Finite element code: PDE

David E. Stewart

July 9th, 2012

Contents

1 Overview

This describes a pure Matlab finite element code for two dimensional problems. It is assumed that a triangulation of the domain is given in the same format as the output from the Persson & Strang Matlab triangulation code [5]. That is, the basic triangulation data is given in the form of a pair (p, t) where p is an array of points: point p_i is `p(i,:)`, while triangle i is the triangle with vertices `p(j,:)`, `p(k,:)`, `p(l,:)` where `j = t(i,1)`, `k = t(i,2)` and `l = t(i,3)`. This can be generalized to tetrahedra in three dimensions, etc. This also works well with the Matlab `trimesh()` function for plotting two-dimensional meshes.

There are a number of different element types, most notably Lagrange elements which represent scalar piecewise polynomials (that is, the restriction of the basis functions to each triangle is a polynomial). The simplest of these is the Lagrange piecewise linear element, but quadratic and cubic Lagrangian elements have also been implemented. Extensions to C^1 elements (such as Bell's triangle and the Argyris element) are also planned, but not yet implemented. Code for the elements can be found in Section 5.

Since the values at certain points are shared by elements on different (but touching) triangles, we need a way to determine when these values are shared. This is done via a geometric feature hash table. Code for these aspects can be found in Section 4.

The core routines are the assembly routines which form the matrices and vectors for the linear systems to be solved. These are for both the Galerkin and Petrov–Galerkin methods. Similar routines are provided for handling boundary values and conditions. Code for matrix assembly can be found in Section 3. Part of this process is the task of numerical integration. Integration rules can be found in Section 6.

Testing codes can be found in Section 12.

1.1 Basic organization

Each element type must be able to compute the values of the basis functions at each point of the reference triangle $\widehat{K} = \operatorname{co}\{\,(0,0),\,(1,0),\,(0,1)\,\}$, along with the values of a number of *operators* applied to the basis functions. That is, for a point $\widehat{\mathbf{x}} \in \widehat{K}$ and basis function $\widehat{\phi}_i$ on $\widehat{K}$ we need to be able to compute not only $\widehat{\phi}_i(\widehat{\mathbf{x}})$, but also $\mathcal{A}\widehat{\phi}_i(\widehat{\mathbf{x}})$ where $\mathcal{A} = \partial/\partial x_1$, $\mathcal{A} = \partial/\partial x_2$; sometimes higher order derivatives are also necessary, such as for 4th order PDEs. In that case, each element can also compute $\mathcal{A}\widehat{\phi}_i(\widehat{\mathbf{x}})$ where $\mathcal{A} = \partial^2/\partial x_1^2$, $\mathcal{A} = \partial^2/\partial x_1\,\partial x_2$ and $\mathcal{A} = \partial^2/\partial x_2^2$. The ordering of these operators is essentially fixed across the different element types.

There are also elements for providing vector-valued basis functions, in which case we also need to use different operators: $\mathcal{A}\widehat{\phi}_i(\widehat{\mathbf{x}}) = \widehat{\phi}_i(\widehat{\mathbf{x}}) \cdot \mathbf{e}_1$, $\mathcal{A}\widehat{\phi}_i(\widehat{\mathbf{x}}) = \widehat{\phi}_i(\widehat{\mathbf{x}}) \cdot \mathbf{e}_2$, $\mathcal{A}\widehat{\phi}_i(\widehat{\mathbf{x}}) = \partial\widehat{\phi}_i(\widehat{\mathbf{x}})/\partial x_1 \cdot \mathbf{e}_1$, etc.

Basis functions on the reference element $\widehat{K}$ are used to create basis functions on the actual elements $K = \operatorname{co}\left\{\,\mathbf{p}_j,\,\mathbf{p}_k,\,\mathbf{p}_\ell\,\right\}$ by means of an affine transformation $\widehat{\mathbf{x}} \mapsto \mathbf{x} = T_K\widehat{\mathbf{x}} + \mathbf{b}_K$ with T_K and $\mathbf{b}_K$ computed from $\mathbf{p}_j$, $\mathbf{p}_k$, $\mathbf{p}_\ell$. This also transforms that values of $\mathcal{A}'\widehat{\phi}_i(\widehat{\mathbf{x}})$ to compute $\mathcal{A}\phi_i(\mathbf{x})$: $\phi_i(\mathbf{x}) = \widehat{\phi}_i(\widehat{\mathbf{x}})$, but

$$\frac{\partial \phi_i}{\partial x_1}(\mathbf{x}) = \frac{\partial \widehat{\phi}_i}{\partial \widehat{x}_1}(\widehat{\mathbf{x}})\frac{\partial \widehat{x}_1}{\partial x_1}(\mathbf{x}) + \frac{\partial \widehat{\phi}_i}{\partial \widehat{x}_2}(\widehat{\mathbf{x}})\frac{\partial \widehat{x}_2}{\partial x_1}(\mathbf{x}),$$

for example. The derivatives $\partial \widehat{x}_i / \partial x_i$ are entries of the T_K matrix. In fact,

$$\nabla \phi_i(\mathbf{x}) = (T_K)^T \nabla \widehat{\phi}_i(\widehat{\mathbf{x}}).$$

The matrix entries are formed by means of integrals

$$\begin{aligned} a_{ij} &= \int_\Omega \sum_{\mathcal{A},\mathcal{B}} c_{\mathcal{A},\mathcal{B}}(\mathbf{x})\, \mathcal{A}\phi_i(\mathbf{x})\, \mathcal{B}\phi_j(\mathbf{x})\, d\mathbf{x} \\ &= \sum_K \int_K \sum_{\mathcal{A},\mathcal{B}} c_{\mathcal{A},\mathcal{B}}(\mathbf{x})\, \mathcal{A}\phi_i(\mathbf{x})\, \mathcal{B}\phi_j(\mathbf{x})\, d\mathbf{x}. \end{aligned}$$

The sum over $\mathcal{A}$ and $\mathcal{B}$ is over all the operators used to define the Galerkin form of the partial differential equations; the sum over K is the sum over all the triangles of the triangulation. To compute the integral over K we use rules for integration over the reference triangle $\widehat{K}$.

For the Petrov–Galerkin method, we can use different basis functions (and thus different element types), but they must be based on the same triangulation:

$$b_{ij} = \sum_K \int_K \sum_{\mathcal{A},\mathcal{B}} c_{\mathcal{A},\mathcal{B}}(\mathbf{x})\, \mathcal{A}\psi_i(\mathbf{x})\, \mathcal{B}\phi_j(\mathbf{x})\, d\mathbf{x}.$$

There are also matrix assembly routines for boundaries. Boundaries of two-dimensional regions are given as sets of edges (each edge being a pair of indexes into the point array `p`).

1.2 PDE representation

The PDE itself is represented by `pde` structure, which is based on the Galerkin (or Galerkin–Petrov) method. If the weak form is of the Galerkin type:

$$\int_\Omega \sum_{\mathcal{A},\mathcal{B}} c_{\mathcal{A},\mathcal{B}}(\mathbf{x})\, \mathcal{A}v(\mathbf{x})\, \mathcal{B}u(\mathbf{x})\, d\mathbf{x} = \int_\Omega \sum_{\mathcal{A}} f_{\mathcal{A}}(\mathbf{x})\, \mathcal{A}v(\mathbf{x})\, d\mathbf{x} \qquad \text{for all } v \in \operatorname{span}\{\phi_i\}_{i=1}^N,$$

then `pde` consists of the maximum order of the operators $\mathcal{A}$ and $\mathcal{B}$, together with the functions $C\colon \Omega \to \mathbb{R}^{M\times M}$ and $\mathbf{f}\colon \Omega \to \mathbb{R}^M$ where M is the number of operators $\mathcal{A}$ considered. For example, for a scalar problem in two dimensions where the Galerkin form only involves function values and first derivatives, $\mathcal{A}$ can be I (identity) for the function values, $\partial/\partial x_1$, or $\partial/\partial x_2$ for the first derivatives. Then $M = 3$.

For the PDE $-\Delta u = f(\mathbf{x})$ with $f(\mathbf{x}) = x_1^2 \exp(x_2)$, we use:

6a ⟨*pde-struct-eg* 6a⟩≡
```
pde = struct('coeffs',@(x)diag([0,1,1]), ...
        'rhs',@(x)[x(1)^2*exp(x(2));0;0],'order',1)
```

1.3 Usage

Were we present an example of the solution of

$$\begin{aligned} -\Delta u &= f(\mathbf{x}) \qquad \text{in } \Omega, \\ u(\mathbf{x}) &= g(\mathbf{x}) \qquad \text{on } \partial\Omega. \end{aligned}$$

As usual $\partial\Omega$ is the boundary of Ω, and $\Delta u = \partial^2 u/\partial x_1^2 + \partial^2 u/\partial x_2^2$ is the Laplacian operator. This is an elliptic partial differential operator of second order with Dirichlet (forced) boundary conditions.

The weak form of this PDE is

$$\int_\Omega \left[\frac{\partial u}{\partial x_1}\frac{\partial v}{\partial x_1} + \frac{\partial u}{\partial x_2}\frac{\partial v}{\partial x_2} \right] d\mathbf{x} = \int_\Omega \nabla u \cdot \nabla v \, d\mathbf{x} = \int_\Omega f(\mathbf{x})\, v(\mathbf{x})\, d\mathbf{x}$$

for all v that is "nice" and satisfies $v(\mathbf{x}) = 0$ for all $\mathbf{x} \in \partial\Omega$. Here "nice" can mean smooth, but it is also true when v is a piecewise linear function over the triangulation with "$v = 0$ on $\partial\Omega$".

The domain Ω has to be defined according to the problem. An example is a square $[-1, +1] \times [-1, +1]$ with a circle center at the origin and radius $1/2$ removed. A triangulation can be computed using `distmesh` from Persson and Strang (URL `http://persson.berkeley.edu/distmesh/`) [5] as follows:

6b ⟨*filelist* 6b⟩≡
```
usage.m \
```

6c ⟨*usage.m* 6c⟩≡
```
fd = @(p)ddiff(drectangle(p,-1,1,-1,1),dcircle(p,0,0,0.5))
fh = @(p)min(4*sqrt(sum(p.^2,2))-1,2)
[p,t]=distmesh2d(fd,fh,0.1,[-1,-1;1,1],[-1,-1;-1,1;1,-1;1,1]);
np = size(p,1)
```

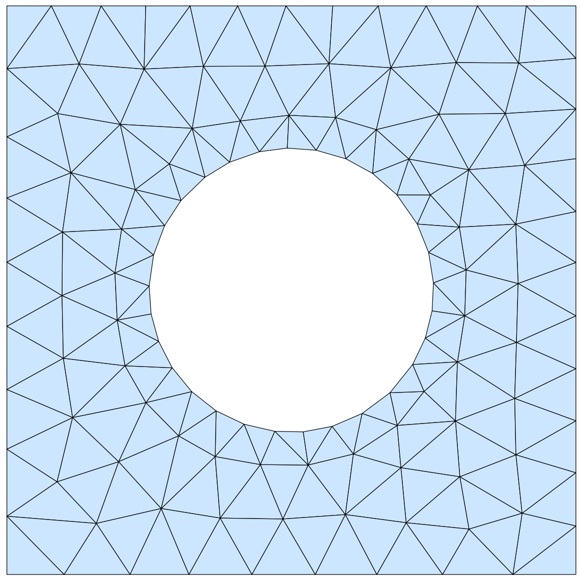

Figure 1: Mesh produced by `distmesh2d()`

The `fd` function defines the region, while `fh` is used to control the variation of the triangle sizes. Then `distmesh2d()` itself creates the triangulation. See the documentation on `distmesh` for more information. The last line just tells us the number of points in the triangulation. The triangulation is shown in Figure 1.

Once the triangulation has been computed, we can create the element type (piecewise linear):

7a ⟨*usage.m* 6c⟩+≡

```
lin2d = lin2d_elt()
```

The triangulation and the element type together determine the variables:

7b ⟨*usage.m* 6c⟩+≡

```
fht = create_fht(p,t,lin2d)
nv = fht_num_vars(fht)
```

Here `fht` is the geometric feature hash table, which relates geometric features (triangles, edges, vertices) with variable indexes. The value of `nv` is the number of variables in the system.

Since the weak form of the PDE is

$$\int_\Omega \left[\frac{\partial u}{\partial x_1}\frac{\partial v}{\partial x_1} + \frac{\partial u}{\partial x_2}\frac{\partial v}{\partial x_2} \right] d\mathbf{x} = \int_\Omega f(\mathbf{x})\, v(\mathbf{x})\, d\mathbf{x}$$

where $v = 0$ on $\partial\Omega$, the PDE structure representing this for $f(\mathbf{x}) = 10x_1^2 \exp(x_2)$ is

8a ⟨*usage.m* 6c⟩+≡

```
f = @(x)(10*x(1)^2*exp(x(2)))
pde = struct('coeffs',@(x)diag([0,1,1]),'rhs',@(x)[f(x);0;0],'order',1)
```

The `coeffs` component is a function returning a matrix that represents the bilinear weak form above:

$$\frac{\partial u}{\partial x_1}\frac{\partial v}{\partial x_1} + \frac{\partial u}{\partial x_2}\frac{\partial v}{\partial x_2} = \begin{bmatrix} v \\ \partial v/\partial x_1 \\ \partial v/\partial x_2 \end{bmatrix}^T \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} u \\ \partial u/\partial x_1 \\ \partial u/\partial x_2 \end{bmatrix}.$$

The `rhs` component is a function returning a vector that represents the linear part of the weak form (on the right):

$$f(\mathbf{x})\, v(\mathbf{x}) = \begin{bmatrix} v(\mathbf{x}) \\ \partial v/\partial x_1(\mathbf{x}) \\ \partial v/\partial x_2(\mathbf{x}) \end{bmatrix}^T \begin{bmatrix} f(\mathbf{x}) \\ 0 \\ 0 \end{bmatrix}.$$

The `order` component is set to one to indicate that only function values and first order derivatives are needed. The maximum value of `order` (in the current code) is two.

Now we need to create the matrix and right-hand side vector for the linear system to solve. First we assemble the matrix and right-hand side vector for the PDE. As yet, this does not deal with boundary values:

8b ⟨*usage.m* 6c⟩+≡

```
% Initialize A and b
A = sparse(nv,nv);
b = zeros(nv,1);
% Assemble matrix and vector
[A,b] = assembly2d(A,b,pde,lin2d,p,t,fht,@int2d_radon7);
```

Note that the matrix A is created as a sparse matrix. It is not necessary to do so, but it is recommended as the systems created are generally very sparse. The vector $\mathbf{b}$ does not need to be created as a sparse vector. These are initialized to zero by the `sparse()` and `zeros()` functions. The `assembly2d()` function adds the assembled matrices and vectors to the pre-existing A and $\mathbf{b}$; this feature is useful if you are combining several different partial differential operators into one. An integration method needed to be selected: Radon's 7-point scheme for triangles is 5th order accurate, which is sufficient for our purposes.

Since we have Dirichlet boundary conditions, we need to explicitly set the values of the variables of the boundary nodes according to $g(\mathbf{x})$. First we need to identify the boundary edges and nodes:

9a ⟨*usage.m* 6c⟩+≡
```
[bedges,bnodes,t_index] = boundary2d(t);
```

There are several ways of setting boundary values. The method presented here essentially solves a least squares problem to find the finite element function that best approximates the given $g(\mathbf{x})$. This shows another usage of the PDE data structure as well as the boundary assembly function for $g(\mathbf{x}) = \cos(x_1)\, x_2$:

9b ⟨*usage.m* 6c⟩+≡
```
g = @(x)(cos(x(1))*x(2))
pde2 = struct('coeffs',@(x)[1],'rhs',@(x)g(x),'order',0)
```

Note that the `order` parameter is set to zero to indicate that no derivatives are involved. Then we assemble the normal equations matrix and right-hand side for the least squares problem:

9c ⟨*usage.m* 6c⟩+≡
```
[Ab,bb,bvlist] = assembly2dbdry(pde2,lin2d,p,t,bedges,t_index,fht,@int1d_gauss5);
```

Now we solve the linear system to obtain values of the boundary variables.

9d ⟨*usage.m* 6c⟩+≡
```
g1 = Ab(bvlist,bvlist) \ bb(bvlist);
```

Now we can solve the remainder of the system for the non-boundary variables. First, find the non-boundary variables (`cbvlist` means "complement of `bvlist`").

10a ⟨*usage.m* 6c⟩+≡

```
% find non-boundary variables (cbvlist)
v_array = ones(nv,1); v_array(bvlist) = 0;
cbvlist = find(v_array ~= 0);
```

Now we solve the linear system for the non-boundary variables and insert the results into the vector **u**; the boundary variables in **u** are in `g1`.

10b ⟨*usage.m* 6c⟩+≡

```
u_int = A(cbvlist,cbvlist) \ (b(cbvlist) - A(cbvlist,bvlist)*g1);
u = zeros(nv,1);
u(cbvlist) = u_int;
u( bvlist) = g1;
```

We can plot the values of our numerical solution $u(\mathbf{x})$ at the vertices of the triangulation by means of `trimesh()` and a helper routine `pvlist()` that returns the variable indexes for the vertices of the triangulation.

10c ⟨*usage.m* 6c⟩+≡

```
pvlist = get_pvlist(fht,np);
figure(2)
trimesh(t,p(:,1),p(:,2),u(pvlist))
```

The result is illustrated in Figure 2.

To repeat the calculation using piecewise quadratic elements, we must first create the associated element type structure, and create the feature hash table for that element type.

10d ⟨*usage.m* 6c⟩+≡

```
% Now using piecewise quadratic elements
quad2d = quad2d_elt()
fht2 = create_fht(p,t,quad2d)
nv2 = fht_num_vars(fht2)
```

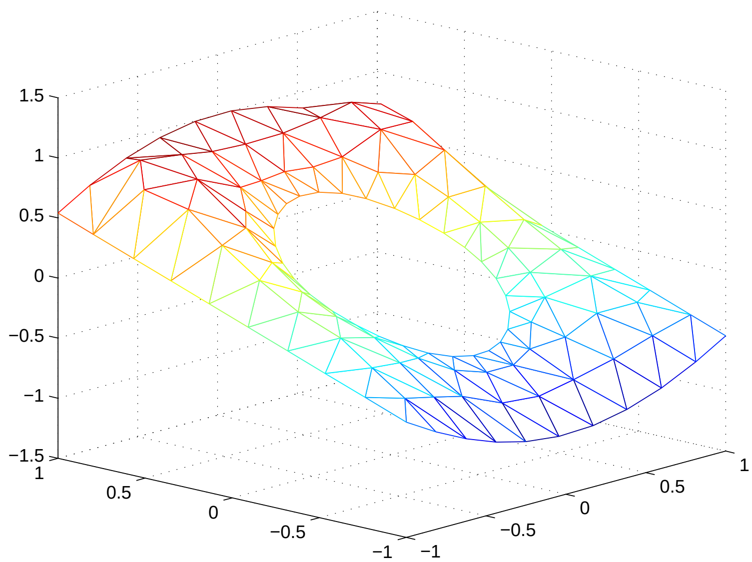

Figure 2: Computed solution to PDE

Then we need to assemble the matrix and right-hand side:

11 ⟨*usage.m* 6c⟩+≡

```
% Initialize A2 and b2
A2 = sparse(nv2,nv2);
b2 = zeros(nv2,1);
% Assemble matrix and vector
[A2,b2] = assembly2d(A2,b2,pde,quad2d,p,t,fht2,@int2d_radon7);
```

Note that the pde structure remains unchanged, as does the triangulation. While the boundary nodes remain unchanged, the boundary *variables* do not. We also use the least-squares approach to the Dirichlet boundary values, which works just as well for the case of piecewise quadratic elements as for piecewise linear elements. So we use the following code:

12a ⟨*usage.m* 6c⟩+≡
```
[Ab2,bb2,bvlist2] = ...
    assembly2dbdry(pde2,quad2d,p,t,bedges,t_index,fht2,@int1d_gauss5);
% find non-boundary variables (cbvlist2)
v_array = ones(nv2,1); v_array(bvlist2) = 0;
cbvlist2 = find(v_array ~= 0);

g2 = Ab2(bvlist2,bvlist2) \ bb2(bvlist2);
u_int = A2(cbvlist2,cbvlist2) \ (b2(cbvlist2) - A2(cbvlist2,bvlist2)*g2);
u2 = zeros(nv2,1);
u2(cbvlist2) = u_int;
u2( bvlist2) = g2;
```

To properly display the results, which are more accurate than the results of piecewise linear elements, we should use sub-meshes. First we create the sub-mesh for the reference element (which is the triangle with vertices $(0,0)$, $(1,0)$, and $(0,1)$):

12b ⟨*usage.m* 6c⟩+≡
```
[pr,tr] = ref_triangle_submesh(4);
```

The returned triangulation of the reference element has the edges of 4 small triangles on each coordinate axis. Then we can generate the values on a sub-mesh of the original triangulation, and plot the results.

12c ⟨*usage.m* 6c⟩+≡
```
[pv,tv,vals] = get_submesh_vals(p,t,fht2,quad2d,u2,pr,tr,0);
figure(3)
trimesh(tv,pv(:,1),pv(:,2),vals)
```

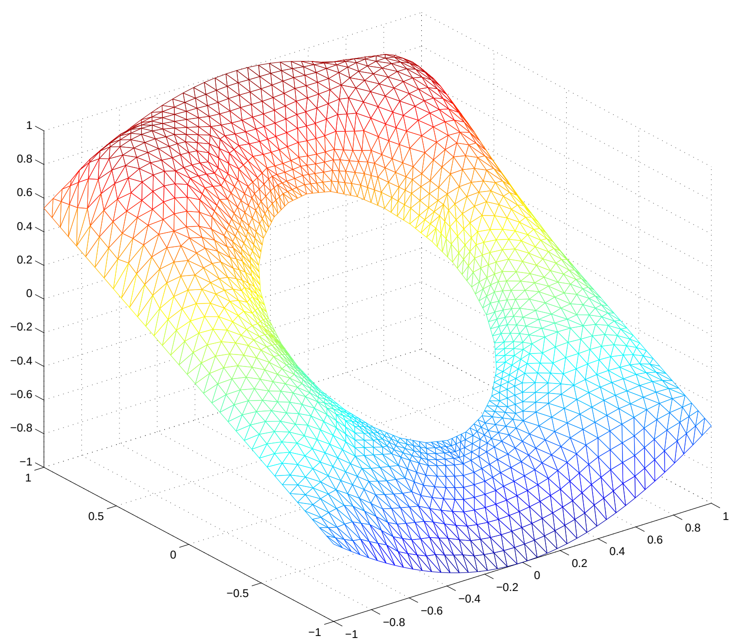

Figure 3: Results of using quadratic elements (plotted using a sub-mesh)

The last input to `get_submesh_vals()` is the maximum order of the derivatives values generated. Since we just want to plot the values of the solution, this is set to zero. The result is shown in Figure 3.

