COLLEGE OF ARTS + SCIENCES
SUNY POLYTECHNIC INSTITUTE

State University of New York Polytechnic Institute
College of Arts and Sciences

Vibration of a Framework of Springs and Masses

Submitted by:
Gregory Georgiades
Mechanical Engineering Student
David Petrushenko
Mechanical Engineering Student

Submitted to:
Project Advisor
Dr. Andrea Dziubek
Assistant Professor
Department of Mathematics

A differential equations project submitted in partial fulfillment of the requirements for the spring 2017 session of Numerical Differential Equations (MAT460)

Contents

1 Project Definition **3**

2 Physical Problem **5**
2.1 Model Assumptions 6
2.2 Formulating Physical Effects 7

3 Numerical Methods **8**
3.1 Euler Explicit 8
3.2 Euler Symplectic 8
3.3 Störmer-Verlet 9
3.4 4^{th} Order Runge-Kutta 9
3.5 Applying the numerical methods 10

4 Algorithm and Programming **10**

5 Parameter Analysis and Discussion of Results **11**
5.1 Symmetry 11
5.2 Energy Under Symmetric Initial Conditions 12
5.3 Time Step Considerations 14
5.4 High Strength Frameworks 15
5.5 High Mass 15

6 Applications **15**
6.1 Floor Truss 15
6.2 Crane 16

7 Summary of the Project **16**

Appendices **19**

A Course Evaluation **19**
A.1 Gregory Georgiades 19
A.2 David Petrushenko 19

B Python Simulation Code **20**

List of Figures

1 A three-dimensional framework observed on a tower crane. Image retrieved from Google. . . . 3
2 An example of a simple roof truss. Image retrieved from Google. . . . 4
3 General planar framework configuration. Image adapted from Project Guide. . . . 5
4 Free-body diagram of node two. . . . 7
5 General planar framework configuration. . . . 10
6 Symmetric Motion Example . . . 11
7 Unsymmetric Motion Example . . . 12
8 Euler Symplectic Simulation with Initial Loading. . . . 13
9 Runge-Kutta 4^{th} Order Explicit Simulation with Initial Loading. . . . 13
10 Euler Symplectic Simulation with Initial Displacement. . . . 14
11 Runge-Kutta 4^{th} Order Explicit Simulation with Initial Displacement. . . . 14
12 Floor Truss Initial Configuration . . . 16
15 Crane Initial Configuration . . . 16
13 Steps during the Floor Truss Framework Simulation . . . 17
14 Energy Plot for the Floor Truss Framework Simulation . . . 17
16 Steps during the Crane Framework Simulation . . . 18
17 Energy Plot for the Crane Framework Simulation . . . 18

List of Tables

1 Variable nomenclature for framework analysis. . . . 5

Nomenclature

Refer to Table 1 for most of the nomenclature used in this project. Certain variable definitions are not reproduced here to avoid unnecessary repetition and potential variable definition conflicts.

- a = Acceleration
- v = Velocity
- F, f = Force
- $t, \Delta t$ = Time
- G = Acceleration due to Gravity
- w = Energy
- C = Generalized Initial Condition

1 Project Definition

Physical systems encountered in many disciplines of engineering, mathematics, and science are the basis of inquiry and motivation for study. It is often the interest of engineers to develop models in order to characterize their behavior based on a set of initial conditions. Once the behavior of a simple system is well understood, a similar approach may be utilized to develop a model for a more complex system, or one that better captures the behavior of initially simplified one. By varying the parameters, it is possible to study separate components in order to quantify their effects to the overall estimation of system behavior.

Applying the theories of mathematics and science provides a basis for developing a rigorous model. It is important to note the various contributions provided by a variety of factors present in a physical system. Modeling these behaviors is often dependent on a series of underlying mathematical and physical principles governing the evolution of the system. Prior to modeling, it is often the case that each system is observed to gain a preliminary understanding of the characteristics that should be captured. Based on these details, a model may be developed in order to study a particular component or components of a system. After applying the relevant equations, models are conceived in the order of complexity that interests the investigator studying the system.

Figure 1: A three-dimensional framework observed on a tower crane. Image retrieved from Google.

It is common practice to begin with a complex model and introduce assumptions to narrow the scope of study. While keeping a model at its original state is possible, often times it becomes difficult to solve analytically or may be computationally expensive to perform the analysis using sophisticated computational logarithms. In some cases, systems may be simplified with the intent of producing a closed form analytical solution. An example is the commonly used method of approximating nonlinear systems as linear by substituting coefficients for nonlinear terms or expanding them using series. Even though these methods may partially skew the system characteristics, they provide good starting points for analysis to the system dynamics. Alternatively, different methods of evaluation may be exercised to solve the system and compared to determine which yielded better results. Then, the differences between solutions may be quantified to determine the errors associated with the particular method. It is typical to analyze a system and compare to experimental results using the methods available in the literature. After formulating the modeling concepts in a generalized manner, it is typical to add complexity of different conditions to the system in order to study the incremental differences that the model outputs based on the varied inputs.

A common system encountered in mechanical and civil engineering is one made of two-force members that create a unified structure of members that is referred to a framework or truss. Figures 1 and 2 show examples of a three-dimensional and two-dimensional framework structures commonly encountered by engineers. Developing a theoretical model to formulate the predictive behavior of these physical systems provides a means of testing them without having a physical model. In the case

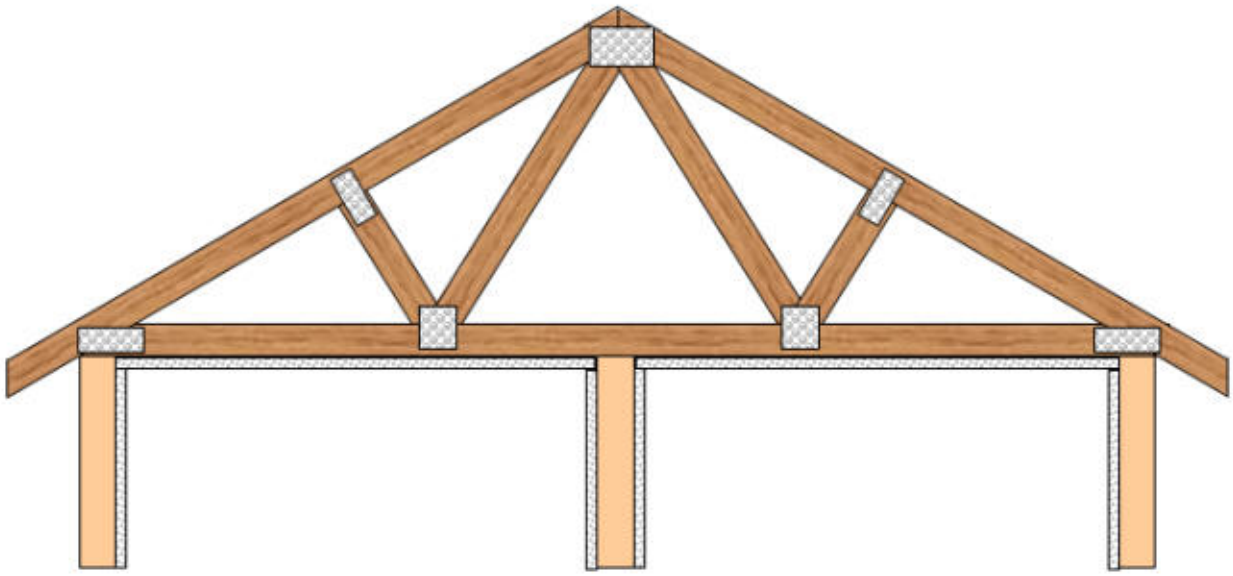

Figure 2: An example of a simple roof truss. Image retrieved from Google.

of designing cranes, it would become very expensive and impractical to study the behavior without developing a theoretical model. The theoretical contributions of modeling systems using simulations available on computer programs show the true potential of combining physical and mathematical concepts. They allow for a viable solution to study engineering systems and provide it for a very low cost as compared to building and testing a system physically.

Trusses and frameworks make a particularly interesting topic to study since they provide an instance of combining theoretical knowledge to applications in industry. These structures are built with minimal material and designed to provide large load to mass ratios which is commonly desired in structural engineering applications when building roofs, floors, highways, and heavy machinery such as the tower crane shown in Fig. 1. The primary interest of our study does not lie in determining the static boundary positions, but rather in the evolution of the system moving towards its final state after the application of some initial conditions. Members of the framework are simplified to be considered as springs and masses connecting to make a general structure able to be defined in any physical shape. Also, it is out intent to quantify the differences in the numerical methods used by applying the conservation of energy principle to quantify some of the numerical errors created while solving the differential equations system. We will explore various methods such as Euler Explicit, Euler Symplectic, Störmer-Verlet, and 4^{th} Order Runge-Kutta to show the differences in their performance as integrators.

2 Physical Problem

The inherent characteristics of a spring-mass framework allow multiple methods to be used in estimating the system geometry as it evolves. Due to its classical connection to mechanical systems, the Newtonian approach was selected as the primary method for propagating changes in the framework. We begin our analysis by introducing the truss framework shown in Fig. 3 and develop a method of analysis. Then, this method is generalized to accommodate any general planar framework system which can be modeled as a spring-mass framework.

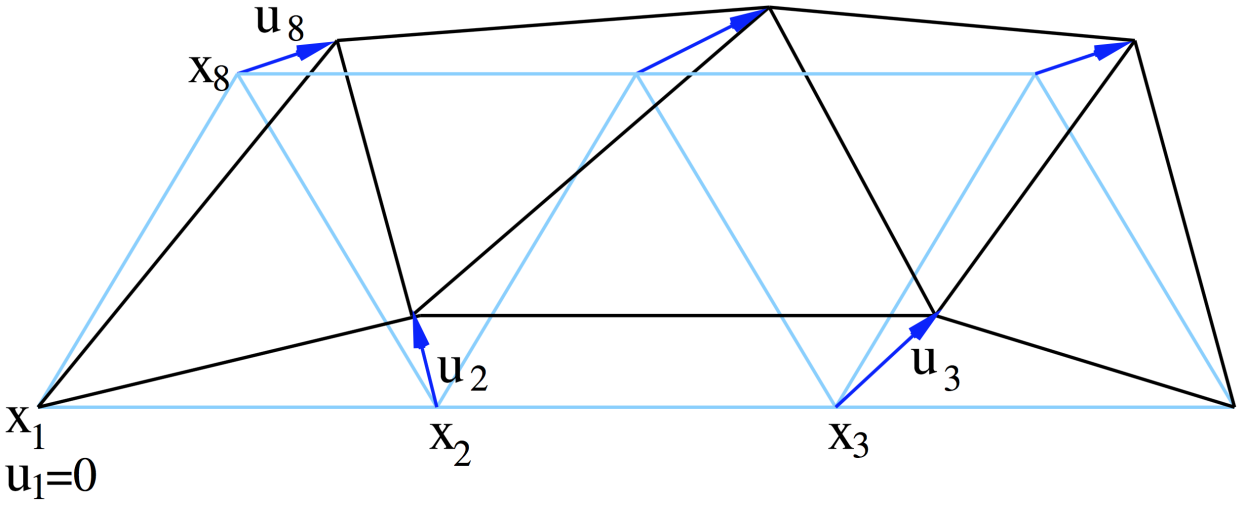

Figure 3: General planar framework configuration. Image adapted from Project Guide.

