**AME50541: Finite Element Methods**
**Homework 5: Due Wednesday, April 10 2019**

*Be sure to email all code to the instructor and TA.*

**Problem 1:** (25 points) Consider a general nonlinear vector-valued function of $m$ variables and its Jacobian matrix

$$\begin{aligned} \boldsymbol{R} : \mathbb{R}^m &\to \mathbb{R}^m \\ \boldsymbol{x} &\mapsto \boldsymbol{R}(\boldsymbol{x}), \end{aligned} \qquad \begin{aligned} \boldsymbol{J} : \mathbb{R}^m &\to \mathbb{R}^{m\times m} \\ \boldsymbol{x} &\mapsto \frac{\partial \boldsymbol{R}}{\partial \boldsymbol{x}}(\boldsymbol{x}), \end{aligned}$$

where $\left[\frac{\partial \boldsymbol{R}}{\partial \boldsymbol{x}}\right]_{ij} = \frac{\partial R_i}{\partial x_j}$. To find a root of $\boldsymbol{R}$, i.e., find $\boldsymbol{x}^* \in \mathbb{R}^m$ such that $\boldsymbol{R}(\boldsymbol{x}^*) = 0$, we will apply the Newton-Raphson iteration

$$\boldsymbol{x}_{k+1} = \boldsymbol{x}_k - \left[\frac{\partial \boldsymbol{R}}{\partial \boldsymbol{x}}(\boldsymbol{x}_k)\right]^{-1} \boldsymbol{R}(\boldsymbol{x}_k),$$

where $\boldsymbol{x}_0 \in \mathbb{R}^m$ is some initial guess for $\boldsymbol{x}^*$ and $\boldsymbol{x}_k \in \mathbb{R}^m$ is the approximation of $\boldsymbol{x}^*$ at the $k$th iteration for $k = 0, 1, \ldots$. The Newton-Raphson iterations will converge quadratically to a root of $\boldsymbol{R}$ under some assumptions. For the algorithm to be practical, we must specify a maximum number of iterations $N$ (in cases where the iterations diverge) and a convergence condition. We will consider iteration $k$ to be sufficiently close to $\boldsymbol{x}^*$ (converged) if

$$\|\boldsymbol{R}(\boldsymbol{x}_k)\|_\infty \leqslant \tau,$$

where $\|\cdot\|_\infty$ is the infinity norm (the entry with the maximum magnitude) and $\tau \in \mathbb{R}$ is the convergence tolerance. If none of the iterations $\{\boldsymbol{x}_0, \ldots, \boldsymbol{x}_N\}$ satisfy the convergence conditions, we assume the algorithm is diverging and terminate the iteration.

(a) Implement the Newton-Raphson method with the signature given below. Be sure to save convergence iteration (see comments below) in order to investigate the convergence properties of the method.

```
function [x, info] = solve_newtraph(fcn, x0, tol, maxit)
%SOLVE_NEWTRAPH Solve a nonlinear system of equations R(x) = 0 using the
%Newton—Raphson method.
%
%Input arguments
%———————————————
%  FCN : fuction : Function that accepts X (Array (m,)) and returns the
%    residual R and Jacobian dRdx of the nonlinear system of equation
%    corresponding to X, i.e., R(X), dRdx(X)
%
%  X0 : Array (m,) : Initial guess for the solution of R(x) = 0
%
%  TOL : number : Convergence tolerance; the system is considered
%    converged when max(abs(R)) < tol
%
%  MAXIT : number : Maximum number of iterations allowed
%
%Output arguments
%————————————————
%  X : Array (m,) : Solution of R(x) = 0
%
%  INFO : struct : Convergence information
%    INFO.SUCC : bool : Whether the iterations converged
%    INFO.NIT : number : Number of iteration required
%    INFO.R_NRM : Array (nit+1,) : Norm of residual (inf norm) at each
%      iteration, including initial guess
%    INFO.DX_NRM : Array (nit,) : Norm of Newton step at each iteration
```

(b) Test your function on any linear system you choose $\boldsymbol{R}(\boldsymbol{x}) = \boldsymbol{A}\boldsymbol{x} - \boldsymbol{b}$ and verify it converges to machine precision after one iteration.