1.3.1 Usage: a convection diffusion problem

This equation is more complex than the previous example, but on the same region:

$$\begin{aligned}
-\nabla\cdot[\alpha(\mathbf{x})\nabla u]+\mathbf{w}(\mathbf{x})\cdot\nabla u &= f(\mathbf{x}) \qquad \text{in } \Omega,\\
u(\mathbf{x}) &= g(\mathbf{x}) \qquad \text{on } \Gamma_1,\\
\beta(\mathbf{x})u(\mathbf{x})+\alpha(\mathbf{x})\frac{\partial u}{\partial n}(\mathbf{x}) &= h(\mathbf{x}) \qquad \text{on } \Gamma_2,
\end{aligned}$$

where Γ_1 is the outer boundary, and Γ_2 is the inner (circular) boundary. The boundary conditions over Γ_2 are called Robin boundary conditions. In the case where $\beta(\mathbf{x})\equiv 0$, they are called Neumann boundary conditions, or natural boundary conditions. To obtain the weak form, multiply by a function $v(\mathbf{x})$ with $v=0$ on Γ_1 and integrate over Ω:

$$\int_\Omega v\,(-\nabla\cdot[\alpha(\mathbf{x})\nabla u]+\mathbf{w}(\mathbf{x})\cdot\nabla u)\,d\mathbf{x}=\int_\Omega v\,f\,d\mathbf{x}.$$

The left-hand side can be re-arranged using

$$\begin{aligned}
\int_\Omega v\nabla\cdot[\alpha\nabla u]\;d\mathbf{x} &= \int_\Omega\{\nabla\cdot(v\alpha\nabla u)-\alpha\nabla v\cdot\nabla u\}\;d\mathbf{x}\\
&= \int_{\partial\Omega} v\alpha\frac{\partial u}{\partial n}\,dS-\int_\Omega \alpha\nabla v\cdot\nabla u\,d\mathbf{x}\\
&= \int_{\Gamma_2} v\alpha\frac{\partial u}{\partial n}\,dS-\int_\Omega \alpha\nabla v\cdot\nabla u\,d\mathbf{x}
\end{aligned}$$

since $v=0$ on Γ_1 , $\overline{\Gamma_1}\cup\overline{\Gamma_2}=\partial\Omega$, and $\Gamma_1\cap\Gamma_2=\emptyset$. Now we can use the boundary conditions on Γ_2:

$$\int_{\Gamma_2} v\alpha\frac{\partial u}{\partial n}\,dS \;=\; \int_{\Gamma_2} v\,[h-\beta u]\;dS.$$

Combining, this gives the weak form

$$\int_\Omega[\alpha\nabla v\cdot\nabla u+v\,\mathbf{w}\cdot\nabla u]\,d\mathbf{x}+\int_{\Gamma_2}\beta u v\,dS=\int_\Omega f\,v\,d\mathbf{x}+\int_{\Gamma_2} h\,v\,dS,$$

for all v with $v = 0$ on Γ_1, and $u = g$ on Γ_1.

We need to define the pde structures for the region and boundary integrals.

15a ⟨*usage.m* 6c⟩+≡
```
f = @(x)10;
%g = @(x)(0.5*cos(x(1)));
g = @(x)0;
h = @(x)exp(x(2));
w = @(x)[1; -2];
alpha = @(x)1;
beta  = @(x)10;
pde  = struct('coeffs',@(x)[0,w(x)'; [0;0],alpha(x)*eye(2)], ...
         'rhs',@(x)[f(x);0;0], 'order',1)
pdeb = struct('coeffs',beta, 'rhs',h, 'order',0)
```

Note that $f(\mathbf{x})$, $\mathbf{w}(\mathbf{x})$, and $\alpha(\mathbf{x})$ are only evaluated once for each value of $\mathbf{x}$.

We also need to separate out the edges on Γ_1 and Γ_2. A simple way to separate them is that points $\mathbf{x}$ on Γ_1 have $\|\mathbf{x}\|_2 > 3/4$ while points $\mathbf{x}$ on Γ_2 have $\|\mathbf{x}\|_2 < 3/4$. Fortunately in this case, there are no edges that intersect both Γ_1 and Γ_2, although this may occur with other problems.

This task can be carried by starting with bnodes, which contains the list of boundary vertices as indexes into the p array.

15b ⟨*usage.m* 6c⟩+≡
```
in_gamma1_bnodes = (sqrt(p(bnodes,1).^2+p(bnodes,2).^2) > 3/4);
```

Note that in_gamma1_bnodes(i) is true (1) or false (0) depending on whether the point indexed by bnodes(i) is in Γ_1. We create a boolean (that is, zero–one) vector indicating whether a given point of the triangulation is in Γ_1:

15c ⟨*usage.m* 6c⟩+≡
```
in_gamma1 = zeros(size(p,1),1);
in_gamma1(bnodes) = in_gamma1_bnodes;
```

Then we can check the boundary edges to see which boundary edges are in Γ_1:

16a $\langle$*usage.m* 6c$\rangle$+≡
```
in_gamma1_bedges = in_gamma1(bedges(:,1)) & in_gamma1(bedges(:,2));
bedges1  = bedges (find( in_gamma1_bedges),:);
t_index1 = t_index(find( in_gamma1_bedges));
bedges2  = bedges (find(~in_gamma1_bedges),:);
t_index2 = t_index(find(~in_gamma1_bedges));
```

The boundaries can be shown by means of the `triplot()` function:

16b $\langle$*usage.m* 6c$\rangle$+≡
```
figure(4)
triplot(t,p(:,1),p(:,2),'k') % plot triangulation
% plot \Gamma_1 in blue and \Gamma_2 in red
triplot([bedges1(:,1),bedges1(:,2),bedges1(:,2)],p(:,1),p(:,2),'b')
triplot([bedges2(:,1),bedges2(:,2),bedges2(:,2)],p(:,1),p(:,2),'r')
```

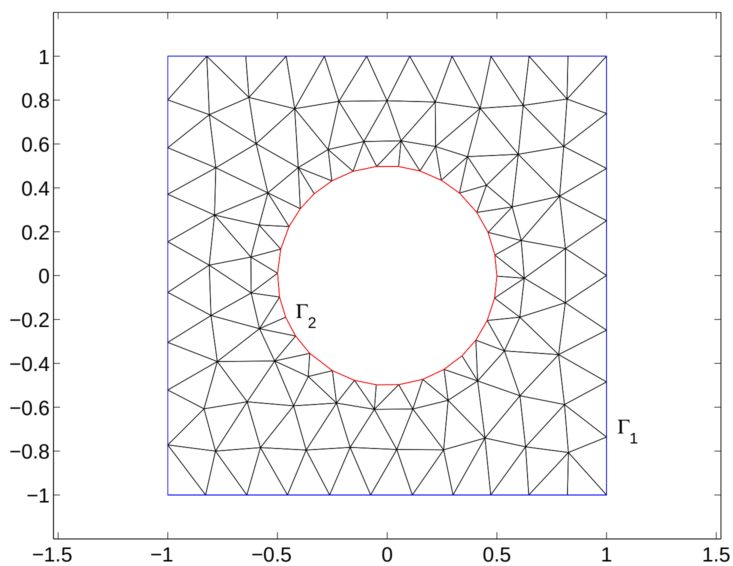

Figure 4: Boundaries Γ_1 and Γ_2

The result is shown in Figure 4.

Then to create the linear system to solve, we need to find all the variables associated with the Dirichlet boundary conditions; that is, we need to find all variables associated with Γ_1. The main matrix and right-hand side can be assembled independently.

18 ⟨*usage.m* 6c⟩+≡

```
intmethod = @int2d_radon7;
[A,b,bvlist2] = assembly2dbdry(pdeb,lin2d,p,t, ...
      bedges2,t_index2,fht,@int1d_gauss5);
[A,b] = assembly2d(A,b,pde,lin2d,p,t,fht,intmethod);
dir_bc_pde = struct('coeffs',@(x)[1],'rhs',@(x)g(x),'order',0)
Ab = sparse(nv,nv);
bb =  zeros(nv,1);
[Ab,bb,dir_bc_vlist] = ...
      assembly2dbdry(dir_bc_pde,lin2d,p,t, ...
      bedges1,t_index1,fht,@int1d_gauss5);
u1 = Ab(dir_bc_vlist,dir_bc_vlist) \ bb(dir_bc_vlist);
% Get complement to dir_bc_vlist
varray = zeros(nv,1);
varray(dir_bc_vlist) = 1;
cvlist = find(varray == 0);
% Now solve linear system
u2 = A(cvlist,cvlist) \ (b(cvlist) - A(cvlist,dir_bc_vlist)*u1);
u = zeros(nv,1);
u(dir_bc_vlist) = u1;
u(cvlist)       = u2;
figure(5)
trimesh(t,p(:,1),p(:,2),u(pvlist))
```

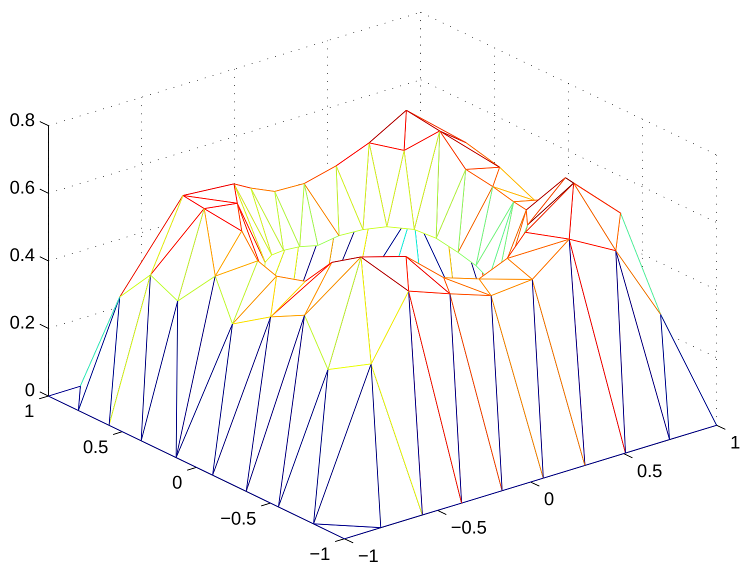

Figure 5: Result for convection–diffusion problem with Robin boundary conditions

The result is shown in Figure 5.

2 Basic assumptions

There are a number of assumptions about the triangulation and the element types (which includes their basis functions) that are necessary for this code to work. It is important to be aware of these issues when, for example, creating new element types, or using different mesh generators.

The triangulation is represented by the pair of arrays (`p`,`t`) where rows of `p` are the (x, y) co-ordinates of the vertices (points) of the triangulation, and each row of `t` contains the row indexes into `p` for the vertices of that triangle. Thus `p` is a floating point array with two columns and `t` is an integer array with three columns. For three-dimensions, `p` is a floating point array with three columns (for x, y and z coordinates) and `t` is an integer array with four columns representing the vertices of tetrahedra.

The triangulation is assumed to be *conforming*. That is, for any two triangles K_1 and K_2 in the triangulation, $K_1 \cap K_2$ is either empty, or a common *geometric feature* (vertex, edge, or triangle). Vertices are represented by a single row index into `p`, edges by a pair of row indexes into `p`, and triangles by three row indexes into `p`.

Each triangle K in the triangulation has a number of associated basis functions $\phi_i(\mathbf{x})$. These are basis functions are related to basis functions on the *reference element* $\widehat{K}$. For this code, the reference element is the triangle with vertices $(0, 0)$, $(1, 0)$, and $(0, 1)$. The basis functions on the reference element are fixed functions $\widehat{\phi}_j(\widehat{\mathbf{x}})$ for $\widehat{\mathbf{x}} \in \widehat{K}$. Each triangle K in the triangulation is the image of the reference element under an affine transformation $\widehat{K} \to K$ given by $\widehat{\mathbf{x}} \mapsto \mathbf{x} = T_K\widehat{\mathbf{x}} + \mathbf{b}_K$. Then each basis function ϕ_i that is nonzero on K is related to some basis function $\widehat{\phi}_j$ on $\widehat{K}$ through $\phi_i(\mathbf{x}) = \widehat{\phi}_j(\widehat{\mathbf{x}})$ where $\mathbf{x} = T_K\widehat{\mathbf{x}} + \mathbf{b}_K$. (This requirement is actually relaxed for some element types to requiring that $\phi_i(\mathbf{x}) = \sum_j c_{ij}\, \widehat{\phi}_j(\widehat{\mathbf{x}})$ for some linear combination of basis functions on the reference element.)

Every basis function (or variable) on the reference element is associated with a unique geometric feature of the reference element. If $\widehat{\mathbf{x}}$ belongs to a geometric feature and $\widehat{\phi}_j$ is *not* associated with that geometric feature or any of its subfeatures, then $\widehat{\phi}_j(\widehat{\mathbf{x}}) = 0$ if the basis functions are continuous across element boundaries. Piecewise constant elements are not continuous across element boundaries, and so do not have to satisfy this requirement.

Any permutation of the vertices of the reference triangle $\widehat{K}$ induces an affine transformation $\widehat{K} \to \widehat{K}$ given by $\widehat{\mathbf{x}} \mapsto \widehat{T}\widehat{\mathbf{x}} + \widehat{\mathbf{b}}$. For each basis function $\widehat{\phi}_j$ and permutation of the vertices of $\widehat{K}$, $\widehat{\mathbf{x}} \mapsto \widehat{\phi}_j(\widehat{T}\widehat{\mathbf{x}} + \widehat{\mathbf{b}})$ is a $\widehat{\phi}_k$ basis functions. That is, for some coefficients c_k,

$$\widehat{\phi}_j(\widehat{T}\widehat{\mathbf{x}} + \widehat{\mathbf{b}}) = \sum_k c_k\, \widehat{\phi}_k(\widehat{\mathbf{x}}) \qquad \text{for all } \widehat{\mathbf{x}} \in \widehat{K}.$$

Very often we can write $\widehat{\phi}_j(\widehat{T}\widehat{\mathbf{x}} + \widehat{\mathbf{b}}) = \widehat{\phi}_k(\widehat{\mathbf{x}})$ for all $\widehat{\mathbf{x}} \in \widehat{K}$ for some k. For example, consider the piecewise linear, quadratic and cubic Lagrange elements: using a nodal basis with nodes that are symmetrically placed with respect to permutations of the vertices means that $\widehat{\phi}_j(\widehat{T}\widehat{\mathbf{x}} + \widehat{\mathbf{b}}) = \widehat{\phi}_k(\widehat{\mathbf{x}})$ for some k.

3 Matrix assembly code

The main matrix assembly code `assembly2d()` is given below. The function adds values in the matrix A and the vector $\mathbf{b}$. In this way, the full assembly process can be accomplished "in pieces", if needed. So, for stand-alone use, A and $\mathbf{b}$ must be initialized to zero. Note that A can (and should) be a sparse matrix.

The PDE is represented by two functions which are in the `pde` structure (see Subsection 1.2).

The element type is defined by the `elt` structure (see Section 5).

The triangulation is given by the pair (`p`,`t`) as described in Section 1.

The hash table for the map from geometric features (triangles, edges, and vertices) to variables is `fht` (see Section 4).

The points and weights for the integration method on the reference element are returned by the function `intmethod()` (see Section 6). These points and weights are computed in *assembly2d-init*.

The line

```
[vlist,slist] = get_var_triangle(t(i,:),fht,elt,np);
```

gets the list of (global) variables indexes (`vlist`) associated with triangle i, along with the list of sign changes needed (`slist`).

3.1 Main two-dimensional assembly function

21 ⟨*filelist* 6b⟩+≡

```
assembly2d.m \
```

22 ⟨*assembly2d.m* 22⟩≡

```
function [A,b] = assembly2d(A,b,pde,elt,p,t,fht,intmethod)
% function [A,b] = assembly2d(A,b,pde,elt,p,t,fht,intmethod)
% Adds the assembled matrix and vector representing the
% given PDE (pde) to the A matrix & b vector.
% This uses a given element (elt) with the triangulation given by (p,t).
% The feature hash table (fht) is used to obtain variable indexes
% for given features. This is obtained by create_fht().
%
% A must be nv x nv and b must be nv x 1 where nv is the total
% number of variables (as returned by fht_num_vars()).
% Reference triangle has vertices (0,0), (1,0), (0,1).
⟨assembly2d-init 23a⟩
⟨assembly2d-precompute-Aphihat 23b⟩
for i = 1:size(t,1) % for all triangles ...
    % obtain variable list and signs for this triangle
    [vlist,slist] = get_var_triangle(t(i,:),fht,elt,np);
    % set up affine transformation xhat :-> x = T.xhat + b0
    i1 = t(i,1);  i2 = t(i,2);  i3 = t(i,3);
    T = [p(i2,:)'-p(i1,:)', p(i3,:)'-p(i1,:)'];
    b0 = p(i1,:)';
    % form weighted sum of integrand at integration points
    intval1 = 0;
    intval2 = 0;
    for k = 1:length(w_int)
        Aphival = elt.trans_Aphihat(T,Aphihatvals{k},order);
        Dmat    = pde.coeffs(T*p_int(k,:)'+b0);
        rhsvec  = pde.rhs(T*p_int(k,:)'+b0);
        integrand_val1 = Aphival*Dmat*Aphival';
        integrand_val2 = Aphival*rhsvec;
        intval1 = intval1 + w_int(k)*integrand_val1;
        intval2 = intval2 + w_int(k)*integrand_val2;
    end
    detT = abs(det(T));
    intval1 = intval1*detT; % scale by Jacobian
    intval2 = intval2*detT;
    intval1 = diag(slist)*intval1*diag(slist); % change signs if needed
    intval2 = slist'.*intval2;
    A(vlist,vlist) = A(vlist,vlist) + intval1; % add to matrix & vec
    b(vlist) = b(vlist) + intval2;
```

```
end
```

Initialization for *assembly2d*:

23a ⟨*assembly2d-init* 23a⟩≡

```
[p_int,w_int] = intmethod(); % points and weights for reference triangle
% np is the total number of points in the triangulation
np = size(p,1);
% compute nv = total number of variables
nv = fht_num_vars(fht);
% nv_elt is the number of variables in one element
nv_elt = sum(elt.nvars);
% order is the order of derivatives used in the assembly;
% we need 0 <= order <= 2
order = pde.order;
intval1 = zeros(nv_elt,nv_elt);
intval2 = zeros(nv_elt,1);
```

For efficiency, we precompute the values of $\mathcal{A}\widehat{\phi}_i(\widehat{\mathbf{x}}_j)$ where $\widehat{\mathbf{x}}_j$ are the integration points on the reference triangle. These depend only on the element type and the reference element $\widehat{K}$.

23b ⟨*assembly2d-precompute-Aphihat* 23b⟩≡

```
% Save get_Aphihat() values for all the integration points
% on the reference element
Aphihatvals = cell(length(w_int),1);
for k = 1:length(w_int)
    Aphihatvals{k} = elt.get_Aphihat(p_int(k,:),order);
end
```

3.2 Petrov–Galerkin method

The Petrov–Galerkin method is supported through the `pgassembly2d()` function. Since there are potentially two different elements used (`elt1` and `elt2`), we need to pass two separate feature hash tables (`fht1` and `fht2`). Otherwise the inputs are identical to those for the standard Galerkin assembly function `assembly2d()`. Note that `assembly2d()` is equivalent to

```
pgassembly2d(A,b,pde,p,t,elt,fht,elt,fht,intmethod)
```

23c ⟨*filelist* 6b⟩+≡

```
pgassembly2d.m \
```

24 ⟨*pgassembly2d.m* 24⟩≡

```
function [A,b] = pgassembly2d(A,b,pde,p,t,elt1,fht1,elt2,fht2,intmethod)
% function [A,b] = pgassembly2d(A,b,pde,p,t,elt1,fht1,elt2,fht2,intmethod)
%
% Petrov-Galerkin matrix assembly.
% Adds the assembled matrix and vector representing the
% given PDE (pde) to the A matrix & b vector.
% This uses a given elements (elt1, elt2) with the triangulation given by (p,t).
% The feature hash tables (fht1 for elt1, fht2 for elt2) are used to obtain
% variable indexes for given features. These are obtained by create_fht().
%
% elt1 represents the test functions, while elt2 represents the basis
% functions.
%
% The two elements can be quite independent, but the triangulation must be
% the same for the two sets of variables.
%
% A must be nv1 x nv2 and b must be nv1 x 1 where nv1 is the total
% number of variables for elt1 and nv2 is the total number of variables
% for elt2 (as returned by fht_num_vars()).
% Reference triangle has vertices (0,0), (1,0), (0,1).
⟨pgassembly2d-init 25⟩
⟨pgassembly2d-precompute-Aphilist 26a⟩
for i = 1:size(t,1) % for all triangles ...
    % obtain variable list and signs for this triangle
    [vlist1,slist1] = get_var_triangle(t(i,:),fht1,elt1,np);
    [vlist2,slist2] = get_var_triangle(t(i,:),fht2,elt2,np);
    % set up affine transformation xhat :-> x = T.xhat + b
    i1 = t(i,1); i2 = t(i,2); i3 = t(i,3);
    T = [p(i2,:)'-p(i1,:)', p(i3,:)'-p(i1,:)'];
    b0 = p(i1,:)';
    % form weighted sum of integrand at integration points
    intval1 = 0;
    intval2 = 0;
    for k = 1:length(w_int)
        Aphival1 = elt1.trans_Aphihat(T,Aphihatvals1{k},order);
        Aphival2 = elt2.trans_Aphihat(T,Aphihatvals2{k},order);
        Dmat   = pde.coeffs(T*p_int(k,:)'+b0);
        rhsvec = pde.rhs(   T*p_int(k,:)'+b0);
        integrand_val1 = Aphival1*Dmat*Aphival2';
```

```
        integrand_val2 = Aphival1*rhsvec;
        intval1 = intval1 + w_int(k)*integrand_val1;
        intval2 = intval2 + w_int(k)*integrand_val2;
    end
    detT = abs(det(T));
    intval1 = intval1*detT; % scale by Jacobian
    intval2 = intval2*detT;
    intval1 = diag(slist1)*intval1*diag(slist2); % change signs if needed
    intval2 = slist1'.*intval2;
    A(vlist1,vlist2) = A(vlist1,vlist2) + intval1; % add to matrix & vec
    b(vlist1)        = b(vlist1)        + intval2;
end % for i
```

The initialization code follows:

25 ⟨*pgassembly2d-init* 25⟩≡

```
[p_int,w_int] = intmethod(); % points and weights for reference triangle
% np is the total number of points in the triangulation
np = size(p,1);
% compute total numbers of variables
nv1 = fht_num_vars(fht1);
nv2 = fht_num_vars(fht2);
% nv_elt is the number of variables in one element
nv_elt1 = sum(elt1.nvars);
nv_elt2 = sum(elt2.nvars);
% order is the order of derivatives used in the assembly;
% we need 0 <= order <= 2
order = pde.order;
intval1 = zeros(nv_elt1,nv_elt2);
intval2 = zeros(nv_elt1,1);
```

For efficiency we pre-compute the `Aphihat` values at the integration points on the reference element.