As a prelude to the analysis, we begin by defining the pertinent variables in Table 1. The variables in the table are primarily defined with respect to the nodes since the method of analysis follows a similar approach.

Notation	General Description
$n \in \mathbb{N}$	Number of nodes in the framework
$m_i > 0$	Mass at node n_i
$s_{ik} \in \mathbb{N}$	Spring connecting node n_i to node n_k
$x_i \in \mathbb{R}^2$	Initial position of rest for node n_i
$u_i \in \mathbb{R}$	Time dependent displacement of node
$F_i \in \mathbb{R}^2$	Force applied to node n_i
$D_i \in \mathbb{R}^2$	Initial displacement applied to node n_i
$k_{ik} \geq 0$	Stiffness of spring connecting node n_i to node n_k
$l_{ik} \geq 0$	Length of spring connecting node n_i to node n_k

Table 1: Variable nomenclature for framework analysis.

In the general case, each node is connected to at least one spring, but any spring may be specified to have zero mass. For this reason, masses are an optional attribute to a particular node but the position of the nodes must be specified for the analysis of the framework.

2.1 Model Assumptions

A few assumptions have been considered in developing the framework model and the motivation for each is briefly described here. In general, a number of considerations were made to narrow the scope of the project but still provide accurate representations to capture the prominent behaviors of the system.

Following the applied character of this study, a few constraints were added to limit the input parameters for the framework. In theory, it is possible to construct a system without any constrained nodes, however since this type of system is physically inconceivable on earth, it will not be considered in the set of analysis cases. Along the same argument, we require that each system has at least one spring and one mass defined. In summary, the input parameters for the system must model a physical system having at least one defined node, spring, and mass. If we consider these inputs, they model a system commonly encountered when studying dynamical systems. A common example is a simple two degree of freedom harmonic oscillator, otherwise known as a simple pendulum, having a spring connecting a mass to a rotating fulcrum (node). It is interesting to note that this set of inputs has a direct connection to simple systems studied in introductory mechanics courses.

As the two main components of the system, the following considerations have been defined for the springs and masses of the framework. The springs in the system are defined in such a way that they connect exactly two nodes and half of the spring's mass is transferred to either node. This is a reasonable assumption since most members in frameworks have linear densities in their force direction. Further, springs are free to rotate about either node that they are attached therefore eliminating any generation of torques within the system. We also allow that springs can act in both tension and compression and they are assumed to exhibit linear stiffness for their full length, experience no fatigue, and provide 100 percent elasticity. We also assume that the masses to which springs connect do not alter the dynamics of the system significantly, in particular the stiffness, since the model follows a lumped mass approach.

In addition to the previously defined assumptions, a few others have been added to simplify the analysis of the system which are more theoretical in character. Although this is not physically possible, we assume that the springs in the system may experience infinite tension and compression and are physically indestructible. This assumption was retained to simplify and speed up computations and as a way of comparing the various methods of analysis. If constraints for compressed and extended lengths were considered, numeric methods that quickly break down would be limited by these constraints and their overall inaccuracy would become masked by such bounds. Futher, most physical springs do not exhibit linear stiffness especially when stretched or compressed passed their intended service lengths, however it should be noted that most physical systems maintain their linear range of stiffness unless they are collapsing to overload. Lastly, we neglect the rotational kinetic energy generated by the small rotations of the springs and masses as they oscillate. The primary motivation for this assumption is that most physical systems do not oscillate significantly about their equilibrium point under stable operating conditions.

In summary, the primary motivation for defining the assumptions is to narrow the scope of study to a system with physical characteristics. Given some initial inputs, the system is expected to behave close to that if a physical system were available for the said conditions.

2.2 Formulating Physical Effects

Considering the assumptions stated above, a general approach of analysis may be formulated using Newtonian mechanics. To begin the analysis, consider node X_2 from Fig. 3. The corresponding free body diagram (FBD) of the node is shown in Fig. 4.

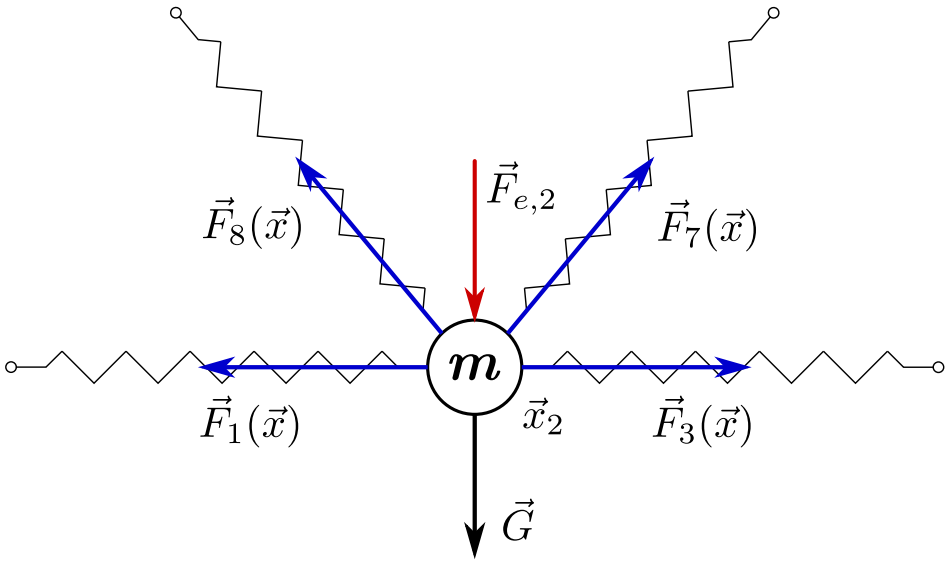

Figure 4: Free-body diagram of node two.

The vector form of Newton's second law of motion is applied to the node as follows,

$$\sum \vec{F}_2 = m_2 \vec{a}_2 \tag{1}$$

where $\sum \vec{F}_2$ is the sum of all of the external loads, m_2 is the mass assigned to the node, and $\vec{a}_2$ is the resulting acceleration of the node. If the effects of gravity, $\vec{G}$, are neglected, the remaining forces acting on the node are external loads and forces caused by the springs attached to the node. For this particular node, four springs are attached exerting forces $\vec{F}_1(\vec{x})$, $\vec{F}_3(\vec{x})$, $\vec{F}_7(\vec{x})$, and $\vec{F}_8(\vec{x})$, where the subscripts indicate the node they are attached to on the opposite end. Each spring in the framework is limited to connecting exactly two nodes. Following Newton's Law, it exerts an equal amount force on either node it connects to, in the direction if its prominent axis. The spring's effective force is defined in Eq. 2 which depends on the position and displacement of the node along with the spring's stiffness.

$$f_{ik}(u_i, u_k) = s_{ik}\left(1 - \frac{\ell_{ik}}{||(x_k + u_k) - (x_i - u_i)||_2}\right)((x_k + u_k) - (x_i + u_i)). \tag{2}$$

The definitions of each variable in Eq. 2 have previously been defined in Table 1 above. In addition to the spring applied spring forces, initial conditions of applied force must also be considered in the general EOM for each node. Because the motion of the nodes are dependent on the integration time step, the variables in Eq. 2 can be more explicitly written with the time dependence shown. Equations 1 and 2 may be combined with the generalized initial conditions of continuously applied force or displacement to give:

$$m_i \ddot{u}_i(t) = \sum_{k=1}^{n} f_{ik}(u_i(t), u_k(t)) + C_i \tag{3}$$

where C_i is a term generalizing the initial displacement applied to the node, D_i, or the persistent force applied to the node, F_i. Both variables have been previously defined in Table 1 above.

Due to the nature of the system, it is possible to incorporate an energy calculation as a benchmark to compare various integrators. The energy of the framework may be generalized by adding the energy of each individual mass concentrated at the node. The following equation summarizes this approach:

$$w(t) = \sum_{i=1}^{n} \frac{m_i}{2} ||\dot{u}_i(t)||_2^2 + \sum_{i=1}^{n} \sum_{k=i+1} \frac{s_{ik}}{2} \left(||(x_k + u_k(t)) - (x_i + u_i(t))||_2 - \ell_{ij} \right)^2 \tag{4}$$

where $w(t)$ is the time dependent energy associated with each node of the framework. This energy calculation does not consider gravitational potential energy, only spring potential energy and kinetic energy. Note that Eqs. 2 - 4 have been adapted from the Project Guide provided at the beginning of this project.

3 Numerical Methods

Once the equations of motion are finalized for our framework, simulation is the next step. Our model allows for the calculations of the net acceleration of each node. The numerical methods employed must be able to step up through the velocity then to the position of each node. Several methods were used including: Euler Explicit, Euler Symplectic, Störmer-Verlet Symplectic, and the classical 4^{th} Order Runge-Kutta methods. Each one is described in a section below. All of these methods are derived to solve second order differential equations representing equations of motion for dynamical systems.

3.1 Euler Explicit

The equations representing the Euler Explicit Method are

$$u^{j+1} = u^j + \Delta t \, v^j, \qquad v^{j+1} = v^j + \Delta t \, a(u^j, v^j, t_j). \tag{5}$$

Comparing these to the generalized kinematics equations for particles, the position values are calculated using the old velocities with no considerations to the old accelerations. The velocity equation looks just like the one for particle kinematics under constant acceleration. Using a constant acceleration is a reasonable method when stepping through a simulation at very small time steps.

Euler explicit is known to be the least useful numerical solving method because it quickly diverges from what would be considered the actual solution. This may be because the position calculation acts as if the acceleration is zero.

3.2 Euler Symplectic

The equations representing the Euler Symplectic Method are

$$v^{j+1} = v^j + \Delta t\, a(u^j, v^j, t_j), \qquad u^{j+1} = u^j + \Delta t\, v^{j+1}. \tag{6}$$

Euler symplectic looks very similar to Euler explicit, but it differs in that the new positions always depend on the newly calculated velocities. This causes the symplectic nature of the method, or the ability to preserve the energy of the system. It is derived to be a solution to Hamilton's Equations, which are inherently energy preserving.

Euler Symplectic still depends on smaller time steps to maintain accuracy, but it will not diverge from the expected solution of a system to the same degree as the explicit version does.

3.3 Störmer-Verlet

The equations representing the Störmer-Verlet Method are

$$u^{j+1} = u^j + \Delta t\, v^j + \frac{\Delta t^2}{2} a(u^j, v^j, t_j), \qquad v^{j+1} = v^j + \Delta t \frac{a(u^j, v^j, t_j) + a(u^{j+1}, v^{j+1}, t_{j+1})}{2}. \tag{7}$$

This method takes care of the issue that the Euler methods had; the acceleration is considered when the positions are calculated. The new velocity is calculated using the average of the current and next acceleration accounting for the semi-implicit or symplectic nature of the method. Our system does not depend on velocity so this method is purely explicit. The symplectic nature of the method in our system is maintained because all of the energy is due to conservative spring forces and kinetic energy.

In terms of the numerical order of the method, the Störmer-Verlet is a second order method, taking advantage of the central difference derivative method. Both Euler methods described above are first order. Numerical error is significantly reduced as the order of the method is increased. The trade-off to higher order methods is the required computation ability is greater.

