Non-Linear Heat Conduction

Consider a slab of material experiencing heat conduction:

\[
\begin{array}{c}
L \\
\hline
s = \text{const} \\
k = k(T)
\end{array}
\quad R
\]

\[
T_L = 400 \text{ K} \\
k(T) = 74.59 - 0.0445T \quad \text{W/m-K}
\]

\[
T_R = 1000 \text{ K} \\
x_L = 0 \\
s = 10 \quad \text{W/m}^3
\]

\[
x_R = 1 \text{ m}
\]

Unlike prior analyses, the thermal conductivity is more likely a function of temperature rather than position. For a steel slab with a constant source, a temperature-dependant thermal conductivity, and Dirichlet boundary conditions as given above, the governing differential equation is (Cengel, *Heat Transfer - A Practical Approach*, McGraw Hill, 1997. Chapt. 2.3)

\[
L(T) = -\frac{d}{dx}\left(k(T) \frac{dT(x)}{dx}\right) - s = 0 \quad \Omega \in (L, R)
\]  
(1)

with boundary conditions of

\[
T(x_L) = T_L \quad \partial \Omega_L \in L \\
T(x_R) = T_R \quad \partial \Omega_R \in R
\]  
(2)

Thus, equation (1) has two Dirichlet boundary conditions and the problem statement is therefore well-posed. Note that the second-order term contains a non-linearity and ultimately some type of iterative algorithm will be required to solve (1). We begin, as usual, by assuming a series expansion approximate form for the unknown solution of temperature T, hence

\[
T(x) \approx T^N(x) = \sum_{\alpha=1}^{N} \Psi_{\alpha} (x) T_{\alpha} = \Psi_{\beta} (x) T_{\beta} \quad \text{for } 1 \leq \alpha \leq N
\]  
(3)

Forming the Galerkin Weak Statement to minimize the error in our approximate solution

\[
GWS^N = \int_{\Omega} \Psi_{\beta} (x)L(T^N)dx = 0 \quad 1 \leq \beta \leq N
\]  
(4)

Applying the operator

\[
GWS^N = \int_{\Omega} \Psi_{\beta} (x) \left(-\frac{d}{dx}\left(k(T^N) \frac{dT(x)}{dx}\right) - s\right)dx = 0 \quad 1 \leq \beta \leq N
\]  
(5)

Integrating (5) by parts and expanding:

\[
GWS^N = \int_{\Omega} \frac{d\Psi_{\beta}}{dx}k(T^N) \frac{dT(x)}{dx}dx - \int_{\Omega} \Psi_{\beta} s dx - \Psi_{\beta} k(T^N) \frac{dT^N}{dx} = 0 \quad 1 \leq \beta \leq N
\]  
(6)

The final step is handling the non-linearity in \(k(T^N)\). We can begin by recognizing the linear variation in \(k\) to be of the form \(k(T^N) = a + bT^N\). Substituting into (6)

\[
GWS^N = \int_{\Omega} \frac{d\Psi_{\beta}}{dx} (a + bT^N) \frac{dT(x)}{dx}dx - \int_{\Omega} \Psi_{\beta} s dx - \Psi_{\beta} k(T^N) \frac{dT^N}{dx} = 0 \quad 1 \leq \beta \leq N
\]  
(7)
Substituting the series expansion and expanding

\[ GWS^N = a \int_\Omega \frac{d\Psi_\beta}{dx} \frac{d\Psi_\alpha}{dx} dT_\alpha + bT_\alpha \int_\Omega \frac{d\Psi_\beta}{dx} \frac{d\Psi_\alpha}{dx} dT_a - \int_\Omega \Psi_\beta s dx - \Psi_\beta k(T^N) \frac{dT^N}{dx} \bigg|_{\partial \Omega} = 0 \quad 1 \leq \alpha, \beta \leq N \quad (8) \]

Discretizing the domain

\[ GWS^h = \sum_e \left[ a \int_\Omega \frac{d\{N_k\}^T}{dx} \{N_k\} dT_e + b\{T\}_e \int \{N_k\} \frac{d\{N_k\}^T}{dx} dT_e - \int \{N_k\}^T \{N_k\} d\{S\}_e \right] = \{0\}_e \quad (9) \]

