**Coding Q&A 3**

**Question:** I was already planning on putting the nodal equations in pre-derived, however, my issue is using them to compile the Jacobian matrix, and converting them from local to global coordinates. If zeta and eta need to be predefined before I can manipulate the equations, does that mean I need to define a zeta\_1, zeta\_2, eta\_1 and eta\_2 and assign them values for Gaussian integration from the beginning, or do zeta and eta need dummy values until the global stiffness matrix is assembled and then they get assigned the proper Gaussian values?

**Answer:**

**The short answer:**

The short answer is that for calculating element stiffness matric, you only need to evaluate everything at the Gaussian points for which zeta\_1, zeta\_2, eta\_1 and eta\_2 are the Gaussian point values.

**The long answer for why or how:**

**Analysis for programming**

Let’s start by looking at what we ultimately need -- the element stiffness matrix

 (1)

where t is the thickness of the body, and

 (2)

Since both  and  are matrices, it’s easier to talk about their generic p-th row, q-th column coefficient:

 (3)



then

 (4)

Using 2-point Gaussian integration, we can express the integration as

 (5)

Since t and the weights of 2-point Gaussian integration  and  are all known, it is very clear that once  can be evaluate at the Gaussian points and , then the computation of  can be done without any problem.

Now look at eq (2):

 (2)

The **D** matrix contains only material constants. The first matrix (**B** transpose) and the third matrix (**B**) contain both natural and global coordinates: functions N1, N2, etc., are functions of natural coordinates  and , but the derivation is with respect to global coordinates x and y. How to get around that? Well, the approach will be to transform all functions to functions of the natural coordinates. Use the equation (10-14) in the text:

 (6)

where the Jacobian is

 (7)

 (8)

with similar expressions for , , and . Note that, , , etc. are known nodal coordinates in the global system from input data file -- they are fixed parameters as far as integration is concerned.

With (6) to (8), everything in (2) is now expressed as functions of the natural coordinates and fixed or known parameters such as material constants and nodal positions, x1, x2, etc. And we also don’t need to bother with general locations of zeta and eta, but instead, just zeta and eta at every Gaussian point. Now, everything is doable with regard to coding.

**Coding**

The above shows the analysis for programming. For coding, it’s much easier to explain things in pseudo codes.

**// pseudo code for calculating the element stiffness matric ke\_matrix**

|  |
| --- |
| //this function computes the element stiffness matrix for a generic element ie. it should be called from the global stiffness matrix calculation function within a loop that goes from 0 to (total\_number\_of\_elems - 1)  //hard code all Gaussian points and weights  zeta[0] = -1/sqrt(3)  zeta[1] = 1/sqrt(3)  eta[0] = -1/sqrt(3)  eta[1] = 1/sqrt(3)  w[0]=1  w[1]=1  // in fact, since the above parameters are the same for all elements, it will be much more efficient to define them as static global scope constants which are defined just once, but can be used repeatedly  //Need a double loop to go through all 16 by 16 coefficients for ke\_matrix and initialize to zero  For p from 0 to 15  For q from 0 to 15  ke\_matrix[p][q] = 0  end q  end p  //Need a double loop to go through all 2 by 2 Gaussian points  For j from 0 to 1  For i from 0 to 1  call cal\_bmatrix\_function(at zeta[i], eta[j], b\_matrix, b\_inverse, det\_jacob)  call cal\_mmatrix\_function(at zeta[i], eta[j], b\_matrix, b\_inverse, det\_jacob, m\_matrix)  //Need a double loop to go through all 16 by 16 coefficients for ke\_matrix and add contribution from each Gaussian point  For p from 0 to 15  For q from 0 to 15  ke\_matrix[p][q] = ke\_matrix[p][q] + t\*m\_matrix[p][q]\*w[i]\*w[j]  end q  end p  end i  end j  //this completes |

**// pseudo code for cal\_bmatrix\_function(at zeta[i], eta[j], b\_matrix, b\_inverse, det\_jacob)**

|  |
| --- |
| // the cal\_bmatruix function computes the B matrix, B transpose, and the determinant of the jacobian at Gausian point zeta[i] and eta[j]. When this function is called, zeta[i], eta[j] are known and passed by the argument list  Calculate dN1dzeta (you need to derive by hand, and “hard” code it, where zeta is zeta[i], eta is eta[j])  Calculate dN2dzeta  Calculate dN3dzeta  Calculate dN4dzeta  Calculate dN5dzeta  Calculate dN6dzeta  Calculate dN7dzeta  Calculate dN8dzeta  Calculate dN1deta (you need to derive by hand, and “hard” code it, where zeta is zeta[i], eta is eta[j])  Calculate dN2deta  Calculate dN3deta  Calculate dN4deta  Calculate dN5deta  Calculate dN6deta  Calculate dN7deta  Calculate dN8deta  Calculate dxdzeta according to (8):  Calculate dxdeta according to  Calculate dydzeta according  Calculate dydeta according to  Form a 2 by 2 jacobian matrix according to  det\_jacob = dxdzeta\*dydeta - dxdeta\*dydzeta //calculate the determinant of the jacobian  Call invert\_matrix(jacobian, jacobina\_inverse) where the invert\_matrix function takes a matrix (jacobian) as input, find its inverse, and store the inverse in jacobian\_inverse as a return parameter. //a Gaussian-Jordan elimination solve will be provided to you for use. But you can use any other matrix inversion method too.  Use dN1dzeta and dN1deta to form a 1 by 2 matrix (a vector) called vector1,  Call matrix\_multiply(jacobian\_inverse, vector1, dN1vector) //the matrix\_multiply function performs matrix multiplication (jacobian\_inverse\*vector1) and store the result in dN1vector, according to  Call matrix\_multiply(jacobian\_inverse, vector2, dN2vector) //similarly, calculate  ……  Call matrix\_multiply(jacobian\_inverse, vector8, dN8vector) //similarly, calculate  Form B matrix b\_matrix according to //pretty much just put dN1dx etc into the right position  Form B matrix inverse b\_inverse by a double loop:  For i from 0 to 3  For j from 0 to 8  b\_inverse[j][i] = b\_matrix[i][j]  End j  End i |

**//pseudo code for calculating M matrix cal\_mmatrix\_function(at zeta[i], eta[j], b\_matrix, b\_inverse, det\_jacob, m\_matrix)**

|  |
| --- |
| //This function computes the M matrix. When cal\_mmatrix\_function is called, zeta[i], eta[j] are known (given), and the determinate of the jacobian is also known (calculated in cal\_bmatrix\_function and passed as a function argument)  Form material D matrix d\_matrix (in fact, since d\_matrix is the same for all elements, it will be much more efficient to define d\_matrix as a static global scope matrix)  Call matrix\_multiply\_function(d\_matrix, b\_matrix, r, s, t, matrix\_holder1) // matrix\_multiply\_function performs matrix multiplication. d\_matrix has size r by s, b\_matrix has size s by t, and matrix\_holder1 = d\_matrix\*b\_matrix and has size r by t. Here, r=3, s=3, t=16  Call matrix\_multiply\_function(b\_inverse, matrix\_holder1, r, s, t, matrix\_holder2) // matrix\_holder2 = b\_inverse\*matrix\_holder1, here r=16, s=3, t=16  //Use a double loop to form M matrix  For i from 0 to 15  For j from 0 to 15  M\_matrix[i][j] = matriux\_holder2[i][j]\*det\_jacob  End j  End i  //this completes the calculation of the m matrix according to |