3.4 4^{th} Order Runge-Kutta

The equations representing the 4^{th} Order Runge-Kutta method are

$$\begin{aligned}
\hat{u}_1 &= u^j, & \hat{v}_1 &= v^j, & \hat{a}_1 &= a(u^j, v^j, t_j) \\
\hat{u}_2 &= u^j + \frac{\Delta t}{2}\hat{v}_1, & \hat{v}_2 &= v^j + \frac{\Delta t}{2}\hat{a}_1, & \hat{a}_2 &= a(\hat{u}_2, \hat{v}_2, t_j + \frac{\Delta t}{2}) \\
\hat{u}_3 &= u^j + \frac{\Delta t}{2}\hat{v}_2, & \hat{v}_3 &= v^j + \frac{\Delta t}{2}\hat{a}_2, & \hat{a}_3 &= a(\hat{u}_3, \hat{v}_3, t_j + \frac{\Delta t}{2}) \\
\hat{u}_4 &= u^j + \Delta t\, \hat{v}_2, & \hat{v}_4 &= v^j + \Delta t\, \hat{a}_2, & \hat{a}_4 &= a(\hat{u}_4, \hat{v}_4, t_{j+1}) \\
u^{j+1} &= u^j + \frac{\Delta t}{6}(\hat{v}_1 + 2\hat{v}_2 + 2\hat{v}_3 + \hat{v}_4), & v^{j+1} &= v^j + \frac{\Delta t}{6}(\hat{a}_1 + 2\hat{a}_2 + 2\hat{a}_3 + \hat{a}_4)
\end{aligned} \tag{8}$$

This method, explicit in nature, takes advantage of multiple intermediate calculations between two time steps. Each sub-iteration essentially calculates the midpoint position, velocity, and accelerations and it does it 4 times to find more accurate results. In the final calculations, the the average of all of the intermediate steps is take as the next iteration values.

As the name suggests, this is a 4^{th} order method which means the numerical accuracy of the method is much greater than than the previous methods. It is not symplectic so it does not preserve energy, but as it will estimate the exact solution accurately for a longer period of time.

3.5 Applying the numerical methods

In the case of our model and the way they were programmed, the above methods were essentially input identically as they are read. All that changes is which one to use. The tricky part arises when any intermediate accelerations must be calculated because it must be done in a certain manner to accommodate the limitless frameworks possible. Intermediate acceleration calculations tax the program significantly so they should be avoided in complex frameworks unless substantial computation times are acceptable.

4 Algorithm and Programming

Figure 5 summarizes the basic operation of the program. The first step is to develop a frame design and transfer the information into Cartesian coordinate pairs into the program, specifying which nodes are grounded and which are free to move.

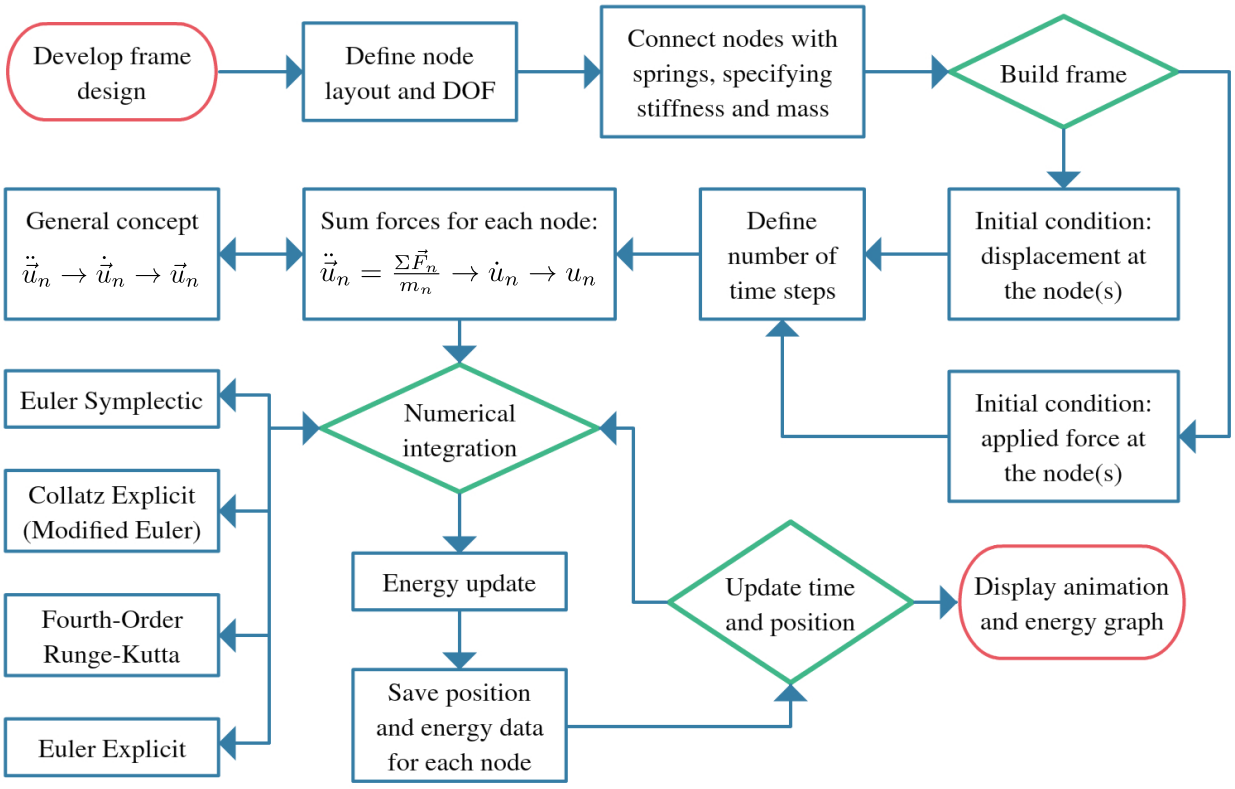

Figure 5: General planar framework configuration.

Then the user needs to indicate which nodes are connected with springs, identifying spring stiffness and spring mass. The program proceeds to internally build a framework which is defined by the input parameters of initial node locations and springs connecting the nodes. If initial conditions are applied to the framework, it will provide a simulation with a changing geometry, otherwise, the frame will remain stationary. There are two ways to disturb the initially static system; with the application of a persistent load or an initial displacement. Each node is programmed to react to the initial conditions, affecting the neighboring nodes as time evolves. The user input also requires a time step value to be specified.

After the initial definition stage, the program proceeds sum the forces at each node, with the general method indicated in the diagram, and computes the next iteration using one of the specified integrators. The energy associated with each node is updated based on the current condition as well as the position. This iterative process of computing the node acceleration, position, and energy continues for the duration of the specified time interval. When the computational work is completed, the program displays an animation of the framework as it changes geometry.

The program algorithm works by updating the node positions and computing the reactions after the updated positions are saved. Essentially, the code moves the nodes due to the respective forces acting on them. This method was chosen over a different one where it was proposed to control the spring positions rather than the nodes. The current algorithm is very robust in nature allowing the user to create various planar configurations beyond the initial intentions.

5 Parameter Analysis and Discussion of Results

This section describes the conclusions about the model in which the program and various simulations lead to.

5.1 Symmetry

Due to the nature of the model, symmetry preserves motion. Building a symmetric framework with all mass and spring constants the same and then applying initial conditions in a symmetric manner should cause the system to move symmetrically. This is because all spring forces and load forces balance in the system. Therefore, testing for symmetrical motion is one way to verify the accuracy of the simulation. Figures 6 and 7 show examples of symmetrical and unsymmetrical motion.

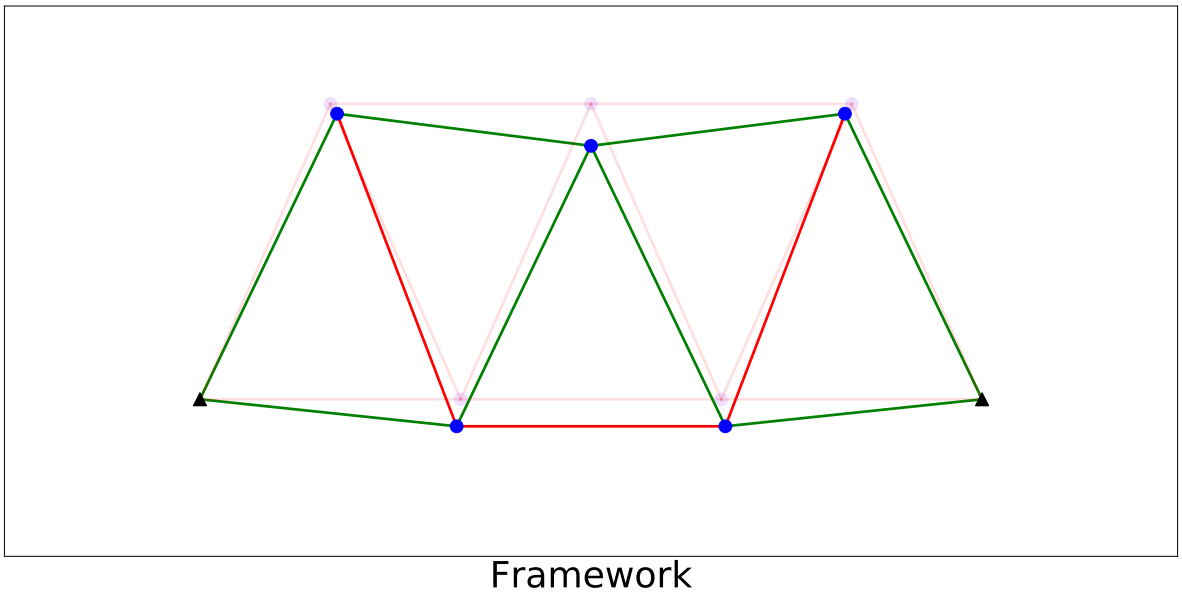

Figure 6: Symmetric Motion Example

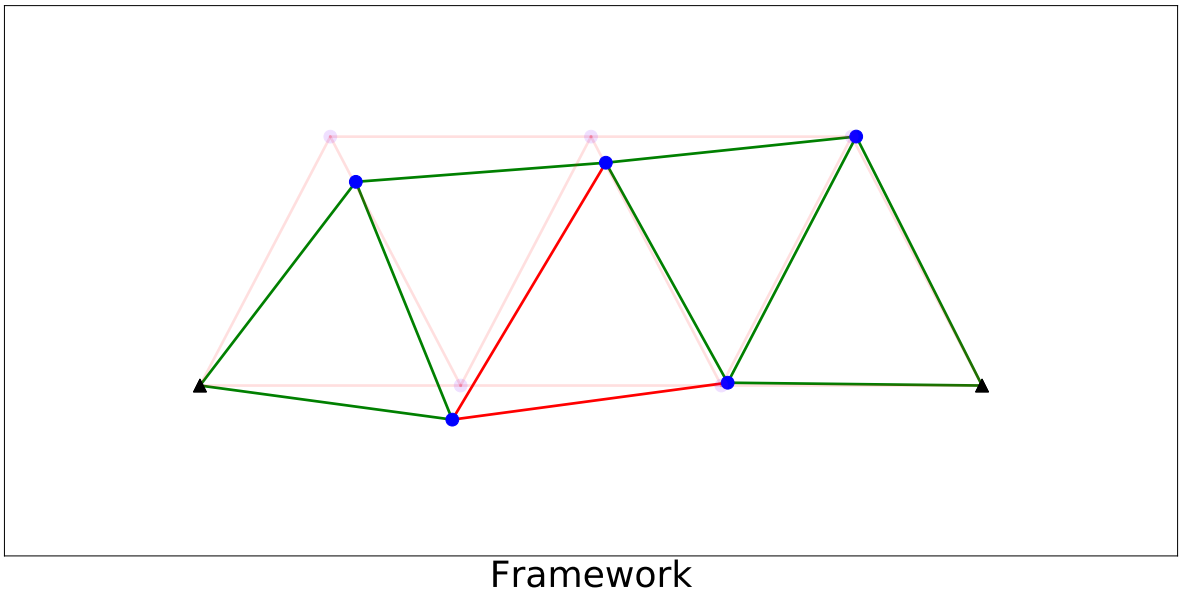

Figure 7: Unsymmetric Motion Example

5.2 Energy Under Symmetric Initial Conditions

Another method to test the accuracy of the simulations is to input a simple framework and check the energy of the system under symmetrical initial conditions.

