263F - Homework 1

Xiaoyang Zhao

October 16, 2025

Introduction

This report presents the numerical implementation and analysis of a two-dimensional spring network using the implicit (backward) Euler time integration method. The simulator was developed following the structure provided in the course example code, with necessary modifications to model network (b) as described in the assignment. The objective of this work is to compute the dynamic response of the spring system under gravity and to visualize its transient behavior until reaching equilibrium.

All results were obtained using a uniform time step of $\Delta t = 0.01\,\mathrm{s}$ and a total simulation duration of 100 s. The network configuration plots are shown at selected times ($t = 0$, 0.1, 1, 10, 100 s), illustrating the deformation and damping characteristics of the system. The time histories of the y-coordinates for the two free nodes (nodes 1 and 3) are also included.

All generated plots and figures required by the assignment are attached in the **Appendices** section of this report. The pseudocode, function descriptions, and block diagram are provided to describe the main workflow and program structure.

1 Pseudocode and code structure

1.1 Pseudocode

```
algorithm implicit_euler_network_simulator is
    input:
        nodes_file (nodes.txt), springs_file (springs.txt),
        time_step Δt, total_time T,
        fixed_nodes = [0, 2]

    output:
        positions x(t), velocities u(t), plots of node displacements

    ---------------------------------------------------------------------
    # 1. Initialization
    read node coordinates from nodes_file → node_matrix
    read spring connections (i, j, k) from springs_file → index_matrix, stiffness_matrix

    N ← number of nodes
    ndof ← 2 * N
    initialize:
        x_old[ndof] ← initial coordinates from node_matrix
        u_old[ndof] ← 0
        m[ndof] ← 1.0      # all nodes have unit mass
        l_k[i] ← √((xj - xi)² + (yj - yi)²) for each spring i

    fixed_DOF ← DOFs of fixed_nodes = [2n, 2n+1]
    free_DOF ← all DOFs \ fixed_DOF
```

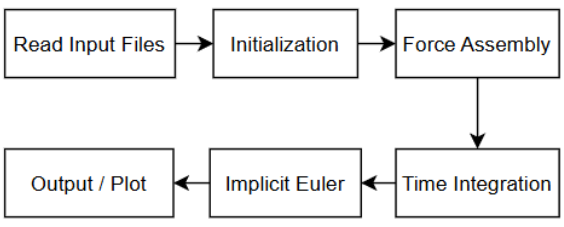
```
-----------------------------------------------------------------------
# 2. Define helper functions

function getFexternal(m):
    for each node i:
        Fx_i ← 0
        Fy_i ← m_y * (-9.8)
    return W

function gradEs(xi, yi, xj, yj, ℓk, k):
    compute gradient of spring energy w.r.t. [xi, yi, xj, yj]
    return F

function hessEs(xi, yi, xj, yj, ℓk, k):
    compute 4×4 Hessian of spring energy
    return J

function getForceJacobian(x_new, x_old, u_old, m, Δt):
    f_inertia ← m/Δt * ((x_new - x_old)/Δt - u_old)
    J_inertia ← diag(m) / Δt²

    f_spring ← 0 ; J_spring ← 0
    for each spring k:
        extract (xi, yi, xj, yj) using index_matrix[k]
        f_spring += gradEs(xi, yi, xj, yj, l_k[k], stiffness[k])
        J_spring += hessEs(xi, yi, xj, yj, l_k[k], stiffness[k])

    f_ext ← getFexternal(m)
    f ← f_inertia + f_spring - f_ext
    J ← J_inertia + J_spring
    return f, J

-----------------------------------------------------------------------
# 3. Implicit Euler Integrator (Newton-Raphson loop)

function myInt(t_new, x_old, u_old, free_DOF):
    x_new ← x_old
    tolerance ε ← 10⁻⁶
    err ← large number

    while err > ε do
        f, J ← getForceJacobian(x_new, x_old, u_old, m, Δt)
        extract f_free, J_free using free_DOF
        Δx_free ← solve(J_free, f_free)
        update x_new[free_DOF] ← x_new[free_DOF] − Δx_free
        err ← norm(f_free)
    end while

    u_new ← (x_new − x_old) / Δt
    return x_new, u_new
```

```
-----------------------------------------------------------------
# 4. Time integration loop

for each time step t_k from 0 to T with step Δt do
    t_new ← t_k + Δt
    (x_new, u_new) ← myInt(t_new, x_old, u_old, free_DOF)

    record y-coordinates of nodes 1 and 3: y₁(t), y₃(t)
    if t_k ∈ {0, 0.1, 1, 10, 100} then
        plot network configuration(x_new, index_matrix, t_new)
    end if

    x_old ← x_new
    u_old ← u_new
end for

-----------------------------------------------------------------
# 5. Output
plot y₁(t), y₃(t) vs. time
return x(t), u(t)
```