26a ⟨*pgassembly2d-precompute-Aphilist* 26a⟩≡

```
% Save get_Aphihat() values for all the integration points
% on the reference element
Aphihatvals1 = cell(length(w_int),1);
Aphihatvals2 = cell(length(w_int),1);
for k = 1:length(w_int)
    Aphihatvals1{k} = elt1.get_Aphihat(p_int(k,:),order);
    Aphihatvals2{k} = elt2.get_Aphihat(p_int(k,:),order);
end
```

3.3 Re-factored two-dimensional assembly function

This is an attempt to re-factor and distill the essence of `assembly2d()`, so that it can be easily extended. The interface differs from `assembly2d()` in that the element type (`elt`) is now together with the feature hash table (`fht1`) in the argument list. These items really go together.

26b ⟨*filelist* 6b⟩+≡

```
assembly2d-rf.m \
```

27 ⟨*assembly2d-rf.m* 27⟩≡

```
function [A,b] = assembly2d_rf(A,b,pde,p,t,elt1,fht1,intmethod)
% function [A,b] = assembly2d_rf(A,b,pde,p,t,elt1,fht1,intmethod)
% Adds the assembled matrix and vector representing the
% given PDE (pde) to the A matrix & b vector.
% This uses a given element (elt1) with the triangulation given by (p,t).
% The feature hash table (fht1) is used to obtain variable indexes
% for given features. This is obtained by create_fht().
%
% A must be nv1 x nv1 and b must be nv1 x 1 where nv1 is the total
% number of variables (as returned by fht_num_vars(fht1)).
% Reference triangle has vertices (0,0), (1,0), (0,1).
⟨assembly2d-rf-init 28a⟩
⟨assembly2d-rf-init-update 28b⟩
⟨precompute-rf-Aphihat 28c⟩
for i = 1:size(t,1) % for all triangles ...
    ⟨assembly-get-variable-list 28d⟩
    ⟨assembly-set-affine-transformation 29a⟩

    % form weighted sum of integrand at integration points
    update_mat = 0;
    update_vec = 0;
    for k = 1:length(w_int)
        ⟨assembly-transform-Aphihat 29b⟩
        ⟨assembly-add-to-update-matrix 29c⟩
        ⟨assembly-add-to-update-vector 29d⟩
    end
    detT = abs(det(T));
    ⟨assembly-scale-and-update-matrix 30a⟩
    ⟨assembly-scale-and-update-vector 30b⟩
end
```

This initialization segment simply sets the integration points and weights, the variables np (number of points), nv1 (total number of variables), nv_elt1 (number of variables in an element), and the creates space for the small update matrices and vectors (update_mat and update_vec respectively).

28a ⟨*assembly2d-rf-init* 28a⟩≡

```
[p_int,w_int] = intmethod(); % points and weights for reference triangle
% np is the total number of points in the triangulation
np = size(p,1);
% order is the order of derivatives used in the assembly;
% we need 0 <= order <= 2
order = pde.order;
```

28b ⟨*assembly2d-rf-init-update* 28b⟩≡

```
% nv_elt1 is the number of variables in one element
nv_elt1 = sum(elt1.nvars);
update_mat = zeros(nv_elt1,nv_elt1);
update_vec = zeros(nv_elt1,1);
```

Pre-computing the $\mathcal{A}\widehat{\phi}(\widehat{\mathbf{x}})$ values for the integration points in the reference element saves repeating this computation on each iteration through the loops over the elements and the integration points.

28c ⟨*precompute-rf-Aphihat* 28c⟩≡

```
% Save get_Aphihat() values for all the integration points
% on the reference element
Aphihatvals = cell(length(w_int),1);
for k = 1:length(w_int)
    Aphihatvals{k} = elt1.get_Aphihat(p_int(k,:),order);
end
```

For each element we need to obtain the associated variable index list (vlist) and sign list (slist). The order of the indexes corresponds to the order of the basis functions as generated by elt1.get_Aphihat(). This must be done once for each element.

28d ⟨*assembly-get-variable-list* 28d⟩≡

```
% obtain variable list and signs for this triangle
[vlist,slist] = get_var_triangle(t(i,:),fht1,elt1,np);
```

This is where the matrix T_K and the vector $\mathbf{b}_K$ are set up for the triangle K with vertices `p(i1,:)`, `p(i2,:)`, `p(i3,:)`. This must be done once for each triangle (= element).

29a ⟨*assembly-set-affine-transformation* 29a⟩≡

```
% set up affine transformation xhat :-> x = T.xhat + b0
i1 = t(i,1);  i2 = t(i,2);  i3 = t(i,3);
T = [p(i2,:)'-p(i1,:)', p(i3,:)'-p(i1,:)'];
b0 = p(i1,:)';
```

The following code segments are executed once for each combination of triangle and integration point.

We must transform the array of $\mathcal{A}\widehat{\phi}_i(\widehat{\mathbf{x}})$ values to obtain the $\mathcal{A}\phi_i(\mathbf{x})$ values, for $\mathcal{A}$ ranging over the operators as defined by the element and the order required.

29b ⟨*assembly-transform-Aphihat* 29b⟩≡

```
Aphival = elt1.trans_Aphihat(T,Aphihatvals{k},order);
```

We compute

$$\sum_{\mathcal{A},\mathcal{B}} \int_K c_{\mathcal{A},\mathcal{B}}(\mathbf{x})\, \mathcal{A}\phi_i(\mathbf{x})\, \mathcal{B}\phi_j(\mathbf{x})\, d\mathbf{x} \approx \sum_p w_p \sum_{\mathcal{A},\mathcal{B}} c_{\mathcal{A},\mathcal{B}}(\widetilde{\mathbf{x}}_p)\, \mathcal{A}\phi_i(\widetilde{\mathbf{x}}_p)\, \mathcal{B}\phi_j(\widetilde{\mathbf{x}}_p)\ |\det T_K|$$

where $\widetilde{\mathbf{x}}_p = T_K\widehat{\mathbf{x}}_p + \mathbf{b}_K$ and $\widehat{\mathbf{x}}_p$ is the integration point in the reference element. The sum over the operators $\mathcal{A}$ and $\mathcal{B}$ is hidden in the matrix multiplications below. The multiplication by $|\det T_K|$ is performed after the loop over integration points. A similar operation is carried out for the right-hand side vector. So far, these operations are carried out on the small update matrices and vectors, which will then be added to main matrix and vector.

29c ⟨*assembly-add-to-update-matrix* 29c⟩≡

```
Dmat    = pde.coeffs(T*p_int(k,:)'+b0);
update_mat = update_mat + w_int(k)*(Aphival*Dmat*Aphival');
```

29d ⟨*assembly-add-to-update-vector* 29d⟩≡

```
rhsvec  = pde.rhs(T*p_int(k,:)'+b0);
update_vec = update_vec + w_int(k)*(Aphival*rhsvec);
```

After the loop over integration points, but before adding the update matrices and vectors to the main matrix and vector, we need to scale by $|\det T_K|$ and apply any sign changes required by the element type. We then use `vlist` to determine the place in the main matrix and vector to update.

30a ⟨*assembly-scale-and-update-matrix* 30a⟩≡
```
update_mat = update_mat*detT;                      % scale by Jacobian
update_mat = diag(slist)*update_mat*diag(slist); % change signs if needed
A(vlist,vlist) = A(vlist,vlist) + update_mat;    % add to matrix
```

30b ⟨*assembly-scale-and-update-vector* 30b⟩≡
```
update_vec = update_vec*detT;                      % scale by Jacobian
update_vec = slist'.*update_vec;                   % change signs if needed
b(vlist)   = b(vlist)          + update_vec;     % add to vector
```

3.4 Mesh-based functions and nonlinear problems

Mesh-based functions are needed for handling nonlinear PDEs. These functions have the form

$$g(\mathbf{x}) = \sum_{i=1}^{N} g_i\, \phi_i(\mathbf{x});$$

g itself may be a solution of a PDE, or a function of one (or more) such solutions. The assembly routine still produces a matrix and right-hand side for a linear system, but the linear system itself depends on a mesh-based function. This makes it easy to implement (for example), Newton's method for nonlinear PDEs. As with the Petrov–Galerkin assembly routine, it is important that g is defined using the same triangulation as we are going to use for the assembly of the linear system. However, the element type used does not need to be the same as for the remainder of the linear system. (This can also be used in a Petrov–Galerkin way, but this has not been implemented as yet.)

The differences in the code with `assembly2d()` can be easily identified: the `pde.coeffs()` and `pde.rhs()` functions have an extra input for the $\mathcal{A}g(\mathbf{x})$ values ($\mathcal{A}$ represents one of the operators I, $\partial/\partial x_1$, $\partial/\partial x_2$, etc.). Note that $g(\mathbf{x})$ can have vector values if desired.

30c ⟨*filelist* 6b⟩+≡
```
assembly2d-nl.m \
```

31 ⟨*assembly2d-nl.m* 31⟩≡

```
function [A,b] = assembly2d_nl(A,b,pde,elt,p,t,fht,intmethod,elt_nl,fht_nl,g_nl)
% function [A,b] = assembly2d_nl(A,b,pde,elt,p,t,fht,intmethod,elt_nl,fht_nl,g_nl)
% Adds the assembled matrix and vector representing the
% given PDE (pde) to the A matrix & b vector.
% This uses a given element (elt) with the triangulation given by (p,t).
% The feature hash table (fht) is used to obtain variable indexes
% for given features. This is obtained by create_fht().
%
% A must be nv x nv and b must be nv x 1 where nv is the total
% number of variables (as returned by fht_num_vars()).
%
% The last three inputs (elt_nl, fht_nl, g_nl) represent an additional
% function defined over the triangulation (this will typically be a
% solution of this or some other PDE over the same domain).
% elt_nl is the element type for the additional function
% fht_nl is the feature hashtable for elt_nl
% g_nl is the vector where variable index i for this element has value g_nl(i)
% The pde.coeffs() & pde.rhs() functions will now have the interfaces
% pde.coeffs(x,g_val)
% pde.rhs(x,g_val)
% where g_val is the (row) vector of A.g(x) values where A is one of the standard
% sets of operators (see elt.get_Aphihat()).
% Reference triangle has vertices (0,0), (1,0), (0,1).
[p_int,w_int] = intmethod(); % points and weights for reference triangle
% np is the total number of points in the triangulation
np = size(p,1);
% compute nv = total number of variables
nv = fht_num_vars(fht);
nv_nl = fht_num_vars(fht_nl);
% nv_elt is the number of variables in one element
nv_elt = sum(elt.nvars);
nv_nl_elt = sum(elt_nl.nvars);
% order is the order of derivatives used in the assembly;
% we need 0 <= order <= 2
order = pde.order;
intval1 = zeros(nv_elt,nv_elt);
intval2 = zeros(nv_elt,1);
% Save get_Aphihat() values for all the integration points
% on the reference element
```

```
Aphihatvals = cell(length(w_int),1);
Aphihatvals_nl = cell(length(w_int),1);
for k = 1:length(w_int)
    Aphihatvals{k}    = elt.get_Aphihat(p_int(k,:),order);
    Aphihatvals_nl{k} = elt.get_Aphihat(p_int(k,:),order);
end
for i = 1:size(t,1) % for all triangles ...
    % obtain variable list and signs for this triangle
    [vlist,   slist]    = get_var_triangle(t(i,:),fht,   elt,   np);
    [vlist_nl,slist_nl] = get_var_triangle(t(i,:),fht_nl,elt_nl,np);
    % set up affine transformation xhat :-> x = T.xhat + b0
    i1 = t(i,1); i2 = t(i,2); i3 = t(i,3);
    T = [p(i2,:)'-p(i1,:)', p(i3,:)'-p(i1,:)'];
    b0 = p(i1,:)';
    % form weighted sum of integrand at integration points
    intval1 = 0;
    intval2 = 0;
    % Compute element integral
    for k = 1:length(w_int)
        Aphival    = elt.trans_Aphihat(T,Aphihatvals{k},   order);
        Aphival_nl = elt.trans_Aphihat(T,Aphihatvals_nl{k},order);
        Dmat = pde.coeffs(T*p_int(k,:)'+b0, ...
            (g_nl(vlist_nl).*slist_nl')'*Aphival_nl);
        rhsvec = pde.rhs(T*p_int(k,:)'+b0, ...
            (g_nl(vlist_nl).*slist_nl')'*Aphival_nl);
        integrand_val1 = Aphival*Dmat*Aphival';
        integrand_val2 = Aphival*rhsvec;
        intval1 = intval1 + w_int(k)*integrand_val1;
        intval2 = intval2 + w_int(k)*integrand_val2;
    end
    detT = abs(det(T));
    intval1 = intval1*detT; % scale by Jacobian
    intval2 = intval2*detT;
    intval1 = diag(slist)*intval1*diag(slist); % change signs if needed
    intval2 = slist'.*intval2;
    A(vlist,vlist) = A(vlist,vlist) + intval1; % add to matrix & vec
    b(vlist)       = b(vlist) + intval2;
end % for
```

3.5 Boundary assembly

Assembling a matrix and vector using integration over the boundary, or part of it, can be useful for dealing with boundary conditions. The main difference with the other assembly routines is that we need to input the relevant boundary edges as a list of pairs of indexes into `p`, and to use `get_edge_vars()` to obtain the list of relevant variables. The integration routine `intmethod()` used must also be a one-dimensional integration method such as `int1d_gauss5()`. At the end there is some extra code to "trim" the matrix and vector assembled to be zero for variables not associated with the boundary.

There is an implicit assumption in this code that values on the boundary are not affected by variables not associated with a geometric feature of the boundary. For example, with piecewise linear elements, we need the value on an edge not to be affected by the value at the opposite vertex. This holds, as it should.

33 ⟨*filelist* 6b⟩+≡

```
assembly2dbdry.m \
```

34 ⟨*assembly2dbdry.m* 34⟩≡

```
function [A,b,bvlist] = assembly2dbdry(pde,elt,p,t,bedges,tidx,fht,intmethod)
% function [A,b,bvlist] = assembly2dbdry(pde,elt,p,t,bedges,tidx,fht,intmethod)
% Adds the assembled matrix and vector representing the
% given PDE (pde) to the A matrix & b vector.
% This uses a given element (elt) with the triangulation given by (p,t,bedges,tidx)
% for the boundary. Note that bedges(i,:) is in triangle t(tidx(i),:).
% The feature hash table (fht) is used to obtain variable indexes
% for given features. This is obtained by create_fht().
%
% A must be nv x nv and b must be nv x 1 where nv is the total
% number of variables (as returned by fht_num_vars()).
% Reference edge has vertices 0 and 1.
[p_int,w_int] = intmethod(); % points and weights for reference triangle
% np is the total number of points in the triangulation
np = size(p,1);
% compute nv = total number of variables
nv = fht_num_vars(fht);
% nv_edge is the number of variables in one edge (and associated points)
%nv_elt = sum(elt.nvars);
nv_edge = 0;
for i = 1:size(elt.flist,1)
    if sum(elt.flist(i,:) ~= 0) <= 2
        nv_edge = nv_edge + elt.nvars(i);
    end
end % for
% order is the order of differentiation used in the "PDE"
order = pde.order;
A = sparse(nv,nv);
b = zeros(nv,1);
bvlist = [];
for i = 1:size(bedges,1) % for all boundary edges ...
    % obtain variable list and signs for this triangle & boundary edge
    bedge = bedges(i,:);
    triangle = t(tidx(i),:);
    [tvlist,slist] = get_var_triangle(t(tidx(i),:),fht,elt,np);
    bvlist1 = get_var_edge(bedges(i,:),fht,np);
    bvlist  = [bvlist,bvlist1];
    match = match_edge_triangle(bedges(i,:),t(tidx(i),:));
    % set up affine transformation xhat :-> x = T.xhat + b0
```

```
    i1 = t(tidx(i),1); i2 = t(tidx(i),2); i3 = t(tidx(i),3);
    T = [p(i2,:)'-p(i1,:)', p(i3,:)'-p(i1,:)'];
    b0 = p(i1,:)';
    % Turn p_int on the interval [0,1] to points on the appropriate
    % edge of the reference triangle
    p_ref = [0 0; 1 0; 0 1];
    p_ref0 = p_ref(match(1),:);
    p_ref1 = p_ref(match(2),:);
    % form weighted sum of integrand at integration points
    intval1 = zeros(length(tvlist),length(tvlist));
    intval2 = zeros(length(tvlist),1);
    for k = 1:length(w_int)
        p_int_ref = (1-p_int(k))*p_ref0+p_int(k)*p_ref1;
        % p_int_val = T*p_int_ref'+b0;
        Aphivalhat = elt.get_Aphihat(p_int_ref,order);
        Aphival = elt.trans_Aphihat(T,Aphivalhat,order);
        Dmat = pde.coeffs(T*p_int_ref'+b0);
        rhsvec = pde.rhs(T*p_int_ref'+b0);
        integrand_val1 = Aphival*Dmat*Aphival';
        integrand_val2 = Aphival*rhsvec;
        intval1 = intval1 + w_int(k)*integrand_val1;
        intval2 = intval2 + w_int(k)*integrand_val2;
    end
    detT = norm(p(t(tidx(i),match(1)),:)-p(t(tidx(i),match(2)),:),2);
    intval1 = intval1*detT;
    intval2 = intval2*detT;
    intval1 = diag(slist)*intval1*diag(slist); % change signs if needed
    intval2 = slist'.*intval2;
    A(tvlist,tvlist) = A(tvlist,tvlist) + intval1; % add to matrix & vec
    b(tvlist)        = b(tvlist) + intval2;
end
bvlist = unique(sort(bvlist));
v_array = ones(nv,1);
v_array(bvlist) = 0;
cbvlist = find(v_array ~= 0);
Ab(cbvlist,:) = 0;
Ab(:,cbvlist) = 0;
b(cbvlist) = 0;
```

4 Handling geometric features

Geometric features are triangles, edges, and points (vertices) of the triangulation. Each variable is associated with a single geometric feature: if several seem possible, then we choose the one of lowest dimension. For example, if we use a piecewise linear finite element space over a given triangulation, then each variable is associated with a vertex of a triangle in the triangulation. Then each vertex has one variable whose value is the same for all the triangles sharing that vertex. This ensures that at any edge shared between two triangles, the value of a piecewise linear function is the same at the ends of the edge and so is the same along the entire edge. This ensures continuity of the piecewise linear function.

A function in the piecewise linear finite element space will have the form

$$v_h(\mathbf{x}) = \sum_{i=1}^{N} v_i \, \phi_i(\mathbf{x})$$

where v_i are the values associated with the vertices of the triangulation; for each triangle K in the triangulation, $\phi_i|K$ is a linear (actually, affine) function. When we assemble the part of a matrix for triangle K, we need to ensure that the same v_i is used. To do this, we have a list of all the variables (in order) associated with a given vertex.

Similarly, for a quadratic Lagrange basis, we typically use a nodal basis using values at the vertices of a triangle, and the values at the midpoints of the edges of the triangle. A function in the finite element space generated by these nodal basis functions must have the same values at every point on an edge shared between two elements. It is sufficient if the values at the shared vertices and shared edge's midpoint are equal: two quadratic functions of one variable that are equal at three points must be the same. The basis functions ϕ_i have an associated value or variable v_i for representing a function in the finite element space

$$v_h(\mathbf{x}) = \sum_{i=1}^{N} v_i \, \phi_i(\mathbf{x}).$$

If ϕ_i is a nodal basis function for a vertex then v_i is associated with that vertex; if it is associated with a midpoint of an edge, it is associated with that edge.

For a cubic Lagrange basis, we use a nodal basis using values at the vertices, values at points along each edge at the 1/3 and 2/3 positions, and one at the centroid of the triangle. This time there is one variables associated with each vertex, one with each triangle, but two with each edge. It is important to distinguish between the two variables associated with a given edge, because they correspond to different basis functions.

