Homework1_SUN

Shixuan Sun

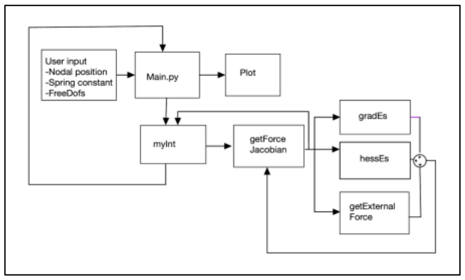
***Abstract*— This is HW1 for 263E**

I. QUESTION1

Pseudocode and explanations of functions are fellows:

```
I devided the scripts into functions and main script. the functions includes getExternalForce,
getForceJacobian, gradEs, hessEs, myInt, and plot, then the main script, which is used to get
user input then call all the necessary function to output simulation.

Main script:
import all the supporting function
create a node,txt file and calls it f
    write the nodal coordinate
create a spring.txt file can calls it f
    write the nodes that the spring connects and the corresponding spring constant

open the file node.txt and stored in path nodes_file_path
initialize a matrix called node_coordinates
try
        open nodes file path
            assign the nodal position to index, and convert into matrix

open spring.txt and stored in springs_file_path
initialize a index_info matrix
initialize a stiffness_info matrix
try
    open spring-file_path
        remove the space, and comma
                if the length of parts = 3
                    try
                        assign nodal location into index, and convert to matrix
                        assign spring constant into index, and convert to matrix

get how many rows does node-matrix has and stored it in N
ndof = 2N

initizalize x_old and u_old
build x_old vector forl node.txt

calculate the spring rest length
initialize the mass vector, and assign 1 to it
specify the dt, maxtime, and t vector used in calculation
create and assign free dof to free_DOF
create a vector to store y position of middle node
for every time step int
    calculate the x_new and u_new by calling the myInt
    plot at time 0.1,1,10, and 100
    update the y-position of middle node, and also update the x_old and u_old
plot the figure
the purpose of this function is to output plot given inputs like the coordinate of nodes, spring
constant, free nodes.

getExternalForce:
define a function called getExternalForce, with input of m
    create a weight matricx that looks lile mass matrix
        go over each node, and assign weight to weight matrix
this function take mass matrix as input and output the weight vector.


getForceJacobian:
define a function called getForceJAcobian
    extract number of dof form x_new
    calculate the inertia force, Jacobian
    calculate spring force and jacobian by calling gradEs, and hessEs
    calculate the force and jacobian due to external force
    sum up the force due to inertia, elastic, and external
    sum up the jacobian due to inertia, elastic, and external
this function takes old and new positions, old velocity, stiffness matrix, index matrix, mass
matrix, dt, and spring rest length, then it utilized gradEs, and hessEs to calculate the negative
force and hessian of the elastic energy.


gradEs:
define a function called gradEs
    create a 1x4 matrix called F
    then it calculate the gradient of elastic energy with respect to xk yk xk+1 yk+1
this function calculate the gradient of elastic energy when inputs xk,yk,xkp1,ykp1,spring rest
length, and k.


hessEs:
define a function called hessEs
    create a 4x4 matric called J
    calculate the hessian of elastic energy and stores them into J
this function calculate the hessian of elastic energy when inputs xk,yk,xkp1,ykp1,spring rest
length, and k.


myInt:
define a function called myInt
    initialize guess that is taken form the x_old
    initialize error and eps
        while err>eps
            excuate the Newton-raphson
this function calculate x_new and y_new given the inputs of t_new, x_old, u_old, free_DOF,
stiffness_matrix, index_matrix, m, dt, l_k.


plot:
define a function called plot
    it loop over each spring to get the position then plot them
this function plot the position of nodes connected by sprinsg given input of x, index matrix, and
time setp
```

Figure1. A block diagram that shows how the functions interact



II. QUESTION 2

The choice of Δt is vital in simulation. Generally, a small Δt will lead to an accurate result, but it will take much longer compared to a large Δt. On the other hand, a large Δt will enable a faster simulation, but will not capture some instances during each Δt. In our simulation, a Δt of 0.1, and 0.01 will result the following displacement of y-axis of free nodes vs. Time plots.