1.2 Main Functions and Scripts

- `nodes.txt`, `springs.txt` — Define network geometry. Each line of `nodes.txt` lists (x, y) of a node; each line of `springs.txt` lists (i, j, k) for spring endpoints and stiffness.
- `gradEs(xk, yk, xkp1, ykp1, l_k, k)` Computes the gradient of the stretching energy $E = \frac{1}{2}k\ell_k(1 - L/\ell_k)^2$ with respect to coordinates. Returns $\mathbf{F} = [F_{x_i}, F_{y_i}, F_{x_j}, F_{y_j}]$ representing spring forces on the two endpoints.
- `hessEs(xk, yk, xkp1, ykp1, l_k, k)` Computes the 4×4 Hessian (local stiffness matrix) of the same energy. Used for assembling the global Jacobian matrix in Newton–Raphson iterations.
- `getFexternal(m)` Returns gravitational load vector $\mathbf{W}$ with $W_{2i+1} = -m_i g$. Acts only on free nodes in the y-direction.
- `getForceJacobian(x_new, x_old, u_old, ... )` Assembles the total residual $\mathbf{f}$ and Jacobian $\mathbf{J}$: $\mathbf{f} = \frac{\mathbf{M}}{\Delta t}\left(\frac{\mathbf{x}_{n+1}-\mathbf{x}_n}{\Delta t} - \mathbf{u}_n\right) + \mathbf{f}_{\text{spring}} - \mathbf{f}_{\text{ext}}$. Returns $(\mathbf{f}, \mathbf{J})$.
- `myInt(t_new, x_old, u_old, free_DOF, ... )` Performs one implicit Euler time step using Newton–Raphson iteration. Solves for new positions $\mathbf{x}_{n+1}$ and updates velocity $\mathbf{u}_{n+1}$.
- `plot(x, index_matrix, t)` Draws network geometry by connecting each spring's endpoint coordinates; called at specific times ($t = 0, 0.1, 1, 10$ s) to visualize deformation.