In the case of initial loading, the energy of the system oscillates between zero energy and a maximum energy. This is because at the instant when the load is applied, the energy of the system is zero. The loading causes a force imbalance which initiates motion in the system. Thus, energy is added until the momentum of the system changes direction and the system returns to its starting position, while decreasing in energy. This repeats forever due to the lack of damping in the model. The initial load is constantly applied so when the springs and nodes settle down, the load force can once again act in full power. Figures 8 and 9 shows two different numerical solvers simulating the same system given an initial, symmetric loading.[1] Their corresponding energy plots are adjacent. It is clear looking at the energy plot that there is basically no difference between the two methods

In the case of initial displacement, energy is added because of the displacement and the total energy remains constant throughout the entire simulation. Adding initial displacement can be compared to priming a slingshot and releasing it. The potential energy in the sling is quickly converted to kinetic energy. In our model, there is nothing to take the energy back so the system keeps the energy and oscillations occur forever. Unlike initial loading, the energy is passed between kinetic and spring strain potential without any losses or gains. Initial loading is not very sensitive to more accurate numerical solvers, but initial displacement is heavily dependent on such accuracy. Figures 10 and 11 shows two different numerical solvers simulating the same system given an initial, symmetric displacement. Their corresponding energy plots are adjacent. The Runge-Kutta simulation shows small energy loss at the micro-unit level, while the Euler sim-

[1]Figures 8, 9, 10, and 11 each simulate 300 steps at a step size of 0.05 for a total of 6000 steps.

ulation energy fluctuated above the mili-unit level.

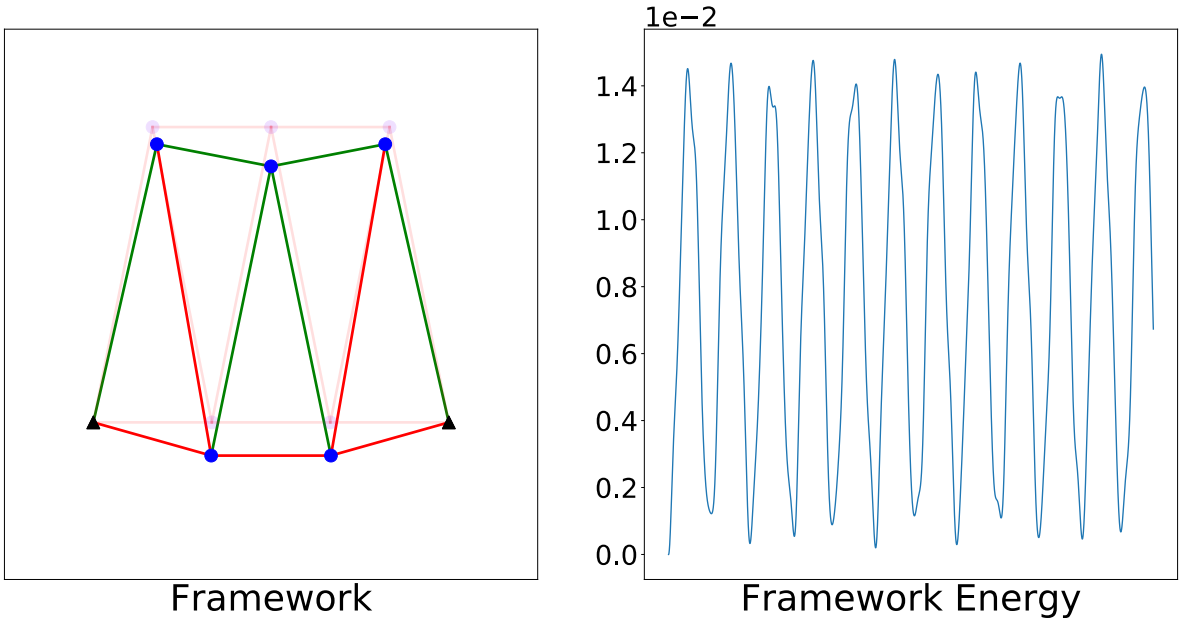

Figure 8: Euler Symplectic Simulation with Initial Loading.

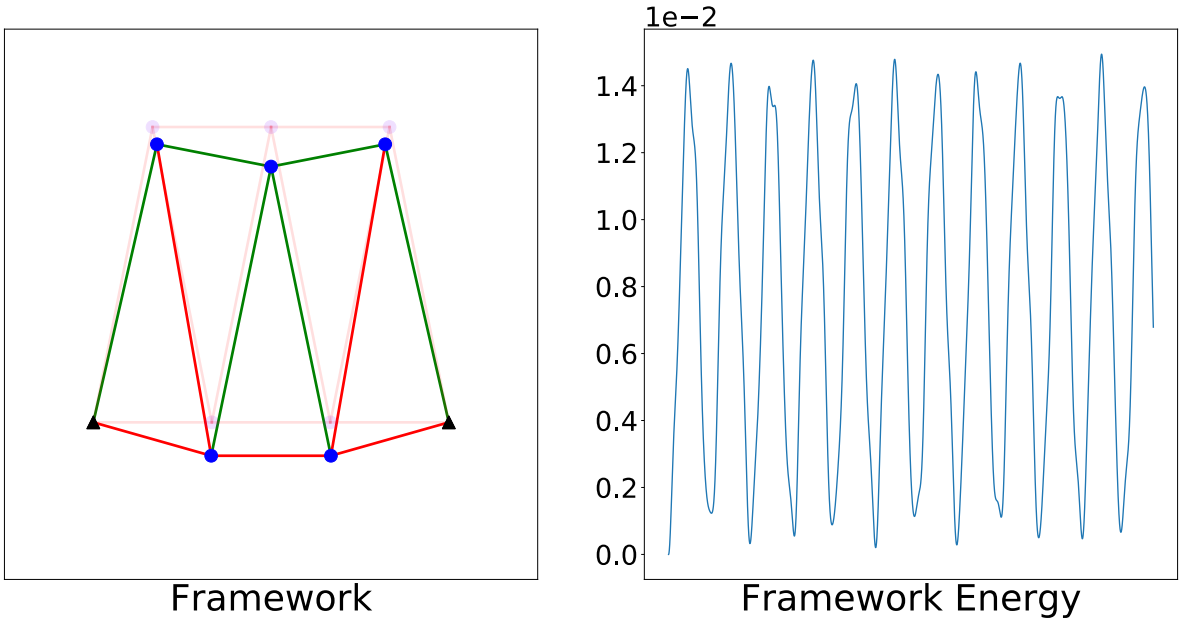

Figure 9: Runge-Kutta 4^{th} Order Explicit Simulation with Initial Loading.

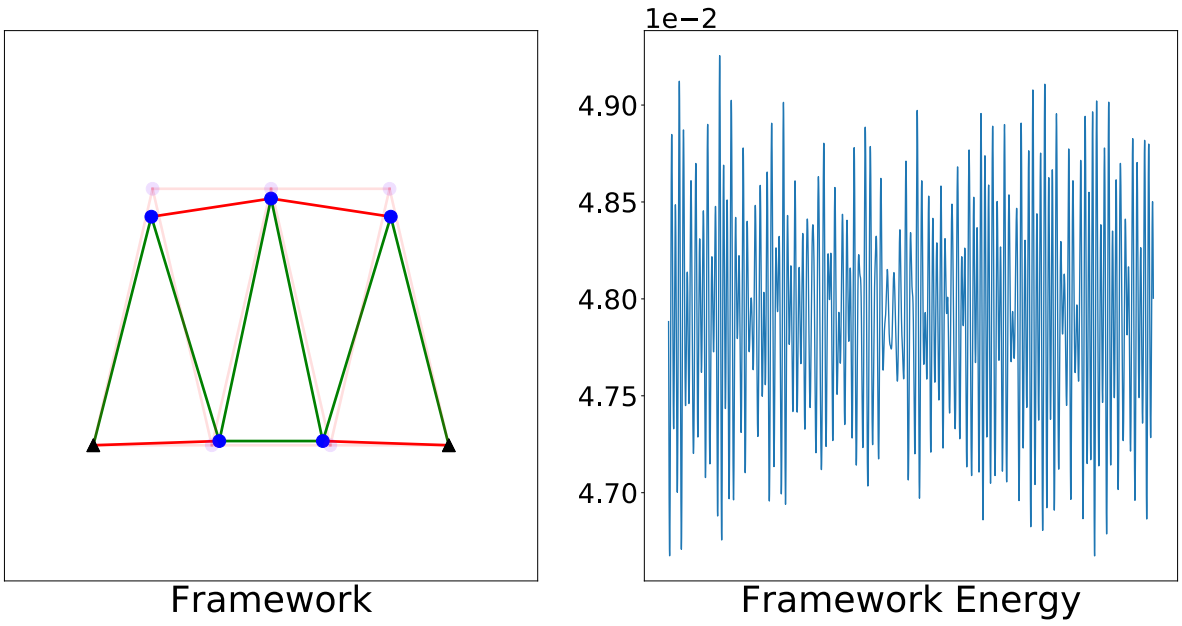

Figure 10: Euler Symplectic Simulation with Initial Displacement.