(c) Consider the following nonlinear function

$$\boldsymbol{R}(\boldsymbol{x}) = \begin{bmatrix} e^{-e^{-(x_1+x_2)}} - x_2(1+x_1^2) \\ x_1\cos(x_2) + x_2\sin(x_1) - \frac{1}{2}. \end{bmatrix}$$

- Derive the Jacobian of $\boldsymbol{R}(\boldsymbol{x})$. As always, you are welcome to use symbolic mathematics software.
- Use your code to find a root $\boldsymbol{R}(\boldsymbol{x})$ from the starting point $\boldsymbol{x} = (0,0)^T$. Use $\tau = 10^{-10}$ and $N = 20$.
- Verify quadratic convergence by computing the ratio

$$m_k = \frac{\log \|\boldsymbol{R}(\boldsymbol{x}_k)\|_\infty}{\log \|\boldsymbol{R}(\boldsymbol{x}_{k-1})\|_\infty}$$

for $k = 1, \ldots, N$ and confirming $m_k \to 2$.

(d) Repeat the previous tasks for the following nonlinear function

$$\boldsymbol{R}(\boldsymbol{x}) = \begin{bmatrix} x_1 + 2x_2 + 4x_3 \\ \sin x_1 \sin x_2 \sin x_3 \\ x_1^2 + 2x_2^2 + 3x_3^2 \end{bmatrix}$$

with the starting point $\boldsymbol{x} = (0.5, 0.5, 0.5)^T$ with $\tau = 10^{-10}$ and $N = 100$. Does the method still converge quadratically? If not, explain why quadratic convergence breaks down in this case.

(e) Check your code to make sure your implementation is reasonably *efficient*. In particular, make sure there are no unnecessary evaluations of $\boldsymbol{R}(\boldsymbol{x})$ and its Jacobian. Even though we have only considered small problems in this assignment, you will use this code in your project to solve 2- and 3-dimensional PDEs using the finite element method.

**Problem 2:** (30 points) Consider the following system of $N$ ordinary differential equations

$$\begin{aligned} \boldsymbol{M}\dot{\boldsymbol{u}}(t) + \boldsymbol{K}\boldsymbol{u}(t) + \boldsymbol{F} = 0, \quad t \in (0,T] \\ \boldsymbol{u}(0) = \bar{\boldsymbol{u}} \end{aligned}$$

where $\boldsymbol{u}(t) \in \mathbb{R}^N$ is the solution of the system of ODEs at each time $t \in (0,T]$, $\boldsymbol{M} \in \mathbb{R}^{N\times N}$ is the mass matrix, $\boldsymbol{K} \in \mathbb{R}^{N\times N}$ is the stiffness matrix, and $\boldsymbol{F}$ is the forcing vector. Partition the time interval into $N_t$ segments $(t_n, t_{n+1})$ for $n = 1, \ldots, N_t$ where $t_1, \ldots, t_{N_t}$ are the discrete time instances used to partition the time domain, $0 = t_1 < t_2 < \cdots < t_{N_t-1} < t_{N_t} = T$, and $\Delta t_n = t_{n+1} - t_n$. Define $\boldsymbol{u}_n = \boldsymbol{u}(t_n)$ and consider the forward and backward Euler discretization of the system of ODEs

$$\begin{aligned} \boldsymbol{M}\frac{\boldsymbol{u}_{n+1} - \boldsymbol{u}_n}{\Delta t_n} + \boldsymbol{K}\boldsymbol{u}_n + \boldsymbol{F} = 0 \qquad \text{(forward Euler)} \\ \boldsymbol{M}\frac{\boldsymbol{u}_{n+1} - \boldsymbol{u}_n}{\Delta t_n} + \boldsymbol{K}\boldsymbol{u}_{n+1} + \boldsymbol{F} = 0 \qquad \text{(backward Euler),} \end{aligned}$$

for $n = 1, \ldots, N_t$, which lead to the respective update formulas, i.e., equation to advance the solution from the current time step $\boldsymbol{u}_n$ to the next time step $\boldsymbol{u}_{n+1}$,

$$\begin{aligned} \boldsymbol{u}_{n+1} &= (\boldsymbol{I} - \Delta t_n \boldsymbol{M}^{-1}\boldsymbol{K})\boldsymbol{u}_n - \Delta t_n \boldsymbol{M}^{-1}\boldsymbol{F} \quad \text{(forward Euler)} \\ \boldsymbol{u}_{n+1} &= (\boldsymbol{M} + \Delta t_n \boldsymbol{K})^{-1}(\boldsymbol{M}\boldsymbol{u}_n - \Delta t_n \boldsymbol{F}) \quad \text{(backward Euler).} \end{aligned}$$

