Numerical Simulation of Sensors and Actuators
Date: April 2020
Assignment due: 2020-05-19 at 11:55pm

Lehrstuhl für Sensorik
Universität Erlangen-Nürnberg
Daniel Kiefer

Assignment 0: FEM Implementation – Modal Analysis and Time Stepping

A typical string of a guitar has scale length $L = 0.660\,\mathrm{m}$ and a linear mass density of $\mu = 3\,\mathrm{g/m}$. Assume that the string is tightened to a tension of $T = 64\,\mathrm{N}$. Denote the coordinate along the string with x and the transversal displacement of the string with u. The free motion of the string that is fixed at both ends is described by the following boundary value problem (BVP):

$$Tu'' = \mu\ddot{u} \quad \text{on} \quad x \in \Omega = [0, L]\,, \tag{1a}$$

$$u(0) = 0\,, \tag{1b}$$

$$u(L) = 0\,, \tag{1c}$$

In this assignment you will implement a Finite Element code to

- compute the harmonic vibrations of the string and
- visualize a disturbance traveling along the string (lossless transmission line).

Note: This assignment builds upon *Assignment 0* of *CAE of Sensors and Actuators* from last semester. You shall extend the code in order to solve eigenvalue problems and time-dependent problems. A reference implementation of last semester's code is provided on StudOn. You can either use this code or your own code as a starting point. Re-use of the code whatever you deem convenient.

1 Preparation: Galerkin discretization

1.1. Write down the weak form of the BVP (1).

1.2. Perform a Galerkin discretization of the resulting weak form. Thereby, do not consider the BCs directly. Instead, use the ansatz

$$u(x) \approx \sum_{j=1}^{N} N_j(x) u_j$$

$$v(x) \approx \sum_{i=1}^{N} N_i(x) v_i$$

for the test function $v(x)$ and the trial function $u(x)$. This will yield a semi-discrete linear system of the form

$$-\mathbf{K} \cdot \underline{u}(t) = \mathbf{M} \cdot \frac{\partial^2}{\partial t^2} \underline{u}(t)\,, \tag{2}$$

with the global stiffness matrix $\mathbf{K}$ and the global mass matrix $\mathbf{M}$.

1.3. Write down the expressions of the components k^e_{ab} of the *element stiffness matrix* using linear shape functions in terms of local coordinates (reference element). Which components are equal?

1.4. Write down the expressions of the components m^e_{ab} of the *element mass matrix* using linear shape functions in terms of local coordinates (reference element). Which components are equal?

2 FE implementation

Your code should be structured in several functions, each contained in a separate file. Make sure that you have at least the following set of files:

2.1. `generateMesh.m`: returns a Matlab structure fully-describing the mesh, including the node connectivity.

2.2. `jacobi.m`: returns the Jacobi matrix of an element. The Jacobi matrix reduces to a scalar in the 1D-case, which is simply the element size.

2.3. `elementStiffness.m`: returns the element stiffness matrix.

2.4. `elementMass.m`: returns the element mass matrix.

2.5. `assembleMatrix.m`: assembles the element matrices into a global matrix.

2.6. `newmarkScheme.m`: performs the time-stepping.

2.7. `modal_analysis.m`: the main script for computing the string's modes.

2.8. `transient_analysis.m`: the main script for performing the transient analysis.

Tip: After implementing a function, test if it is doing what you expect. Only if it passes the test you should proceed to the next function.

You can and should re-use the functions from last semester, whenever possible and suitable. We suggest to proceed as follows:

2.9. Implement the `generateMesh(domain, nElems)` function.

2.10. Implement the `jacobi(mesh, elem)` function.

2.11. Implement the `elementStiffness(..)` function. Pass as argument whatever you deem necessary. You can choose whether to perform the integration beforehand on paper and hard-code the solution or if you prefer to perform a numerical integration. In any case, the code should be valid for any equidistant mesh and tension T.