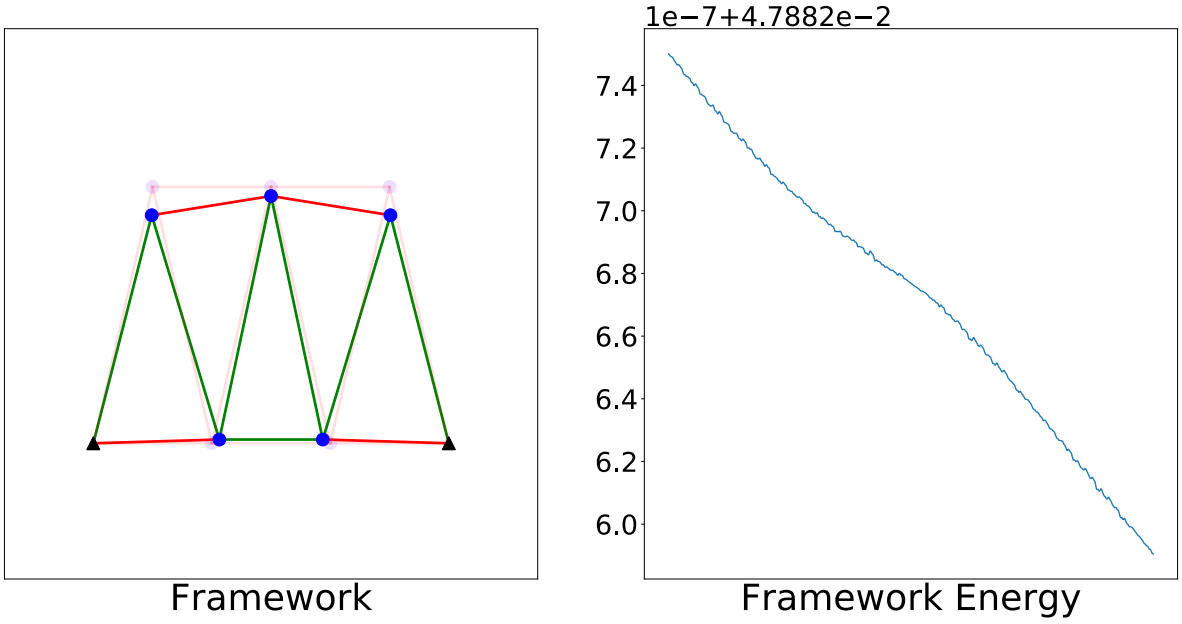

Figure 11: Runge-Kutta 4^{th} Order Explicit Simulation with Initial Displacement.

5.3 Time Step Considerations

Changing the step size of the numerical solver improves the accuracy of the simulation, regardless of what method is employed. The trade-off is the large increase in calculation time. In the case of our system, the improved accuracy is helpful, but it is not necessary so long as the symmetry of motion is maintained in such configurations that should be symmetric. Also, as show in the previous figures, initial loading is not as sensitive to the numerical accuracy of the simulations

As our project and program is designed to be inclusive of any framework system, more complex systems can be simulated. Each free node in the framework is essentially a system of 2 differential equations. One DE to solve the velocity of the node, the other to solve the position of the node. Therefore, the total number of equations that need to be solved every step is two times the number of free nodes. For a system with 10 free nodes, the program must simulate 20 differential equations every time step. In higher order numerical methods, the acceleration must be re-calculated multiple times at intermediate time steps. This places a massive load on the computation time in order to improve the accuracy of the simulation. Distinguishing when a particular method is the most useful is important when simulating far more complex frameworks. By using a smaller time step, more times must be simulated in order to produce the same simulation time. Simulating 100 seconds with a time step of 0.05 seconds is faster than simulating 100 seconds with a time step of 0.01 seconds. Therefore, the quality of the simulation must be weighed with the computation time also. For most of the frameworks tested, a time step of 0.05 or 0.01 seconds was sufficient to produce accurate simulations depending on the strengths desired.

5.4 High Strength Frameworks

In the case when substantial spring strength is desired, the spring constants are set to high values. This causes the spring force to be large. This sometimes serves to create quasi-rigid springs. In certain applications, the high spring forces makes it near impossible for the spring length to change from its undeformed length. This broadens the scope of application of the project to particle pendulums and other rigid-length dynamic systems. A side effect of rigidity is the dependence on smaller time steps. A high force has more potential to cause a node to escape so the smaller time step will catch it before any harm is done to the simulation. Essentially this means smaller time steps are required to simulate stronger frameworks.

5.5 High Mass

Changing the mass of the springs which correlates to increasing the mass of the nodes causes the net acceleration of the nodes to be smaller. Over the duration of the simulation, this causes everything to just happen slower. There is not much change in the motion of the framework. This could be because we are not considering rotational energy in our model.

6 Applications

After verifying the consistency and the appropriateness of the program, frameworks can be built to simulate pseudo-realistic structures such as truss-works and even crane bodies. A few examples are described.

6.1 Floor Truss

A simple 2D floor truss was simulated. Figure 12 shows the initial position of the structure with the locations of the applied distributed load along the top. The magnitude of each load $\vec{F}$ is 0.25. The spring constant for all of the springs is 200 and the masses are 1. These initial conditions represent what a floor may experience when heavy objects are uniformly spread along it.

The simulation was run on the floor truss framework using a time step of 0.01 for a total of 10,000 iterations. The numerical method employed was the Störmer-Verlet method. Figure 13

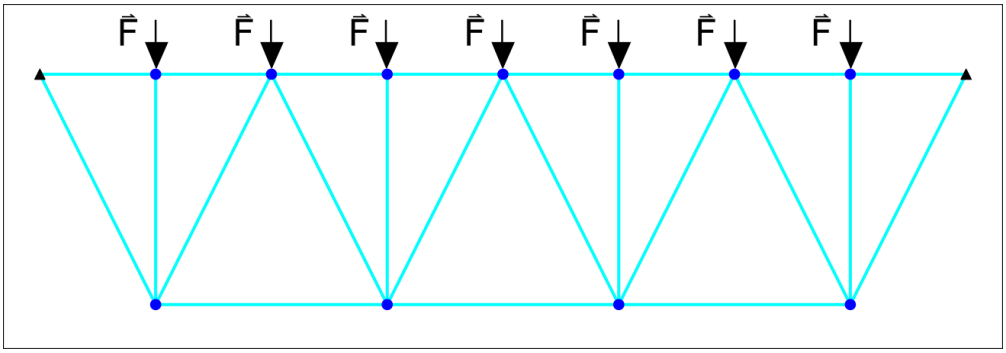

Figure 12: Floor Truss Initial Configuration

shows the framework motion over time. The end nodes of the framework were fixed in place. It is interesting to see how the truss center bends downwards under the distributed load. Figure 14 shows the associated energy of the framework. Because of the symmetrical framework and loading, the energy follows a consistent cyclic manner.

6.2 Crane

A simplified 2D projection of a crane was simulated. Figure 15 shows the initial position of the structure with the locations of the applied loading. The magnitude of the applied loads $\vec{F}_1$ and $\vec{F}_2$ are 0.1 and 0.2, respectively. The spring constant for all of the springs is 200 and the masses are 1. These initial conditions produce a simulation that demonstrates the scope of the program while showing the periodicity aspect inherent in applying certain loadings.

The simulation was run on the crane framework using a time step of 0.01 for a total of 10,000 iterations. The numerical method employed was the Störmer-Verlet method. Figure 16 shows the framework motion over time. The base of the framework was fixed in place. It is interesting to see how the crane tower in the center is stretched and compressed as the head of crane deforms. Figure 17 shows the associated energy of the framework. It shows oscillatory motion, but it is not as nice as other cases due to the unsymmetric loading and framework.

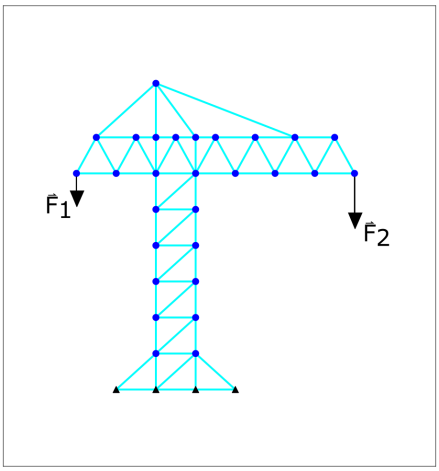

Figure 15: Crane Initial Configuration

7 Summary of the Project

Overall, this project and course in general have been a great learning experience. From the practical perspective, this adds to the fundamental courses of our engineering coursework. Understanding the limitations of analytical solutions in modeling applications is key to studying numerical differential equations. The practical use of the concepts learned in this project extend far beyond the boundaries of this course. The immediate goal of the project was to provide a robust program to accept various framework geometries and simulate their motion with evolving time. Based on the initially stated assumptions, the goal of this project was satisfied. Multiple geometries have been simulated and presented in the class showing the robustness

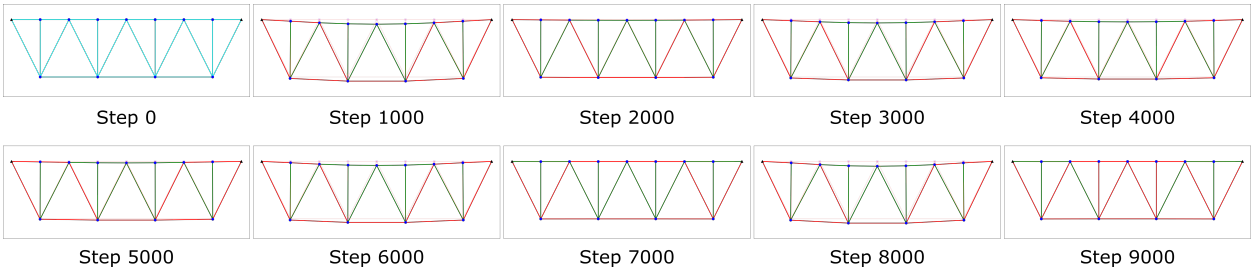

Figure 13: Steps during the Floor Truss Framework Simulation

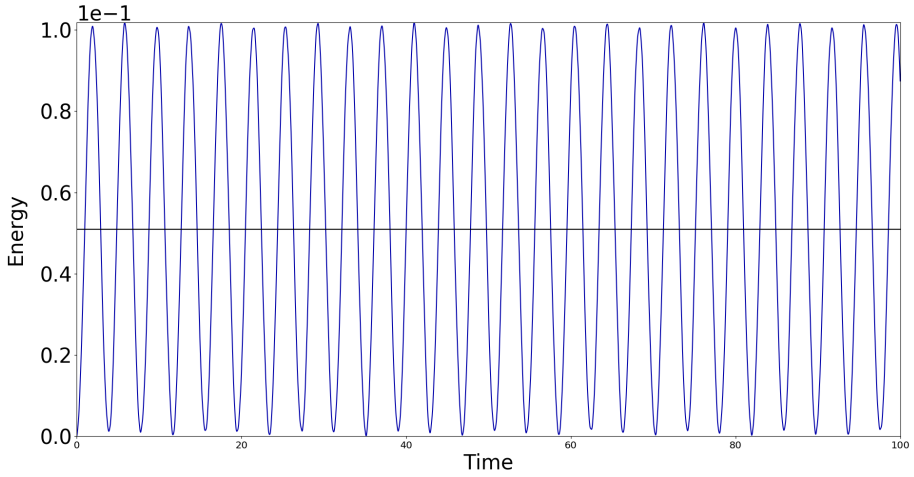

Figure 14: Energy Plot for the Floor Truss Framework Simulation

of the programming algorithm.

Multiple issues were encountered throughout this project which extended our interest and learning. The initial layout of program structure proved to be a big help. This served as a limitation device since we were constrained by the given amount of time as well as a plan for development. Other programming decisions such as the choice to use object-oriented programming proved to be very useful for this particular situation. General programming debugging was also an unavoidable recurring issue. In general, however, not many issues arose after the initial meetings with project advisor Dr. Andrea Dziubek.