(a) Implement a function that advances the ODE solution at time $n$ to the solution at time $n+1$ using the forward Euler method. Your function should have the following signature:

```
function [u_np1] = advance_soln_fe_lin(u_n, dt, M, K, F)
%ADVANCE_SOLN_FE_LIN Advance a linear system of ODEs defined by
%
```

```
%                   M * \dot{u} + K * u + F = 0
%
% from time n to n+1 (step size = dt) using Forward Euler.
%
%Input arguments
%———————————————
%  u_n : Array (m,) : Solution vector at step n
%
%  dt : number : Time step
%
%  M : Array (m, m) : Mass matrix of ODE system
%
%  K : Array (m, m) : Stiffness matrix of ODE system
%
%  F : Array (m,) : Right—hand side (force vector) of ODE system
%
%Output arguments
%————————————————
%  u_np1 : Array (m,) : Solution vector at step n+1
```

(b) Implement a function that advances the ODE solution at time $n$ to the solution at time $n+1$ using the backward Euler method. Your function should have the following signature:

```
function [u_np1] = advance_soln_bdf1_lin(u_n, dt, M, K, F)
%ADVANCE_SOLN_BDF1_LIN Advance a linear system of ODEs defined by
%
%                   M * \dot{u} + K * u + F = 0
%
% from time n to n+1 (step size = dt) using Backward Euler (BDF1).
%
%Input arguments
%———————————————
%  u_n : Array (m,) : Solution vector at step n
%
%  dt : number : Time step
%
%  M : Array (m, m) : Mass matrix of ODE system
%
%  K : Array (m, m) : Stiffness matrix of ODE system
%
%  F : Array (m,) : Right—hand side (force vector) of ODE system
%
%Output arguments
%————————————————
%  u_np1 : Array (m,) : Solution vector at step n+1
```

(c) Consider the linear, scalar ODE often used as a model problem in ODE stability theory:

$$\begin{aligned} y' + \lambda y &= 0 \qquad t \in (0, T] \\ y(0) &= \beta. \end{aligned}$$

- What is the critical time step for the forward Euler method to be stable?
- Use this model problem with $\lambda = \beta = 1$ to test your implementation of forward and backward Euler. That is, compute the solution numerically using your code and compare to the exact solution

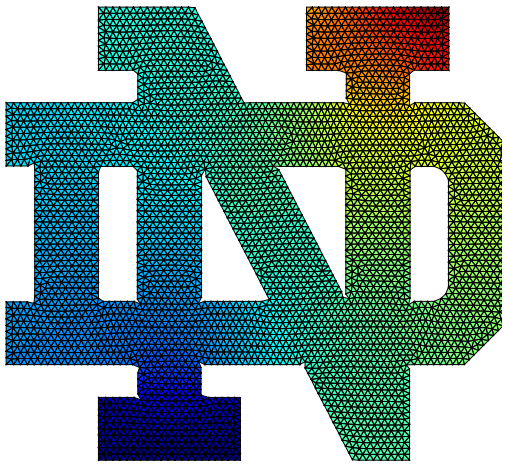
$$y(t) = \beta e^{-\lambda t}.$$

  Make sure you choose your time step sufficiently small so both calculations are stable and accurate. In a single figure, plot the exact solution and the solution you obtain using the forward and backward Euler methods.