4.1 Feature hash tables

Each geometric feature then has an associated ordered list of variables. To store these we use a hash table. Matlab's `container.Map`, however, allows only string or integer keys, so we need to convert a feature (given as a list of indexes into the `p` array) into an integer. This is done as follows:

37a ⟨*filelist* 6b⟩+≡

```
get-feature-ref.m \
```

37b ⟨*get-feature-ref.m* 37b⟩≡

```
function ref = get_feature_ref(f,np)
% function ref = get_feature_ref(f,np)
%
% Return a unique integer for the given feature, for
% use in the feature hashtable.
% np is the number of points in the triangulation.
ref = sum(int64(f) .* int64(np).^int64(0:length(f)-1));
end
```

From the triangulation and the element type we can create the entire hash table (see *create-fht.m*). Note that we need certain information from the element type (`elt`): `elt.flist` is a list of geometric features to which variables are associated for the reference element. Note that these are lists of integers in the set $\{1, 2, 3\}$; these integers refer to the vertices of the reference element: vertex 1 is $(0, 0)$, vertex 2 is $(1, 0)$, and vertex 3 is $(0, 1)$. The number of variables associated with the feature `elt.flist(i,:)` is `elt.nvars(i)`. For more details, see Section 5. Variables are numbered sequentially as they are discovered. Note that if a geometric feature has already been found, then it is skipped.

It should be noted that all geometric features entered into `fht` must be *normalized*; that is, they must be an increasing list of indexes into the `p` array. Zero padding at the end should be stripped.

37c ⟨*filelist* 6b⟩+≡

```
create-fht.m \
```

38 ⟨*create-fht.m* 38⟩≡

```
function [fht,v2tnum,v2fnum,v2fidx] = create_fht(p,t,elt)
% function [fht,v2tnum,v2fnum,v2fidx] = create_fht(p,t,elt)
%
% Create feature hash table (fht) which shows what variables
% are associated with which geometric features.
% The geometric features inserted into fht must be
% in normalized form (that is, a sorted vector of point indexes).
%
% This routine also returns a variable-to-triangle-number array (v2tnum)
% a variable-to-feature-number array (v2fnum), and a variable-to-feature-index
% array (v2fidx).
% The variable (or basis function) with global index k is the v2fidx(k)'th
% basis function associated with the v2fnum(k)'th geometric feature of
% triangle v2tnum(k).
fht = containers.Map('KeyType','int64','ValueType','any');
nvars = elt.nvars;
flist = elt.flist; % list of features with associated variables
v2tnum = [];
v2fnum = [];
v2fidx = [];
% flist is assumed normalized except for trailing zeros
np = size(p,1);
counter = 0;
for i = 1:size(t,1) % for each triangle ...
    triangle = [0, t(i,:)];
    tflist = triangle(flist+1);
    for j = 1:size(flist,1)
        % for each feature ...
        f = tflist(j,:);
        f = f(find(f ~= 0));
        f = sort(f);
        % Is this feature already in fht? If not add its variables.
        ref = get_feature_ref(f,np);
        if ~ isKey(fht,ref)
            fht(ref) = [(counter+1):(counter + nvars(j))];
            counter = counter + nvars(j);
            v2tnum = [v2tnum, i*ones(1,nvars(j))];
            v2fnum = [v2fnum, j*ones(1,nvars(j))];
            v2fidx = [v2fidx, 1:nvars(j)];
```

```
        end
    end % for each feature
    size_fht = size(fht);
end % for each triangle
end % function create_fht
```

Using the feature hash table (`fht`) can be done through a number of functions; the main one (used by the assembly functions) is `get_var_triangle()`. This finds all variables associated with any geometric feature (triangle, edge, vertex) of the triangle. This triangle is represented as a triple of indexes into the `p` array.

39 ⟨*filelist* 6b⟩+≡

```
get-var-triangle.m \
```

40a ⟨*get-var-triangle.m* 40a⟩≡

```
function [vlist,slist] = get_var_triangle(tri,fht,elt,np)
% function [vlist,slist] = get_var_triangle(tri,fht,elt,np)
%
% Get the list of variables (vlist) and the list of sign changes (slist)
% for a given triangle tri using the feature ahstable (fht) for
% the given element type (see elt data structure).
% Note that np is the number of points.
%
% tri is a 1 x 3 array of indexes into the p array of points in the
% triangulation
tri2 = [0,tri];
flist = elt.flist;
flist = tri2(flist+1); % use point indexes
vlist = [];
slist = [];
for i = 1:size(flist,1) % for each feature
f = flist(i,:);
f = f(find(f ~= 0)); % strip zeros from f
[fn,px] = sort(f); % normalize f: fn(px) == f
ref = get_feature_ref(fn,np);
if ~ isKey(fht,ref)
    error('flexPDE:missing value','get_var_triangle: Missing feature',fn,ref)
return
else
    fvlist = fht(ref);
    [pxvars,fslist] = elt.pxfeature(px);
    fvlist = fvlist(pxvars);
end
% concatenate the list of variable indexes & signs
vlist = [vlist,fvlist];
slist = [slist,fslist];
end
```

For performing boundary integrals we use the corresponding function for edges:

40b ⟨*filelist* 6b⟩+≡

```
get-var-edge.m \
```

41a ⟨*get-var-edge.m* 41a⟩≡

```
function [vlist] = get_var_edge(edge,fht,np)
% function [vlist] = get_var_edge(edge,fht,np)
%
% Returns list of variable indexes for given edge (including end-point
% variables). The feature hash table (fht) is used to look up variable
% lists. Also np is the number of points in the triangulation.
vlist = [];
ref = get_feature_ref(sort(edge),np);
if isKey(fht,ref)
    vlist = [vlist, fht(ref)];
end
ref = get_feature_ref(edge(1),np);
if isKey(fht,ref)
    vlist = [vlist, fht(ref)];
end
ref = get_feature_ref(edge(2),np);
if isKey(fht,ref)
    vlist = [vlist, fht(ref)];
end
```

The total number of variables can be found using the following routine, which simply adds the lengths of all the lists of variables in `fht`. Note that this assumes that every variable is associated with exactly one geometric feature.

41b ⟨*filelist* 6b⟩+≡

```
fht-num-vars.m \
```

42a ⟨*fht-num-vars.m* 42a⟩≡

```
function nvars = fht_num_vars(fht)
% function nvars = fht_num_vars(fht)
%
% Returns the total number of variables for a discretization
% based on the feature hash table (fht), which stores
% variable index lists for each geometric feature.
nvars = 0;
vals = values(fht);
for i = 1:length(vals)
    nvars = nvars + length(vals{i});
end
end
```

4.2 Geometric utilities

4.2.1 Find boundary

Finding boundary edges can be done directly from the `t` array: a boundary edge is an edge of exactly one triangle. The basic code is here:

42b ⟨*filelist* 6b⟩+≡

```
boundary2d.m \
```

43a ⟨*boundary2d.m* 43a⟩≡

```
function [bedges,bnodes,t_index] = boundary2d(t)
% function [bedges,bnodes,t_index] = boundary2d(t)
%
% Construct boundary edge list from triangle list t
% t is ntriangles x 3, bedges = nedges x 2
% Edge k joins points bd(k,1) and bd(k,2).
%
% Simply check when edges only appear once in the triangle list.
%
% Also returns the triangle index for each boundary edge
t = sort(t,2); % sort each row of t
bd1 = sortrows([t(:,1),t(:,2),(1:size(t,1))';
                t(:,2),t(:,3),(1:size(t,1))';
                t(:,1),t(:,3),(1:size(t,1))']);
[bd2,idx1] = unique(bd1(:,1:2),'rows','first');
[bd2,idx2] = unique(bd1(:,1:2),'rows','last');
eqlist = find(idx1 == idx2);
bedges = bd1(idx1(eqlist),1:2);
t_index = bd1(idx1(eqlist),3);
bnodes = unique(sort(bedges(:)));
```

Note that this routine also returns `bnodes`, a list of all vertices in the boundary, and `t_index` where `t_index(i)` is the row index into `t` for the edge `bedges(i,:)`.

4.2.2 Matching edges to triangles

This returns a two-integer vector matching a single given edge to a single given triangle. It is assumed that the edge is an edge of the triangle. All objects given as lists of indexes into `p`.

43b ⟨*filelist* 6b⟩+≡

```
match-edge-triangle.m \
```

44a ⟨*match-edge-triangle.m* 44a⟩≡

```
function match = match_edge_triangle(edge,triangle)
% function match = match_edge_triangle(edge,triangle)
%
% Returns index vector (2 elements) match so that
% edge(i) = triangle(match(i)), i = 1, 2.
for i = 1:2
    for j = 1:3
        if edge(i) == triangle(j)
            match(i) = j;
        end
    end % for j
end % for i
end % function
```

4.2.3 Generating transformation for an element

Given the points of a triangle as rows of a 3×2 array, compute the matrix T and vector $\mathbf{b}$ so that $\widehat{\mathbf{x}} \mapsto T\widehat{\mathbf{x}} + \mathbf{b}$ maps the reference triangle (vertices at $(0,0)$, $(1,0)$ and $(0,1)$) to the triangle given. Note that the ordering of the vertices is respected.

44b ⟨*filelist* 6b⟩+≡

```
gen-transform2d.m \
```

44c ⟨*gen-transform2d.m* 44c⟩≡

```
function [T,b] = gen_transform2d(p)
% function [T,b] = gen_transform2d(p)
%
% Returns T, b so that x \mapsto T*x+b maps
% the reference triangle co{(0,0),(1,0),(0,1)} to
% the given triangle, mapping (0,0) to p(1,:)',
% (1,0) to p(2,:)' and (0,1) to p(3,:)'
b = p(1,:)';
T = [p(2,:)' - p(1,:)', p(3,:)' - p(1,:)'];
```

5 Element types

Each element type is represented by a corresponding data structure. An element type must provide the basis functions $\widehat{\phi}_i$ on the reference element and the various functions $\mathcal{A}\widehat{\phi}_i$ for $\mathcal{A} = I$, $\partial/\partial x_1$, $\partial/\partial x_2$ etc., and it must also provide the information as to which basis functions $\widehat{\phi}_i$ are associated with which geometric feature. The reference element has the basis functions ordered in a specific way. For a real element K, there must be an identification of the geometric features (triangle, edges, vertices) of K with the corresponding geometric features of the reference element $\widehat{K}$.

For a description of the different components of the element type data structure, see Section 5.1.

Each element type structure contains the following fields: `get_Aphihat()`, `nvars`, `flist`, `pxfeature()`, `vnodes` and `trans_Aphihat()`. The fields with "`()`" are functions. The call `Aphihatvals = get_Aphihat(xhat,order)` computes the values of $\mathcal{A}\widehat{\phi}_i(\widehat{\mathbf{x}})$ for all reference element basis functions $\widehat{\phi}_i$, and various operators $\mathcal{A}$ (including the identity operator). Specifically, `Aphihatvals(i,j)` is$\mathcal{A}\widehat{\phi}_i(\widehat{\mathbf{x}})$ where $\mathcal{A}$ is the jth operator in the list of applicable operators ($\mathcal{A}$ is the identity if $j = 1$). The value `nvars(k)` is the number of variables (or basis functions) associated with geometric feature `flist(k,:)` of the reference element. The call `pxvars = pxfeature(px)` returns the permutation of the variables (or basis functions) for a geometric feature of dimension `length(px)-1`. The point `vnodes(i,:)` is the coordinate vector for the ith basis function's node. (Basis functions do not necessarily need to be nodal, but this gives a useful point associated with a given basis function.) The call `Aphivals = trans_Aphihat(T,Aphihatvals,order)` computes the values $\mathcal{A}\phi_k(\mathbf{x})$ where ϕ_k is the actual basis function.

5.1 Piecewise linear elements

First we consider piecewise linear elements. The main function called is `lin2d_elt()`. It creates a structure which contains the other routines needed to properly define the element type. The component `flist` lists the geometric features of the reference element to which variables (or basis functions) are associated. A geometric feature is here described by a list or row of non-zero vertex indices for the reference element. The number of variables associated with geometric feature `flist(i)` is `nvars(i)`. Thus the row `[1, 0, 0]` of `flist` shows that that variables are associated with vertex 1 of the reference element. Indeed, for the piecewise linear element here, all variables are associated with vertices.

On the other hand, for the piecewise cubic element, one row of `flist` is `[1, 2, 0]`, indicating that the edge joining vertices one and two of the reference element has variables associated with it. The corresponding entry of `nvars` is 2, showing that there are exactly two variables associated with this edge. Another row of `flist` is `[1, 2, 3]`, and the corresponding entry of `nvars` is 1, indicating that exactly one variable is associated with the triangle (and not any sub-feature).

Note that each variables is associated with *exactly one* geometric feature; for a nodal basis, a variable or basis function is associated with the geometric feature (triangle, edge, vertex) of lowest dimension that contains the nodal point.

The order of the geometric features in `flist` must correspond to the order in which the basis functions occur in the `elt.get_Aphihat()` function.

46a ⟨*filelist* 6b⟩+≡

```
lin2d-elt.m \
```

46b ⟨*lin2d-elt.m* 46b⟩≡

```
function elt = lin2d_elt()
% function elt = lin2d_elt()
%
% Returns the linear 2-D (3-point) element data structure.
nvars = [1;1;1];
flist = [1 0 0;
         2 0 0;
         3 0 0]; % the three vertices
vnodes = [0 0;
          1 0;
          0 1];
elt = struct('get_Aphihat',@lin2d_get_Aphihat, ...
    'nvars',nvars,'flist',flist, ...
    'pxfeature',@lin2d_pxfeature,'vnodes',vnodes, ...
    'trans_Aphihat',@trans2d_Aphilist);
end
```

5.1.1 Piecewise linear elements: get_Aphihat()

The first component of the structure is `get_Aphihat`, which is set to be the function `lin2d_get_Aphihat()`. This computes $\mathcal{A}\widehat{\phi}_i(\widehat{\mathbf{x}})$ for $i = 1, 2, \ldots, M$, where M is the number of basis functions for the reference element:

47 ⟨*lin2d-elt.m* 46b⟩+≡

```
function Aphilist = lin2d_get_Aphihat(xhat,order)
% Aphilist = lin2d_get_Aphihat(xhat,order)
%
% Returns array of basis function values, their gradient and Hessian entries
% for linear (affine) basis functions on a 2-D reference triangle at xhat.
% The vertices of the reference triangle are (0,0), (1,0), and (0,1).
% Aphilist(i,j) is the value of the j'th operator on phi_i at xhat.
% Here phi_i is the affine function where phi_i(xhat_j) == 1
% if i == j, and zero otherwise; xhat_i is the i'th vertex listed above.
%
% Order of operators: Aphi(xhat) = phi(xhat), d/dx1 phi(xhat),
% d/dx2 phi(xhat), d^2/dx1^2 phi(xhat), d^2/dx1.dx2 phi(xhat),
% d^2/dx2^2 phi(xhat).  Note that x1 = x and x2 = y.
x = xhat(1);  y = xhat(2);
% Basis function values
Aphilist0 = [1-x-y;
             x;
             y];
if order >= 1
    % Basis gradient values (along rows)
    Aphilist1 = [-1 -1;
                  1  0;
                  0  1];
end
if order >= 2
    % Basis hessian values (along rows: dx1^2, dx1.dx2, dx2^2)
    Aphilist2 = [0 0 0;
                 0 0 0;
                 0 0 0];
end
if order == 0
    Aphilist = Aphilist0;
elseif order == 1
```

```
    Aphilist = [Aphilist0,Aphilist1];
elseif order == 2
    Aphilist = [Aphilist0,Aphilist1,Aphilist2];
end % if
end % function
```

This is the main workhorse of the element type; it is the function that is called the most of all of the functions in the structure.

5.1.2 Piecewise linear elements: pxfeature()

In addition, when a geometric feature is found as part of some element, the orientation of the element may result in certain variables being permuted. In the case of piecewise linear elements, the geometric features are vertices, which do not have an orientation. Consequently this code is essentially trivial. This component of the element type structure becomes important for certain other more complex elements.

48 ⟨*lin2d-elt.m* 46b⟩+≡

```
function [px_vars,signs] = lin2d_pxfeature(px)
% function [px_vars,signs] = lin2d_pxfeature(px)
%
% Returns the permutation of the variables (px_vars),
% and the sign changes (signs) resulting from a permutation (px)
% applied to a feature of the appropriate dimension (== length(px)).
% This is for the linear (or affine) 2-D triangle elements.
dimp1 = sum(px ~= 0);  % dimp1 == dimension plus 1
switch dimp1
    case 1 % points
        px_vars = [1]; signs = [1];
    otherwise % not a valid feature
        px_vars = []; signs = [];
end % switch
end % function
```

5.2 Transformation routines: scalar Lagrange elements

The last component is the transformation routine to transform the basis functions and their derivatives from the reference triangle to a given real triangle using the map $\widehat{\mathbf{x}} \mapsto \mathbf{x} = T_K\widehat{\mathbf{x}} + \mathbf{b}_K$:

The basis function on the reference element $\widehat{K}$ is $\widehat{\phi}$, while the basis function on the real element K is ϕ where $\phi(\mathbf{x}) = \widehat{\phi}(\widehat{\mathbf{x}})$ provided $\mathbf{x} = T_K\widehat{\mathbf{x}} + \mathbf{b}_K$. The chain rule then says that

$$\begin{aligned} \frac{\partial\phi}{\partial x_i}(\mathbf{x}) &= \frac{\partial}{\partial x_i}\widehat{\phi}(\widehat{\mathbf{x}}) \\ &= \sum_p \frac{\partial \widehat{x}_p}{\partial x_i}\,\frac{\partial\widehat{\phi}}{\partial\widehat{x}_p}(\widehat{\mathbf{x}}). \end{aligned}$$

Now $\widehat{\mathbf{x}} = T_K^{-1}(\mathbf{x} - \mathbf{b}_K)$, so if we write $S_K = T_K^{-1}$, then

$$\frac{\partial \widehat{x}_p}{\partial x_i} = s_{pi} = (S_K)_{pi}.$$

In terms of the gradient vector,

$$\nabla\phi(\mathbf{x}) = S_K^T\,\nabla\widehat{\phi}(\widehat{\mathbf{x}}).$$

Since the gradient vectors below are given as *row* vectors, we do not need the transpose.

For second derivatives,

$$\begin{aligned} \frac{\partial^2\phi}{\partial x_i\,\partial x_j}(\mathbf{x}) &= \frac{\partial}{\partial x_i}\sum_p s_{pj}\,\frac{\partial\widehat{\phi}}{\partial\widehat{x}_p}(\widehat{\mathbf{x}}) \\ &= \sum_q s_{qi}\frac{\partial}{\partial\widehat{x}_q}\sum_p s_{pj}\,\frac{\partial\widehat{\phi}}{\partial\widehat{x}_p}(\widehat{\mathbf{x}}) \\ &= \sum_{p,q} s_{qi}s_{pj}\frac{\partial^2\widehat{\phi}}{\partial\widehat{x}_q\,\partial\widehat{x}_p}(\widehat{\mathbf{x}}) \;=\; \sum_{p,q} s_{qi}\frac{\partial^2\widehat{\phi}}{\partial\widehat{x}_q\,\partial\widehat{x}_p}(\widehat{\mathbf{x}})s_{pj}. \end{aligned}$$

If Hess $\psi$is the usual Hessian matrix of 2nd derivatives, this can be written in the form

$$\operatorname{Hess}\phi(\mathbf{x}) = S^T\operatorname{Hess}\widehat{\phi}(\widehat{\mathbf{x}})\,S.$$

If $H = \begin{bmatrix} h_{11} & h_{12} \\ h_{12} & h_{22} \end{bmatrix}$ (symmetric) and $S = \begin{bmatrix} s_{11} & s_{12} \\ s_{21} & s_{22} \end{bmatrix}$ (which is usually not symmetric), we can easily verify that

$$S^T H S = \begin{bmatrix} (h_{11}s_{11}^2 + 2h_{12}s_{11}s_{21} + h_{22}s_{21}^2) & (h_{11}s_{11}s_{12} + h_{12}(s_{11}s_{22} + s_{12}s_{21}) + h_{22}s_{21}s_{22}) \\ \text{(symmetric)} & (h_{11}s_{12}^2 + 2h_{12}s_{12}s_{22} + h_{22}s_{22}^2) \end{bmatrix}.$$

50a ⟨*filelist* 6b⟩+≡

```
trans2d-Aphilist.m \
```

50b ⟨*trans2d-Aphilist.m* 50b⟩≡

```
function Aphilist2 = trans2d_Aphilist(T,Aphilist,order)
% function Aphilist2 = trans2d_Aphilist(T,Aphilist,order)
%
% Transforms Aphilist into Aphilist2 according to matrix T (2 x 2)
% Note: Order of colums is [val, d/dx1, d/dx2, d^2/dx1^2, d^2/dx1.dx2, d^2/dx2^2]
Aphilist2 = zeros(size(Aphilist));
Aphilist2(:,1) = Aphilist(:,1); % values unchanged
if order >= 1
    S = inv(T);
    Aphilist2(:,2:3) = Aphilist(:,2:3)*S; % chain rule for 1st derivatives
end % if
if order >= 2
    % chain rule for 2nd derivatives (affine transformation)
    Aphilist2(:,4) = Aphilist(:,4)*(S(1,1)^2)+ ...
           Aphilist(:,5)*(2*S(2,1)*S(1,1))+ ...
           Aphilist(:,6)*(S(2,1)^2);
    Aphilist2(:,5) = Aphilist(:,4)*(S(1,1)*S(1,2))+ ...
           Aphilist(:,5)*(S(1,1)*S(2,2)+S(1,2)*S(2,1))+ ...
           Aphilist(:,6)*(S(2,2)*S(2,1));
    Aphilist2(:,6) = Aphilist(:,4)*(S(1,2)^2)+ ...
           Aphilist(:,5)*(2*S(1,2)*S(2,2))+ ...
           Aphilist(:,6)*(S(2,2)^2);
end % if
end % function
```

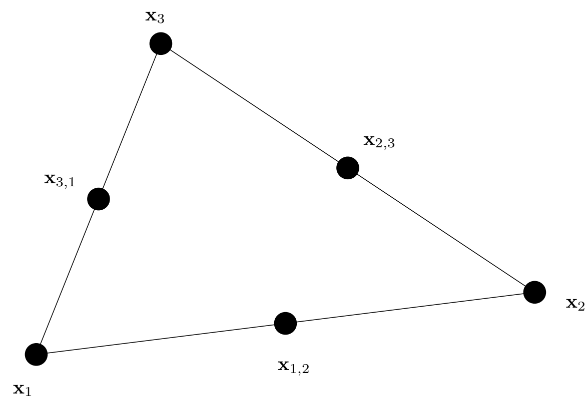

Figure 6: Quadratic Lagrange element

The `Aphilist_trans2d()` function is common to all scalar Lagrange elements. However, for vector-valued elements or for certain C^1 elements (such as Bell's triangle or the Argyris element), this needs to be modified.

