathsf{dim}}(\ ); \qquad = 1:::N_{\mathsf{dim}}(\hat{}) \\ & \qquad\qquad\qquad\qquad\qquad\qquad i = 1:::N_{\mathsf{loc}}(\ ); \quad j = 1:::N_{\mathsf{loc}}(\hat{}) \end{aligned} \tag{16.13}$$

Where the arguments are as given in table 16.1. Implicit summation occurs over both the indices and so that the dimensions of , ^ and $^{=}$ must match accordingly.

16.1.14 dshape_outer_dshape

Module: fetools

```
function dshape_dot_dshape(dshape1, dshape2, detwei)
  real, dimension(:,:,:), intent(in) :: dshape1, dshape2
  real, dimension(shape1%ngi), intent(in) :: detwei

  real, dimension(dshape_dim(dshape1), dshape_dim(dshape2), &
       &           dshape_loc(dshape1), dshape_loc(dshape2)) :: dshape_dot_dshape
```

This routine constructs the following matrix:

$$\mathrm{M}(\ ;\ ;i;j) = \int_E (\nabla\ _i)_\alpha (\nabla\hat{}_j)_\beta \mathrm{d}V \qquad \begin{array}{l} =1:::N_{\mathrm{dim}}(\ ); \qquad =1:::N_{\mathrm{dim}}(\hat{}) \\ i=1:::N_{\mathrm{loc}}(\ ); \quad j=1:::N_{\mathrm{loc}}(\hat{}) \end{array} \tag{16.14}$$

Where the arguments are as given in table 16.1.

16.1.15 shape_curl_shape_2d

Module: fetools

```
function shape_curl_shape_2d(shape, dshape, detwei)
  type(element_type), intent(in) :: shape
  real, dimension(:,:,:), intent(in) :: dshape
  real, dimension(shape%ngi) :: detwei

  real, dimension(2,shape%loc,dshape_loc(dshape)) :: shape_curl_shape_2d
```

This routine constructs the following matrix:

$$\begin{aligned} \mathrm{M}(:;i;j) &= \int_E {}_i\nabla \times \hat{}_i \mathrm{d}V \\ &= \int_E \begin{bmatrix} i\frac{\mathrm{d}\hat{\phi}_{1,j}}{\mathrm{d}y} \\ -\ i\frac{\mathrm{d}\hat{\phi}_{2,j}}{\mathrm{d}x} \end{bmatrix} \mathrm{d}V \end{aligned} \tag{16.15}$$

Note that due to the dimension-specific nature of the curl operator, this routine only works in two dimensions. For information on the treatment of vector shape functions in femtools, see section 9.1

16.2 Linear Forms

In contrast to the bilinear forms in the previous section, the routines here do not accept a trial function argument. This is primarily useful for assembling right hand side contributions for which all the functions in the integral other than the test function are already known and have been multiplied by `detwei` in the function arguments.

16.2.1 shape_rhs

Module: fetools

```
function shape_rhs(shape, detwei)
  type(element_type), intent(in) :: shape
  real, dimension(shape1%ngi), intent(in) :: detwei
```

```
  real, dimension(shape%loc) :: shape_rhs
```

This routine constructs the following vector:

$$\mathsf{I}(i) = \int_E {}_i \mathrm{d}V \quad i = 1:::N_{\mathsf{loc}}(\ ) \tag{16.16}$$

Where the arguments are as given in table 16.1.

16.2.2 shape_vector_rhs

Module: fetools

```
function shape_vector_rhs(shape, vector, detwei)
  type(element_type), intent(in) :: shape
  real, dimension(:,:), intent(in) :: vector
  real, dimension(shape1%ngi), intent(in) :: detwei

  real, dimension(vector_dim(vector), shape%loc) :: shape_vector_rhs
```

This routine constructs the following matrix:

$$\mathsf{I}(\ ;i) = \int_E {}_i \mathsf{v}_\alpha \mathrm{d}V \qquad = 1:::N_{\mathsf{dim}}(\mathsf{v}); \quad i = 1:::N_{\mathsf{loc}}(\ ) \tag{16.17}$$

Where the arguments are as given in table 16.1.

16.2.3 shape_tensor_rhs

Module: fetools

```
function shape_tensor_rhs(shape, tensor, detwei)
  type(element_type), intent(in) :: shape
  real, dimension(:,:,:), intent(in) :: tensor
  real, dimension(shape1%ngi), intent(in) :: detwei

  real, dimension(tensor_dim(tensor,1), tensor_dim(tensor,2), shape%loc) &
       & :: shape_tensor_rhs
```

This routine constructs the following tensor:

$$\mathsf{I}(\ ;\ ;i) = \int_E {}_i \bar{=}_{\alpha\beta} \mathrm{d}V \qquad ; \quad = 1:::N_{\mathsf{dim}}(\mathsf{v}); \quad i = 1:::N_{\mathsf{loc}}(\ ) \tag{16.18}$$

Where the arguments are as given in table 16.1.