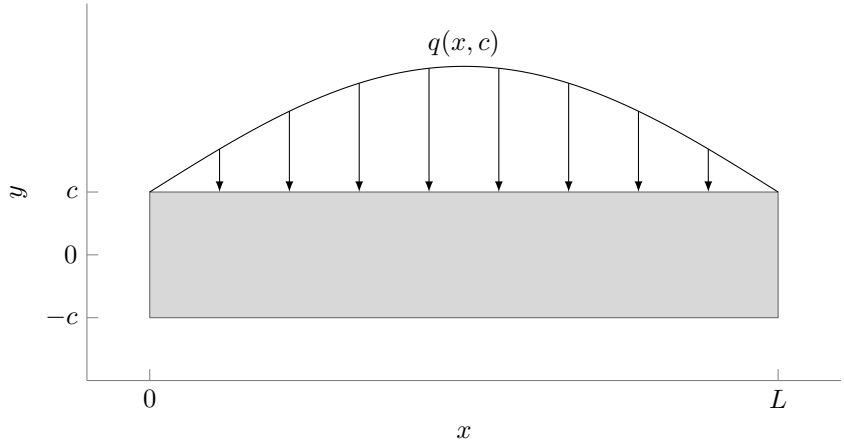
(d) Now consider the finite element system provided for you in `fem_mat_for_hwk5prob2d.mat`. You are provided the finite element mass ($\boldsymbol{M}_{ff}$) and stiffness ($\boldsymbol{K}_{ff}$) matrices and the force vector ($\boldsymbol{F}_f$) after the application of static condensation. You are also provided information about the mesh and essential boundary conditions of the problem as well as the initial condition for the free degrees of freedom (those without essential boundary conditions) $\boldsymbol{u}_{f,0}$. The system of ODEs governing the FEM approximation of the time-dependent PDE under consideration is

$$\boldsymbol{M}_{ff}\dot{\boldsymbol{u}}_f(t) + \boldsymbol{K}_{ff}\boldsymbol{u}_f(t) + \boldsymbol{F}_f = 0, \quad t \in (0, 2]$$
$$\boldsymbol{u}_f(0) = \boldsymbol{u}_{f,0}$$

- What is the critical time step size for the forward Euler method to be stable? Since this cannot be derived analytically in this case, examine the eigenvalues of the update matrix for a number of different time step sizes.
- Solve the system of ODEs using the backward Euler method. Take $\Delta t_n = 0.01$. The code provided in `prob2.m` will assist with this task and generate an animation of the PDE solution evolving over time. Submit a plot of the solution at the final time step (should look like the solution in the figure below).



**Problem 3:** (30 points) From S. Govindjee, UC Berkeley. Consider the two-dimensional beam subject to a transverse sinusoidal load below.




Let $u(x, y)$ denote the $x$-dispalcement and $v(x, y)$ the $y$-displacement. The exact solution of the $y$-displacement along the centerline of the beam with the boundary conditions

$$u(0, 0) = v(0, 0) = v(L, 0) = 0, \qquad q(x, c) = q_0 \sin\left(\frac{\pi x}{L}\right),$$

and the geometric condition $L \gg c$, is

$$v(x,0) = -\frac{3q_0 L^4}{2c^3\pi^4 E}\sin\left(\frac{\pi x}{L}\right)\left[1 + \frac{1+\nu}{2}\frac{\pi c}{L}\tanh\left(\frac{\pi c}{L}\right)\right].$$

This analytical solution was derived under the plane stress assumptions. Select reasonable values for the geometry (the $L/c$ ratio should be at least 10) and material properties (stiffness $E$ and Poissson ratio $\nu$).

(a) What is the analytical solution at $(x, y) = (L/2, 0)$ for the parameters you chose?

(b) Model this system in COMSOL. Be sure to use the plane stress assumption and apply boundary conditions exactly as specified above i.e., do not fix the displacements along an entire edge. For the discretization, consider both linear triangles and quadrilateral elements on a sequence of at least four meshes each of increasing refinement. For each mesh, compute the solution and output $v(L/2, 0)$.

- COMSOL does not allow you to take the sine of a number with *units*. Therefore, to prescribe the distributed load $\sin(\pi x/L)$, this must be entered as `sin(pi*(x/L[m]))` if you are working in units of meters.

(c) On a single figure, plot $v(L/2, 0)$ versus the number of elements for both mesh sequences. Also include the exact solution as a horizontal line (since it does not depend on the number of elements). What do you observe about the accuracy of triangular vs. quadrilateral elements for bending problems?