With the availability of time, our plan is to further generalize the existing program. We also plan to create a GUI allowing the user to specify optional input parameters such as three-dimensional geometry, the effects of gravitational force and gravitational potential energy, and member damping models. Other program upgrades would focus on the limiting physical factors encountered in physical systems. Such factors include the non-linearity of springs past a certain extension and compression length and compensation of framework members' inertia. Adding these parameters to the program would give it a higher realistic factor as well as a broader applications footprint. In addition, if we are successful in implementing these program upgrades, we will consider attending the Student Project Showcase next year.

If this project was to be repeated with the current knowledge, it would be very helpful to be-

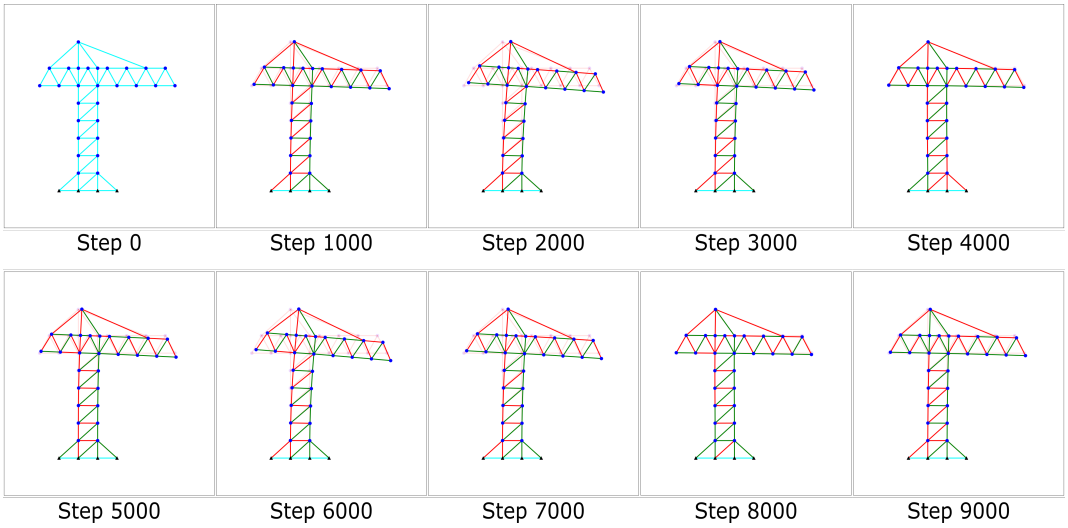

Figure 16: Steps during the Crane Framework Simulation

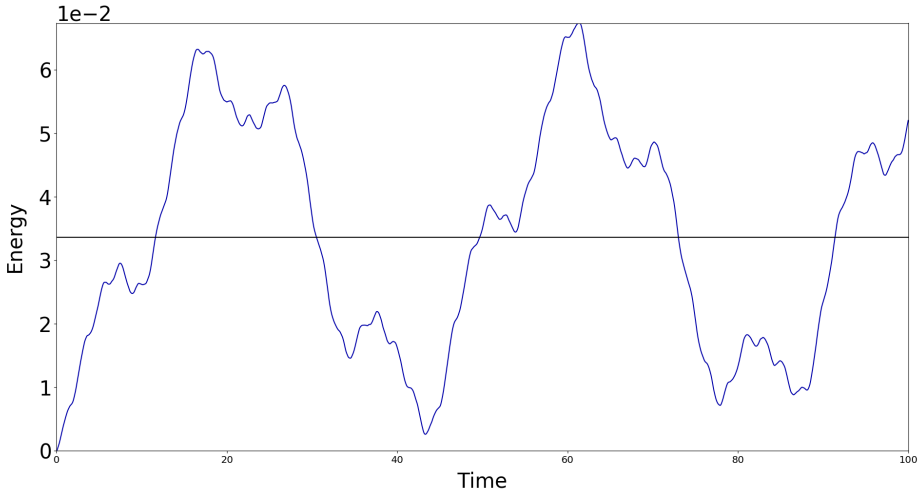

Figure 17: Energy Plot for the Crane Framework Simulation

gin the project prior to the time we did. The extra time could have been utilized to ask additional questions and provide more clarification to issues that arose in project areas outside of the numerical differential equations knowledge.

Appendices

A Course Evaluation

A.1 Gregory Georgiades

In Numerical Differential Equations, I learned a lot about numerical methods and how they work, what kinds of issues they have, and their usage in general. I also learned a lot more about numerical computations. I was able to improve my python skills and I was ecstatic that I could bring object orientated programming into the project. My strong programming skills made the course a lot simpler than it could've been. Many other students were struggling the entire semester to learn how to program at the level required. Programming ability is an absolute must in order to enjoy this course. I believe the instructor taught at an appropriate level for a senior undergraduate course. Working as a team with David was nice for bouncing ideas around and he was an excellent partner on the project. It was difficult for me to share coding assignments with him because I wanted to do them all, but I managed to let some slide. I would say the availability of the instructor was not very strong. Especially since she cancelled many of our pre-arranged appointments. This did not hurt us because we were confident in our position, but it was still very very sad. In the case of other groups, from what I saw, the meetings were never long enough for them. I believe the two tutors were not good. They could not provide much help to other students and would often waste students' time when they were in dire need. The grading system was completely flawed. Anyone who doesn't know what a zip file is or how to handle them, should not be grading a computer-based course where assignments were uploaded.

A.2 David Petrushenko

In general, I am happy about the knowledge and experience I gained through taking Numerical Differential Equations. I learned the fundamentals of an open source programming language while having constant support from my partner Greg and Dr. Dziubek. I am very thankful for all of the discussions that we had together about homework and project related work. All has contributed to the advancement in my knowledge of mathematical applications. In general, I think the lectures were a little too theoretical. Since the course was primarily filled with mechanical engineering students, I feel that the topics covered were lacking a direction of their direct applications. Outside support was definitely very weak. It seems like the assigned tutors had little interest in preparing and staying current with the coursework which showed with multiple frustrated students. In general, it seemed better to spend a few hours searching online rather than being confused by the tutors. In general, I feel that the presentations were a bit repetitive and excessive. Most groups seemed to stretch their presentations and present the same content multiple times just to fulfill the requirements. Alternatively, it may be beneficial to rotate group presentations such that each group does not present one out of the three times. The grading system may need an additional handout to avoid repetitive student questions and confusion of total points allowed for a particular exercise. To conclude, I would like to say that one of the best things about the classroom experience was the positive student-professor interactions. Although questions were not always answered, it did feel like the instructor was genuinely concerned about student performance.