We can interpolate the constant source \( s \) over each element via \( s_e = \{N_k\}^T \{S\}_e \) for no reason other than to keep our usual matrix structure for sources and loads. Note that the imposed Dirichlet boundary conditions removed the need for generating the discretized form of the unknown boundary fluxes. Identifying the terms in (9):

\[
\begin{align*}
[DIFFA]_e &= a \int_\Omega \frac{d\{N_k\}^T}{dx} \{N_k\} \frac{d\{N_k\}^T}{dx} d\bar{x} \\
&= (a) \{e\}_e \{e\}_e (-1) \{A211k\}_e \quad (10a) \\
[DIFFB]_e &= b\{T\}_e \int_\Omega \{N_k\} \frac{d\{N_k\}^T}{dx} d\bar{x} \\
&= (b) \{e\}_e \{e\}_e (-1) \{A3011k\}_e \\
[SRC]_e &= \int_\Omega \{N_k\}^T \{N_k\} d\bar{x} \{S\}_e \\
&= (1) \{e\}_e \{e\}_e \{A200k\} \{S\}_e \\
\end{align*}
\]

Upon assembly, (9) can be simplified to the following syntactical matrix statement:

\[ [DIFFA + DIFFB]\{T\} = \{SRC\} \quad (11) \]

and we recover our friend \([LHS]^*\{Q\} = \{RHS\}\). However, the unknown values of \{T\} are inside of \([DIFFB]\) - thus we need to know \{T\} to form \([DIFFB]\) to solve for \{T\} to form \([DIFFB]\) ... we need an iterative scheme to solve!

Many iterative algorithms exist including Jacobi, Gauss-Seidel, Successive Over-Relaxation, and Picard. However, these algorithms are limited in that they converge very slowly (linearly). A better algorithm is Newton which converges quadratically. The downside of Newton is that additional calculus is required but the speed of convergence is well worth the additional effort.

Without derivation, the Newton Iteration Algorithm operates in the following manner:

The new and improved unknown solution \( \{Q\} \) at the \( p+1 \) (current) iteration is given by:

\[ \{Q\}^{p+1} = \{Q\}^p + \{\delta Q\}^{p+1} \quad (12) \]

where \( \{Q\}^p \) is the previous value of the unknown solution vector and \( \{\delta Q\}^{p+1} \) is the current value of the incremental change in \( \{Q\}^{p+1} \). To obtain the incremental change \( \{\delta Q\}^{p+1} \) requires solving the statement

\[ [JAC]^p \{\delta Q\}^{p+1} = -[FQ]^p \quad (13) \]
where \( [JAC]^p \) is the jacobian evaluated using the previous values of \( \{Q\}^p \) and \( \{FQ\}^p \) is the residual evaluated using the previous values of \( \{Q\}^p \). Note that the jacobian is always a square matrix and the residual is always a column vector. The final issue is forming \( [JAC]^p \) and \( \{FQ\}^p \).

The residual \( \{FQ\}^p \) is simply the entire weak statement (9) evaluated using the previous values of \( \{Q\}^p \). Thus, for our heat conduction problem, the residual is

\[
\{FQ\} = S_e \left( a \int \frac{d[N_k]}{dx} \frac{d[N_k]^T}{dx} d\bar{x} [T]_e^p + b[T]_e^p \int \frac{d[N_k]}{dx} \frac{d[N_k]^T}{dx} d\bar{x} [T]_e^p - \int \frac{d[N_k]}{dx} \frac{d[N_k]^T}{dx} d\bar{x} [S]_e = [0]_e \right)
\] (14)

where \( [T]_e^p \) are the nodal values of the solution from the previous iteration.