2.12. Implement the `elementMass(..)` function. Pass as argument whatever you deem necessary. You can choose whether to perform the integration beforehand on paper and hard-code the solution or if you prefer to perform a numerical integration. In any case, the code should be valid for any equidistant mesh and mass density μ.

2.13. Implement the `assembleMatrix()` function. Pass as argument whatever you deem necessary. One argument could be a function handle to either of the element matrices, i.e., `@elementStiffness` or `@elementMass`. That way, you can use the same function to assemble both of the global matrices.

The main scripts and the Newmark scheme will be implemented in the next two sections.

3 Modal Analysis

In this section we assume a harmonic time-dependence of the displacements, i.e.,

$$\underline{u}(t) = \underline{\hat{u}}\, \mathrm{e}^{\mathrm{i}\omega t}\,, \tag{3}$$

with angular frequency $\omega = 2\pi f$. We consider it as understood, that only the real part of the complex valued function $\underline{u}(t)$ in (3) is the actual solution.

3.1. Insert the ansatz (3) into your semi-discrete Galerkin formulation from task 1.2. and simplify the resulting expression as much as possible. If correct, no dependence on the exponential function should remain. The obtained equation is often referred to as a *generalized eigenvalue problem* and the eigenvalue is $\lambda = \omega^2$.

3.2. Convert the generalized eigenvalue problem of task 3.1. to an ordinary eigenvalue problem, i.e., of the form $\mathbf{D} \cdot \underline{u} = \lambda \underline{u}$.

Now you can implement the main script `modal_analysis.m`:

3.3. Define all physical and mesh-relevant parameters you need.

3.4. Generate the mesh.

3.5. In the mesh, set the nodes where Dirichlet boundary condition will be applied.

3.6. Assemble the global stiffness and mass matrices. Remember that these are the full matrices that do not yet incorporate the boundary conditions.

3.7. Incorporate the boundary conditions into the obtained matrices.

3.8. Using the resulting $\mathbf{K}$ and $\mathbf{M}$ matrices, compute the matrix $\mathbf{D}$ that you obtained in task 3.2.

3.9. Inspect the $\mathbf{D}$ matrix you obtained:

3.9.1. Do you have zero-valued columns and rows? Which ones? Why? What would happen if you were to compute the eigenvalues of this matrix? Try it out using `eig()`. What do you observe?

3.9.2. Is the matrix symmetric? Should it be symmetric? What kind of eigenvalues does a symmetric Matrix have? Hint: You can use the Matlab function `issymmetric()`.

3.9.3. We computed the matrix $\mathbf{D}$ numerically and the asymmetry could be due to numerical inaccuracy. Is the matrix almost symmetric? You can test this by inspecting the maximum deviation from zero of $\mathbf{D} - \mathbf{D}^T$ because this should be exactly zero for a symmetric matrix. Floating point arithmetic always generates errors *relative to the absolute* values of the numbers. That is why you should compare the deviation to the maximum absolute entry in $\mathbf{D}$. Better even: first normalize $\mathbf{D}$ in Matlab to `D/norm(D)`. How close is this maximum deviation to the machine precision `eps`? Can it be said, that the matrix $\mathbf{D}$ is symmetric within machine precision?
Hint: the transposition operator in Matlab is `.'`, while the `'`-operator is the conjugate transpose!

3.10. Reduce the $\mathbf{D}$ matrix to the degrees of freedom that need to be solved for. Hint: `setdiff()` might be a useful function.

3.11. Compute the eigenvalues λ and corresponding eigenvectors $\underline{u}$ using the Matlab function `eig()`.

3.12. Sort the eigenvalues from lowest to highest. Make sure that the eigenvectors are sorted in the same way. Hint: use `diag()` to extract a matrix's diagonal and use `sort()` to obtain the permutation that you need to apply to the columns of the matrix containing all the eigenvectors.