B Python Simulation Code

```
"""
    Numerical Differential Equations
    Framework Project Code: Master Simulation, Plot, Animation, Saving
    Instructor: Dr. Andrea Dziubek
    Prepared by Gregory Georgiades and David Petrushenko
    Last Updated: 5/1/2017
"""

"""
  TODO:
    - Spring Force is getting inf/nan somewhere? - based on step, spring constant!

    # === If time/bored ===
    - Interactive framework building
"""

import time
start_time = time.process_time()            # exectution time includes other
    import times

import numpy as np
# import scipy.linalg as sci
from matplotlib import use, pyplot as plt, animation as ani, pylab
import subprocess
import os
import sys
sys.path.append(os.getcwd()) # Fix path

np.set_printoptions(precision = 10, suppress = True, threshold = np.inf)
use('TkAgg')
plt.ion()
plt.close('all')
params = {'legend.fontsize': 'x-large',
          'figure.figsize': (16, 8),
          'axes.labelsize': '40',
          'axes.titlesize':'40',
          'xtick.labelsize':'x-large',
          'ytick.labelsize':'30'
          }
pylab.rcParams.update(params)


from famework_structures import *
from framework_solvers import solve


# Build a framework, initial conditions deifined in build function
# nodes, springs = buildframework1(10)
# nodes, springs = buildBigPendulum(1200)
nodes, springs = buildSmallPendulum(20000) # step 0.01, k=20000, runge: is good
    pendulum!
# nodes, springs = buildCrane(100) # ~640 is cap
# nodes, springs = buildFloorTruss(30)

# Simulate the framework
# methods = {'eulex', 'collatz', 'eulsym', 'runge', 'stormer'}
```

```
55 times, energy = solve(nodes, springs, 0.01, 100, method = 'runge')

57 print("Simulation Runtime: %s seconds" % (time.process_time() - start_time))     #
       print simulation execution time

59
   # ====================================================
61 # === Plotting and Animation ========================
   # ====================================================
63

65 # === Plot Bounds ====================================
   # figure out plot bounds based on intial positon and max displacements, then scale
       a bit based on center
67
   minx, miny, maxx, maxy = 0, 0, 0, 0
69
   for n in nodes:
71     nx = [x[0] + n.x[0] for x in n.um]
       ny = [x[1] + n.x[1] for x in n.um]
73     nmaxx = np.max(nx)
       nminx = np.min(nx)
75     nmaxy = np.max(ny)
       nminy = np.min(ny)
77
       if nminx < minx:
79         minx = nminx
       if nmaxx > maxx:
81         maxx = nmaxx
       if nminy < miny:
83         miny = nminy
       if nmaxy > maxy:
85         maxy = nmaxy

87 c = [(maxx - minx) / 2, (maxy - miny) / 2]                        # Geometric center
       of bounding box
   c = np.dot(0.5, c)                                               # Upscaling Factor
89 bounds = [minx - c[0], maxx + c[0], miny - c[1], maxy + c[1]]    # Scale each side
       of the bounding box up by its distance to the center

91
   # === Plotting function ==============================
93 # Plot the system at any time t (integer...)

95 def plotTime(t):

97     # Have to store the artists to return to animation func for using blit
       artists = list()
99
       # === Update Framework ============================
101
       for s in springs:
103        artists.append(s.drawSpring(t))

105    for n in nodes:
           artists.append(n.drawNode(t))
107
       # === Update Energy ===============================
109
```

```
#     en.set_data(times[:t], energy[:t])
#     artists.append(en)

    return artists

# Plotting and animation initiation function. Builds plot parameters and creates the artist objects
def aniinit():

    for s in springs:
        s.initDraw(ax[0])

    for n in nodes:
        n.initDraw(ax[0])

    # Framework Plot Parameters
    ax[0].axis('scaled')
    ax[0].axis(bounds)
    ax[0].set_xlabel('Framework')
    ax[0].yaxis.set_visible(False)
    ax[0].tick_params(axis = 'x',          # changes apply to the x-axis
                      which = 'both',      # both major and minor ticks are affected
                      bottom = 'off',      # ticks along the bottom edge are off
                      top = 'off',         # ticks along the top edge are off
                      labelbottom = 'off') # labels along the bottom edge are off


    # Energy Plot parameters

#     ax[1].axhline(np.average(energy), times[0], times[-1], linestyle = '-', color = 'black')
#     ax[1].set_xlabel('Framework Energy')
#     ax[1].set_xlim((0, round(times[-1]))) # show times
#
# #     e = np.ma.masked_equal(energy, 0.0, copy = False)   # for excluding zero .....only in special cases!
# #     ax[1].set_ylim((e.min(), e.max()))                  # ^^^
#     ax[1].set_ylim((energy.min(), energy.max()))
#     ax[1].ticklabel_format(style = 'sci', axis = 'y', scilimits = (0, 0)) # Force scientific axis tick labels


    return


# dpi and bitrate control output quality
# fps controls output speed (not the time it takes to write the file...)

# python ffmpeg src code - with slight modification
# https://stackoverflow.com/questions/30965355/speedup-matplotlib-animation-to-video-file

# Replacement for animation.save...in fact replaces animation completely if given a wait timer..., a looot better...before blit stuff, but still more reliable for saving + output

def anisave():
```

```
    # Open an ffmpeg process
    outfile = 'Z:\\users\\gregory\\desktop\\frameani.mp4'

    cmdstring = ('ffmpeg',                                                  # ffmpeg is part of windows path, otherwise just give abs path or put in script dir?
                 '-y',                                                      # overwrite existing
                 '-r', '5',                                                 # fps
                 '-s', '%dx%d' % fig.canvas.get_width_height(), # size of image string
                 '-pix_fmt', 'argb',                                        # format: argb pixel data
                 '-f', 'rawvideo', '-i', '-',                               # tell ffmpeg to expect raw video from the pipe
                 '-vb', '2.5M',                                             # Bitrate - ie quality - this param is for rawvid only?, affects output filesize!
                 '-vcodec', 'mpeg4',                                        # output codec
                 outfile,                                                   # output file - need proper extension
                 )

    # Execute as an open pipe process
    p = subprocess.Popen(cmdstring, stdin = subprocess.PIPE)

    # Draw frames and write data to the pipe
    for frame in pts:
        plotTime(frame)                     # Call the animation function
        fig.canvas.draw()                   # Draw the updated frame
        string = fig.canvas.tostring_argb() # Extract the figure image as an ARGB string - is what ffmpeg was told it would get above
        p.stdin.write(string)               # Write to ffmpeg pipe

    p.communicate() # Send 'next' input to ffmpeg - essentially tells it the vid is finished so finish writing and close it.

    return


# === Create The Animation ============================

plt.close('all')
fig, ax = plt.subplots(1, 1, num = 'Awesomist~~~Yeaaah~~~', figsize = (19.2, 10.8), dpi = 100)
if type(ax) != np.ndarray:
    ax = np.array([ax])
fig.subplots_adjust(left = 0.1, right = 0.97, bottom = 0.1, top = 0.95, hspace = 0.35)
plt.get_current_fig_manager().window.state('zoomed')

pts = np.linspace(0, len(times), int(len(times) / 15), dtype = int) # Subset points to make animation faster - does not affect accuracy of the motion relative to the simulation performed.

# en, = ax[1].plot([0], [0], '-', color = '#0000aa')

aniinit()
anim = ani.FuncAnimation(fig, plotTime, pts, interval = 1, repeat = True, blit = True)
# anim._stop()
```

```
# plt.show()

# === Saving Animations ==============================
print('Saving Animation...')
# anisave()

# === Plot Specific Times ============================

# Plots the initial configuration
# plotTime(0)

# Plots final configuration based on pts
# plotTime(pts[-1])

# ====================================================


print('=' * 150, '\n')
print("--- Runtime: %s seconds ---" % (time.process_time() - start_time))     #
    print program execution time
```

```
"""
    Numerical Differential Equations
    Framework Project Code: Class Structures
    Instructor: Dr. Andrea Dziubek
    Prepared by Gregory Georgiades and David Petrushenko
    Last Updated: 5/1/2017
"""

import numpy as np
import scipy.linalg as sci
import matplotlib.pyplot as plt

# ====================================================
# === Creating the Classes ===========================
# ====================================================

class Node:
    """
    A Node is where springs connect and it holds spring mass
    """

    def __init__(self, x, fixed = False, grav = False):
        self.x = np.array(x)                    # Save initial position
        self.u = np.array([0, 0])               # Current Displacement
        self.um = list([self.u])                # Node remembers the path it has
    traveled
        self.v = np.array([0, 0])               # Current Velocity
        self.vm = list([self.v])                # Node remembers the path it has
    traveled
        self.fixed = fixed                      # Fixed or free node
        self.springs = list()                   # List of all springs attached to
    this node
        self.m = 0                              # Node mass
        self.force = list([[0, 0]])
        self.load = np.array([0, 0])            # Initially no loads on this node
#         if grav:
#             self.load = np.array([0, -9.8])         # All nodes get gravity

        return

    # Add spring to this node
    def addSpring(self, Spring):
        self.springs.append(Spring)             # Add this springs to the attached
     springs list
        self.m = self.m + Spring.m / 2          # Add half the spring mass to this
     the node's mass

        return

    # Add loading (vector force), additive, can do multiple times, though why?
    # Loads are constant!
    def addLoad(self, load):
        if self.fixed:
            print('LOADING A FIXED NODE IS POINTLESS (Does nothing in this program
    )')
            return
        self.load = self.load + np.array(load)
```

```
54          return

56      # Additive initial displacement
        # Not constant
58      def addDisplacement(self, u):
            if self.fixed:
60              print('CANNOT MOVE A FIXED NODE')
                return
62          self.u = self.u + np.array(u)

64          return

66      # Add all spring forces and loads occuring at this node
        def forceSum(self, sav = True):
68          if self.fixed:                      # A fixed node has force, but it does
    not matter because it will never move so make zero for simplicity
                self.force.append([0, 0])
70              return np.array([0, 0])

72          netforce = 0

74          for spring in self.springs:         # Add all forces from the Springs
    attached to this node.
                sf = spring.springForce(self)
76              netforce = netforce + sf

78          self.force.append(netforce)# only springs

80          netforce = netforce + self.load            # Add inital loading to spring
    forces

82          return netforce

84      def updateNode(self, u, v, sav = True):
            if sav:                             # Only saving position if this update
    is a final result of a solver
86              if self.fixed:
                    self.um.append([0, 0])      # Have to save zero displacement for
    fixed nodes otherwise it breaks other code
88                  self.vm.append([0, 0])
                else:
90                  self.um.append(u)
                    self.vm.append(v)
92
            self.u = u                          # Update displacement
94          self.v = v                          # Update velocity

96          return


98
        def initDraw(self, ax, color = 'blue', marker = 'o', text = 0):
100         if self.fixed:
                color = 'black'
102             marker = '^'
            self.p0 = ax.scatter([self.x[0]], [self.x[1]], color = '#8000FF20', marker
     = marker, s = 100)    # Draw inital position
104         self.p = ax.scatter([self.x[0]], [self.x[1]], color = color, marker =
    marker, s = 100)          # Draw mutable point
            self.p.set_zorder(100)
```

```
106
        return self.p
108

110     def drawNode(self, t):

112         self.p.set_offsets([[self.x[0] + self.um[t][0]], [self.x[1] + self.um[t
    ][1]]])  # Update position
            self.p.set_zorder(100)
              # Make sure nodes over springs
114
#           if text and not self.fixed:
116 #               ax.text(self.x[0] + self.um[t][0], self.x[1] + self.um[t][1], self.
    vm[t])
#               ax.text(self.x[0] + self.um[t][0], 0.15, self.um[t])
118
        return self.p
120
class Spring:
122     """
        A Spring connects two nodes together
124     """

126     nodes = None    # Global nodes list - must be passed after nodes are built and
         before springs are built!
        k = 1           # Global spring constant
128     m = 1           # Global mass value

130     def __init__(self, ids = '0,0', sm = 0, sk = 0):
            self.ids = ids.split('+')
132         self.node1 = self.nodes[int(self.ids[0])]       # The first node
            self.node2 = self.nodes[int(self.ids[1])]       # The second node, note
    order of nodes does not matter
134         if sk:
                self.k = sk                                 # This springs custom
    stiffness
136         if sm:
                self.m = sm                                 # This springs custom mass
138         self.node1.addSpring(self)                      # Tell node1 that this
    spring is connected
            self.node2.addSpring(self)                      # Tell node2 that this
    spring is connected
140         self.l = sci.norm(self.node1.x - self.node2.x)  # Save undeformed length
    of this spring
            self.fm = list([[0, 0]])
142         self.lm = list([self.l])

144         return


146
        # Calculate the force on a node due to this spring
