Rotation of Momentum Degrees of Freedom
M. Behr

http://www.cats.rwth-aachen.de/library/research/technotes

Abstract

This note outlines the steps necessary to locally rotate the coordinate system at selected nodes, in order to align that system with the boundary.

1. Introduction

Dirichlet boundary conditions that do not coincide with the Cartesian coordinate system used to formulate the fluid flow or mesh displacement problem present an implementation headache in most finite element codes. Such conditions are often used to impose slip or partial slip conditions at some of the domain boundaries. One can introduce the boundary conditions through Lagrange multiplier method, with the constraint of no flux in a specified direction; the multiplier will then represent the normal stress at that point. A simpler method is to apply local coordinate transformation for the affected boundary nodes. The locally-transformed system will have then separate degrees of freedom corresponding to the normal and tangent direction, and the constrained degrees of freedom are trivially handled by removing their rows from the system, and transferring their column to the right-hand side.

A pair of nodes $p$ and $q$ which have common element support has an $n_{dof} \times n_{dof}$ associated block in the left-hand side matrix, $A_{pq}$. Each node $p$ has an orthonormal $n_{ad} \times n_{sd}$ rotation matrix associated with it:

$$Q_p = \begin{bmatrix} \cos \theta_p & \sin \theta_p \\ -\sin \theta_p & \cos \theta_p \end{bmatrix}$$ \text{ in 2D.} \hspace{1cm} (1)

For the nodes on the slip boundary, this matrix contains the direction cosines of the normal; otherwise, it is an identity matrix. Let us extend that matrix to $n_{dof} \times n_{dof}$ matrix $Q_p$, which coincides with $Q_p$ for the velocity degrees of freedom, and is an $n_{dof} \times n_{dof}$ identity matrix elsewhere:

$$Q_p = \begin{bmatrix} \cos \theta_p & \sin \theta_p & 0 \\ -\sin \theta_p & \cos \theta_p & 0 \\ 0 & 0 & 1 \end{bmatrix}$$ \text{ in 2D UVP.} \hspace{1cm} (2)

The original left-hand side is the collection of block matrices $A = [A_{pq}]_{p,q=1,...,n_a}$, with most blocks being empty. We can also construct a global orthonormal rotation matrix $Q$ by placing the nodal rotation matrices $Q_p$ as its diagonal blocks: $Q = [I|Q_r]_{r=1,...,n_{rot}}$. Naturally, only non-identity rotation matrices are stored, numbered from 1 to $n_{rot}$. 

Then the degree of freedom rotation on the original system $A \mathbf{x} = \mathbf{b}$ is represented globally as:

$$ (QAQ^T)(Q\mathbf{x}) = (Q\mathbf{b}). \quad (3) $$

Each individual block $A_{pq}$ undergoes then rotation (no sum):

$$ A_{pq} \rightarrow Q_p A_{pq} Q_q^T. \quad (4) $$

On a single partition, a typical system matrix becomes:

\[
\begin{bmatrix}
\mathbf{I} & Q_1 & & & \\
& \mathbf{I} & & & \\
& & \ddots & & \\
& & & Q_{n\text{rot}} & \\
& & & & \mathbf{I}
\end{bmatrix}
\begin{bmatrix}
A_{11} & A_{12} & A_{13} \\
A_{21} & A_{22} & A_{23} \\
A_{31} & A_{32} & A_{33} \\
& & \\
& & \ddots & \\
& & & \ddots & \\
& & & & \mathbf{I}
\end{bmatrix}
\begin{bmatrix}
\mathbf{I} & Q_1^T & & & \\
& \mathbf{I} & & & \\
& & \ddots & & \\
& & & Q_{n\text{rot}}^T & \\
& & & & \mathbf{I}
\end{bmatrix}
\begin{bmatrix}
\mathbf{I} & Q_1 & & & \\
& \mathbf{I} & & & \\
& & \ddots & & \\
& & & Q_{n\text{rot}} & \\
& & & & \mathbf{I}
\end{bmatrix}.
\quad (5)
\]

or

\[
\begin{bmatrix}
A_{11} & A_{12} Q_1^T & & & \\
Q_1 A_{21} & Q_1 A_{22} Q_1^T & A_{13} & & \\
A_{31} & A_{32} & A_{33} & & \\
& & & \ddots & \\
& & & & \mathbf{I}
\end{bmatrix}
\begin{bmatrix}
Q_{n\text{rot}} A_{n\text{rot}2} Q_1^T & & & & \\
& \mathbf{I} & Q_{n\text{rot}} A_{n\text{rot}n} Q_{n\text{rot}}^T & & \\
& & & \ddots & \\
& & & & \mathbf{I}
\end{bmatrix}.
\quad (6)
\]