5.3 Piecewise quadratic elements

Piecewise quadratic elements are based on nodal interpolation at the vertices and the midpoints of the edges, as illustrated in Figure 6. It is important to use the midpoint as when two elements share a common edge, it is important that the nodal points are the same for both elements.

The function that creates the element type data structure for piecewise quadratic elements is very similar to that for piecewise linear elements.

51 ⟨*filelist* 6b⟩+≡
```
  quad2d-elt.m \
```

52 ⟨*quad2d-elt.m* 52⟩≡

```
function elt = quad2d_elt()
% function elt = quad2d_elt()
%
% Returns the quadratic 2-D (6-point) element data structure
nvars = [1;1;1;1;1;1];
flist = [1 0 0;
         2 0 0;
         3 0 0;
         1 2 0;
         1 3 0;
         2 3 0];
vnodes = [0   0;
          1   0;
          0   1;
          1/2 0;
          0   1/2;
          1/2 1/2];
elt = struct('get_Aphihat',@quad2d_get_Aphihat, ...
     'nvars',nvars,'flist',flist, ...
     'pxfeature',@quad2d_pxfeature,'vnodes',vnodes, ...
     'trans_Aphihat',@trans2d_Aphilist);
end % function
```

5.3.1 Piecewise quadratic elements: get_Aphihat()

This is the main workhorse of the quadratic element data structure:

53 ⟨*quad2d-elt.m* 52⟩+≡

```
function Aphilist = quad2d_get_Aphihat(xhat,order)
% function Aphilist = quad2d_get_Aphihat(xhat,order)
%
% Returns array of basis functions, their gradient and Hessian entries
% for quadratic basis functions on a 2-D reference triangle at xhat.
% The vertices of the reference triangle are (0,0), (1,0), and (0,1).
% Aphilist(i,j) is the value of the j'th operator on phi_i at xhat.
% Here phi_i is the affine function where phi_i(xhat_j) == 1
% if i == j, and zero otherwise; xhat_i is the i'th vertex listed above.
%
% order is the maximum order of derivatives considered (order <= 2)
%
% Order of operators: Aphi(xhat) = phi(xhat), d/dx1 phi(xhat),
% d/dx2 phi(xhat), d^2/dx1^2 phi(xhat), d^2/dx1.dx2 phi(xhat),
% d^2/dx2^2 phi(xhat).
x = xhat(1);  y = xhat(2);
% basis function values
Aphilist = [2*(1-x-y)*(0.5-x-y);
            2*x*(x-0.5);
            2*y*(y-0.5);
            4*x*(1-x-y);
            4*y*(1-x-y);
            4*x*y];
if order >= 1
    % gradients (rows) of basis functions
    Aphilist1 = [2*(2*(x+y)-1.5), 2*(2*(x+y)-1.5);
                 4*x-1,           0;
                 0,               4*y-1;
                 4*(1-y)-8*x,     -4*x;
                 -4*y,            4*(1-x)-8*y;
                 4*y,             4*x];
    Aphilist = [Aphilist, Aphilist1];
end
if order >= 2
    % Hessian matrix entries of basis functions: dx1^2, dx1.dx2, dx2^2
```

```
    Aphilist2 = [4,  4,  4;
                 4,  0,  0;
                 0,  0,  4;
                -8, -4,  0;
                 0, -4, -8;
                 0,  4,  0];
    Aphilist = [Aphilist, Aphilist2];
end % if
end % function
```

5.3.2 Piecewise quadratic elements: pxfeature()

Again this code is essentially trivial, even though we now have variables associated with edges rather than only with vertices as in the piecewise linear case.

54 ⟨*quad2d-elt.m* 52⟩+≡

```
function [px_vars,signs] = quad2d_pxfeature(px)
% function [px_vars,signs] = quad2d_pxfeature(px)
%
% Returns the permutation of the variables (px_vars),
% and the sign changes (signs) resulting from a permutation (px)
% applied to a feature of the appropriate dimension (== length(px)).
% This is for the quadratic 2-D triangle elements.
dim = sum(px ~= 0)-1;
switch dim
    case 0 % points
        px_vars = [1]; signs = [1];
    case 1 % edges
        px_vars = [1]; signs = [1];
    otherwise % not a valid feature
        px_vars = []; signs = [];
end % switch
end % function
```

5.4 Piecewise cubic elements

These elements have a nodal basis with nodes at the vertices, at the 1/3 and 2/3 points on each edge, and the centroid of the triangle. Because they have two node points on each edge, there may need to be some permutations to ensure correct references to variables (see `pxfeature()` below).

55a ⟨*filelist* 6b⟩+≡

```
cub2d-elt.m \
```

55b ⟨*cub2d-elt.m* 55b⟩≡

```
function elt = cub2d_elt()
% function elt = cub2d_elt()
%
% Returns cubic 2-D (10-point) element data structure
nvars = [1;1;1;2;2;2;1];
flist = [1 0 0;
         2 0 0;
         3 0 0;
         1 2 0;
         1 3 0;
         2 3 0;
         1 2 3];
vnodes = [0   0;
          1   0;
          0   1;
          0   1/3;
          0   2/3;
          1/3 0;
          2/3 0;
          1/3 2/3;
          2/3 1/3;
          1/3 1/3];
elt = struct('get_Aphihat',@cub2d_get_Aphihat, ...
     'nvars',nvars,'flist',flist, ...
     'pxfeature',@cub2d_pxfeature,'vnodes',vnodes, ...
     'trans_Aphihat',@trans2d_Aphilist);
end % function
```

5.4.1 Piecewise cubic elements: get_Aphihat

There are six basis functions on the reference element.

56 ⟨*cub2d-elt.m* 55b⟩+≡

```
function Aphihat = cub2d_get_Aphihat(xhat,order)
% function Aphihat = cub2d_get_Aphihat(xhat,order)
%
% Returns basis function values, gradients and Hessian
% entries for Lagrangian cubic basis functions on the
% reference triangle with vertices (0,0), (1,0), and (0,1).
% Each row contains the value, 1st derivatives, and 2nd derivatives
% of the corresponding basis function on the reference element.
x = xhat(1);
y = xhat(2);
Aphihat0 = [(9/2)*(1-x-y)*(2/3-x-y)*(1/3-x-y);
    (9/2)*x*(x-1/3)*(x-2/3);
    (9/2)*y*(y-1/3)*(y-2/3);
    (27/2)*x*(2/3-x-y)*(1-x-y);
    (27/2)*x*(x-1/3)*(1-x-y);
    (27/2)*y*(2/3-x-y)*(1-x-y);
    (27/2)*y*(y-1/3)*(1-x-y);
    (27/2)*x*y*(x-1/3);
    (27/2)*x*y*(y-1/3);
    27*x*y*(1-x-y)];
if order >= 1
    Aphihat1 = [ ...
      18*x + 18*y - 27*x*y - (27*x^2)/2 - (27*y^2)/2 - 11/2, 18*x + 18*y - 27*x*y -
      (27*x^2)/2 - 9*x + 1, 0;
      0, (27*y^2)/2 - 9*y + 1;
      (81*x^2)/2 + 54*x*y - 45*x + (27*y^2)/2 - (45*y)/2 + 9, (9*x*(6*x + 6*y - 5))
      36*x + (9*y)/2 - 27*x*y - (81*x^2)/2 - 9/2, -(27*x*(x - 1/3))/2;
      (9*y*(6*x + 6*y - 5))/2, (27*x^2)/2 + 54*x*y - (45*x)/2 + (81*y^2)/2 - 45*y +
      -(27*y*(y - 1/3))/2, (9*x)/2 + 36*y - 27*x*y - (81*y^2)/2 - 9/2;
      (9*y*(6*x - 1))/2, (27*x*(x - 1/3))/2;
      (27*y*(y - 1/3))/2, (9*x*(6*y - 1))/2;
      -27*y*(2*x + y - 1), -27*x*(x + 2*y - 1)];
end
if order >= 2
    Aphihat2 = [ ...
```

```
      18 - 27*y - 27*x, 18 - 27*y - 27*x, 18 - 27*y - 27*x;
      27*x - 9, 0, 0;
      0, 0, 27*y - 9;
      81*x + 54*y - 45, 54*x + 27*y - 45/2, 27*x;
      36 - 27*y - 81*x, 9/2 - 27*x, 0;
      27*y, 27*x + 54*y - 45/2, 54*x + 81*y - 45;
      0, 9/2 - 27*y, 36 - 81*y - 27*x;
      27*y, 27*x - 9/2, 0;
      0, 27*y - 9/2, 27*x;
      -54*y, 27 - 54*y - 54*x, -54*x];
end
if order == 0
    Aphihat = Aphihat0;
elseif order == 1
    Aphihat = [Aphihat0, Aphihat1];
elseif order == 2
    Aphihat = [Aphihat0, Aphihat1, Aphihat2];
end % if
end % function
```

5.4.2 Piecewise cubic elements: pxfeature()

Here we need to permute the two edge variables.

58a ⟨*cub2d-elt.m* 55b⟩+≡

```
function [px_vars,signs] = cub2d_pxfeature(px)
% function [px_vars,signs] = cub2d_pxfeature(px)
%
% Returns the permutation of the variables (px_vars),
% and the sign changes (signs) resulting from a permutation (px)
% applied to a feature of the appropriate dimension (== length(px)).
% This is for the quadratic 2-D triangle elements.
dim = sum(px ~= 0)-1;
switch dim
    case 0 % points
        px_vars = [1]; signs = [1];
    case 1 % edges
        px_vars = px; signs = [1 1];
    case 2 % triangles
        px_vars = [1]; signs = [1];
    otherwise % not a valid feature
        px_vars = []; signs = [];
end % switch
end % function
```

5.5 Piecewise constant elements

Piecewise constant functions are either completely constant, or are discontinuous. This limits their applicability, but they can still be useful. Their definition for this system follows.

58b ⟨*filelist* 6b⟩+≡

```
const2d-elt.m \
```

59 ⟨*const2d-elt.m* 59⟩≡

```
function elt = const2d_elt()
% function elt = const2d_elt()
%
% Returns constant 2-D triangle element
nvars  = [1];
flist  = [1 2 3];
vnodes = [1/3, 1/3];
elt = struct('get_Aphihat',@const2d_get_Aphihat, ...
    'nvars',nvars,'flist',flist, ...
    'pxfeature',@const2d_pxfeature,'vnodes',vnodes, ...
    'trans_Aphihat',@trans2d_Aphilist);
end % function
```

The basis functions are easy to compute:

60 $\langle$*const2d-elt.m* 59$\rangle+\equiv$

```
function Aphilist = const2d_get_Aphihat(xhat,order)
% function Aphilist = const2d_get_Aphihat(xhat,order)
%
% Returns array of basis function values, their gradient and Hessian entries
% for constant basis functions on a 2-D reference triangle at xhat.
% Basis function values
Aphilist0 = [1];
if order >= 1
    % Basis gradient values (along rows)
    Aphilist1 = [0 0];
end
if order >= 2
    % Basis hessian values (along rows: d1^2, d1.d2, d2^2)
    Aphilist2 = [0 0 0];
end
if order == 0
    Aphilist = Aphilist0;
elseif order == 1
    Aphilist = [Aphilist0,Aphilist1];
elseif order == 2
    Aphilist = [Aphilist0,Aphilist1,Aphilist2];
end % if
end % function
```

There is only one variable so changing the orientation does not do much to the variables.

61a ⟨*const2d-elt.m* 59⟩+≡

```
function [px_vars,signs] = const2d_pxfeature(px)
% function [px_vars,signs] = const2d_pxfeature(px)
%
% Returns the permutation of the variables (px_vars),
% and the sign changes (signs) resulting from a permutation (px)
% applied to a feature of the appropriate dimension (== length(px)).
% This is for the quadratic 2-D triangle elements.
dim = sum(px ~= 0)-1;
switch dim
    case 2 % triangles
        px_vars = [1]; signs = [1];
    otherwise % not a valid feature
        px_vars = []; signs = [];
end % switch
end % function
```

5.6 Vector elements

Rather than create a vector element type separately for each scalar element type, we can create them automatically from the scalar element type. There is a new transformation routine, and it is here that the vector character is made apparent. The final result after the transformation has the columns of the output ordered as $\mathbf{e}_1 \cdot \phi_i(\mathbf{x})$, $\mathbf{e}_2 \cdot \phi_i(\mathbf{x})$, $\mathbf{e}_1 \cdot \partial\phi_i/\partial x_1(\mathbf{x})$, $\mathbf{e}_2 \cdot \partial\phi_i/\partial x_1(\mathbf{x})$, etc. That is, the components alternate.

61b ⟨*filelist* 6b⟩+≡

```
eltx2-elt.m \
```

62 ⟨*eltx2-elt.m* 62⟩≡

```
function eltx2 = eltx2_elt(elt)
% function eltx2 = eltx2_elt(elt)
%
% Returns element data structure with the same basis
% functions as defined by elt, but with 2 components for
% each component of elt.
nvars = elt.nvars;
nvarsx2 = 2*nvars;
flist = elt.flist;
flistx2 = flist;
trans_Aphihat = @(T,Aphilist,order)(transx2(elt.trans_Aphihat,T,Aphilist,order));
pxfeature = @(px)(pxfeaturex2(elt.pxfeature,px));
vnodes = elt.vnodes();
vnodesx2 = zeros(2*size(vnodes,1),size(vnodes,2));
vnodesx2(2*(1:size(vnodes,1))-1,:) = vnodes;
vnodesx2(2*(1:size(vnodes,1)) ,:) = vnodes;
eltx2 = struct('get_Aphihat',elt.get_Aphihat, ...
    'nvars',nvarsx2,'flist',flistx2, ...
    'pxfeature',pxfeature,'vnodes',vnodesx2, ...
    'trans_Aphihat',trans_Aphihat);
end % function
```

Note that the original get_Aphihat() function is used. But we need new permutation and transformation functions based on the originals.

5.6.1 Vector elements: pxfeature()

63 ⟨*eltx2-elt.m* 62⟩+≡

```
function [px_varsx2,signsx2] = pxfeaturex2(base_pxfeature,px)
% function [px_varsx2,signsx2] = pxfeaturex2(base_pxfeature,px)
%
% Uses base_pxfeature to create pxfeature() function for the "x2" element
[px_vars,signs] = base_pxfeature(px);
px_varsx2 = zeros(1,2*length(px_vars));
signsx2   = zeros(1,2*length(px_vars));
px_varsx2(2*(1:length(px_vars))-1) = 2*px_vars-1;
signsx2(2*(1:length(px_vars))-1)   = signs;
px_varsx2(2*(1:length(px_vars)) )  = 2*px_vars;
signsx2(2*(1:length(px_vars)) )    = signs;
end % function
```

5.6.2 Vector elements: trans_Aphilist()

64 ⟨*eltx2-elt.m* 62⟩+≡

```
function Aphilistx2 = transx2(base_trans,T,Aphilist,order)
% function Aphilistx2 = transx2(base_trans,T,Aphilist,order)
%
% Uses base_trans to create trans_Aphilist function for the "x2" element
Aphilist = base_trans(T,Aphilist,order);
nb = size(Aphilist,1);
Aphilistx2 = zeros(2*nb,size(Aphilist,2));
Aphilistx2(2*(1:nb)-1,1) = Aphilist(:,1);
Aphilistx2(2*(1:nb) ,2) = Aphilist(:,1);
if order >= 1
    Aphilistx2(2*(1:nb)-1,3:4) = Aphilist(:,2:3);
    Aphilistx2(2*(1:nb) ,5:6) = Aphilist(:,2:3);
end
if order >= 2
    Aphilistx2(2*(1:nb)-1,7:9) = Aphilist(:,4:6);
    Aphilistx2(2*(1:nb) ,10:12) = Aphilist(:,4:6);
end
end % function
```

5.7 C^1 elements: Bell's triangle

Work in progress.

5.8 Stokes' equations: the Arnold–Brezzi–Fortin elements

Work in progress.

Stokes' equations have the form

$$\begin{aligned} -\mu \Delta \mathbf{u} + (\mathbf{w} \cdot \nabla)\, \mathbf{u} &= -\nabla p + \mathbf{f}(\mathbf{x}), \\ \nabla \cdot \mathbf{u} &= 0. \end{aligned}$$

This is a natural template for the Navier–Stokes equations for incompressible Newtonian fluidswhere $\mathbf{u}$ is the velocity field, and p is the pressure field. For $\mathbf{w} = 0$, the Stokes equations can be treated as a constrained optimization problem:

$$\min \int_\Omega \left[\frac{1}{2} \mu \, |\nabla \mathbf{u}|^2 - \mathbf{f} \cdot \mathbf{u} \right] d\mathbf{x} \qquad \text{subject to}$$
$$\nabla \cdot \mathbf{u} = 0,$$

where p takes the role of Lagrange multiplier. A global condition has to be placed on p as otherwise replacing p with $p + c$ for any constant c gives a new solution. For uniqueness we typically require that $\int_\Omega p \, d\mathbf{x} = 0$.

The weak form of Stokes' equations are: for all suitable $\mathbf{v}$ and q,

$$\begin{aligned} \int_\Omega [\mu \nabla \mathbf{v} : \nabla \mathbf{u} + \mathbf{v} \cdot ((\mathbf{w} \cdot \nabla)\, \mathbf{u}) - (\nabla \cdot \mathbf{v})\, p] \, d\mathbf{x} &= \int_\Omega \mathbf{v} \cdot \mathbf{f} \, d\mathbf{x} + (\text{boundary integrals}), \\ \int_\Omega q\, (\nabla \cdot \mathbf{u}) \, d\mathbf{x} &= 0. \end{aligned}$$

This can be dealt with in separate ways. We can create separate elements for $\mathbf{u}$ and p, and use `pgassembly()` to combine them, or we can create a single element for both together. The Arnold–Brezzi–Fortin (ABF) finite element method for this problem involves using an enriched piecewise linear element for each component of $\mathbf{u}$ and piecewise linear elements for p. The additional basis function for each u_i has the form $\lambda_1 \lambda_2 \lambda_3$ in barycentric coordinates, or in standard coordinates for the reference triangle, $27xy(1 - x - y)$. We can create a scalar ABF element which contains just the four basis functions necessary, and then use `eltx2_elt()` to create the vector element for $\mathbf{u}$.

The vector element for **u** is simply the ABF scalar element "$\times 2$".

66a ⟨*filelist* 6b⟩+≡
```
abf2d-elt.m \
```

66b ⟨*abf2d-elt.m* 66b⟩≡
```
function elt = abf2d_elt()
% function elt = abf2d_elt()
%
% ABF vector element for velocity field.
% This is the ABF scalar element "x2".
% The rationale for using this set of basis functions is
% given in Arnold, Brezzi and Fortin,
%
elt = eltx2_elt(abfs2d_elt());
end % function
```

The scalar element is defined below.

66c ⟨*abf2d-elt.m* 66b⟩+≡
```
function elt = abfs2d_elt()
% function elt = abfs2d_elt()
%
% Returns the ABF scalar 2-D (3-point) element data structure.
% The basis functions for this element are the same as for
% the piecewise linear element, plus the "bubble" function
% \phi_4(x,y) = 27xy(1-x-y).
nvars = [1;1;1;1]
flist = [1 0 0
         2 0 0
         3 0 0
         1 2 3];
elt = struct('get_Aphihat',@abfs2d_get_Aphihat, ...
    'nvars',nvars,'flist',flist, ...
    'pxfeature',@abfs2d_pxfeature,'vnodes',abfs2d_vnodes(), ...
    'trans_Aphihat',@trans2d_Aphilist);
end
```

Note that the first three basis functions are associated with the vertices of the triangle; these are linear basis functions. The fourth is the "bubble" function, which is associated with the interior of the triangle. The basis functions are given below:

67 ⟨*abf2d-elt.m* 66b⟩+≡

```
function Aphilist = abfs2d_get_Aphihat(xhat,order)
% Aphilist = abfs2d_get_Aphihat(xhat,order)
%
% ABF scalar element: linear basis functions plus "bubble" function
% \lambda_1\lambda_2\lambda_3 in barycentric coordinates.
x = xhat(1);  y = xhat(2);
% Basis function values
Aphilist0 = [1-x-y;
             x;
             y
             27*x*y*(1-x-y)];
if order >= 1
    % Basis gradient values (along rows)
    Aphilist1 = [-1 -1;
                  1  0;
                  0  1;
                 27*y*(1-y-2*x), 27*x*(1-x-2*y)];
end
if order >= 2
    % Basis hessian values (along rows: dx1^2, dx1.dx2, dx2^2)
    Aphilist2 = [0 0 0;
                 0 0 0;
                 0 0 0;
                 -54*y, 27 - 54*y - 54*x, -54*x];
end
if order == 0
    Aphilist = Aphilist0;
elseif order == 1
    Aphilist = [Aphilist0,Aphilist1];
elseif order == 2
    Aphilist = [Aphilist0,Aphilist1,Aphilist2];
end % if
end % function
```

Permutations of the geometric features do not change the ordering (or signs) of the variables.

