3. Automating Basis Functions and Assembly

Anders Logg
logg@tti-c.org

Toyota Technological Institute at Chicago

Sixth Winter School in Computational Mathematics
Geilo, March 5-10 2006

Outline

Automating the tabulation of basis functions
- Finite element basis functions
- Tabulating polynomial spaces
- Tabulating spaces with constraints

Automating the assembly of the discrete system
- Assembling the global tensor
- Implementing the local-to-global mapping
- Generating the local-to-global mapping

- Chapters 4 and 6 in lecture notes

The reference finite element

The reference finite element is a triple

$$(K_0, \mathcal{P}_0, \mathcal{N}_0)$$

- K_0 is a bounded closed subset of $\mathbb{R}^d$ with nonempty interior and piecewise smooth boundary
- $\mathcal{P}_0$ is a function space on K_0 of dimension $n_0 < \infty$
- $\mathcal{N}_0 = \{\nu_1^0, \nu_2^0, \ldots, \nu_{n_0}^0\}$ is a basis for $\mathcal{P}_0'$ (the bounded linear functionals on $\mathcal{P}_0$)

The nodal basis

A special basis $\{\Phi_i\}_{i=1}^{n_0}$ for $\mathcal{P}_0$ that satisfies

$$\nu_i^0(\Phi_j) = \delta_{ij}, \quad i, j = 1, 2, \ldots, n_0$$

Implies that

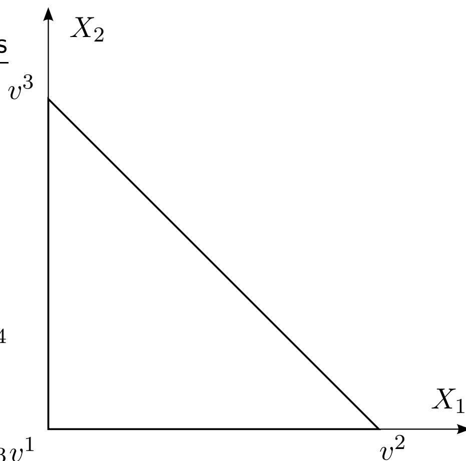
$$v = \sum_{i=1}^{n_0} \nu_i^0(v) \Phi_i$$

for any $v \in \mathcal{P}_0$

- Can be worked out analytically in simple cases
- Can be generated automatically

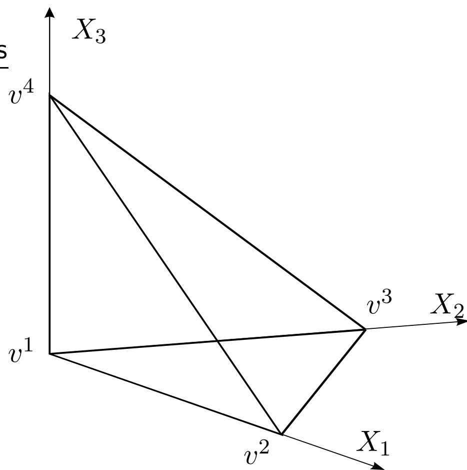
The reference triangle

PSfrag replacements



The reference tetrahedron

PSfrag replacements



$P_1(K_0)$ on the reference triangle

- K_0 is the reference triangle in $\mathbb{R}^2$
- $\mathcal{P}_0 = P_1(K_0)$ is the space of first-degree polynomials on K_0
- The nodal basis is given by

$$
\begin{aligned}
\Phi_1(X) &= 1 - X_1 - X_2 \\
\Phi_2(X) &= X_1 \\
\Phi_3(X) &= X_2
\end{aligned}
$$

$P_2(K_0)$ on the reference triangle

- K_0 is the reference triangle in $\mathbb{R}^2$
- $\mathcal{P}_0 = P_2(K_0)$ is the space of second-degree polynomials on K_0
- The nodal basis is given by

$$
\begin{aligned}
\Phi_1(X) &= (1 - X_1 - X_2)(1 - 2X_1 - 2X_2) \\
\Phi_2(X) &= X_1(2X_1 - 1) \\
\Phi_3(X) &= X_2(2X_2 - 1) \\
\Phi_4(X) &= 4X_1X_2 \\
\Phi_5(X) &= 4X_2(1 - X_1 - X_2) \\
\Phi_6(X) &= 4X_1(1 - X_1 - X_2)
\end{aligned}
$$