3.13. Extend all eigenvectors (which have been computed only at the inner/free nodes) with the Dirichlet values.

3.14. Compute the eigenfrequencies f from the eigenvalues λ.

3.15. What is the fundamental frequency of the guitar string we are analyzing? Which note does it correspond to?

Now assess and visualize your solution:

3.16. Compute the eigenfrequencies of the string using the analytical formula

$$f_n = \frac{n}{2L}\sqrt{\frac{T}{\mu}}, \quad n \in \mathbb{Z}\backslash 0\,. \tag{4}$$

3.17. Compute the relative deviation between the analytical and the numerical solution. How large are the deviations for the first four modes in percent? Is this a good correspondence?

3.18. Plot the modal structures (eigenvectors) of the first four modes in one single figure. Remember to always label the axes and add a legend if necessary.

3.19. Generate an animation of the modal structures of the first four modes. Remember that the time-dependence of the vibrations is given by the real part of (3). You will find a skeleton for how to generate the animation in the code provided on StudOn.

Remark: In practice, you would not first reduce the generalized eigenvalue problem to an ordinary eigenvalue problem before solving. Instead, numerical solution methods are directly applied to the generalized eigenvalue problem as obtained in task 3.1. You can also do this using the Matlab function `eig()`. Try it out if you want! A direct solution procedure does not only avoid the computation of the form $\mathbf{A}^{-1}\mathbf{B}$, but is also more general because it might be used even when $\mathbf{A}$ is not invertible!

4 Transient analysis

In this section, we analyze the same fixed string as before but with respect to another application: a transmission line. We are, therefore, interested in how a given initial disturbance propagates along the string.

4.1. What is the wave speed at which this disturbance propagates?

For a full analysis we need to solve the transient problem as given in (2). To solve this problem, you will implement and apply the Newmark scheme (Newmark-beta method). You learned this method in *CAE of Sensors and Actuators* for solving transient acoustics and mechanics.

4.2. Perform the Newmark time-discretization by writing down the discrete system of equations. These should depend only on

- the quantities that are used in the semi-discrete Galerkin system (2),
- the known values $\underline{u}_\mathrm{n}$, $\underline{\dot{u}}_\mathrm{n}$ and $\underline{\ddot{u}}_\mathrm{n}$ of the "current time step", as well as
- the unknown values for the "next time step" $\underline{u}_{\mathrm{n}+1}$, $\underline{\dot{u}}_{\mathrm{n}+1}$ and $\underline{\ddot{u}}_{\mathrm{n}+1}$.

The latter are the ones that need to be solved for. How many equations do you need for this? Do you have enough equations?

4.3. Re-write the equations so that only the unknowns $\underline{u}_{\mathrm{n}+1}$, $\underline{\dot{u}}_{\mathrm{n}+1}$ and $\underline{\ddot{u}}_{\mathrm{n}+1}$ are on the left hand side while the right hand side contains only known quantities.

Implement the time-stepping scheme in the function `newmarkScheme(M, K, u0, v0, Tend, dt,... betta, gamma)`. As arguments you will pass the mass and stiffness matrix with incorporated boundary conditions, the initial displacement and velocity vector, the ending time of the investigated time interval [0,Tend], the time-step size and the Newmark parameters β and γ. For the implementation you can proceed as follows:

4.4. Initialize variables for the current values $\underline{u}_\mathrm{n}$, $\underline{\dot{u}}_\mathrm{n}$ and $\underline{\ddot{u}}_\mathrm{n}$ (the acceleration is zero).

4.5. Determine the total number of time steps.

4.6. Allocate memory for the solution by creating a zero-matrix `u` of appropriate dimensions. The nth column shall contain the solution vector at time $t = (n-1)\,\mathrm{d}t = t_{n-1}$.

4.7. Initialize the first column of `u` with `u0`.

4.8. Compute the matrices appearing in the equations you derived in task 4.3..

4.9. Loop over all time steps (remember that the first one is known) and compute the new values:

4.9.1. In which order do you have to compute $\underline{u}_{\mathrm{n}+1}$, $\dot{\underline{u}}_{\mathrm{n}+1}$ and $\ddot{\underline{u}}_{\mathrm{n}+1}$? Compute these vectors one after another in this order.