The jacobian \( [JAC]^p \) is a bit more difficult to obtain. It is mathematically defined as

\[
[JAC] = S_e \left( \frac{\partial \{FQ\}_e^p}{\partial \{Q\}_e^p} \right)
\] (15)

Thus we must first differentiate the entire weak statement with respect to the unknowns to form \( [JAC] \) and then evaluate any hypermatrices within \( [JAC] \) with the previous values of \( \{Q\}_e^p \) to obtain \( [JAC]^p \). The best way to perform this differentiation is to employ the discretized form of the governing equation (9)

\[
GWS^b = S_e \left( a \int \frac{d[N_k]}{dx} \frac{d[N_k]^T}{dx} d\bar{x} [T]_e^p + b[T]_e^p \int \frac{d[N_k]}{dx} \frac{d[N_k]^T}{dx} d\bar{x} [T]_e^p - \int \frac{d[N_k]}{dx} \frac{d[N_k]^T}{dx} d\bar{x} [S]_e = [0]_e \right)
\] (18)

Differentiating our GWS^b with respect to \( [T]_e \) has one caveat - \( [T]_e \) must be outside the integral AND on the right side so as not to muck up the hypermatrix structure. Let’s begin by differentiating the element level constant source term with respect to temperature

\[
\frac{\partial S}{\partial [T]_e} = \left( \frac{\partial [S]_e}{\partial [T]_e} \right) = \left( \begin{array}{c} 0 \\ 0 \end{array} \right)
\] (17)

Note that the master matrix is not a function of temperature, hence our differentiation will default to the source term. Assuming a linear basis for demonstrations and doing the vector calculus

\[
\frac{\partial \{Q\}_e}{\partial [T]_e} = \left( \begin{array}{c} \frac{\partial S}{\partial T_L} \\ \frac{\partial S}{\partial T_R} \end{array} \right) = \left( \begin{array}{c} 0 \\ 0 \end{array} \right)
\]

Thus, upon assembly

\[
S_e \left( \frac{\partial \{Q\}_e}{\partial [T]_e} = \left( \begin{array}{c} 0 \\ 0 \end{array} \right) \right) = S_e \left( \int \frac{d[N_k]}{dx} \frac{d[N_k]^T}{dx} d\bar{x} [0]_e \right)
\] (18)
where \( \mathbf{0} \) is the zero matrix with size appropriate for the selected basis degree. Clearly the assembly will yield a matrix of zeros and thus the differentiation of the constant source with respect to temperature is does not result in a contribution to the jacobian.

The element level linear conductivity term follows suit via

\[
\frac{\partial}{\partial \{T\}_e} \left( a \int_{\alpha_e} \frac{d\{N_k\}}{dx} \frac{d\{N_k\}^T}{dx} d\{T\}_e \right) = a \int_{\alpha_e} \frac{d\{N_k\}}{dx} \frac{d\{N_k\}^T}{dx} d\{T\}_e^T \frac{\partial \{T\}_e}{\partial \{T\}_e} \tag{19}
\]

Again, assuming a linear basis for demonstration

\[
\frac{\partial \{T\}_e}{\partial \{T\}_e} = \begin{bmatrix} \frac{\partial T_L}{\partial T_L} & \frac{\partial T_L}{\partial T_R} \\ \frac{\partial T_R}{\partial T_L} & \frac{\partial T_R}{\partial T_R} \end{bmatrix}_e = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}_e
\tag{20}
\]

Thus, upon assembly

\[
S_e \left( \frac{\partial}{\partial \{T\}_e} \left( a \int_{\alpha_e} \frac{d\{N_k\}}{dx} \frac{d\{N_k\}^T}{dx} d\{T\}_e \right) \right) = S_e \left( a \int_{\alpha_e} \frac{d\{N_k\}}{dx} \frac{d\{N_k\}^T}{dx} d\{1\}_e \right) \tag{21}
\]

where \( \mathbf{1} \) is the identity matrix with size appropriate for the selected basis degree.

To handle the non-linear conductivity term, we must recognize that the product rule will be required and that for correct vector differentiation, the temperature must be outside the integral on the right side, i.e. a column vector: Preparing the element level non-linear term by rearranging the scalar terms for the impending product rule

\[
b\{T\}_e^T \int_{\alpha_e} \frac{d\{N_k\}}{dx} \frac{d\{N_k\}^T}{dx} d\{T\}_e d\bar{x} = b \int_{\alpha_e} \frac{d\{N_k\}^T}{dx} \frac{d\{N_k\}}{dx} \{T\}_e^T d\bar{x}
\]

\[
= b \int_{\alpha_e} \frac{d\{N_k\}^T}{dx} \frac{d\{N_k\}}{dx} \{T\}_e^T \{N_k\}_e d\bar{x}
\]

\[
= b \int_{\alpha_e} \frac{d\{N_k\}^T}{dx} \frac{d\{N_k\}}{dx} \{N_k\}_e^T \{N_k\}_e d\bar{x}
\]

\[
= b \{T\}_e^T \int_{\alpha_e} \frac{d\{N_k\}}{dx} \frac{d\{N_k\}^T}{dx} d\bar{x} \{T\}_e \tag{22}
\]