Each individual right-hand side vector segment $b_p$ also undergoes rotation (no sum):

$$ b_p \rightarrow Q_p b_p. \quad (7) $$

A typical right-hand side vector becomes:

\[
\begin{bmatrix}
\mathbf{I} & Q_1 & & & \\
& \mathbf{I} & & & \\
& & \ddots & & \\
& & & Q_{n\text{rot}} & \\
& & & & \mathbf{I}
\end{bmatrix}
\begin{bmatrix}
b_1 \\
b_2 \\
b_3 \\
& & \\
& & \ddots & \\
& & & b_{n\text{rot}}
\end{bmatrix}
= 
\begin{bmatrix}
b_1 \\
Q_1 b_2 \\
& & b_3 \\
& & & \ddots & \\
& & & & \mathbf{I}
\end{bmatrix}.
\quad (8)
\]

2. Implementation of Matrix Rotation

The non-trivial (partition or mesh) matrices $Q_p$ are stored in a compressed array

\[
\text{real* 8 r(nsd,nsd,nnr)}
\]
where \( \text{nsd} \) is \( n_{sd} \) and \( \text{nnr} \) is \( n_{rot} \). The mapping from a spatial dimension to the degree of freedom is defined by an input array:

\[
\text{adf}[1:\text{nsd}] \rightarrow 1:\text{ndf}
\]

(9)

The mapping from a rotation matrix counter to the (partition or mesh) node number is defined by another input array:

\[
i[1:\text{nnr}] \rightarrow 1:\text{nnl}
\]

(10)

Rotation of a matrix \( A_{pq} \rightarrow Q_{p}A_{pq}Q_{q}^{T} \) is accomplished in the subroutine

\[
\text{ewd:ewdbsr.F:ewdbsrrotate()}
\]

Depending on \( n_{sd} \), it calls either one of the subroutines \text{ewdbsrrotate2()} or \text{ewdbsrrotate3()}.

The 2D case is described here. The 3D case doesn’t involve any new concepts compared to the 2D case; all the unrolled \( n_{sd} \) loops are simply larger.

First, the BSR pointers (described in a separate TN) are recovered, and the \( n_{sd} \) to \( n_{dof} \) mapping is looked up. The rotation matrix counters for rows are also initialized.

\[
\begin{align*}
\text{bsrbegptr} &= \text{sbsrbegptr}(\text{id}) \\
\text{bsrendptr} &= \text{sbsrendptr}(\text{id}) \\
\text{bsridxptr} &= \text{sbsridxptr}(\text{id}) \\
i1 &= \text{adf}(1) \\
i2 &= \text{adf}(2) \\
inr &= 0 \\
inext &= 0
\end{align*}
\]

Now enter the loop over rows, and advance the rotation matrix counter until the corresponding node number is greater than or equal to the current row number.

\[
\begin{align*}
do \text{in}=1,:\text{nnl} \\
do \text{while (inext.lt.in.and.inr.lt.nnr)} \\
& \quad \text{inr} = \text{inr} + 1 \\
& \quad \text{inext} = i(\text{inr})
\end{align*}
\]

The \text{inext} index is thus leapfrogging past \text{in}, and if they are equal, we have a row that corresponds to a rotating node. In that case, retrieve the rotation matrix entries. Also, initialize rotation matrix counter for this row.

\[
\begin{align*}
\text{if (in.eq.inext) then} \\
& \quad r111 = r(1,1,inr) \\
& \quad r121 = r(2,1,inr) \\
& \quad r112 = r(1,2,inr) \\
& \quad r122 = r(2,2,inr)
\end{align*}
\]

end if

\[
\begin{align*}
\text{jnr} &= 0 \\
\text{jnext} &= 0
\end{align*}
\]
For that particular row, loop over non-zero $A_{pq}$ blocks, and for each of them retrieve the column number.