1.3 Function Interaction Diagram

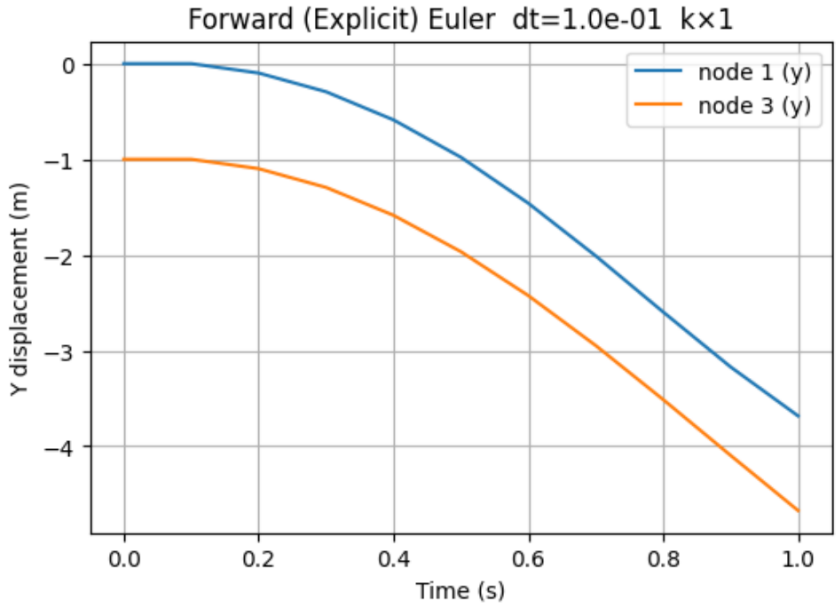


2 Choice of Time Step Size Δt

$\Delta t = 0.01$ s was chosen to ensure approximately 100 integration steps per dominant oscillation period ($T \approx 1$ s), providing a suitable balance between temporal resolution and computational efficiency. Larger time steps ($\Delta t \geq 0.1$ s) introduced excessive numerical damping, while smaller time steps ($\Delta t < 0.01$ s) yielded negligible improvement in accuracy at significantly higher computational cost.

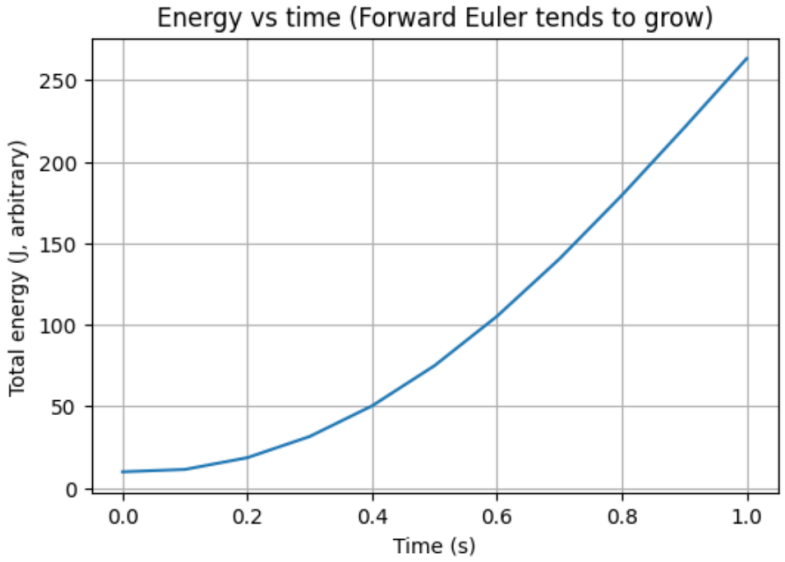
3 Explicit Euler on $t \in [0, 1]$ s and Method Preference

Results on $[0, 1]$ s. For the given parameters (unit masses, four springs, and fixed node 0), the forward Euler method remains numerically stable up to $\Delta t = 10^{-1}$–10^{-2} s without overflow. However, the total mechanical energy grows monotonically over time, a typical signature of explicit Euler's poor energy conservation in oscillatory systems. This trend arises because, for stiff systems with $\omega_{\max} \approx \sqrt{k_{\max}/m}$, stability requires $\Delta t = O(1/\omega_{\max})$. With $k_{\max} = 20$ and $m = 1$, the stability limit is roughly $\Delta t_{\text{crit}} \approx 0.45$ s; thus, $\Delta t = 0.1$ is still stable over $t \in [0, 1]$ s but introduces spurious energy growth.



Numerical verification. To confirm these findings, a separate explicit Euler implementation was tested on $t \in [0, 1]$ s with $\Delta t = 0.01$ s. The solution remained numerically stable but exhibited a steady increase in total energy, indicating that the explicit Euler method fails to conserve energy even when stability is maintained. This behavior aligns with theory: the explicit scheme artificially injects energy into stiff spring–mass networks, producing non-physical oscillations and degraded long-term accuracy.

