Chapter 9

Assembling the stiffness matrix

I will now present the main computation of the finite element method, namely, the assembly of the stiffness matrix. In Chapter 6, I presented an outline of an element-oriented algorithm for assembling $K$ (see Algorithm 6.1). I can now express this algorithm in terms of the mesh data structure presented in Chapter 7.

A fundamental operation needed for implementing element-oriented algorithms is the determination of the vertices of a given triangle. To be precise, it is necessary to know, for each vertex, its coordinates, whether it is free or constrained, and its index in the list of free or constrained nodes.

The reader will recall that the mesh data structure stores, for each triangle in a mesh, (pointers to) the edges of the triangle. It also stores, for each edge, (pointers to) the endpoints of the edge. By a simple algorithm, then, one can extract pointers to the vertices of a given triangle. If the triangle is $T_k$, then, according to the notation developed earlier, these pointers are $n_{k,r}$, $r = 1, 2, 3$. Given $n_{k,1}, n_{k,2}, n_{k,3}$, the coordinates of $v_{n_{k,1}}, v_{n_{k,2}}, v_{n_{k,3}}$ can be extracted from the Nodes array. Using the same pointers, the flags indicating whether the nodes are free or constrained can be extracted from NodePtrs. This algorithm, which I call getNodes, is so fundamental that I describe it fully in Algorithm 9.1. The reader will recall that the sign of $T.\text{Elements}(k,j)$ indicates whether edge $j$ of triangle $k$ is traversed forward or backward. This fact is used in extracting the pointers to the vertices of the triangle.

As I discussed at the end of the previous chapter, the three basis functions corresponding to a given triangle $T_k$ are computed by inverting the following $3 \times 3$ matrix:

\[
M = \begin{bmatrix}
1 & x_{n_{k,1}} & y_{n_{k,1}} \\
1 & x_{n_{k,2}} & y_{n_{k,2}} \\
1 & x_{n_{k,3}} & y_{n_{k,3}}
\end{bmatrix}.
\]  

(9.1)

The matrix $M^{-1}$ contains the coefficients of the basis functions as its columns, and
for $j = 1, 2, 3$

\[
\text{eptr}(j) = \text{T.Elements}(k, j)
\]

if $\text{eptr}(1) > 0$

\[
\text{indices}(1) = \text{T.Edges}(\text{eptr}(1), 1)
\]

\[
\text{indices}(2) = \text{T.Edges}(\text{eptr}(1), 2)
\]

else

\[
\text{indices}(1) = \text{T.Edges}(\text{eptr}(1), 2)
\]

\[
\text{indices}(2) = \text{T.Edges}(\text{eptr}(1), 1)
\]

if $\text{eptr}(2) > 0$

\[
\text{indices}(3) = \text{T.Edges}(\text{eptr}(2), 2)
\]

else

\[
\text{indices}(3) = \text{T.Edges}(\text{eptr}(2), 1)
\]

for $i = 1, 2, 3$

\[
\text{ptrs}(i) = \text{T.NodePtrs}(\text{indices}(i))
\]

for $j = 1, 2$

\[
\text{coords}(i, j) = \text{T.Nodes}(\text{indices}(i, j))
\]

Table 9.1. The getNodes algorithm: Given a mesh $T$ and a triangle index $k$, determines the coordinates of the vertices of $T_k$ and the corresponding entries in $\text{T.NodePtrs}$. The coordinates are returned in a $3 \times 2$ array coords and the pointers in the $3 \times 1$ array ptrs.

since the basis functions are linear, these coefficients include the partial derivatives:

\[
M^{-1} = \begin{bmatrix}
    \frac{\partial \psi_{n,1}}{\partial x} & \frac{\partial \psi_{n,1}}{\partial y} & \frac{\partial \psi_{n,3}}{\partial x} \\
    \frac{\partial \psi_{n,2}}{\partial x} & \frac{\partial \psi_{n,2}}{\partial y} & \frac{\partial \psi_{n,3}}{\partial y}
\end{bmatrix}
\]

Therefore, upon computing $C = M^{-1}$, the gradients of the three basis functions are immediately available.

The integral

\[
\int_{T_k} \kappa
\]

is estimated using the one-point rule

\[
\int_{T_k} \kappa \approx A \kappa(x, y),
\]

where $A$ is the area and $(x, y)$ the centroid of the triangle. The computation of the area was explained at the end of the last chapter (see page 134).

Algorithm 9.2 is the complete element-oriented algorithm for assembling $K$. I assume that a routine for computing a matrix inverse is available. MATLAB contains such a routine. If the code is to be written in a high-level language such as
9.1. The MATLAB implementation

Fortran, C, or C++, it is recommended that code from the LAPACK package [?] be used. Since $K$ is symmetric, the upper triangle is first computed and then copied to the lower triangle. Depending on the software to be used to solve $KU = F$, it may not be necessary to fill in the lower triangle of $K$.

<table>
<thead>
<tr>
<th>Initialize $K$ to the zero matrix</th>
</tr>
</thead>
<tbody>
<tr>
<td>for $k = 1, 2, \ldots, N_t$</td>
</tr>
<tr>
<td>Call <code>getNodes</code> to get <code>coords</code> and <code>ptrs</code></td>
</tr>
<tr>
<td>Compute the matrix $M$ and its inverse $C$</td>
</tr>
<tr>
<td>for $r = 1, 2, 3$</td>
</tr>
<tr>
<td>for $s = r, \ldots, 3$</td>
</tr>
<tr>
<td>$G(r,s) = \nabla \phi_r \cdot \nabla \phi_s$</td>
</tr>
<tr>
<td>Estimate $I = \int_{T_k} \kappa$ using the one-point quadrature rule</td>
</tr>
<tr>
<td>for $r = 1, 2, 3$</td>
</tr>
<tr>
<td>for $s = r, \ldots, 3$</td>
</tr>
<tr>
<td>if ptr(r)&gt;0 and ptr(s)&gt;0</td>
</tr>
<tr>
<td>$i = \min{\text{ptr}(r), \text{ptr}(s)}$</td>
</tr>
<tr>
<td>$j = \max{\text{ptr}(r), \text{ptr}(s)}$</td>
</tr>
<tr>
<td>Add $G(r,s)I$ to $K(i,j)$</td>
</tr>
<tr>
<td>for $i = 2, 3, \ldots, N_f$</td>
</tr>
<tr>
<td>for $j = 1, 2, \ldots, i-1$</td>
</tr>
<tr>
<td>$K(i,j) = K(j,i)$</td>
</tr>
</tbody>
</table>

Table 9.2. The complete algorithm for assembling $K$. The matrix $M$ is defined by (9.1) and $C = M^{-1}$ contains $\nabla \phi_1, \nabla \phi_2, \nabla \phi_3$ in its second and third rows. The stiffness matrix $K$ is symmetric, so the upper triangle is computed in the main loop, and then the lower triangular entries are assigned at the end.

9.1 The MATLAB implementation

I have written a routine called `Stiffness1` that implements Algorithm 9.2. It takes as input the mesh data structure and a function implementing the coefficient $\kappa$. The second input is optional; if it is omitted, then the constant function 1 is used.

I hope that the reader is pleasantly surprised at how compactly the algorithm can be expressed and coded. Indeed, `Stiffness1` has only 26 lines of executable code! The key to this conciseness is the mesh data structure, which makes it easy to obtain the necessary information. MATLAB makes it easy to manipulate vectors and matrices with concise commands, which is also a factor in reducing the number of lines of code.