\[
\text{do } iblk=bsrbeg(in),bsrend(in)-1
\]
\[
jn = bsridx(iblk)
\]

Advance the rotation matrix counter until the corresponding node number is greater or equal than the current column number. The $j_{\text{next}}$ index is thus leapfrogging past $jn$, and if they are equal, we have both a non-zero matrix block, and a rotating node column. This may not happen even once for any particular row which does not correspond to a rotating node.

\[
\text{do while } (j_{\text{next}}.lt.jn.and.jnr.lt.nnr)
\]
\[
jnr = jnr + 1
\]
\[
j_{\text{next}} = i(jnr)
\]
\[
\text{end do}
\]

Having found $p$ (in) and $q$ (jn) for that block, multiply the block from the left by the rotation matrix. Only $n_{sd}$ rows are affected.

\[
\text{if } (\text{in.eq.inext}) \text{ then}
\]
\[
\text{do } jdf=1,ndf
\]
\[
a1 = a(i1,jdf,iblk)
\]
\[
a2 = a(i2,jdf,iblk)
\]
\[
a(i1,jdf,iblk) = rl11 * a1 + rl12 * a2
\]
\[
a(i2,jdf,iblk) = rl21 * a1 + rl22 * a2
\]
\[
\text{end do}
\]
\[
\text{end if}
\]

We know that the row corresponds to a rotating node. If the column also corresponds to a rotating node, we need to retrieve another rotation matrix.

\[
\text{if } (jn.eq.j_{\text{next}}) \text{ then}
\]
\[
rr11 = r(1,1,jnr)
\]
\[
rr21 = r(2,1,jnr)
\]
\[
rr12 = r(1,2,jnr)
\]
\[
rr22 = r(2,2,jnr)
\]

Now we multiply the block from the right by the transpose of that rotation matrix. Only $n_{sd}$ columns are affected.

\[
\text{do } idf=1,ndf
\]
\[
a1 = a(idf,i1,iblk)
\]
\[
a2 = a(idf,i2,iblk)
\]
\[
a(idf,i1,iblk) = a1 * rr11 + a2 * rr12
\]
\[
a(idf,i2,iblk) = a1 * rr21 + a2 * rr22
\]
\[
\text{end do}
\]
Terminate the loops.

\[
\end{if}
\]
\[
\end{do}
\]
\[
\end{do}
\]

3. Implementation of Vector Rotation

Rotation of a vector \( \mathbf{d}_p \rightarrow \mathbf{Q}_p \mathbf{d}_p \) is accomplished in the subroutine

\[
\text{ns:align.F:alignvector()}
\]

As before, the non-trivial (partition or mesh) matrices \( \mathbf{Q}_p \) are stored in a compressed array \( \mathbf{rot}(\text{nsd,nsd,nnlrot}) \). The two aforementioned mapping arrays are named here \( \text{adfalign[]} \) and \( \text{rot[]} \), to avoid conflicts with other XNS arrays. Additionally, the parameter \( \text{nnr} \) is named here \( \text{nnlrot} \).

First, the \( n_{sd} \) to \( n_{dof} \) mapping is looked up.

\[
\text{xdf} = \text{adfalign}(\text{xsd})
\]
\[
\text{ydf} = \text{adfalign}(\text{ysd})
\]

Then the vector segment is multiplied by the rotation matrix from the left.

\[
\begin{align*}
&\text{do inlrot=1,nnlrot} \\
&\quad \text{inl} = \text{idxrot(inlrot)} \\
&\quad \text{du} = \text{d(xdf,inl)} \\
&\quad \text{dv} = \text{d(ydf,inl)} \\
&\quad \text{d(xdf,inl)} = \text{rot(xsd,xsd,inlrot)} \times \text{du} \\
&\quad \text{&} \\
&\quad \text{d(ydf,inl)} = \text{rot(ysd,xsd,inlrot)} \times \text{du} \\
&\quad \text{&} \\
&\quad \text{end do} \\
&\end{align*}
\]

The inverse transformation \( \mathbf{d}_p \rightarrow \mathbf{Q}_p^T \mathbf{d}_p \) is accomplished in the subroutine

\[
\text{ns:align.F:dealignvector()}
\]

which is nearly identical to the above implementation.

History

February 22, 2001  Written.
March 22, 2001  Completed, along with code revisions.
April 5, 2001  Fixed incomplete rotation code.