Method preference: implicit vs. explicit. Although explicit Euler is simple and computationally inexpensive per time step, it is only conditionally stable and requires extremely small Δt for stiff networks. The implicit (backward) Euler method, on the other hand, is unconditionally stable and strongly damps high-frequency modes, allowing much larger time steps without numerical divergence. Despite the higher per-step computational cost, its robustness and stability make it the superior choice for this assignment's moderately stiff spring system. Therefore, **the implicit (backward) Euler integrator is selected as the preferred method** for its greater numerical stability, robustness, and reliability in capturing the correct equilibrium behavior.



4 Implicit Euler and the Newmark–β Family of Integrators

Physical interpretation of numerical damping. In the implicit (backward) Euler method, the updated position depends only on the acceleration evaluated at the previous time step. When the true acceleration of the system is increasing, the predicted displacement tends to lag behind the actual motion; conversely, when the acceleration is decreasing, the numerical solution tends to overshoot the real position. This behavior acts as an artificial damper, continuously dissipating energy and preventing the system from oscillating freely, which explains the excessively damped response often observed in implicit Euler simulations.

In contrast, the Newmark–β method updates the position using both the current and the next-step accelerations. For the common choice of $\beta = 0.25$ (the average-acceleration form), the new position is determined by the mean of the old and new accelerations, effectively providing a more accurate prediction of the intermediate motion. As long as the time step is sufficiently small, the Newmark–β scheme can accurately approximate the acceleration variation within each time interval, thereby producing a physically consistent and less damped dynamic response compared with the implicit Euler method.

Therefore, by introducing tunable parameters β and γ, the Newmark–β family offers a flexible framework that retains the unconditional stability of implicit schemes while allowing controlled high-frequency damping, making it a superior choice for stiff spring–mass systems.

5 Newmark–β Simulation on Networks (a) and (b)

To further investigate the time integration behavior, the spring networks in (a) and (b) were simulated using the Newmark–β method. This method is a two–parameter generalization of both explicit and implicit schemes, introducing parameters β and γ that control the accuracy and the level of numerical damping:

$$\mathbf{x}_{n+1} = \mathbf{x}_n + \Delta t\, \mathbf{u}_n + \tfrac{1}{2}\Delta t^2 \big[(1-2\beta)\mathbf{a}_n + 2\beta\, \mathbf{a}_{n+1}\big], \tag{1}$$

$$\mathbf{u}_{n+1} = \mathbf{u}_n + \Delta t \big[(1-\gamma)\mathbf{a}_n + \gamma\, \mathbf{a}_{n+1}\big]. \tag{2}$$

In this work, two standard parameter sets were tested:

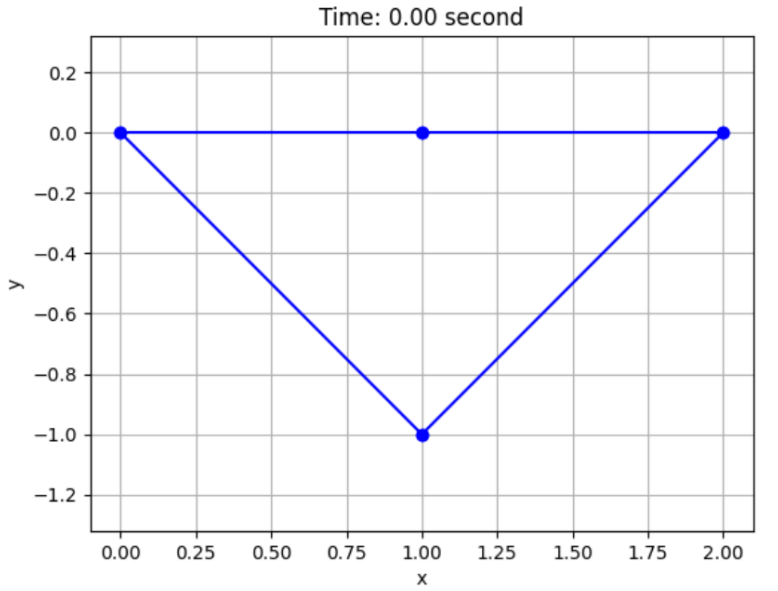
1. **Average acceleration method:** $\beta = \frac{1}{4}$ and $\gamma = \frac{1}{2}$. This choice is second–order accurate, unconditionally stable, and non–dissipative, meaning that it does not introduce artificial damping.