Differentiating via the product rule

\[
\frac{\partial}{\partial \{T\}_e} \left( b\{T\}_e^T \int_{\alpha_e} \frac{d\{N_k\}}{dx} \frac{d\{N_k\}^T}{dx} d\{T\}_e \right) = b\{T\}_e^T \int_{\alpha_e} \frac{d\{N_k\}}{dx} \frac{d\{N_k\}^T}{dx} d\bar{x} \frac{\partial \{T\}_e}{\partial \{T\}_e} \tag{23}
\]
Thus, upon assembly

\[
S_e \left( \frac{\partial}{\partial [T]_e} \left( b [T]_e^T \int_{\Omega_e} \left\{ \frac{d[N_k]}{dx} \frac{d[N_k]^T}{dx} d\bar{x} \right\}_T \, dx \right) \right) = S_e \left( b [T]_e^T \int_{\Omega_e} \left\{ \frac{d[N_k]}{dx} \frac{d[N_k]^T}{dx} d\bar{x} \right\}_T \, dx \right),
\]

(24)

Summing the contributions to the Newton jacobian, (21) and (24)

\[
[\text{JAC}] = S_e \left( \frac{\partial [FQ]_e}{\partial [Q]_e} \right) = S_e \left( a \int_{\Omega_e} \frac{d[N_k]}{dx} \frac{d[N_k]^T}{dx} d\bar{x} \right)_e
\]

(25)

Evaluating the jacobian with the previous values of \([T]^p\)

\[
[\text{JAC}]^p = S_e \left( \frac{\partial [FQ]_e}{\partial [Q]_e} \right)^p
\]

(26)

Homework:

For a generic form of the thermal conductivity to be \(k(T) = a + bT + cT^2\), evaluate the residual and jacobian as in (14) and (26). Comment on the matrix which arises from the new quadratic term when forming the jacobian.

Hint: Employ a *grouped variable approximation* to the quadratic component of the conductivity via \(T^2 \approx T^{N,2} = \Psi_a T_a^2\) instead of introducing the \([A4ddddk]\) mega-hypermatrix.

Non-linear heat conduction
% FEm.PSE template, Non-linear Heat Conduction

% SOLVE FOR TEMPERATURE
% iteration loop
Q = pi*ones(nnodes,1); % initialize Q
Q(1,1) = Tl; % impose Dirichlet data
Q(nnodes,1) = Tr; % initialize dQ
dQ = pi*ones(nnodes,1);

iter_num = 1; % iteration counter
Q_big(:,iter_num) = Q; % big matrix of nodal Q

while max(dQ)>=newt_crit
    % Assemble all contributions to {FQ}
    res = asres1d(a,[],[],-1,A211L, Q);
    res = res + asres1d(b,[],Q,-1,A3011L,Q);
    res = res + asres1d(-1,[],[],1, A200L, S);

    % Assemble all contributions to [JAC]
    jac = asjac1d(a,[],[],-1,A211L, []);
    jac = jac + asjac1d(b,[],Q,-1,A3011L, []);
    jac = jac + asjac1d(b,[],Q,-1,A3110L, []);

    % Modify [JAC] for Dirichlet data
    jac(1,:) = zeros(1,nnodes);
    jac(1,1) = 1;
    jac(nnodes,:) = zeros(1,nnodes);
    jac(nnodes,nnodes) = 1;

    % Modify (FQ) for Dirichlet data
    res(1,1) = 0;
    res(nnodes,1) = 0;

    % Solve for dQ
    dQ = jac \ (-res);

    % Update Q
    Q = Q + dQ;

    iter_num = iter_num+1; % increment iteration counter
    Q_big(:,iter_num) = Q; % append current value of Q
    max(dQ) % echo max(dQ)
end