Figure2. displacement of y-axis of free nodes vs. Time, Δt = 0.1s

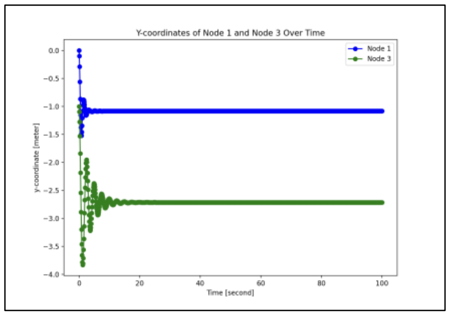
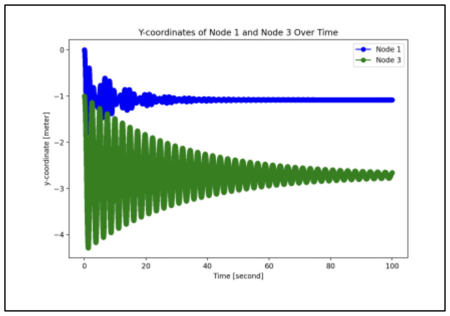


Figure3. displacement of y-axis of free nodes vs. Time, Δt = 0.01s



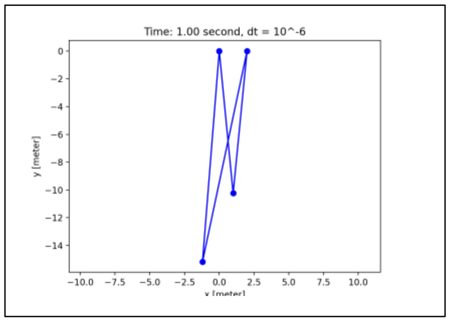
As we can see from Figure2, the oscillation dies out quickly, but the system does not have damping; this artificial damping is introduced by the implicit Erler itself and may not reflect the true physical behavior of the system.

In Figure3, the 3rd node keeps oscillating towards the end of the simulation, which is more reasonable for an actual physical system, since there is no damping exists in the system.

III. Question 3

The explicit method still falls, even if the Δt is 0.000001, as shown in Figure 4.
In this case, the implicit Euler is more stable, thus it is preferred in this case.

Figure4. displacement of y-axis of nodes vs. Time, $\Delta t = 10^{-6}s$



IV. Question 4

The explicit Euler introduce artificial or numerical damping because it takes the xtkp1, which is the next time step to predict the motion, thus the Explicit Euler only uses the old and new x to predict the motion; the physics in between each time step is not taken into account, there for creating artificial damping that makes the energy die out.

On the other hand, Newmark also uses linear interpolation to address the physics during each time step, thus conserving the energy.
Consider a simple 1-DOF system:

$$m\ddot{u} + ku = 0$$

Then the natural frequency is :

$$\omega = \sqrt{\frac{k}{m}}$$

The Newmark-β integration formulas are :

$$u_{n+1} = u_n + \Delta t\, \dot{u}_n + \Delta t^2 \left[(\tfrac{1}{2} - \beta)\ddot{u}_n + \beta \ddot{u}_{n+1} \right],$$
$$\dot{u}_{n+1} = \dot{u}_n + \Delta t \left[(1-\gamma)\ddot{u}_n + \gamma \ddot{u}_{n+1} \right].$$

For an undamped system, $\ddot{u} = -\omega_n^2 u$, if we slove for u_{n+1}

$$u_{n+1} = a_1 u_n + a_2 u_{n-1},$$

Where $a_1$and a_2 depends on β, γ, ω_n, and Δt
If we substitute the parameters in, we have:

$$u_{n+1} = \frac{(2 - \omega_n^2 \Delta t^2)}{(1 + \frac{\omega_n^2 \Delta t^2}{4})} u_n - \frac{(1 - \frac{\omega_n^2 \Delta t^2}{4})}{(1 + \frac{\omega_n^2 \Delta t^2}{4})} u_{n-1}.$$

If we assume a harmonic solution, where $u_n = e^{in\theta}, u_{n+1} = e^{i\theta}, and\ u_{n-1} = e^{-i\theta}$, and plplugut them in:

$$e^{i\theta} = \frac{(2 - \omega_n^2 \Delta t^2)}{(1 + \frac{\omega_n^2 \Delta t^2}{4})} - \frac{(1 - \frac{\omega_n^2 \Delta t^2}{4})}{(1 + \frac{\omega_n^2 \Delta t^2}{4})} e^{-i\theta}.$$

If we calculate the magnitude, it will be 1, thus no energy loss.

Acknowledgment

The source code is taken from Dr.khalid's Google Colab notebook shared on Slack, then modified by Shixuan Sun to accomplish HW1