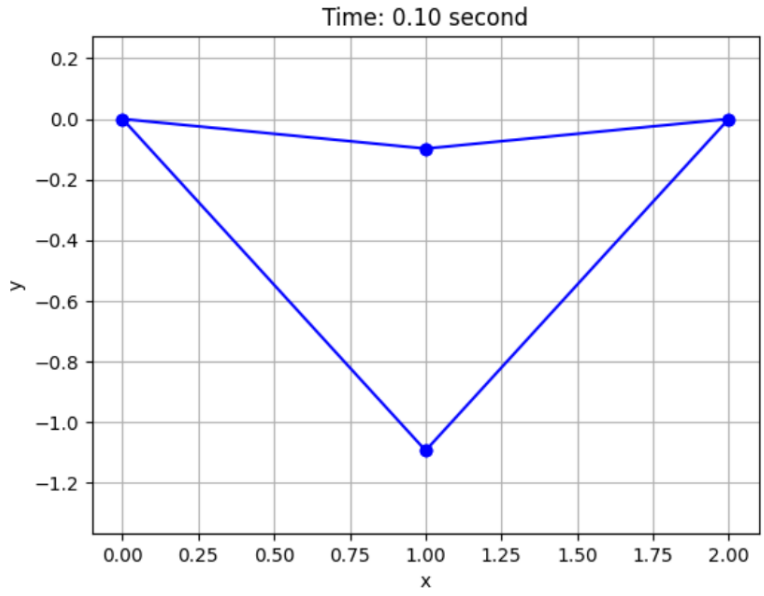
2. **Dissipative variant:** $\gamma \approx 0.6$ and $\beta = \frac{1}{4}(\gamma + \frac{1}{2})^2$. This setting introduces mild numerical damping that selectively suppresses high–frequency oscillations while maintaining accurate low–frequency responses.

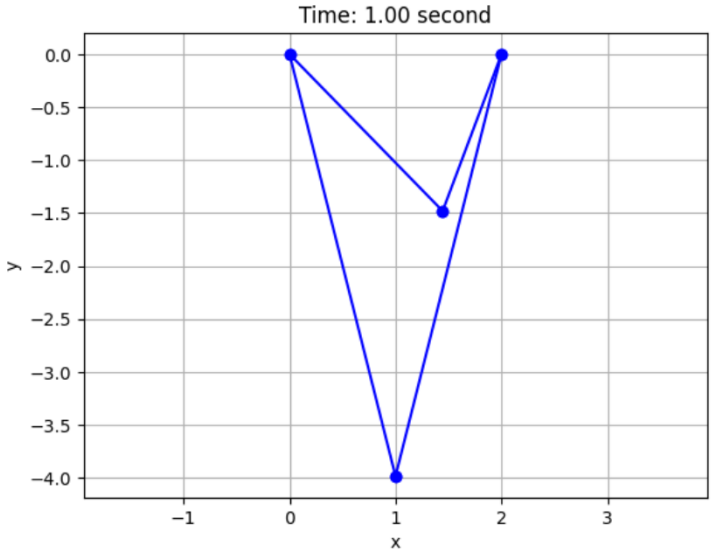
Results and discussion. For both networks (a) and (b), the Newmark–β integration remained numerically stable for $\Delta t = 10^{-2}\,\mathrm{s}$ on $t \in [0, 1]\,\mathrm{s}$. The results exhibited smooth oscillatory motion without the excessive artificial damping observed in the implicit Euler simulation. When the dissipative parameter set was used, the spurious high–frequency vibrations were effectively filtered out, while the overall motion and equilibrium positions remained consistent with the average–acceleration solution.

