1. Pseudocode and code structure: Write pseudocode for your simulator describing its main logic and workflow. Briefly describe the main functions and scripts in your implementation (2–3 sentences each), including their inputs and outputs. Create a simple block diagram showing how the functions or scripts interact (i.e., which one calls which). There is no strict format requirement—use your best judgment to make it understandable to someone with an undergraduate-level engineering background.

Pseudocode:

1. INITIALIZATION:

- Read node coordinates from nodes.txt
- Read spring properties from springs.txt
- Initialize positions (x_old), velocities (u_old), masses (m)
- Calculate spring rest lengths (l_k)
- Define free degrees of freedom

2. TIME INTEGRATION LOOP (for each time step):

- Call myInt() integrator with current state
- myInt() uses Newton-Raphson iteration:
  a. Compute force residual f and Jacobian J
  b. Solve J_free * Δx_free = f_free
  c. Update positions: x_new = x_old - Δx
- Update velocities: u_new = (x_new - x_old)/Δt
- Store results and plot periodically

3. POST-PROCESSING:

- Plot final configuration
- Output displacement vs time graphs

Main Functions Description:

gradEs(xk, yk, xkp1, ykp1, l_k, k)

- **Input**: Coordinates of two connected nodes, rest length, stiffness
- **Output**: 4×1 gradient vector of stretching energy
- **Purpose**: Computes spring forces using analytical derivatives

hessEs(xk, yk, xkp1, ykp1, l_k, k)

- **Input**: Same as gradEs
- **Output**: 4×4 Hessian matrix (stiffness matrix)
- **Purpose**: Provides second derivatives for Newton-Raphson solver

getForceJacobian(x_new, x_old, u_old, stiffness_matrix, index_matrix, m, dt, l_k)

- **Input**: Current/previous states, physical properties
- **Output**: Force residual vector and Jacobian matrix
- **Purpose**: Assembles global force vector and tangent stiffness matrix

myInt(t_new, x_old, u_old, free_DOF, stiffness_matrix, index_matrix, m, dt, l_k)

- **Input**: Time, previous state, physical parameters
- **Output**: Updated positions and velocities
- **Purpose**: Implicit Euler integrator using Newton-Raphson iteration

Function Interaction Diagram:

Text, main script

↓

myInt() [implicit integrator]

↓

getForceJacobian()

↓

gradEs() + hessEs() [per spring]

↓

Newton-Raphson solver

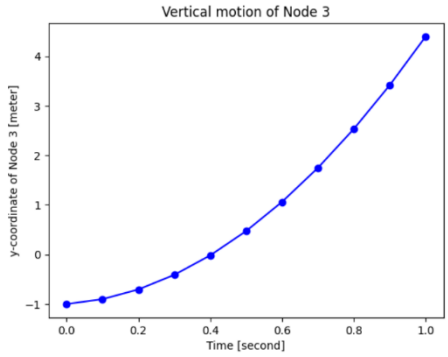
2. How do you choose an appropriate time step size Δt?

Choosing an appropriate time step size Δt involves trying to balance the numerical stability and the computational efficiency. For explicit methods like Explicit Euler, Δt must be small enough to solve the fastest dynamics in the system. Specifically, it must be smaller than $2/\omega_{max}$ where ω_{max} is the highest natural frequency, to prevent instability. This will usually require very small, time steps (as demonstrated by the failure even at $\Delta t = 10^{-6}$ s in problem 3). In contrast, implicit methods like Implicit Euler are unconditionally stable, allowing for much larger time steps (like $\Delta t = 0.1$ s) without numerical divergence. Therefore, the choice depends on the integrator: explicit methods demand very small Δt for stability, while implicit methods allow larger Δt focused on accuracy rather than stability.

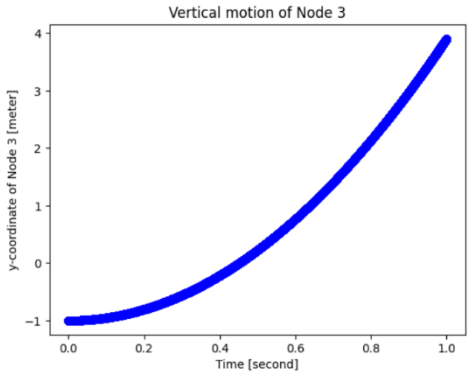
3. Simulate the spring network using Explicit Euler for t ∈ [0, 1] s. If it becomes unstable, reduce Δt and try again. If it still fails even at Δt = 10−6 s, state this in the report. Explain which method (implicit vs. explicit) is preferable for this spring network and why.

Results:

- With $\Delta t = 0.1$ s: **UNSTABLE**



- With $\Delta t = 10^{-6}$ s: **STILL UNSTABLE**



The Explicit Euler method proved completely unsuitable for this spring network, failing even with an extremely small time-step of $\Delta t = 10^{-6}$ seconds. The simulation became unstable almost immediately, with node positions diverging to infinity due to numerical instability. This failure demonstrates why Implicit Euler is much more preferable for stiff systems like spring networks while Explicit Euler demands tiny time steps for basic stability, Implicit Euler remains stable with much larger steps ($\Delta t = 0.1$ s in our case), making it both computationally efficient and reliable for this type of problem.

4. The simulation with implicit Euler appears to reach a static state (numerical damping). Read about the Newmark–β family of time integrators and explain how such integrators can mitigate such artificial damping.

The artificial damping in Implicit Euler that makes the system settle unnaturally fast comes from its numerical formulation, which inherently dissipates energy. The Newmark-β family of integrators solves this by providing a tunable alternative: by selecting specific parameters ($\gamma = 0.5$, $\beta = 0.25$), you get a method that is both unconditionally stable and second-order accurate. This particular combination, known as the "constant average acceleration" method, introduces no numerical damping, allowing the spring network to oscillate realistically for as long as it physically should, rather than being artificially damped.