68a ⟨*abf2d-elt.m* 66b⟩+≡
```
function [px_vars,signs] = abfs2d_pxfeature(px)
% function [px_vars,signs] = abfs2d_pxfeature(px)
%
% Returns the permutation of the variables (px_vars),
% and the sign changes (signs) resulting from a permutation (px)
% applied to a feature of the appropriate dimension (== length(px)).
% This is for the linear (or affine) 2-D triangle elements.
dimp1 = sum(px ~= 0);  % dimp1 == dimension plus 1
switch dimp1
    case 1 % points
        px_vars = [1]; signs = [1];
    case 3 % triangles
        px_vars = [1]; signs = [1];
    otherwise % not a valid feature
        px_vars = []; signs = [];
end % switch
end % function
```

The basis is a nodal basis with nodes at the vertices and the centroid of the triangle, except that ϕ_i are not zero at the centroid for $i = 1, 2, 3$.

68b ⟨*abf2d-elt.m* 66b⟩+≡
```
function vnodes = abfs2d_vnodes()
% function vnodes = abfs2d_vnodes()
%
% Returns the positions of the variable nodes
% with respect to the reference element.
% Same format as p (2 x m)
vnodes = [0   0;
          1   0;
          0   1;
          1/3 1/3];
end % function
```

We need to compute the integrals $\int_\Omega q \, \nabla \cdot \mathbf{u} \, d\mathbf{x}$ for q piecewise linear and $\mathbf{u}$ formed using the ABF vector element, which can be achieved using the following PDE data structure:

```
qdiv_form = struct('coeffs',@(x)[0,0,1,0,0,1;zeros(2,6)], ...
    'rhs',@(x)zeros(6,1), 'order',1);
```

5.9 Hsieh–Clough–Tocher C^1 element

This is a C^1 "macro" element with piecewise cubic basis functions; these basis functions are cubic on subtriangles. This element involves normal derivatives at the midpoints of the edges. Affine transformations do not preserve normal derivatives, so there is an additional complication in the computation of the transformation of the basis functions so that the nodal basis property is preserved in the basis functions on the real elements.

The element is illustrated in Figure 7. At each vertex, there are three variables: one for the value at the point, and two for the two partial derivatives ($\partial\phi/\partial x_1$, $\partial\phi/\partial x_2$). Also, at the midpoint of each edge there is the normal derivative. This gives a total of 12 nodal basis functions for this element. Integration over these elements should be done using a composite integration method: the basis functions are cubic over the sub-triangles formed by an edge and the centroid ($\mathbf{x}_c = (\mathbf{x}_1 + \mathbf{x}_2 + \mathbf{x}_3)/3$).

69 ⟨*filelist* 6b⟩+≡

```
hct2d-elt.m \
```

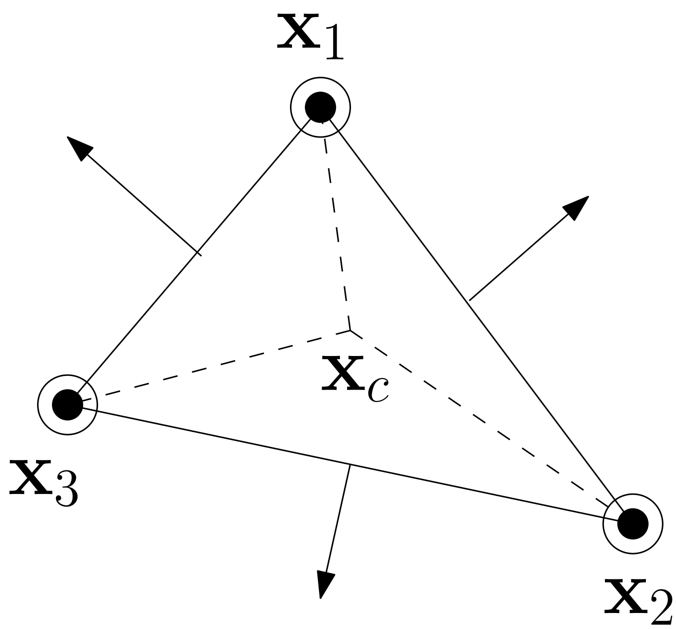

Figure 7: Hsieh–Clough–Tocher (HCT) element

70 ⟨*hct2d-elt.m* 70⟩≡

```
function elt = hct2d_elt()
% function elt = hct2d_elt()
%
% Hsieh-Clough-Tocher element in two dimensions.
nvars = [3;3;3;1;1;1];
flist = [1 0 0;  1 0 0;  1 0 0;
         2 0 0;  2 0 0;  2 0 0;
         3 0 0;  3 0 0;  3 0 0;
         1 2 0;
         1 3 0;
         2 3 0];
vnodes = [0 0;  0 0;  0 0;
          1 0;  1 0;  1 0;
          0 1;  0 1;  0 1;
          1/2 0; 0 1/2; 1/2 1/2];
elt = struct('get_Aphihat',@hct2d_get_Aphihat, ...
    'nvars',nvars,'flist',flist, ...
    'pxfeature',@hct2d_pxfeature,'vnodes',vnodes, ...
```

```
    'trans_Aphihat',@hct2d_trans_Aphilist);
end
end % function
```

There are twelve basis functions for one HCT element, but the formula used depends on which part of the reference triangle needs to be evaluated.

71a ⟨*hct2d-elt.m* 70⟩+≡

5.10 Three-dimensional elements

5.10.1 Piecewise linear three-dimensional elements

This is the simplest useful three-dimensional element. Assembly routines for three-dimensional problems need to be written. (See Section 3.) See Section 5.1 for the two-dimensional piecewise linear element.

Note that the order of the operators for scalar three-dimensional elements is $\mathcal{A} = I$ (identity), $\partial/\partial x_1$, $\partial/\partial x_2$, $\partial/\partial x_3$, $\partial^2/\partial x_1^2$, $\partial^2/\partial x_1\partial x_2$, $\partial^2/\partial x_1\partial x_3$, $\partial^2/\partial x_2^2$, $\partial^2/\partial x_2\partial x_3$, $\partial^2/\partial x_3^2$. We often use the notation $x = x_1$, $y = x_2$ and $z = x_3$ for convenience.

Note that all basis functions are associated with vertices of the tetrahedron. We also need a new transformation routines for transforming from the reference element in three dimensions (which is the tetrahedron with the vertices $(0,0,0)$, $(1,0,0)$, $(0,1,0)$, and $(0,0,1)$) to the actual element (with vertices $\mathbf{p}_1$, $\mathbf{p}_2$, $\mathbf{p}_3$ and $\mathbf{p}_4$).

71b ⟨*filelist* 6b⟩+≡

```
  lin3d-elt.m \
```

72 ⟨*lin3d-elt.m* 72⟩≡

```
function elt = lin3d_elt()
% function elt = lin3d_elt()
%
% Returns the linear 3-D (4-point) element data structure.
nvars = [1;1;1;1];
flist = [1 0 0 0;
         2 0 0 0;
         3 0 0 0;
         4 0 0 0];
vnodes = [0 0 0;
          1 0 0;
          0 1 0;
          0 0 1];
elt = struct('get_Aphihat',@lin3d_get_Aphihat, ...
    'nvars',nvars,'flist',flist, ...
    'pxfeature',@lin3d_pxfeature,'vnodes',vnodes, ...
    'trans_Aphihat',@trans3d_Aphilist);
end
```

The basis functions are easily computed. Note that the first derivatives are constant, and the second derivatives are all zero.

73 ⟨*lin3d-elt.m* 72⟩+≡

```
function Aphilist = lin3d_get_Aphihat(xhat,order)
% Aphilist = lin3d_get_Aphihat(xhat,order)
%
% Returns array of basis function values, their gradient and Hessian entries
x = xhat(1);  y = xhat(2);  z = xhat(3);
% Basis function values
Aphilist0 = [1-x-y-z;
             x;
             y;
             z];
if order >= 1
    % Basis gradient values (along rows)
    Aphilist1 = [-1 -1 -1;
                  1  0  0;
                  0  1  0;
                  0  0  1];
end
if order >= 2
    % Basis Hessian values (along rows: dx1^2, dx1.dx2, dx1.dx3, dx2^2, dx2.dx3, dx
    Aphilist2 = [0 0 0 0 0 0;
                 0 0 0 0 0 0;
                 0 0 0 0 0 0];
end
if order == 0
    Aphilist = Aphilist0;
elseif order == 1
    Aphilist = [Aphilist0,Aphilist1];
elseif order == 2
    Aphilist = [Aphilist0,Aphilist1,Aphilist2];
end % if
end % function
```

Permutation of the geometric features does not change the order of the variables since there is only one variable for each vertex, just as for the two-dimensional piecewise linear element.

74a ⟨*lin3d-elt.m* 72⟩+≡

```
function [px_vars,signs] = lin3d_pxfeature(px)
% function [px_vars,signs] = lin3d_pxfeature(px)
%
% Returns the permutation of the variables (px_vars),
% and the sign changes (signs) resulting from a permutation (px)
% applied to a feature of the appropriate dimension (== length(px)).
% This is for the linear (or affine) 3-D tetrahedral elements.
dimp1 = sum(px ~= 0);  % dimp1 == dimension plus 1
switch dimp1
    case 1 % points
        px_vars = [1]; signs = [1];
    otherwise % not a valid feature
        px_vars = []; signs = [];
end % switch
end % function
```

5.11 Three-dimensional scalar transformations

The affine transformation in three dimensions $\widehat{\mathbf{x}} \mapsto \mathbf{x} = T\widehat{\mathbf{x}} + \mathbf{b}$ must modify the derivative values. For derivation of the basic equations, see Section 5.2.

74b ⟨*filelist* 6b⟩+≡

```
trans3d-Aphilist.m \
```

75 ⟨*trans3d-Aphilist.m* 75⟩≡

```
function Aphilist2 = trans3d_Aphilist(T,Aphilist,order)
% function Aphilist2 = trans3d_Aphilist(T,Aphilist,order)
%
% Transforms Aphilist into Aphilist2 according to matrix T (3 x 3)
% Note: Order of colums is [val, d/dx1, d/dx2, d/dx3, d^2/dx1^2, d^2/dx1.dx2, d^2/d
Aphilist2 = zeros(size(Aphilist));
Aphilist2(:,1) = Aphilist(:,1); % values unchanged
if order >= 1
    S = inv(T);
    Aphilist2(:,2:4) = Aphilist(:,2:4)*S; % chain rule for 1st derivatives
end % if
if order >= 2
    % chain rule for 2nd derivatives (affine transformation)
    Aphilist2(:,5)  = Aphilist(:,5)*(S(1,1)^2)+ ...
           Aphilist(:,6)*(2*S(2,1)*S(1,1))+ ...
           Aphilist(:,7)*(2*S(3,1)*S(1,1))+ ...
           Aphilist(:,8)*(S(2,1)^2)+ ...
           Aphilist(:,9)*(2*S(3,1)*S(2,1))+ ...
           Aphilist(:,10)*(S(3,1)^2);
    Aphilist2(:,6)  = Aphilist(:,5)*(S(1,1)*S(1,2))+ ...
           Aphilist(:,6)*(S(1,1)*S(2,2)+S(1,2)*S(2,1))+ ...
           Aphilist(:,7)*(S(3,1)*S(1,2)+S(1,1)*S(3,2))+ ...
           Aphilist(:,8)*(S(2,2)*S(2,1))+ ...
           Aphilist(:,9)*(S(3,1)*S(2,2)+S(2,1)*S(3,2))+ ...
           Aphilist(:,10)*(S(3,1)*S(3,2));
    Aphilist2(:,7)  = Aphilist(:,5)*(S(1,1)*S(1,3))+ ...
           Aphilist(:,6)*(S(2,1)*S(1,3)+S(1,1)*S(2,3))+ ...
           Aphilist(:,7)*(S(3,1)*S(1,3)+S(1,1)*S(3,3))+ ...
           Aphilist(:,8)*(S(2,1)*S(2,3))+ ...
           Aphilist(:,9)*(S(3,1)*S(2,3)+S(2,1)*S(3,3))+ ...
           Aphilist(:,10)*(S(3,1)*S(3,3));
    Aphilist2(:,8)  = Aphilist(:,5)*(S(1,2)^2)+ ...
           Aphilist(:,6)*(2*S(2,2)*S(1,2))+ ...
           Aphilist(:,7)*(2*S(3,2)*S(1,2))+ ...
           Aphilist(:,8)*(S(2,2)^2)+ ...
           Aphilist(:,9)*(2*S(3,2)*S(2,2))+ ...
           Aphilist(:,10)*(S(3,2)^2);
    Aphilist2(:,9)  = Aphilist(:,5)*(S(1,2)*S(1,3))+ ...
           Aphilist(:,6)*(S(2,2)*S(1,3)+S(1,2)*S(2,3))+ ...
```

```
            Aphilist(:,7)*(S(3,2)*S(1,3)+S(1,2)*S(3,3))+ ...
            Aphilist(:,8)*(S(2,2)*S(2,3))+ ...
            Aphilist(:,9)*(S(3,2)*S(2,3)+S(2,2)*S(3,3))+ ...
            Aphilist(:,10)*(S(3,2)*S(3,3));
    Aphilist2(:,10) = Aphilist(:,5)*(S(1,3)^2)+ ...
            Aphilist(:,6)*(2*S(2,3)*S(1,3))+ ...
            Aphilist(:,7)*(2*S(3,3)*S(1,3))+ ...
            Aphilist(:,8)*(S(2,3)^2)+ ...
            Aphilist(:,9)*(2*S(3,3)*S(2,3))+ ...
            Aphilist(:,10)*(S(3,3)^2);
end % if
end % function
```

6 Numerical integration

Numerical integration is basic to the assembly process. The numerical integration routine should be exact for products of the basis functions that are used. Since the assembly routines perform the transformation from reference elements, these routines just need to return the points and weights for the method on a reference element, which is the triangle with vertices $(0,0)$, $(1,0)$, and $(0,1)$.

6.1 Two-dimensional integration methods

6.1.1 Centroid method

This is a one-point method with order 1. (That is, it is exact for all polynomials of order ≤ 1.)

76a ⟨*filelist* 6b⟩+≡

```
int2d-centroid1.m \
```

76b ⟨*int2d-centroid1.m* 76b⟩≡

```
function [p,w] = int2d_centroid1()
% function [p,w] = int2d_centroid1()
%
% This is the simplest triangle integration method.
p = [1/3, 1/3];
w = 1/2;
```

6.1.2 Radon's method

This is a 7-point method with order 5 [6].

77a ⟨*filelist* 6b⟩+≡
```
int2d-radon7.m \
```

77b ⟨*int2d-radon7.m* 77b⟩≡
```
function [p,w] = int2d_radon7()
% function [p,w] = int2d_radon7()
%
% Returns the points (p) and weights (w) of J. Radon's 7-point
% integration formula for the triangle with vertices (0,0), (1,0), (0,1).
% This formula is exact for polynomials up to degree 5.
% Points are the rows of p.
%
% Reference: J. Radon, Zur mechanischen Kubatur. (German)
% Monatsh. Math. 52, (1948), pp. 286-300.
p = [1/3,               1/3;
    (6+sqrt(15))/21,   (9-2*sqrt(15))/21;
    (9-2*sqrt(15))/21, (6+sqrt(15))/21;
    (6+sqrt(15))/21,   (6+sqrt(15))/21;
    (6-sqrt(15))/21,   (9+2*sqrt(15))/21;
    (9+2*sqrt(15))/21, (6-sqrt(15))/21;
    (6-sqrt(15))/21,   (6-sqrt(15))/21];
w = [9/80;
    (155+sqrt(15))/2400;
    (155+sqrt(15))/2400;
    (155+sqrt(15))/2400;
    (155-sqrt(15))/2400;
    (155-sqrt(15))/2400;
    (155-sqrt(15))/2400];
end
```

6.1.3 Gatermann's method

This is a 12-point method with order 7 [4].

77c ⟨*filelist* 6b⟩+≡
```
int2d-gatermann12.m \
```

78 ⟨*int2d-gatermann12.m* 78⟩≡

```
function [p,w] = int2d_gatermann12()
% function [p,w] = int2d_gatermann12()
%
% Returns the points (p) and weights (w) of K. Gatermann's 12-point
% integration formula for the triangle with vertices (0,0), (1,0), (0,1).
% This formula is exact for polynomials up to degree 7.
% Points are the rows of p.
%
% Reference: The Construction of Symmetric Cubature Formulas
% for the Square and the Triangle, Computing 40, 229 - 240 (1988)
p = [0.06751786707392436, 0.8700998678316848;  % 1
     0.06238226509439084, 0.06751786707392436; % 2
     0.8700998678316848,  0.06238226509439084; % 3
     0.3215024938520156,  0.6232720494910644;  % 4
     0.05522545665692000, 0.3215024938520156;  % 5
     0.6232720494910644,  0.05522545665692000; % 6
     0.6609491961867980,  0.3047265008681072;  % 7
     0.03432430294509488, 0.6609491961867980;  % 8
     0.3047265008681072,  0.03432430294509488; % 9
     0.2777161669764050,  0.2064414986699949;  % 10
     0.5158423343536001,  0.2777161669764050;  % 11
     0.2064414986699949,  0.5158423343536001]; % 12
w = [0.02651702815743450;
     0.02651702815743450;
     0.02651702815743450;
     0.04388140871444811;
     0.04388140871444811;
     0.04388140871444811;
     0.02877504278497528;
     0.02877504278497528;
     0.02877504278497528;
     0.06749318700980879;
     0.06749318700980879;
     0.06749318700980879];
end
```

6.1.4 A method of Dunavant

This is a 33-point method with order 12 [3].

79 ⟨*filelist* 6b⟩+≡

```
int2d-dunavant33.m \
```

80 ⟨*int2d-dunavant33.m* 80⟩≡

```
function [p,w] = int2d_dunavant33()
% function [p,w] = int2d_dunavant33()
%
% Returns points and weights for Dunavant (1978)'s
% 12th order 33 point triangle integration method.
% Values taken directly from Dunavant's paper:
% "High degree efficient symmetrical Gaussian quadrature rules for the
% triangle", Internat. J. Numer. Methods Eng. vol 21, pp. 1129-1148 (1985)
% p=12 ng=33 nsige17.8 ssqa9.d-58 error= 1.d-27 ifn= 2439 infers1 time= 74
% weight alpha beta gamma
% 0.025731066440455 0.023565220452390 0.488217389773805 0.488217389773805
% 0.043692544538038 0.120551215411079 0.439724392294460 0.439724392294460
% 0.062858224217885 0.457579229975768 0.271210385012116 0.271210385012116
% 0.034796112930709 0.744847708916828 0.127576145541586 0.127576145541586
% 0.006166261051559 0.957365299093579 0.021317350453210 0.021317350453210
% 0.040371557766381 0.115343494534698 0.275713269685514 0.608943235779788
% 0.022356773202303 0.022838332222257 0.281325580989940 0.695836086787803
% 0.017316231108659 0.025734050548330 0.116251915907597 0.858014033544073
table = [...
0.025731066440455 0.023565220452390 0.488217389773805 0.488217389773805
0.043692544538038 0.120551215411079 0.439724392294460 0.439724392294460
0.062858224217885 0.457579229975768 0.271210385012116 0.271210385012116
0.034796112930709 0.744847708916828 0.127576145541586 0.127576145541586
0.006166261051559 0.957365299093579 0.021317350453210 0.021317350453210
0.040371557766381 0.115343494534698 0.275713269685514 0.608943235779788
0.022356773202303 0.022838332222257 0.281325580989940 0.695836086787803
0.017316231108659 0.025734050548330 0.116251915907597 0.858014033544073];
idx = 1; t_idx = 1;
p = zeros(33,2); w = zeros(32,1);
for t_idx = 1:size(table,1)
    if table(t_idx,3) == table(t_idx,4)
        % 3 entries
        w(idx:(idx+2)) = table(t_idx,1);
        p(idx+0,:) = table(t_idx,[2,3]);
        p(idx+1,:) = table(t_idx,[3,2]);
        p(idx+2,:) = table(t_idx,[3,4]);
        idx = idx+3;
    else
        % 6 entries
```

```
        w(idx:(idx+5)) = table(t_idx,1);
        p(idx+0,:) = table(t_idx,[2,3]);
        p(idx+1,:) = table(t_idx,[2,4]);
        p(idx+2,:) = table(t_idx,[3,2]);
        p(idx+3,:) = table(t_idx,[3,4]);
        p(idx+4,:) = table(t_idx,[4,2]);
        p(idx+5,:) = table(t_idx,[4,3]);
        idx = idx+6;
    end % if
end % for
w = w/2;
end % function
```

6.2 One-dimensional integration methods

6.2.1 Gauss–Legendre quadrature

Here we use the 5-point Gauss–Legendre quadrature method which is a 9th order method [1].

81a ⟨*filelist* 6b⟩+≡

```
int1d-gauss5.m \
```

81b ⟨*int1d-gauss5.m* 81b⟩≡

```
function [p,w] = int1d_gauss5()
% function [p,w] = int1d_gauss5()
%
% Returns points and weights for a 5-point Gauss rule
% in 1-D for the interval [0,1]
p = [1/2; (1+sqrt(5-2*sqrt(10/7))/3)/2; (1-sqrt(5-2*sqrt(10/7))/3)/2; ...
     (1+sqrt(5+2*sqrt(10/7))/3)/2; (1-sqrt(5+2*sqrt(10/7))/3)/2];
w = 0.5*[128/225; (322+13*sqrt(70))/900; (322+13*sqrt(70))/900; ...
     (322-13*sqrt(70))/900; (322-13*sqrt(70))/900];
```

6.3 Three-dimensional integration

6.3.1 Centroid method

This integration method is exact for 1st order polynomials (affine functions), and uses one point.

82a ⟨*filelist* 6b⟩+≡

```
int3d-centroid1.m \
```

82b ⟨*int3d-centroid1.m* 82b⟩≡

```
function [p,w] = int3d_centroid1()
% function [p,w] = int3d_centroid1()
%
% This is the simplest triangle integration method.
p = [1/4, 1/4, 1/4];
w = 1/6;
```

6.4 Composite integration rules

Composite integration rules allow the same integration rule to be replicated across a collection of sub-triangles. These are particularly useful for "macro" elements, that are formed by piecewise polynomial functions on sub-triangles of the reference triangle, such as the HCT element. In the same way, we can create composite rules that replicate a pre-existing two-dimensional rule (`intmethod`) across a triangulation (`pr`,`tr`) of the reference element.