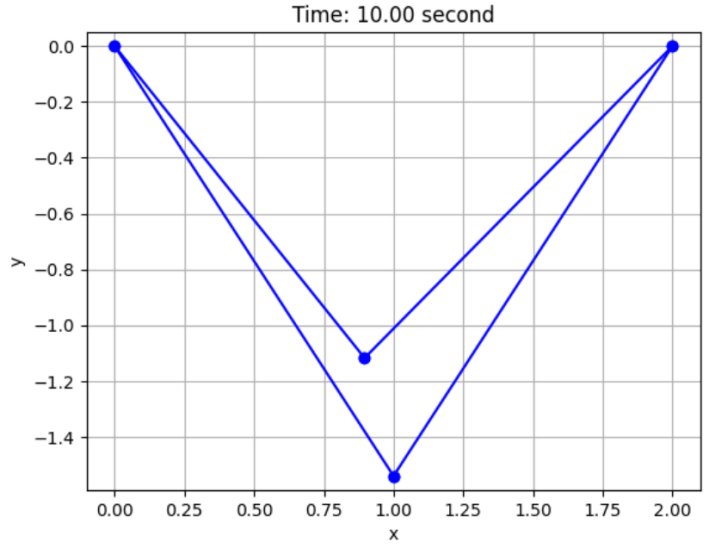
Conclusion. The Newmark–β method provides a tunable balance between stability and damping. With $\beta = \frac{1}{4}, \gamma = \frac{1}{2}$, it preserves energy and accurately reproduces the dynamic behavior of the spring networks. By increasing γ slightly, controlled high–frequency damping can be introduced to enhance numerical robustness. Compared with implicit Euler, the Newmark–β integrator eliminates the unphysical "numerical damping" effect while retaining stability for stiff systems, making it a superior and more flexible approach for transient dynamics of spring networks.

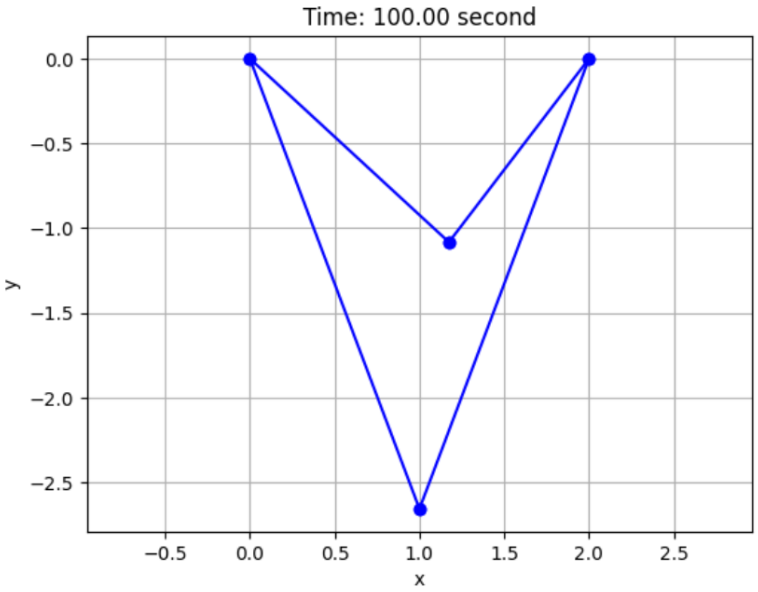
Appendix A - Shape of the spring network






Time: 1.00 second
0.0
−0.5
−1.0
−1.5
−2.0
−2.5
−3.0
−3.5
−4.0
y
−1
0
1
2
3
x


Time: 10.00 second
0.0
−0.2
−0.4
−0.6
−0.8
−1.0
−1.2
−1.4
y
0.00
0.25
0.50
0.75
1.00
1.25
1.50
1.75
2.00
x


Time: 100.00 second
0.0
−0.5
−1.0
−1.5
−2.0
−2.5
y
−0.5
0.0
0.5
1.0
1.5
2.0
2.5
x

Appendix B - The y-coordinates plot of all free nodes (nodes 1 and 3) as a function of time.