Generating the nodal basis

- Choose a simple (non-nodal) basis $\{\Psi_i\}_{i=1}^{n_0}$ for $\mathcal{P}_0$
- Orthogonal bases exist for the reference triangle and reference tetrahedron
- Recurrence relations for evaluation of basis functions and derivatives
- Refer to $\{\Psi_i\}_{i=1}^{n_0}$ as the *prime basis*

Generating the nodal basis

Write each Φ_i as a linear combination of the prime basis functions:

$$\Phi_i = \sum_{j=1}^{n_0} \alpha_{ij} \Psi_j, \quad i = 1, 2, \ldots, n_0$$

The condition $\nu_i^0(\Phi_j) = \delta_{ij}$ gives

$$\delta_{ij} = \nu_i^0(\Phi_j) = \sum_{k=1}^{n_0} \alpha_{jk} \nu_i^0(\Psi_k), \quad i, j = 1, 2, \ldots, n_0$$

This is a linear system for α

Generating the nodal basis

Expansion coefficients α for the nodal basis obtained by solving a linear system:

$$\mathcal{V}\alpha^\top = I$$

The matrix $\mathcal{V}$ is given by

$$\mathcal{V}_{ij} = \nu_i^0(\Psi_j), \quad i,j = 1,2,\ldots,n_0$$

- Need to evaluate the nodes on the prime basis
- Can be automated

Generating the nodal basis

- Implement the prime basis $\{\Psi_i\}_{i=1}^{n_0}$
- Compute the (Vandermonde) matrix $\mathcal{V}$
- Solve the linear system $\mathcal{V}\alpha^\top = I$
- Nodal basis can be evaluated at any point
- Derivatives of nodal basis can be evaluated at any point
- Operations on the nodal basis translated to linear algebraic operations on the coefficients

Constrained spaces

- $\mathcal{P}_0 \subset P_q(K_0)$
- Basis functions or derivatives constrained
- Examples:
  - Raviart–Thomas
  - Brezzi–Douglas–Fortin–Marini
  - Arnold–Winther
- Need to compute a constrained prime basis
- Compute nodal basis from prime basis as before

Constrained spaces in p-refinement

- $\mathcal{P}_0 \subset P_q(K_0)$
- Constrained to $P_{q-1}(\gamma_0)$ on some part γ_0 of the boundary:

$$\begin{aligned} \mathcal{P}_0 &= \{v \in P_q(K_0) : v|_{\gamma_0} \in P_{q-1}(\gamma_0)\} \\ &= \{v \in P_q(K_0) : l(v) = 0\} \end{aligned}$$

- The linear functional l is given by integration against the qth degree Legendre polynomial along γ_0:

$$l(v) = \int_{\gamma_0} v \, p_q \, \mathrm{d}s$$

Computing a constrained prime basis

In general, there may be a set of constraints:

$$l_i : P_q(K_0) \to \mathbb{R}, \quad i = 1, 2, \ldots, n_c$$

Each constraint l_i defines a null space on $P_q(K_0)$:

$$\{v \in P_q(K_0) : l_i(v) = 0\}$$

Define $\mathcal{P}_0$ as the intersection of the null spaces on $P_q(K_0)$:

$$\mathcal{P}_0 = \{v \in P_q(K_0) : l_i(v) = 0, \quad i = 1, 2, \ldots, n_c\}$$

Computing a constrained prime basis

- Let $\{\bar{\Psi}_i\}_{i=1}^{|P_q(K_0)|}$ be a basis for $P_q(K_0)$
- Let $\Psi = \sum_{i=1}^{|P_q(K_0)|} \beta_i \bar{\Psi}_i$
- Determine coefficients β by

$$0 = l_i(\Psi) = \sum_{j=1}^{|P_q(K_0)|} \beta_j l_i(\bar{\Psi}_j)$$

or

$$L\beta = 0$$

where

$$L_{ij} = l_i(\bar{\Psi}_j), \quad i = 1, 2, \ldots, n_c, \quad j = 1, 2, \ldots, |P_q(K_0)|$$

Singular value decomposition (SVD)

Compute singular value decomposition of L:

$$U^\top L V = \mathrm{diag}(\sigma_1, \sigma_2, \dots, \sigma_p)$$

where

$$\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_r > \sigma_{r+1} = \cdots = \sigma_p = 0$$