82c ⟨*filelist* 6b⟩+≡

```
int2d-comp.m \
```

83a ⟨*int2d-comp.m* 83a⟩≡

```
function compmethod = int2d_comp(pr,tr,intmethod)
% function compmethod = int2d_comp(pr,tr,intmethod)
%
% Returns function handle for composite integration
% method that uses intmethod() as the basic method.
% replicated across all triangles of the triangulation
% (pr,tr) of the reference element.
compmethod = @int2d_comp_func(pr,tr,intmethod);
end function

function [p_int,w_int] = int2d_comp_function(pr,tr,intmethod)
% function [p_int,w_int] = int2d_comp_function(pr,tr,intmethod)
%
% Internal function for int2d_comp().
% This is where the work gets done.
[p_base,w_base] = intmethod(); % base method points & weights
base_len = size(p_base,1);
p_int = zeros(size(tr,1)*base_len,2);
w_int = zeros(size(tr,1)*base_len,1);
for j = 1:size(tr,1)
    % For sub-triangle j...
    % Create affine transformation
    i1 = tr(i,1);  i2 = tr(i,2);  i3 = tr(i,3);
    T = [pr(i2,:)'-pr(i1,:)', pr(i3,:)'-pr(i1,:)'];
    b0 = pr(i1,:)';
    % transform weights and points and add to list
    detT = abs(det(T));
    p_int(((j-1)*base_len+1):(j*base_len),:) = p_base*T'+b0';
    w_int(((j-1)*base_len+1):(j*base_len))   = w_int*detT;
end % for
end % function
```

For the HCT element, the appropriate integration method can be created using something like:

83b ⟨*HCT integrator* 83b⟩≡

```
pr = [0,0; 1,0; 0,1; 1/3,1/3];
tr = [1 2 4; 1 3 4; 2 3 4];
int2d_hct = int2d_comp(pr,tr,@int2d_radon7);
```

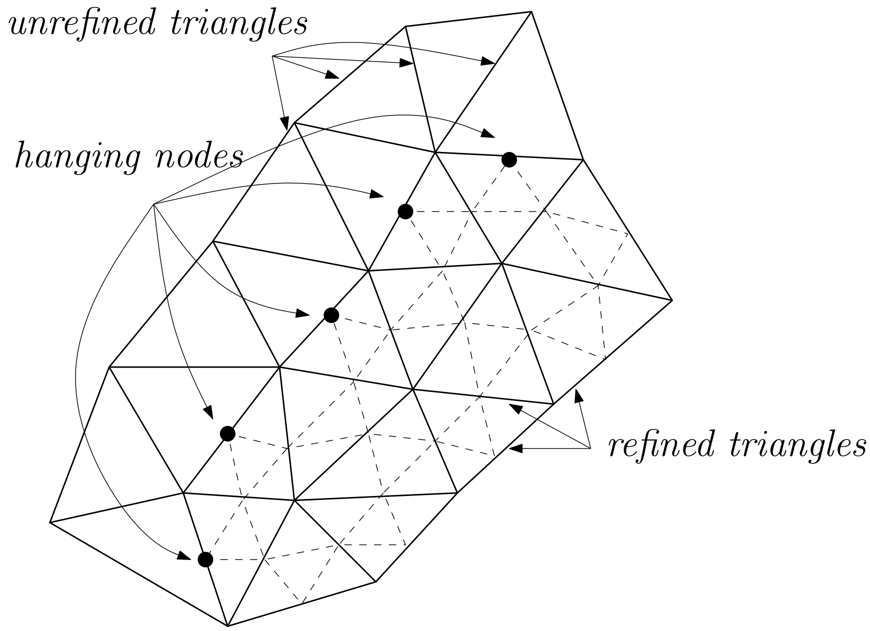

Figure 8: Adaptation of mesh showing hanging nodes

7 Adaptation

Adaptation of the mesh is a very useful technique to improve accuracy at minimal cost. There are two parts to this: one is to use *a posteriori* error estimation to identify triangles that should be refined; the other is to solve the refined system of equations. Simple adaptation techniques result in non-conforming triangulations. This is illustrated in Figure 8, which shows hanging nodes from the refined triangles that do not match nodes in the unrefined triangles on the upper-left.

The fact that we have a non-conforming mesh means that we cannot use the output of assembly routines "as is". The variables associated with the non-conforming part of the mesh need to be represented in terms of variables in the unrefined triangles.

The approach we take to mesh refinement starts with the refinement for the reference element; we take this to be the *reference refinement*. This reference refinement is a triangulation of the reference element. The basic assumption is that the basis functions on the reference element $\widehat{\phi}_i$ can be represented as linear combinations of the basis functions of the reference refinement. This is clearly the case for piecewise polynomial Lagrange elements, since on each triangle all polynomials up to a specified degree can be represented. This is not so for Bell's triangle, since in that element, the basis functions are 5th order polynomials where the normal derivatives on the boundaries are cubic. Thus for a non-trivial refinement of the reference element, the edges of the triangles in the interior of the reference element will not typically have cubic normal derivatives.

The basic idea in this code is to represent variables in the refined triangles in terms of the variables in the unrefined triangles where they meet. Thus we only need to consider the variables associated with the edges common to both refined and unrefined triangle. We assume that every basis function on the reference element is a linear combination of basis functions for the reference refinement:

$$\widehat{\phi}_i = \sum_j b_{ij}\,\widetilde{\phi}_j,$$

where $\widetilde{\phi}_j$ are the basis functions on the reference refinement.

Assuming that the corresponding basis function on the real element is $\phi_k(\mathbf{x}) = \widehat{\phi}_i(\widehat{\mathbf{x}})$ where $\mathbf{x} = T_K\widehat{\mathbf{x}} + \mathbf{b}_K$, this relation between $\widehat{\phi}_i$ and $\widetilde{\phi}_j$ can be transformed from the reference element to the real unrefined and real refined elements.

The refined and unrefined triangles are then treated as logically separated (and the assembly is performed on them that way) until they are "glued" together using sparse matrix-matrix multiplication with the b_{ij} matrices. Note that the computational cost of this extra work depends on the number of basis functions associated with the common boundary between the refined and unrefined triangles. It is desirable to keep this from growing too large, especially after many steps of adaptation. The following approach is taken to prevent excessively large boundaries: the original triangulation is kept, and each triangle, even if refined many times, has a link to its "parent" triangle at the next coarser level. A group of refined triangles can be unrefined whenever desired. Thus there can be a hierarchy of levels of refinement. This is illustrated in Figure 9.

Identifying triangles for refinement is done using a residual-based error estimation method [2, Chap. 9]. If the problem is to solve $\mathcal{L}u = b$ where $\mathcal{L}$ is a linear differential operator $V \to V'$, V a Hilbert space (typically a Sobolev space), then we need to estimate the norm of $\mathcal{L}u - b$ in V' (the dual space of V). This in turn involves estimating

$$\sup_{w\in V} \frac{\langle \mathcal{L}u - b,\, w\rangle_{V\times V'}}{\|w\|_V}.$$

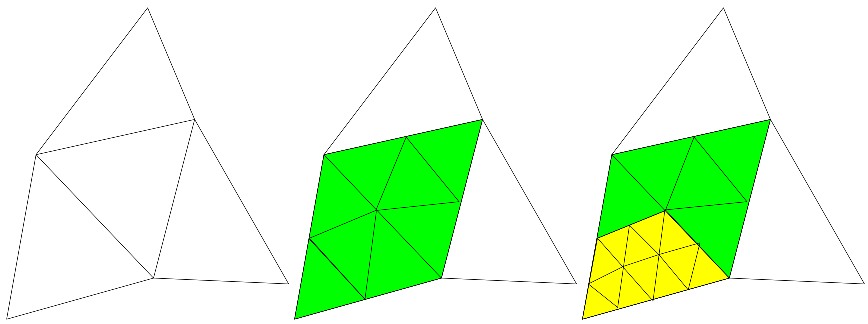

Figure 9: Refinement levels

Since differential operators are local operators (that is, if $u(\mathbf{x}) = v(\mathbf{x})$ for all $\mathbf{x}$ near $\mathbf{x}^*$, then $\mathcal{L}u(\mathbf{x}^*) = \mathcal{L}v(\mathbf{x}^*)$), this residual can be estimated on each element and its boundary. If V_h is the finite element space (generated by all basis functions over all elements in the triangulation), then the Galerkin method already implies that $\langle \mathcal{L}u - b, w \rangle_{V \times V'} = 0$ for all $w \in V_h$. So we need to go beyond just the basis functions of V_h in order to estimate the error in the solution. For the problem

$$\begin{aligned} -\nabla \cdot (\alpha \nabla u) &= f && \text{in } \Omega, \\ u &= g && \text{on } \partial\Omega, \end{aligned}$$

for example, [2, Chap. 9] develops the *a posteriori* residual estimate

$$\|e_h\|_{H^{-1}(\Omega)} \leq \text{constant} \left[\sum_K \|\nabla \cdot (\alpha \nabla u_h) + f\|_{L^2(K)}^2 \, h_K^2 + \sum_e \|[\alpha \mathbf{n} \cdot \nabla u_h]_{\mathbf{n}}\|_{L^2(e)}^2 \, h_e \right]^{1/2}$$

where K ranges over all triangles in the triangulation, and e ranges over all edges in the triangulation in the interior of the domain Ω. The quantity $[\alpha \mathbf{n} \cdot \nabla u_h]_{\mathbf{n}}$ is the jump in the value of $\alpha \mathbf{n} \cdot \nabla u_h$ between the values in the two elements incident to the edge e; $\mathbf{n}$ is the unit normal vector to the edge e. Note that it does not matter which sign is chosen for $\mathbf{n}$ in determining $\|[\alpha \mathbf{n} \cdot \nabla u_h]_{\mathbf{n}}\|_{L^2(e)}^2$. Also note that h_K is the diameter of triangle K, which is equal to the length of the longest edge; h_e is the length of edge e.

For a triangle K in the triangulation, we can use

$$\left[\|\nabla \cdot (\alpha \nabla u_h) + f\|_{L^2(K)}^2 \, h_K^2 + \sum_e \|[\alpha \mathbf{n} \cdot \nabla u_h]_{\mathbf{n}}\|_{L^2(e)}^2 \, h_e \right]^{1/2}$$

where e ranges over the edges of K as an estimate of the error due to the triangle K. If this exceeds a threshold, then that triangle should be marked for refinement. Since the basis functions of refined triangles associated with the common boundary between refined and unrefined triangles must be made "slaves" to the basis functions on the adjacent unrefined triangles, we should create an additional "buffer" region of refined triangles. Thus: a triangle should also be refined if it is adjacent to a triangle with an excessively large error estimate.

7.1 Representation of refined mesh

The refinement of a single element is represented by the standard refinement, which is a triangulation of the reference element (`p_refref`,`t_refref`) (here "refref" indicates "reference element refinement"), being the points and triangles of the triangulation in the manner of (`p`,`t`) described above. We need a data structure similar to that for element types: in order to join refined triangles from different master triangles, we need to associate a unique geometric feature of the master triangle to each node of the refinement. We also need to identify permuations of nodes that occur due to the permutations of the geometric features. These should work very much like `pxfeature()`, `nvars` and `flist` in the element type data structures. (Here we replace `nvars` with `npts`.) This will give us a "refref" or "reference element refinement" data structure. An example reference refinement is shown in Figure 10. Note that for this reference refinement using piecewise linear element (`lin2d_elt()`) we have

$$B = \begin{bmatrix} 1 & & & & 1/2 & 1/2 \\ & 1 & & 1/2 & & 1/2 \\ & & 1 & 1/2 & 1/2 & \end{bmatrix}.$$

87 ⟨*reference refinement example* 87⟩≡

```
p_refref = [0 0; 1 0; 0 1; 1/2 1/2; 0 1/2; 1/2 0];
t_refref = [1 5 6; 2 4 6; 3 4 5; 4 5 6];
npts     = [1;     1;     1;     1;     1;     1];  % number of points in each geom
flist    = [1 0 0; 2 0 0; 3 0 0; 2 3 0; 1 3 0; 1 2 0];
Brefref  = [1 0 0   0 1/2 1/2;
            0 1 0 1/2   0 1/2;
            0 0 1 1/2 1/2   0];
refref2d = struct('p',p_refref,'t',t_refref,'npts',npts,'flist',flist, ...
                  'Brefref',Brefref,'pxfeature',@px_refref1);
```

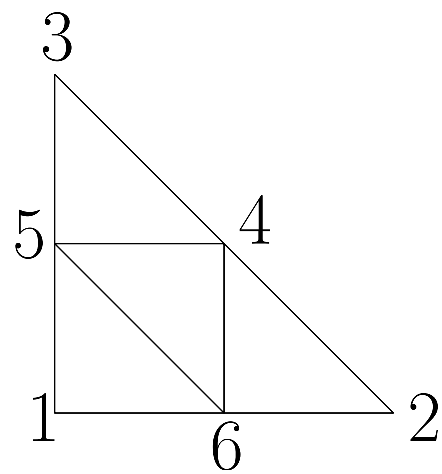

Figure 10: Reference refinement example

Note that the points in `p_refref` must occur in the order described by `flist` and `npts`. Also, if a vertex of the reference refinement is also a vertex of the reference element, then there can only be one point associated with that geometric feature: if `flist(k,:)` represents a vertex (of the reference element), then `npts(k)` must be one.

The relationship between the basis functions on the reference element, and the basis functions on the standard refinement is given by the $[b_{ij}]$ matrix `B_refref`. The indexes into this matrix is given by the standard ordering of basis functions as defined by `get_Aphihat()` for i, but for j we can use the basis function numbers assigned by `create_fht()`.

A test mesh for applying the refinement to is shown in Figure 11. The dashed lines show the refined mesh. The triangulation of this test mesh is given below.

88 ⟨*refinement test triangulation* 88⟩≡

```
p_testref = [0 0; 1.5 0; 2.5 0; 3.5 0; 3.5 1; ...
    2.5 0.7; 1.5 1; 0 1; 1.5 2; 2.5 1.5];
t_testref = [1 2 7; 2 7 6; 2 3 6; 4 3 6; 4 5 6; ...
    10 5 6; 6 10 7; 1 7 8; 8 7 9; 7 9 10];
```

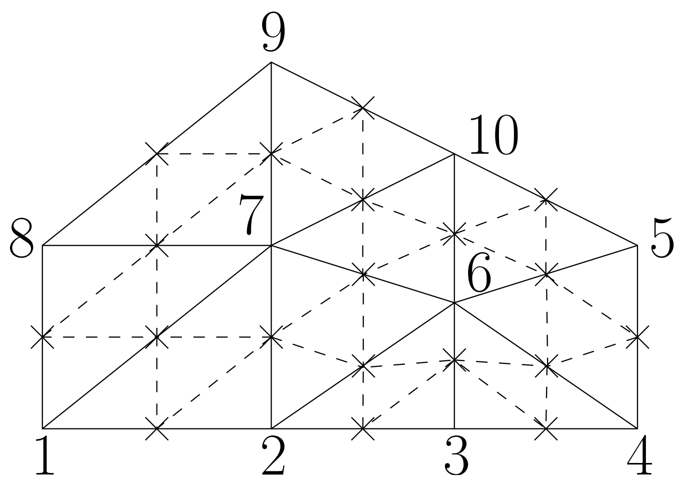

Figure 11: Test mesh for refinement

7.2 Generating refined meshes

The algorithm to create the refined mesh basically involves processing each triangle in the "master triangulation". We process each geometric feature of each master triangle using an initially empty hash table of geometric features (`rfht`). For each geometric feature that is not in the hash table, and is not a vertex, we create new points according to the points associated with that geometric feature of the reference refinement. The geometric feature is then added as a key to the hash table; the value of the key in the table is the ordered list of point indexes in the refined triangulation. For each master triangle to be refined, we need to create a map (`p_rr_trx`) that translates point indexes in the reference refinement into point indexes in the actual refined mesh. The points in the reference refinement must be transformed to actual points in the refined mesh (this involves the transformation $\widehat{\mathbf{x}} \mapsto T\widehat{\mathbf{x}} + \mathbf{b} = \mathbf{x}$.

89 $\langle$*filelist* 6b$\rangle{+}{\equiv}$

```
create-refinement.m \
```

90 ⟨*create-refinement.m* 90⟩≡

```
function [p_ref,t_ref,master_ref,idx_ref,rfht] = ...
    create_refinement(p,t,refine_list,refref)
% function [p_ref,t_ref,master_ref,idx_ref,rfht] = ...
%    create_refinement(p,t,refine_list,refref)
%
% Create refined mesh using reference refinement refref.
% (p,t) represent the unrefined "master" triangulation,
% where refine_list is the list of triangle indexes that are
% to be refined: the triangles to be refined are
%    t(refine_list(i),:), i = 1, ..., length(refine_list).
% The returned items are:
%    (p_ref,t_ref) are the triangulation of the refined mesh
%    master_ref(i) is the row index into t of the master triangle
%         of triangle t_ref(i,:)
%    idx_ref(i) is the row index into refref.t identifies
%         which sub-triangle of triangle master_ref(i) is t_ref(i,:).
% Note that idx_ref(i) == 0 iff t_ref(i,:) is an unrefined triangle.
np = size(p,1);
rfht = containers.Map('KeyType','int64','ValueType','any');
rr_npts  = refref.npts;
rr_flist = refref.flist;
p_ref = p;  % include all points of unrefined triangles
t_ref = [];
master_ref = [];
nt_ref = 0;
np_ref = size(p,1);
ismaster = zeros(size(t,1),1);
ismaster(refine_list) = 1;
p_rr_trx = zeros(size(refref.p,1),1);
idx_ref = [];
for i = 1:size(t,1)
    % master_triangle_i = t(i,:)
    % master_triangle_points = [p(t(i,1),:); p(t(i,2),:); p(t(i,3),:)]
    p_rr_trx(:) = 0;
    if ismaster(i)
        % vertices of master triangle
        p1 = p(t(i,1),:);  p2 = p(t(i,2),:);  p3 = p(t(i,3),:);
        % transformation from ref element to master triangle
        T = [p2'-p1', p3'-p1'];
```

```
    b = p1';
    % transform points in reference refinement
    % Tp_rr = zeros(size(refref.p));
    % for j = 1:size(refref.p,1)
    %     Tp_rr(j,:) = refref.p(j,:)*T' + b';
    % end
    Tp_rr = bsxfun(@plus,refref.p*T',b');
    p_rr_idx = 1; % index into refref.p
    % for each geometric feature of the master triangle ...
    for k = 1:size(rr_flist,1)
        if length(find(rr_flist(k,:))) ~= 1 % not a vertex
            % if geometric feature not in hash table, add the points
            f = rr_flist(k,:);
            f = f(find(f));
            [f,px] = sort(t(i,f));
            fref = get_feature_ref(f,np);
            pt_px = refref.pxfeature(px);
            if ~ isKey(rfht,fref)
                pt_list = np_ref + (1:refref.npts(k));
                rfht(fref) = pt_list;
                p_ref  = [p_ref; Tp_rr(p_rr_idx:p_rr_idx+refref.npts(k)-1,:)];
                np_ref = np_ref + refref.npts(k);
            else % isKey(rfht,fref)
                pt_list = rfht(fref);
            end % if
            p_rr_trx(p_rr_idx:p_rr_idx+refref.npts(k)-1) = pt_list(pt_px);
        else % is a vertex of a master triangle, so use the corresponding
             % vertex of the master triangle
            p_rr_trx(p_rr_idx) = t(i,rr_flist(k,1));
        end % if
        p_rr_idx = p_rr_idx + refref.npts(k);
    end % for k
    % p_rr_trx

    % now add the refined triangles to the mesh
    t_ref   = [t_ref; p_rr_trx(refref.t)];
    idx_ref = [idx_ref; (1:size(refref.t,1))'];
    nt_ref  = nt_ref + size(refref.t,1);
    master_ref = [master_ref, i*ones(1,size(refref.t,1))];
else % ~ ismaster(i)
```

```
        % no extra points, just one extra triangle
        nt_ref = nt_ref + 1;
        t_ref = [t_ref; t(i,:)];
        master_ref(nt_ref) = i;
        idx_ref(nt_ref) = 0; % not a refined triangle
    end % if ismaster(i)
end % for i
```

The next task is to determine the edges (or faces in 3-D) common to both the unrefined and refined master triangles. The main idea is to find all edges of triangles that appear exactly once in both the refined and unrefined sets of triangles, but appear exactly twice in the union of these sets of edges.