16.2.4 shape_tensor_dot_vector_rhs

Module: fetools

```
function shape_tensor_dot_vector_rhs(shape, tensor, vector, detwei)
  type(element_type), intent(in) :: shape
  real, dimension(:,:,:), intent(in) :: tensor
  real, dimension(:,:,:), intent(in) :: vector
  real, dimension(shape1%ngi), intent(in) :: detwei

  real, dimension(tensor_dim(tensor,1), shape%loc) :: shape_tensor_rhs
```

This routine constructs the following tensor:

$$\mathsf{I}(\ ;i) = \int_E {}_i{}^{=}{}_{\alpha\beta}\mathsf{v}_\beta \mathrm{d}V \quad ; \quad = 1:::N_{\mathsf{dim}}(\mathsf{v}); \quad i = 1:::N_{\mathsf{loc}}(\ ) \tag{16.19}$$

Where the arguments are as given in table 16.1. Implicit summation is applied to so the corresponding dimensions of $^{=}$ and v must match accordingly.

16.2.5 dshape_rhs

Module: fetools

```
function dshape_rhs(dshape, detwei)
  real, dimension(:,:,:), intent(in) :: dshape
  real, dimension(:), intent(in) :: detwei

  real, dimension(dshape_loc(dshape)) :: dshape_rhs
```

This routine constructs the following tensor:

$$\begin{aligned} \mathsf{I}(\ ;i) &= \int_E \nabla\ {}_i \mathrm{d}V \\ &= \int_E (\nabla\ {}_i)_\alpha \mathrm{d}V \quad = 1:::N_{\mathsf{dim}}(\nabla\ ); \quad i = 1:::N_{\mathsf{loc}}(\ ) \end{aligned} \tag{16.20}$$

Where the arguments are as given in table 16.1. Implicit summation occurs over so the dimensions of and v must match accordingly.

16.2.6 dshape_dot_vector_rhs

Module: fetools

```
function dshape_dot_vector_rhs(dshape, vector, detwei)
  real, dimension(:,:,:), intent(in) :: dshape
  real, dimension(:,:), intent(in) :: vector
  real, dimension(:), intent(in) :: detwei

  real, dimension(dshape_loc(dshape)) :: dshape_dot_vector_rhs
```

This routine constructs the following tensor:

$$\begin{aligned} \mathsf{I}(i) &= \int_E \nabla\ {}_i \cdot \mathsf{v} \mathrm{d}V \\ &= \int_E (\nabla\ {}_i)_\alpha \mathsf{v}_\alpha \mathrm{d}V \quad = 1:::N_{\mathsf{dim}}(\mathsf{v}); \quad i = 1:::N_{\mathsf{loc}}(\ ) \end{aligned} \tag{16.21}$$

Where the arguments are as given in table 16.1. Implicit summation occurs over so the dimensions of and v must match accordingly.

16.2.7 dshape_dot_tensor_rhs

Module: fetools

```
function dshape_dot_tensor_rhs(dshape, tensor, detwei)
  real, dimension(:,:,:), intent(in) :: dshape
  real, dimension(:,:,:), intent(in) :: tensor
```

```
real, dimension(:), intent(in) :: detwei

real, dimension(tensor_dim(tensor,2), dshape_loc(dshape)) :: dshape_dot_tensor_rh
```

This routine constructs the following tensor:

$$\begin{aligned} I(\ ;i) &= \int_{E} \nabla\ _i \cdot \bar{=} \mathrm{d}V \\ &= \int_{E} (\nabla\ _i)_\alpha \bar{=}_{\alpha\beta} \mathrm{d}V \quad ; \quad = 1:::N_{\mathrm{dim}}(\bar{=}); \quad i = 1:::N_{\mathrm{loc}}(\ ) \end{aligned} \tag{16.22}$$

Where the arguments are as given in table 16.1. Implicit summation occurs over so the dimensions of and $\bar{=}$ must match accordingly.

16.3 Auxiliary form functions

These functions are used to provide the sizes of the linear and bilinear forms. Note that these functions are written to make the manual make sense and do not yet actually exist in the code! These will get put in the code, probably as preprocessor macros, when I get back on dry land.

16.3.1 dshape_loc

Module: none

```
function dshape_loc(dshape)
  integer :: dshape_loc
  real, dimension(:,:,:) :: dshape
```

This routine returns the number of nodes in the shape function gradient given by `dshape`. This is equivalent to **`size`**`(dshape,1)`.

16.3.2 dshape_ngi

Module: none

```
function dshape_ngi(dshape)
  integer :: dshape_ngi
  real, dimension(:,:,:) :: dshape
```

This routine returns the number of quadrature points of the shape function gradient given by `dshape`. This is equivalent to **`size`**`(dshape,2)`.