- L is a $n_c \times |P_q(K_0)|$ matrix
- U and V are orthogonal matrices
- $p = \min(n_c, |P_q(K_0)|)$
- $\mathrm{rank}(L) = r$
- $\mathrm{ran}(L) = \mathrm{span}\{u_1, u_2, \dots, u_r\}$
- $\mathrm{null}(L) = \mathrm{span}\{v_{r+1}, v_{r+2}, \dots, v_{|P_q(K_0)|}\}$

Generating the constrained nodal basis

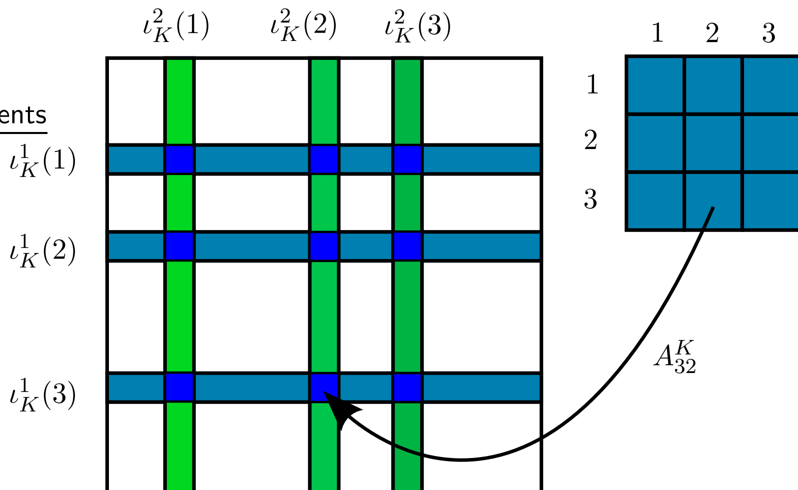
- Implement a basis $\{\bar{\Psi}_i\}_{i=1}^{n_0}$ for $P_q(K_0)$
- Compute the constraint matrix L
- Compute null space by singular value decomposition of L to obtain the prime basis
- Compute the (Vandermonde) matrix $\mathcal{V}$
- Solve the linear system $\mathcal{V}\alpha^\top = I$ to obtain the nodal basis

The assembly algorithm

$A = 0$
for $K \in \mathcal{T}$
  (Compute the element tensor A^K)
  Add A^K to A according to $\{\iota_K\}_{K \in \mathcal{T}}$
end for

- Straightforward if we know the local-to-global mappings $\{\iota_K\}_{K \in \mathcal{T}}$
- More efficient if we don't need to touch all the entries $A_{\iota_K(i)}$ for all $i \in \mathcal{I}_K$ individually

Adding the element tensor A^K



Adding the element tensor A^K

- Compute a tuple

$$\iota_K^j([1, n_K^j]) = (\iota_K^j(1), \iota_K^j(2), \ldots, \iota_K^j(n_K))$$

  for each $j = 1, 2, \ldots, r$
- Give tensors A and A^K and the tuples (arrays) to an optimized library routine
- Flatten the element tensor A^K into a contiguous array

Assembly in DOLFIN

```
for (CellIterator cell(mesh); !cell.end(); ++cell)
{
  map.update(*cell);

  a.update(map);
  a.eval(block, map);

  test_element.nodemap(test_nodes, *cell, mesh);
  trial_element.nodemap(trial_nodes, *cell, mesh);

  A.add(block, test_nodes, m, trial_nodes, n);
}
```

Adding the block of values with PETSc

```
class Matrix
{
public:
  void add(const real block[],
           const int rows[], int m,
           const int cols[], int n)
  {
    MatSetValues(A, m, rows, n, cols,
                 block, ADD_VALUES);
  }
private:
  Mat A;
};
```

Different element — different mappings

- The local-to-global mapping looks very different for different elements
- The mapping is an *algorithm*, not *data*
- A different implementation is needed for each specific element
- Can be implemented efficiently in straight-line code

The local-to-global mapping for linears on tetrahedra

```
void nodemap(int nodes[], const Cell& cell,
             const Mesh& mesh)
{
  nodes[0] = cell.vertexID(0);
  nodes[1] = cell.vertexID(1);
  nodes[2] = cell.vertexID(2);
  nodes[3] = cell.vertexID(3);
}
```