148     # Direction is dependendent on which node calls this method and if the spring
    is in tension/compression
        def springForce(self, referrer, sav = True):
150         a = 1 if referrer == self.node1 else -1     # Object Comparison to check
    which node called this function

152         x = (self.node2.x + self.node2.u) - (self.node1.x + self.node1.u)
            li = sci.norm(x)
```

```
154         f = np.dot(a * self.k * (1 - (self.l / li)) , x)

156         if sav:
                self.fm.append(f)
158         return f


160
        def savelength(self):
162         x = (self.node2.x + self.node2.u) - (self.node1.x + self.node1.u)
            li = sci.norm(x)
164         self.lm.append(li)


166
        def initDraw(self, ax, text = 0):
168         self.s0 = plt.Line2D([self.node1.x[0], self.node2.x[0]], [self.node1.x[1],
         self.node2.x[1]], color = '#FF000020', lw = 3) # Draw Intial Position
            ax.add_line(self.s0)
170
            self.s = plt.Line2D([self.node1.x[0], self.node2.x[0]], [self.node1.x[1],
        self.node2.x[1]], color = 'cyan', lw = 3) # Draw mutable line
172         ax.add_line(self.s)

174         return self.s


176
        def drawSpring(self, t):
178         if self.lm[t] < self.l:
                color = 'green'
180         elif self.lm[t] > self.l:
                color = 'red'
182         else:
                color = 'cyan'
184
            self.s.set_data([self.node1.um[t][0] + self.node1.x[0], self.node2.um[t
        ][0] + self.node2.x[0]], [self.node1.um[t][1] + self.node1.x[1], self.node2.um[
        t][1] + self.node2.x[1]])
186         self.s.set_color(color)

188 #         if text:
    #             xc = (self.node1.um[t][0] + self.node1.x[0] + self.node2.um[t][0] +
        self.node2.x[0]) / 2
190 # #             yc = (self.node1.um[t][1] + self.node1.x[1] + self.node2.um[t][1]
        + self.node2.x[1]) / 2
    #             ax.text(xc, 0.1, self.lm[t])
192 #             ax.text(xc, 0.03, '%s\n%s' % (self.fm[t][0], self.fm[t][1]))

194         return self.s
```

```
"""
    Numerical Differential Equations
    Framework Project Code: Numerical Solvers
    Instructor: Dr. Andrea Dziubek
    Prepared by Gregory Georgiades and David Petrushenko
    Last Updated: 5/1/2017
"""

import numpy as np
import scipy.linalg as sci

# ====================================================
# ==== Solving the systems ===========================
# ====================================================

def solve(nodes, springs, dt, n, method = 'eulex'):
    methods = {'eulex':eulexstep, 'collatz':collatzstep, 'eulsym':eulsymstep, 'runge':rungeexstep, 'stormer':stormer}
    # Assign appropriate function to stepper func reference
    stepper = methods[method]

    # set up time series to simulate under
    times = np.arange(0, n, dt)
    pts = int(n / dt) # rounds down
    p = pts * 5 / 100 # updated every 5% with current position in simulation

    energy = np.zeros(pts + 1)

    print('Starting Simulation')

    # Main loop containing the numerical solver call and energy calculations
    for i in range(len(times)):
        if i % p == 0:
            print("%s%% Simulated..." % int(i / pts * 100))

        # == Energy ======================================
        # Have to calculate energy first in order to capture the starting energy before updating the simulation step.

        kin, spr = 0, 0

        for n in nodes:
            kin = kin + n.m / 2 * np.power(sci.norm(n.v), 2)

        for s in springs:
            spr = spr + s.k / 2 * np.power(sci.norm((s.node1.x + s.node1.u) - (s.node2.x + s.node2.u)) - s.l, 2)

        energy[i] = kin + spr

        # === Data at current time =======================

        aj = np.array([n.forceSum() / n.m for n in nodes])
        vj = np.array([n.v for n in nodes])
        uj = np.array([n.u for n in nodes])

        # === Solver Step ================================
```

```
            vj1, uj1 = stepper(aj, vj, uj, dt, nodes)

            # Update nodes with final solver results for this step and have the nodes save the results to stored data
            for i, n in enumerate(nodes):
                n.updateNode(uj1[i], vj1[i])

            # Save intermediate spring lengths
            for s in springs:
                s.savelength()

        print("Simulation Complete...")
        return times, energy


# ====================================================
# ==== Numerical Solvers ============================
# ====================================================

# Euler Explicit
def eulexstep(aj, vj, uj, dt, nodes):

    vj1 = vj + dt * aj
    uj1 = uj + dt * vj

    return vj1, uj1


# Euler Symplectic
def eulsymstep(aj, vj, uj, dt, nodes):

    vj1 = vj + dt * aj
    uj1 = uj + dt * vj1

    return vj1, uj1


# Explicit Collatz
def collatzstep(aj, vj, uj, dt, nodes):

    vh = vj + dt / 2 * aj
    uh = uj + dt / 2 * vj

    # preliminary update
    for i, n in enumerate(nodes):
        n.updateNode(uh[i], vh[i], sav = False)
    # intermediate acceleration
    ah = np.array([n.forceSum(False) / n.m for n in nodes])

    vj1 = vj + dt * ah
    uj1 = uj + dt * vh

    return vj1, uj1


# Explicit Runge-Kutta 4th Order
def rungeexstep(aj, vj, uj, dt, nodes):

    u2h = uj + dt / 2 * vj
```

```
    v2h = vj + dt / 2 * aj
    for i, n in enumerate(nodes):
        n.updateNode(u2h[i], v2h[i], sav = False)
    a2h = np.array([n.forceSum(False) / n.m for n in nodes])
    u3h = uj + dt / 2 * v2h
    v3h = vj + dt / 2 * a2h
    for i, n in enumerate(nodes):
        n.updateNode(u3h[i], v3h[i], sav = False)
    a3h = np.array([n.forceSum(False) / n.m for n in nodes])
    u4h = uj + dt * v3h
    v4h = vj + dt * a3h
    for i, n in enumerate(nodes):
        n.updateNode(u4h[i], v4h[i], sav = False)
    a4h = np.array([n.forceSum(False) / n.m for n in nodes])

    uj1 = uj + dt / 6 * (vj + 2 * v2h + 2 * v3h + v4h)
    vj1 = vj + dt / 6 * (aj + 2 * a2h + 2 * a3h + a4h)

    return vj1, uj1


# Symplectic Stormer-Verlet Method
def stormer(aj, vj, uj, dt, nodes):

    uj1 = uj + dt * vj + dt ** 2 / 2 * aj

    for i, n in enumerate(nodes):
        n.updateNode(uj1[i], vj[i], sav = False)
    aj1 = np.array([n.forceSum(False) / n.m for n in nodes])

    vj1 = vj + dt / 2 * (aj + aj1)

    return vj1, uj1
```

```
"""
    Numerical Differential Equations
    Framework Project Code: Framework Building
    Instructor: Dr. Andrea Dziubek
    Prepared by Gregory Georgiades and David Petrushenko
    Last Updated: 5/1/2017
"""


from framework_classes import Node, Spring

# ======================================================
# === Buidling the Framework ===========================
# ======================================================


def buildframework1(k = 1):

    nodes = [
            Node([-5, 0], fixed = True),
            Node([0, 0]),
            Node([5, 0], fixed = True)
            ]
    Spring.nodes = nodes # sets spring class var - all springs have knowledge of
    all nodes
    Spring.k = k
    springs = [
                Spring('0+1'),
                Spring('1+2')
              ]
    Spring.nodes = None # clear it so space saved??? - needed?
    nodes[1].addLoad([-3, -0.01])
#     nodes[1].addDisplacement([0, -0.1])

    return nodes, springs

def buildframework2(k = 1):

    nodes = [
            Node([ 0, 5], fixed = True),
            Node([ 0, -5]),
            Node([ 2.5, 0]),
            Node([-2.5, 0])
            ]
    Spring.nodes = nodes
    Spring.k = k
    springs = [
                Spring('0+2'),
                Spring('1+2'),
                Spring('0+3'),
                Spring('1+3'),
                Spring('2+3'),
              ]

    nodes[1].addLoad([0, -1])
#     nodes[2].addDisplacement([0, 1])

    return nodes, springs
```

```
59
def buildSmallPendulum(k = 1):
61
    nodes = [
63              Node([ 0, 0], fixed = True),
                Node([ 0, 2]),
65              Node([ 0.1, 3])
                ]
67  Spring.nodes = nodes
    Spring.k = k
69  springs = [
                Spring('0+1'),
71              Spring('1+2')
                ]
73  for n in nodes:
        n.addLoad([0, -9.8])
75 #     nodes[2].addDisplacement([1, 1])

77  return nodes, springs

79
def buildBigPendulum(k = 1):
81
    nodes = [
83              Node([ 0, 0], fixed = True),
                Node([ 0, 1]),
85              Node([ 0, 2]),
                Node([ 0, 3]),
87              Node([ 0, 4]),
                Node([ 0, 5]),
89              Node([ 0, 6]),
                Node([ 0.01, 7])
91              ]
    Spring.nodes = nodes
93  Spring.k = k
    springs = [
95                Spring('0+1'),
                  Spring('1+2'),
97                Spring('2+3'),
                  Spring('3+4'),
99                Spring('4+5'),
                  Spring('5+6'),
101               Spring('6+7'),
                ]
103 for n in nodes:
        n.addLoad([0, -9.81])
105#     nodes[2].addDisplacement([1, 1])

107 return nodes, springs

109
def buildSmallTruss():
111
    nodes = [
113           Node([0, 0], True),
              Node([0.5, 1]),
115           Node([1, 0]),
              Node([1.5, 1]),
117           Node([2, 0]),
```

```
            Node([2.5, 1]),
            Node([3, 0], True),
            ]
    springs = [
              Spring('0+1'),
              Spring('0+2'),
              Spring('1+2'),
              Spring('1+3'),
              Spring('2+3'),
              Spring('2+4'),
              Spring('3+4'),
              Spring('3+5'),
              Spring('4+5'),
              Spring('4+6'),
              Spring('5+6'),
              ]
    nodes[3].addLoad([0, -0.05])
    # nodes[3].addDisplacement([0, -0.25])

    return nodes, springs

def buildFloorTruss(k = 1):

    nodes = [
            Node([ -4, 1], fixed = True),
            Node([ -3, 1]),
            Node([ -3, 0]),
            Node([ -2, 1]),
            Node([ -1, 0]),
            Node([ -1, 1]),
            Node([ 0, 1]),
            Node([ 1, 0]),
            Node([ 1, 1]),
            Node([ 2, 1]),
            Node([ 3, 0]),
            Node([ 3, 1]),
            Node([ 4, 1], fixed = True),
            ]
    Spring.nodes = nodes
    springs = [
              Spring('0+1'),
              Spring('0+2'),
              Spring('1+2'),
              Spring('1+3'),
              Spring('2+3'),
              Spring('2+4'),
              Spring('3+4'),
              Spring('3+5'),
              Spring('5+4'),
              Spring('5+6'),
              Spring('4+6'),
              Spring('4+7'),
              Spring('6+7'),
              Spring('6+8'),
              Spring('8+7'),
              Spring('8+9'),
              Spring('7+9'),
              Spring('7+10'),
              Spring('9+11'),
```

```
                Spring('9+10'),
                Spring('11+10'),
                Spring('11+12'),
                Spring('12+10'),
                ]
    Spring.k = k
    for i in [1, 3, 5, 6, 8, 9, 11]: # top, free nodes only
        nodes[i].addLoad([0, -0.025])
    # nodes[2].addDisplacement([1, 1])

    return nodes, springs


def buildCrane(k = 1):
    nodes = [
            Node([ 0, 0], fixed = True),
            Node([ 1, 0], fixed = True),
            Node([ 2, 0], fixed = True),
            Node([ 3, 0], fixed = True),
            Node([ 1, 1]),
            Node([ 2, 1]),
            Node([ 1, 2]),
            Node([ 2, 2]),
            Node([ 1, 3]),
            Node([ 2, 3]),
            Node([ 1, 4]),
            Node([ 2, 4]),
            Node([ 1, 5]),
            Node([ 2, 5]),
            Node([ -1, 6]),
            Node([ 0, 6]),
            Node([ 1, 6]),
            Node([ 2, 6]),
            Node([ 3, 6]),
            Node([ 4, 6]),
            Node([ 5, 6]),
            Node([ 6, 6]),
            Node([ -0.5, 7]),
            Node([ 0.5, 7]),
            Node([ 1, 7]),
            Node([ 1.5, 7]),
            Node([ 2, 7]),
            Node([ 2.5, 7]),
            Node([ 3.5, 7]),
            Node([ 4.5, 7]),
            Node([ 5.5, 7]),
            Node([ 1, 8.5]),
            ]
    Spring.nodes = nodes
    Spring.k = k
    springs = [
                Spring('0+1'),
                Spring('1+2'),
                Spring('2+3'),
                Spring('0+4'),
                Spring('1+4'),
                Spring('1+5'),
                Spring('2+5'),
                Spring('3+5'),
```

```
                Spring('4+5'),
                Spring('4+6'),
                Spring('4+7'),
                Spring('7+5'),
                Spring('6+7'),
                Spring('6+8'),
                Spring('6+9'),
                Spring('9+7'),
                Spring('8+9'),
                Spring('8+10'),
                Spring('8+11'),
                Spring('9+11'),
                Spring('10+11'),
                Spring('10+12'),
                Spring('10+13'),
                Spring('11+13'),
                Spring('12+13'),
                Spring('12+16'),
                Spring('12+17'),
                Spring('13+17'),

                Spring('14+15'),
                Spring('15+16'),
                Spring('16+17'),
                Spring('17+18'),
                Spring('18+19'),
                Spring('19+20'),
                Spring('20+21'),

                Spring('22+23'),
                Spring('23+24'),
                Spring('24+25'),
                Spring('25+26'),
                Spring('26+27'),
                Spring('27+28'),
                Spring('28+29'),
                Spring('29+30'),

                Spring('14+22'),
                Spring('22+15'),
                Spring('15+23'),
                Spring('23+16'),
                Spring('16+24'),
                Spring('16+25'),
                Spring('17+25'),
                Spring('17+26'),
                Spring('17+27'),
                Spring('18+27'),
                Spring('18+28'),
                Spring('19+28'),
                Spring('19+29'),
                Spring('20+29'),
                Spring('20+30'),
                Spring('21+30'),

                Spring('22+31'),
                Spring('24+31'),
                Spring('26+31'),
```

```
295                 Spring('29+31'),
                    ]
297     Spring.nodes = None
        nodes[21].addLoad([0, 1])
299     nodes[14].addLoad([0, 2])

301     return nodes, springs
```