4.9.2. Save the result of $\underline{u}_{\mathrm{n}+1}$ to the corresponding position in the solution matrix `u`.

4.9.3. Prepare for the next iteration by updating $\underline{u}_{\mathrm{n}}$, $\dot{\underline{u}}_{\mathrm{n}}$ and $\ddot{\underline{u}}_{\mathrm{n}}$ to $\underline{u}_{\mathrm{n}+1}$, $\dot{\underline{u}}_{\mathrm{n}+1}$ and $\ddot{\underline{u}}_{\mathrm{n}+1}$, respectively.

Lastly, implement the main script `transient_analysis.m`:

4.10. You can re-use the initial part of `modal_analysis.m`, which you implemented in tasks 3.3. to 3.7..

4.11. Define the initial displacement $\underline{u}_0$ and velocity $\underline{v}_0$:

4.11.1. $\underline{u}_0$: This is the initial "disturbance". Use a Gauss-modulated cosine centered at $x_0 = L/3$, a 1-sigma width of $w = 4\,\mathrm{cm}$, and a center wavelength $\lambda_\mathrm{c} = 6\,\mathrm{cm}$. Make sure that the phase of the cosine is zero at x_0. Write down the expression for $\underline{u}_0$ before implementing it.

4.11.2. $\underline{v}_0 = \dot{\underline{u}}_0 = \underline{0}$.

4.11.3. Plot $\underline{u}_0$ and verify it corresponds to what you expected.

4.12. Define your simulation length such that a pulse traveling in positive x-direction and being reflected at $x = L$ has enough time to get back to its original position at $x_0 = L/3$. Write down the expression for T_end.

4.13. Set $\beta = 1/4$ and $\gamma = 1/2$.

4.14. Choose an appropriate time step $\mathrm{d}t$. What values for the time step yield stable solutions? What if you had chosen β and γ differently?

4.15. Call the function `newmarkScheme()` to perform the time-stepping.

4.16. Create a distance-time plot: use `imagesc()` to plot the local amplitude at x and t with a color code. Make sure to use a *perceptually uniform* color map, e.g., `parula`.

4.17. Animate the solution. You will find a skeleton for how to generate animations in the code provided on StudOn.

Analyze the result:

4.1. Describe the physical phenomena you see in your animation.

Remark: A lossless electrical transmission line obeys the same equation as the mechanical transmission line considered here. In practice, however, losses need to be considered, because they do not only produce damping of the waves, but also results in a dispersive wave propagation.

5 OPTIONAL task

5.1. Generate an animation showing how a Gaussian pulse propagates in a lossy electrical transmission line (telegrapher's equation):

$$u''(x,t) - LC\ddot{u}(x,t) = (RC + GL)\dot{u}(x,t) + GRu(x,t)\,, \tag{5}$$

where the unknown $u(x,t)$ this time represents the voltage. At $f = 1\,\mathrm{kHz}$, a telephone cable has $L = 0.61\,\mathrm{mH/km}$ and $C = 51.6\,\mathrm{nF/km}$, while the losses are $R = 172\,\Omega/\mathrm{km}$ and $G = 0.072\,\mu\mathrm{S/km}$.
Note 1: Because there is no new term involving a spacial derivative, you do not need to setup any additional FE-matrix.
Note 2: If you prefer, you can use Matlab's `ode45()` function for time stepping.

Submitting your assignment

Hand in your

- code and
- a `Results.pdf` file with short answers to all questions within this assignment sheet (label each answer with its task number). All plots you are asked to create should also be contained in this file at the position of the corresponding task. Animations should be reduced to just one frame at a "representative" time. You can, for example, use Word or LaTeX to produce the PDF file.

Upload the above files to the corresponding Assignment section on StudOn:
https://www.studon.fau.de/exc2992793.html.

Important: Make sure that your code **executes** without need of changes. Executing `modal_analysis.m` or `transient_analysis.m` should yield the solutions and plot the results on any computer running MATLAB R2019b or newer.