The local-to-global mapping for quadratics on tetrahedra

```
void nodemap(int nodes[], const Cell& cell,
             const Mesh& mesh)
{
  nodes[0] = cell.vertexID(0);
  nodes[1] = cell.vertexID(1);
  nodes[2] = cell.vertexID(2);
  nodes[3] = cell.vertexID(3);
  int offset = mesh.numVertices();
  nodes[4] = offset + cell.edgeID(0);
  nodes[5] = offset + cell.edgeID(1);
  nodes[6] = offset + cell.edgeID(2);
  nodes[7] = offset + cell.edgeID(3);
  nodes[8] = offset + cell.edgeID(4);
  nodes[9] = offset + cell.edgeID(5);
}
```

Generating the local-to-global mapping

- Tedious to implement by hand for each different element
- Generate the code automatically at compile-time
- Need a simple description of the nodes from which we can generate the local-to-global mapping

- FIAT provides simple description of the nodes
- FFC generates the local-to-global mapping
- DOLFIN calls the local-to-global mapping

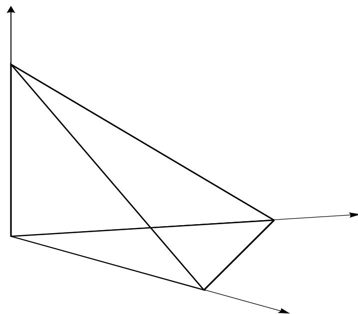
A simple description of the nodes

- Associate nodes with geometric entities:
  - vertices
  - edges
  - faces
  - cells
- Order geometric entities by topological dimension to get a dimension-independent description:
  - topological dimension 0: vertices
  - topological dimension 1: edges
  - topological dimension 2: faces, cells (triangles)
  - topological dimension 3: cells (tetrahedra)
- List the local node numbers associated with the geometric entities within each topological dimension

Specifying the nodes for linears on tetrahedra

PSfrag replacements

$d = 0$	(1) – (2) – (3) – (4)

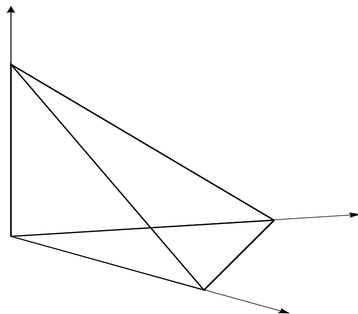


- One node is associated with each vertex
- Ordering not important

Specifying the nodes for quadratics on tetrahedra

PSfrag replacements

$d = 0$	(1) – (2) – (3) – (4)
$d = 1$	(5) – (6) – (7) – (8) – (9) – (10)



- One node is associated with each vertex and each edge
- Ordering not important

Specifying the nodes for quintics ($q = 5$) on tetrahedra

$d = 0$	$(1) - (2) - (3) - (4)$
$d = 1$	$(5, 6, 7, 8) - (9, 10, 11, 12) - (13, 14, 15, 16) -$ $(17, 18, 19, 20) - (21, 22, 23, 24) - (25, 26, 27, 28)$
$d = 2$	$(29, 30, 31, 32, 33, 34) - (35, 36, 37, 38, 39, 40) -$ $(41, 42, 43, 44, 45, 46) - (47, 48, 49, 50, 51, 52)$
$d = 3$	$(53, 54, 55, 56)$

- One node is associated with each vertex
- Four nodes are associated with each edge
- Six nodes are associated with each face
- Four nodes are associated with the cell itself
- Ordering is important!

The local-to-global mapping for quintics on tetrahedra

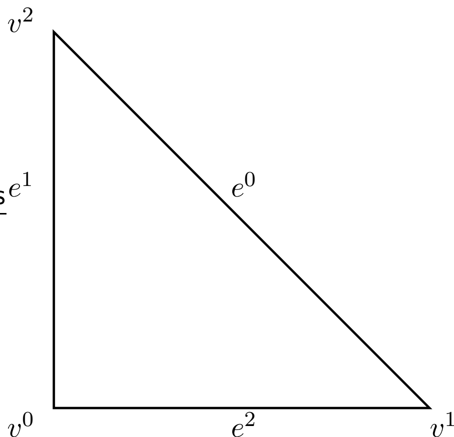
```
void nodemap(int nodes[], const Cell& cell, const Mesh& mesh)
{
  static unsigned int edge_reordering[2][4]
    = {{0, 1, 2, 3}, {3, 2, 1, 0}};
  static unsigned int face_reordering[6][6]
    = {{0, 1, 2, 3, 4, 5},
       {0, 3, 5, 1, 4, 2},
       {5, 3, 0, 4, 1, 2},
       {2, 1, 0, 4, 3, 5},
       {2, 4, 5, 1, 3, 0},
       {5, 4, 2, 3, 1, 0}};

  nodes[0] = cell.vertexID(0);
  nodes[1] = cell.vertexID(1);
  nodes[2] = cell.vertexID(2);
  nodes[3] = cell.vertexID(3);
  ...
```