92 ⟨*filelist* 6b⟩+≡

```
get-internal-boundary2d.m \
```

93 ⟨*get-internal-boundary2d.m* 93⟩≡

```
function [bedges,bnodes,t_idx1,t_idx2] = get_internal_boundary2d(t,t_list)
% function [bedges,bnodes,t_idx1,t_idx2] = get_internal_boundary2d(t,t_list)
%
% Returns the boundary between the triangles in t_list and its complement.
% t_list is a list of row indexes i into t(i,j)
% bedges is an m x 2 array listing the edges in the boundary.
% bnodes is a  p x 1 array listing the nodes in the boundary.
% t_idx1(i) is the triangle containing bedges(i) in t_list.
% t_idx2(i) is the triangle containing bedges(i) in the complement of t_list.
t_list = t_list';
% compute complement of t_list
tf = zeros(size(t,1),1);
tf(t_list) = 1;
ct_list = find(tf == 0);
% compute boundary of each part ...
[bedges1,bnodes1,t_idx1a] = boundary2d(t( t_list,:));
[bedges2,bnodes2,t_idx2a] = boundary2d(t(ct_list,:));
% ... and find the common part
temp = sortrows([bedges1, t_list(t_idx1a);
                 bedges2,ct_list(t_idx2a)+size(t,1)]);
[temp2,idx1] = unique(temp(:,1:2),'rows','first');
[temp2,idx2] = unique(temp(:,1:2),'rows','last');
difflist = find(idx1 ~= idx2);
bedges = temp(idx1(difflist),1:2);
t_idx1 = temp(idx1(difflist),3)';
t_idx2 = temp(idx2(difflist),3)' - size(t,1);
bnodes = unique(sort(bedges(:)));
```

7.3 Combining multiple levels of refinement in matrix assembly

7.4 Error estimation and identification of triangles to refine

8 Output and visualization

8.1 Visualization

Visualization of the results is greatly simplified by Matlab's `trimesh()` function. For example, to see a triangulation as given by (`p`,`t`), we use

94a ⟨*trimesh-example* 94a⟩≡
```
trimesh(t,p(:,1),p(:,2))
```

8.1.1 Visualizing solutions (and mesh functions)

One way of seeing the solution of a PDE (or some other quantity defined on a mesh for a certain element) is to get the list of variables associated with each point and plot those variable values. This will work for scalar element types of Lagrange type (such as the piecewise linear, quadratic, and cubic elements described in Sections 5.1, 5.3, and 5.4). We can use the following routine:

94b ⟨*filelist* 6b⟩+≡
```
get-pvlist.m \
```

94c ⟨*get-pvlist.m* 94c⟩≡
```
function pvlist = get_pvlist(fht,np)
% function pvlist = get_pvlist(fht,np)
%
% Get list of variable indexes for the points.
% That is, pvlist(i) is the variable number for the point
% in the triangulation with index i.
% As usual, np is the number of points.
pvlist = zeros(np,1);
for i = 1:np
    pvlist(i) = fht(get_feature_ref(i,np));
end
```

Then we can view the solution via

95a ⟨*visualization-example* 95a⟩≡
```
u = ...;  % compute solution
np = size(p,1);
pvlist = get_pvlist(fht,np);
trimesh(t,p(:,1),p(:,2),u(pvlist))
```

8.1.2 Boundary visualization

There are two functions for boundary visualization: one which plots just the boundary and one with the boundary and normal vectors.

95b ⟨*filelist* 6b⟩+≡
```
plot-boundary2d.m \
```

95c ⟨*plot-boundary2d.m* 95c⟩≡
```
function plot_boundary2d(p,t,bb)
% function plot_boundary2d(p,t,bb)
%
% Plots the boundary of a mesh in 2D given by p and t.
% The i'th point is p(i,:), and triangle j is given by
% points with indexes t(j,1), t(j,2) & t(j,3).
%
% The bounding box is given in bb = [xmin, xmax, ymin,ymax].
%
% See distmesh.m etc.
[bedges,bnodes] = boundary2d(t);
bdry_tri = [bedges(:,1),bedges(:,2),bedges(:,2)];
triplot(bdry_tri,p(:,1),p(:,2));
axis(bb)
```

And now with normal vectors (use with equal axes to preserve orthogonality):

95d ⟨*filelist* 6b⟩+≡
```
plot-boundary2dwnormals.m \
```

96a ⟨*plot-boundary2dwnormals.m* 96a⟩≡

```
function plot_boundary2dwnormals(p,t,bb)
% function plot_boundary2dwnormals(p,t,bb)
%
% Plots the boundary of a mesh in 2D given by p and t.
% The i'th point is p(i,:), and triangle j is given by
% points with indexes t(j,1), t(j,2) & t(j,3).
% The normals are also plotted from the center of each edge.
% The length of the normal vectors is the length of the edge.
%
% The bounding box is given in bb = [xmin, xmax, ymin,ymax].
%
% See distmesh.m etc.
[bedges,bnodes,normals] = boundary2d_2(p,t);
bdry_tri = [bedges(:,1),bedges(:,2),bedges(:,2)];
midpts = 0.5*(p(bedges(:,1),:)+p(bedges(:,2),:));
len_edges = sqrt(sum((p(bedges(:,1),:)-p(bedges(:,2),:)).^2,2));
arrowpts = midpts + diag(sparse(len_edges))*normals;
hold on
triplot(bdry_tri,p(:,1),p(:,2));
arrow(midpts,arrowpts);
axis(bb);
hold off
```

8.2 Refined output

The results of `trimesh()` do not capture the higher order behavior of the mesh function or solution as it assumes the function is piecewise linear. To capture the quadratic or higher order behavior we need to create a submesh and plot on the submesh. This can be done by, for example, creating a sub-mesh for the reference element, and then replacing each triangle in the triangulation with the reference element's sub-mesh transformed in the usual way ($\widehat{\mathbf{x}} \mapsto \mathbf{x} = T_K\widehat{\mathbf{x}} + \mathbf{b}_K$). The values can then be computed on the sub-mesh of the original triangulation, and the result plotted using `trimesh()`.

First we have code to create a sub-mesh for the reference element.

96b ⟨*filelist* 6b⟩+≡

```
ref-triangle-submesh.m \
```

97 ⟨*ref-triangle-submesh.m* 97⟩≡

```
function [p,t] = ref_triangle_submesh(n)
% function [p,t] = ref_triangle_submesh(n)
%
% Creates a standard mesh on the reference triangle
% (vertices at (0,0), (1,0) and (0,1)).
% n+1 is the number of grid points on each edge
% Generate points
p = zeros(n*(n+1)/2,2);
k = 1;
for i = 0:n
    for j = 0:n-i
        p(k,:) = [i,j];
        k = k+1;
    end
end
p = p / n;
% Create triangulation
t = zeros(n*n,3);
k = 1;
idx = 1;
for i = 0:n-1
    for j = 0:n-i-1
        if j > 0
            t(k,:) = [idx, idx+n-i, idx+n-i+1];
            k = k+1;
        end
        t(k,:) = [idx, idx+n-i+1, idx+1];
        k = k+1;
        idx = idx+1;
    end % for j
    idx = idx+1;
end % for i
```

Once a sub-mesh for the reference element has been created, the creation of the sub-mesh of the original triangulation, and the computation of the values at the sub-mesh nodes, is handled by the following code. Let $g(\mathbf{x})$ be the mesh-based function represented by the values of `vars`. Then `vals(i,1)` contains $g(\widetilde{\mathbf{x}}_i)$ where $\widetilde{\mathbf{x}}_i$ where $\widetilde{\mathbf{x}}_i$ is point i in the sub-mesh of the original triangulation, provided `elt` is a scalar element type. However, `vals(i,r)` contains $\mathcal{A}g(\widetilde{\mathbf{x}}_i)$ where $\mathcal{A}$ is the rth operator (of order $\leq$`order`) from the list of the operators for `elt`. In this way, derivative information can be also displayed; for vector element types, the different components can also be evaluated and displayed. For problems in elasticity, for example, components of the stress and strain tensors can be computed and displayed this way.

Note that in the submesh created, the submeshes for different triangles are separated; that is, variables in the submesh are not shared between different submeshes corresponding to different triangles.

98 ⟨*filelist* 6b⟩+≡

```
get-submesh-vals.m \
```

99 ⟨*get-submesh-vals.m* 99⟩≡

```
function [pv,tv,vals] = get_submesh_vals(p,t,fht,elt,vars,p_ref,t_ref,order)
% function [pv,tv,vals] = get_submesh_vals(p,t,fht,elt,vars,p_ref,t_ref,order)
%
% Return triangulation (pv,tv) and values (vals) for
% the given variable values (vars).
% Each triangle in the master triangulation (p,t)
% is subdivided according to the triangulation given for
% the reference element (p_ref,t_ref).
%
% The relationship between the elements of the master
% triangulation (p,t) is given by fht and elt.
%
% Values and derivatives up to the given order are returned in vals.
% Get the values for the basis functions on p_ref
Aphihat = cell(size(p_ref,1),1);
for i = 1:size(p_ref,1)
    Aphihat{i} = elt.get_Aphihat(p_ref(i,:),order);
end
np = size(p,1);
np_ref = size(p_ref,1);
nt_ref = size(t_ref,1);
pv = zeros(size(t,1)*np_ref,2);
tv = zeros(size(t,1)*nt_ref,3);
vals = zeros(size(t,1)*np_ref,size(Aphihat{1},2));
for i = 1:size(t,1)
    T = [p(t(i,2),:)'-p(t(i,1),:)', p(t(i,3),:)'-p(t(i,1),:)'];
    b = p(t(i,1),:)';
    pv((i-1)*np_ref+(1:np_ref),:) = p_ref*T'+ones(np_ref,1)*b';
    tv((i-1)*nt_ref+(1:nt_ref),:) = t_ref+(i-1)*np_ref;
    % get variable indexes
    [vlist,slist] = get_var_triangle(t(i,:),fht,elt,np);
    % basis function values
    for j = 1:np_ref
        Aphival = elt.trans_Aphihat(T,Aphihat{j},order);
        vals((i-1)*np_ref+j,:) = (vars(vlist)'.*slist)*Aphival;
    end % for j
end % for i
end % function
```

9 Geometric feature hash tables

Throughout this code we need hash tables keyed by geometric features. The initial code used the `containers.Map` structure that is available in Matlab. There are a few problems with this. One is that these are only keyed by numbers or strings. Hence a `get_feature_ref()` function was needed to obtain a unique integer for each geometric feature. The number of points (`np`) parameter was needed for this to work. Overflow can occur when the numbers become too large, as is likely to happen for large three-dimensional triangulations. Instead the key should be a geometric feature. The other issue is that the `containers.Map` may not be a stable part of Matlab. Instead, we should base the hash table on more basic Matlab features.

The hash table consists of a hash function `hash()`, an `index` array, a `next` array, and `key` and `val` cell arrays. If `h = hash(item)`, we set `idx = index(h)`. If `idx` is zero, the item is not in the hash table. Otherwise we then check `key(idx)` to see if this is equal to `item`; if so, then we return `val(idx)`. If `key(idx)` is not `item`, we set `idx = next(idx)`.

!!! continue here !!!

10 Utility routines

This is where we put routines that are generally useful.

The following is useful for anonymous functions where conditionals are needed.

100 ⟨*filelist* 6b⟩+≡

```
  ifte.m \
```

101a ⟨*ifte.m* 101a⟩≡

```
function val = ifte(condition,affirmative,negative)
% function val = ifte(condition,affirmative,negative)
%
% Returns affirmative if condition is true (not zero)
% and negative otherwise.
% This is useful for anonymous functions.
if condition
    val = affirmative;
else
    val = negative;
end
```

There is also a vectorized version of this where all the inputs are vectors of equal size.

101b ⟨*filelist* 6b⟩+≡

```
iftev.m \
```

101c ⟨*iftev.m* 101c⟩≡

```
function val = iftev(condition,affirmative,negative)
% function val = iftev(condition,affirmative,negative)
%
% Returns affirmative(i) if condition(i) is true (not zero)
% and negative(i) otherwise.
% This is useful for anonymous functions.
aff_idx = find(condition);
neg_idx = find(~condition);
val = zeros(size(condition));
val(aff_idx) = affirmative(aff_idx);
val(neg_idx) = negative(neg_idx);
```

For example, the componentwise maximum of two vectors `a` and `b` can be computed by

```
iftev(a<b,a,b)
```

11 Installation

This article is a simple test for using Noweb for mixing code and documentation. One difficulty with using Noweb is that there is no automatic way of generating all code files. However, we can use a *Makefile* to identify all actual code files and so that we can obtain all the code files by means of the following code fragment:

102a ⟨*gen-all-files* 102a⟩≡
```
notangle -t8 -RMakefile pde-code.nw > Makefile
make all
```

The "`-t8`" option is to ensure that tabs are passed without conversion to spaces.

The *Makefile* will know which files to create and the procedure for creating them. The code chunk *filelist* contains the list of files to create (on separate lines but with "\" at the end of each line).

102b ⟨*Makefile* 102b⟩≡
```
files1 = ⟨filelist 6b⟩
files2 = pde-code.tex filelist
files = $(files1) $(files2)
source = pde-code.nw
all: $(files)
$(files): $(source)
	notangle -R$@ $(source) > $@
pde-code.tex: $(source)
	noweave -delay -index $(source) > $@
```

Unfortunately Noweb does not like underscores (_) while Matlab M-files cannot have dashes (-) in the file name. So in this file all the Matlab source files have underscores replaced by dashes. To fix that and be able to run in Matlab, a script has been included to help you do this:

102c ⟨*filelist* 6b⟩+≡
```
filenamehack.bash \
```

103a ⟨*filenamehack.bash* 103a⟩≡
```
#!/bin/bash
cat filelist | tr -d '\\' | tr -d \\r > templist
for f in `cat templist`
do
  if [[ "$f" =~ .*-.*\.m ]]
  then mv "$f" `echo "$f" | sed -e s/-/_/g`
  fi
done
rm templist
```

To use this script in Unix (or Cygwin or ...), just use the command "`bash filenamehack.bash`". In Cygwin, you may need to remove carriage returns from the script, just as for the `Makefile` (see above).

There is an annoying "feature" in Cygwin where carriage returns are inserted into files (for compatibility with Microsoft Windows, I presume) which causes problems with make. So to remove them, use

```
tr -d \\r < Makefile > temp; mv temp Makefile
```

In Unix systems we can use a shell script to automate the entire process. One such script is below (which uses the above scripts and `Makefile`):

103b ⟨*filelist* 6b⟩+≡
```
lyx2code.bash \
```

103c ⟨*lyx2code.bash* 103c⟩≡
```
#!/bin/bash
lyx -e literate $1.lyx
notangle -t8 -RMakefile $1.nw > Makefile
make all
bash filenamehack.bash
```

12 Test code

12.1 Checking consistency of element values and derivatives

Checking the consistency between the element values and the derivatives the element structure provides is an important part of testing. The following code checks consistency of derivatives and values up to the 2nd order derivatives for two-dimensional scalar elements. this uses centered difference approximations; **d** should be small. This code returns

$$\frac{\phi_i(\mathbf{x}+\mathbf{d}) - \phi_i(\mathbf{x}-\mathbf{d}) - 2\mathbf{d}^T \nabla \phi_i(\mathbf{x})}{\|\mathbf{d}\|}$$

and

$$\frac{\nabla \phi_i(\mathbf{x}+\mathbf{d}) - \nabla \phi_i(\mathbf{x}-\mathbf{d}) - 2 \mathrm{Hess}\, \phi_i(\mathbf{x})\, \mathbf{d}}{\|\mathbf{d}\|}$$

where $\mathrm{Hess}\, \psi(\mathbf{x})$ is the Hessian matrix of 2nd order partial derivatives: $(\mathrm{Hess}\, \psi(\mathbf{x}))_{pq} = \partial^2 \psi / \partial x_p \partial x_q(\mathbf{x})$. Provided **d** is small on the scale of **x**, both ratios should be $\mathcal{O}(\|\mathbf{d}\|^2)$, and so be small compared to $\|\mathbf{d}\|$. If you are not sure how small is "small", reduce the size of **d** by a factor of two or ten, and repeat the computation. The returned values should be reduced by a factor of four or a hundred, respectively.

104 ⟨*filelist* 6b⟩+≡

```
  check-derivs.m \
```

105a ⟨*check-derivs.m* 105a⟩≡

```
function [err_dphi,err_ddphi] = check_derivs(Aphifunc,x,d)
% function [err_dphi,err_ddphi] = check_derivs(Aphifunc,x,d)
%
% Returns errors in derivative test: err_dphi is the error vector for
% (phi(x+d)-phi(x-d)-2*grad phi(x)'*d)/norm(d),
% err_ddphi is the error vector for
% (grad phi(x+d)-grad phi(x-d)-2*Hess phi(x)*d)/norm(d).
%
% Assumes scalar element: order of rows:
% [phi(x), (d/dx1)phi(x), (d/dx2)phi(x), (d^2/dx1^2)phi(x), ...
% (d^2/dx1.dx2)phi(x), (d^2/dx2^2)phi(x)]
Aphivalx   = Aphifunc(x,2);
Aphivalxpd = Aphifunc(x+d,2);
Aphivalxmd = Aphifunc(x-d,2);
phixpd = Aphivalxpd(:,1);
phixmd = Aphivalxmd(:,1);
dphix  = Aphivalx(:,2:3);
err_dphi = (phixpd-phixmd-2*dphix*d)/norm(d);
dphixpd = Aphivalxpd(:,2:3);
dphixmd = Aphivalxmd(:,2:3);
ddphix  = Aphivalx(:,4:6);
err_ddphi = (dphixpd-dphixmd-2*(d(1)*ddphix(:,1:2)+d(2)*ddphix(:,2:3)))/norm(d);
```

12.2 Testing basic geometric operations

For testing basic geometric operations, we need a simple example of a mesh that includes interior nodes and non-congruent triangles.

105b ⟨*filelist* 6b⟩+≡

```
test-geom.m \
```

105c ⟨*test-geom.m* 105c⟩≡

```
p = [0 0; 1 0; 0 1; 1 2.5; 1.5 2.5; ...
      1 1; 2 0; 2 1; 2.5 2.5; 3 3; 3 2; 3 1];
t = [1 2 6; 8 11 9; 11 10 9; 7 8 12; 3 1 6; 6 4 3; ...
      2 7 6; 8 11 12; 7 6 8; 4 5 6; 5 8 6; 9 5 8];
```

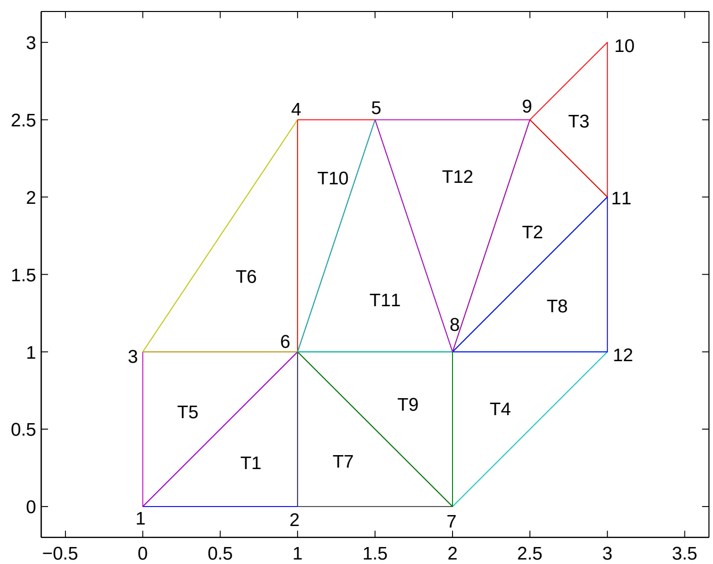

Figure 12: Test geometry (simple)

This triangulation is shown in Figure 12. Note that the vertices are labelled by their vertex number, and the triangles are labelled by their triangle number after a "T".

The correct boundary information is given below (for `boundary2d()`):

106a ⟨*test-geom.m* 105c⟩+≡

```
[bedges,bnodes,t_index] = boundary2d(t)
bedges_exact  = [1 2; 1 3; 2 7; 3 4; 4 5; 5 9; 7 12; 9 10; 10 11; 11 12]
t_index_exact = [1  ; 5  ; 7  ; 6  ; 10 ; 12 ; 4   ; 3   ; 3    ; 8    ]
bnodes_exact  = [1; 2; 3; 4; 5; 7; 9; 10; 11; 12] % only 6 & 8 are not
```

To see how this is used for creating the variables, we can use the piecewise quadratic Lagrange element. Recall that for this element type, there is a variable associated with each vertex and a variable associated with each edge.

106b ⟨*test-geom.m* 105c⟩+≡

```
quad2d = quad2d_elt()
fht_quad2d = create_fht(p,t,quad2d)
np = size(p,1)
```

12.3 Testing overall system

13 To do

Here is a list of things that would be worth doing with this code.

- Properly and correctly implement a C^1 element, which could be Bell's triangle, Argyris element, or a composite element like the Hsieh–Clough–Tocher (HCT) element.

- Implement equations of linearized elasticity.

- Implement adaptive refinement. I have some ideas for doing that using hanging nodes and non-conforming meshes. The trick is to post-process the assembled matrices by sparse matrix–matrix multiplies involving matrices defining the relationship between the hanging nodes and the "real" nodes.

- Implement discontinuous Galerkin methods. The elements are easy to create, but there would need to be new assembly routines which involve integrations over edges and faces.

- Implement three dimensional version. This is already started with three dimensional elements, but we need a three-dimensional assembly routines, and three-dimensional boundary functions.

- More testing code for testing items from bottom up.

- Optional inputs/outputs for A and $\mathbf{b}$ in assembly routines. They can be input as empty matrices (`[]`) to indicate "do not create".

- Replace the `container.Map` structure and `get_feature_ref()` with something more appropriate for the feature hash table (`fht`). A problem with the current approach is that with three-dimensional meshes there is a very good chance of overflow, even with the use of `int64` keys for the hash table.

- Modify to handle convection dominated problems (e.g., high Reynolds number Stokes' problems).

14 Conclusions

108a ⟨*filelist* 6b⟩+≡
```
dummy.txt
```

108b ⟨*dummy.txt* 108b⟩≡
```
This must be the last code scrap.
That's all folks!
```

References

[1] K. E. Atkinson. *An Introduction to Numerical Analysis*. J. Wiley and sons, 1978. 1st edition.

[2] Susanne C. Brenner and L. Ridgway Scott. *The mathematical theory of finite element methods*, volume 15 of *Texts in Applied Mathematics*. Springer-Verlag, New York, second edition, 2002.

[3] D. A. Dunavant. High degree efficient symmetrical Gaussian quadrature rules for the triangle. *Internat. J. Numer. Methods Engrg.*, 21(6):1129–1148, 1985.

[4] Karin Gatermann. The construction of symmetric cubature formulas for the square and the triangle. *Computing*, 40(3):229–240, 1988.

[5] Per-Olof Persson and Gilbert Strang. A simple mesh generator in Matlab. *SIAM Rev.*, 46(2):329–345 (electronic), 2004.

[6] Johann Radon. Zur mechanischen Kubatur. *Monatsh. Math.*, 52:286–300, 1948.

Index