16.3.3 dshape_dim

Module: none

```
function dshape_dim(dshape)
  integer :: dshape_dim
  real, dimension(:,:,:) :: dshape
```

This routine returns the topological dimension of the shape function gradient given by `dshape`. This is equivalent to **`size`**`(dshape,3)`.

16.3.4 vector_dim

Module: none

```
function vector_dim(vector)
  integer :: vector_dim
  real, dimension(:,:) :: vector
```

This routine returns the data dimension of vector, where vector is a real array providing a vector value at each quadrature point of an element. This is equivalent to `size(vector,1)`.

16.3.5 tensor_dim

Module: none

```
function tensor_dim(tensor, dim)
  integer :: tensor_dim
  real, dimension(:,:,:) :: tensor
```

This routine returns the `dim`th data dimension of tensor, where tensor is a real array providing a rank 2tensor value at each quadrature point of an element. This is equivalent to `size(tensor,dim)`. Valid values of `dim` are 1 and 2.

Chapter 17

Diagnostic statistics

17.1 Diagnostic I/O routines

document ewrite and friends here

17.2 Memory statistics

The memory statistics are automatically output in the **.stat** file, if one is in use, however there are some routines which may be useful for debugging purposes.

17.2.1 print_current_memory_stats

```
subroutine print_current_memory_stats(priority)
  integer, intent(in) :: priority
```

This routine prints to screen the current memory allocated to each memory heading. The priority determines whether or not the print will occur: the verbosity of the simulation must be at least equal to the priority or printing will be supressed. Printing is also suppressed if the priority is 0 and there is no memory currently allocated. This facilitates the use of this routine at the end of a simulation to highlight memory leaks.

17.2.2 print_memory_stats

```
subroutine print_memory_stats(priority)
  integer, intent(in) :: priority
```

This routine prints to screen the current, minimum and maximum memory allocated to each memory heading. The priority determines whether or not the print will occur: the verbosity of the simulation must be at least equal to the priority or printing will be supressed.

17.2.3 reset_memory_logs

```
subroutine reset_memory_logs
```

This routine resets the minimum and maximum statistics of all the memory headings to the current value for that heading. This is useful for recording the memory peaks in some part of a simulation, for example one timestep. If a stat file is being generated, this routine will automatically be called

each time a new row in the stat file is generated.

17.3 Register diagnostics in the .stat file

The way registered diagnostics are added to the **.stat** file is similar to the way the options are checked. The script `scripts/make_register_diagnostics` is executed when Fluidity is compiled. All the modules that contain a subroutine called `MODULE_NAME_register_diagnostic` are in turn listed in `preprocessor/register_diagnostics.F90`. These subroutines are executed before the simulation starts and they register the diagnostics appearing in the module.

A diagnostic has the type `registered_diagnostic_item`, i.e.

```
type registered_diagnostic_item
  integer :: dim
  character(len=FIELD_NAME_LEN) :: name
  character(len=FIELD_NAME_LEN) :: statistic
  character(len=FIELD_NAME_LEN) :: material_phase
  logical :: have_material_phase
  real, dimension(:), allocatable :: value
  type(registered_diagnostic_item), pointer :: next => null()
```

Each diagnostic is registered by the subroutine:

```
subroutine register_diagnostic(dim, name, statistic, material_phase)
  integer, intent(in) :: dim
  character(len=*), intent(in) :: name, statistic
  character(len=*), intent(in), optional ::  material_phase
```

It initialises all the attributes of the diagnostic and appends it to a list of registered diagnostics. An error occurs if the diagnostic has already been registered. The registered diagnostics are added to the header of the **.stat** file in the subroutine `initialise_diagnostics`. After the initialisation, the attributes of each registered diagnostic are printed in the log by:

```
subroutine print_registered_diagnostics
```

The value of a diagnostic is set by:

```
subroutine set_diagnostic(name, statistic, material_phase, value)
  character(len=*), intent(in) :: name, statistic
  character(len=*), intent(in), optional ::  material_phase
  real, dimension(:), intent(in) :: value
```

which searches for the appropriate diagnostic in the list of registered diagnostics and sets its value. Finally, the diagnostics are destroyed by:

```
subroutine destroy_registered_diagnostics
```

This subroutine is called to clean up the diagnostics in `uninitialise_diagnostics`.

To summarise, the procedure to register diagnostics in the **.stat** file is the following.

1. In the module where the diagnostic is created, add a public subroutine called `MODULE_NAME_register_diagnostic`. In this subroutine, call `register_diagnostic` for each diagnostic to register, specifying its dimension, statistic name, statistic type and material phase (the latter being optional).
2. Set the value of each diagnostic by calling `set_diagnostic`.