The local-to-global mapping for quintics on tetrahedra

```
  ...
  int alignment = cell.edgeAlignment(0);
  int offset = mesh.numVertices();
  nodes[4]  = offset + 4*cell.edgeID(0)  + edge_reordering[alignment][0];
  nodes[5]  = offset + 4*cell.edgeID(0)  + edge_reordering[alignment][1];
  nodes[6]  = offset + 4*cell.edgeID(0)  + edge_reordering[alignment][2];
  nodes[7]  = offset + 4*cell.edgeID(0)  + edge_reordering[alignment][3];
  ...
  alignment = cell.faceAlignment(0);
  offset = offset + 4*mesh.numEdges();
  nodes[28] = offset + 6*cell.faceID(0)  + face_reordering[alignment][0];
  nodes[29] = offset + 6*cell.faceID(0)  + face_reordering[alignment][1];
  nodes[30] = offset + 6*cell.faceID(0)  + face_reordering[alignment][2];
  nodes[31] = offset + 6*cell.faceID(0)  + face_reordering[alignment][3];
  nodes[32] = offset + 6*cell.faceID(0)  + face_reordering[alignment][4];
  nodes[33] = offset + 6*cell.faceID(0)  + face_reordering[alignment][5];
  ...
  offset = offset + 6*mesh.numFaces();
  nodes[52] = offset + 4*cell.id() + 0;
  nodes[53] = offset + 4*cell.id() + 1;
  nodes[54] = offset + 4*cell.id() + 2;
  nodes[55] = offset + 4*cell.id() + 3;
}
```

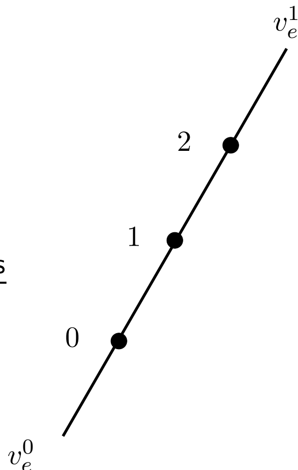
Ordering of entities on the reference triangle

PSfrag replacements



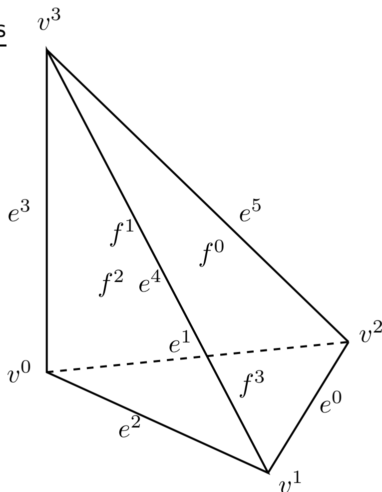
Ordering of nodes on edges

PSfrag replacements



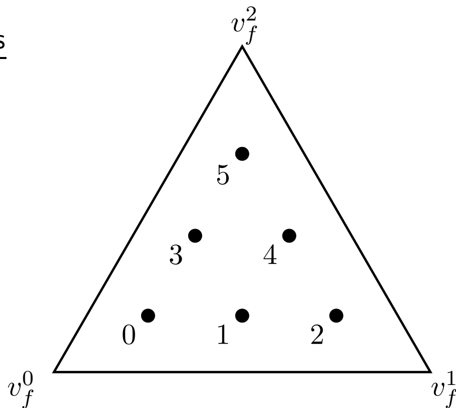
Ordering of entities on the reference tetrahedron

PSfrag replacements



Ordering of nodes on faces

PSfrag replacements



Upcoming lectures

0. Automating the Finite Element Method
1. Survey of Current Finite Element Software
2. The Finite Element Method
3. Automating Basis Functions and Assembly
4. Automating and Optimizing the Computation of the Element Tensor
5. FEniCS and the Automation of CMM
6. FEniCS Demo Session