Kuunal Mahtani
114544951

PHY604 Final Project

December 14th 2021

Contents

1 Introduction/Motivation

The 2D pendulum system is a classic example of a system often taught in introductory physics courses, as it is useful in introducing ubiquitous concepts such as harmonic oscillation and Taylor expansion to first order in order to solve a second order differential equation. The motion of this system can be solved with Newtonian physics and basic trigonometry. Figure 1 illustrates the basic diagram of a 2D pendulum, and indicates the basic way the system of equations will be set up henceforth in the paper. This system can be expanded to the 2D pendulum, a classical example of a chaotic system often explored in advanced Classical Mechanics courses; that is to say small differences in initial conditions can lead to very large differences in motion in time of the pendulum system. The motion of this system can be solved by exploring Lagrangian mechanics, and results in two coupled second-order ordinary differential equations of 2 variables when solved in terms of the angles that each mass makes with respect to equilibrium (the y-axis, for our diagram). Figure 2 illustrates a diagram of such a 2D double pendulum.

1.1 The n-pendulum

The triple pendulum is an even more interesting system, calculating the Lagrangian for this system results in a system of 3-coupled 2^{nd} order ODEs with 3 variables thereby increasing the degrees of freedom to a total of 18 - it is a somewhat more chaotic system. This process of adding pendulums can be repeated with the physics remaining the same; only the complexity of the problem increases with each additional pendulum. In fact, as will be explored later, for a system of n-pendulums, we can generalize the equations of motion to a system of n-coupled differential equations of n variables. Figure 3 illustrates a diagram of such an n-pendulum system in the coordinate axis that will be used henceforth. Note that in doing so, we will be taking into account the first order Taylor expansions of $\sin(\theta)$ and $\cos(\theta)$, where appropriate, such that the equation of motion can be reasonably determined. That is to say, we are taking small angle approximation in order to solve for the system of equations. The reasoning will become apparent later on.

When considering an n-pendulum, there are 2 main modes of solving for a system of equations of motion: Newtonian and Lagrangian. The Newtonian method and Lagrangian method both result in a system of second order differential equations to solve for the system of motion. However, the Newtonian method results in a recursive differential equation relationship that can be solved in terms of the Cartesian x **or** y position; thereby requiring one to either solve for both x and y at the same time, or to solve for the angles that the masses make with equilibrium alongside either the x or y positions, then using those values one may determine the other position (i.e. solve for x_i and keep the values of θ_i, then use those to determine y_i after the system has been solved). Either way, this increases numerical error and computation time. On the other hand, the Lagrangian method results in a system of coupled ordinary differential equations which can be solved for the θ_i, and the results can be used to determine both Cartesian positions and velocities at all points in time at once, after the computation has completed. Hence we shall opt to use the Lagrangian

method to solve for the system of equations. We shall present the Newtonian method purely for completeness.

1.2 Competing Algorithms

There are many competing algorithms used to solve a system of Ordinary Differential Equations (ODEs). Principally, the Euler Method and the Runge Kutta methods are usually standard due to their high degrees of accuracy and high reliability. It should be noted that the Runge Kutta method is essentially the Euler method to higher order. Taking the Runge Kutta method to 4^{th} order further increases accuracy with relatively little added work for the computer. The 4^{th} order Runge-Kutte algorithm also lends itself well to vectorization, thus making it very useful in solving large systems of ODEs. Moreover, the Leapfrog method, Euler-half-step method, and Bulirsch-Stoer methods are additional methods of solving ODEs, and though these methods alone are not as accurate as the 4^{th} order Runge Kutta method they have strength in that they have time-reversal symmetry[2].

One may incorporate adaptive step sizes and/or local extrapolation into any of these methods for increased accuracy at no extra cost. In fact, incorporating adaptive time-step measures into our program is necessary for the n-pendulum system considering it's chaotic nature.

Note however, that if we aim to solve a system of n-pendulums, the number of degrees of freedom increases dramatically with each additional mass, and therefore so does the amount of work that the computer has to do at each step - unless we incorporate elements of linear algebra, which we shall do. The motion of each particle can be organized into a $2 \times n \times n$ tensor (for Cartesian coordinates, or an $n \times n$ matrix for polar coordinates through which the x and y components can be determined), which changes our system to a large, single matrix differential equation. Note that this may be solved as a large eigenvalue problem[2], however we shall continue to aim to solve this as a system of ODEs.

Since we are interested in a chaotic system, adaptive step size implementation is integral to an accurate representation of the motion of the system. Recall that the basis of solving the system of equations using the Lagrangian method is founded on minimizing the action potential S. Hence if we are going to solve the equation using the Lagrangian method we are require conservation of energy. Therefore we aim to choose a numerical method that is time-reversal symmetric - so we must eliminate the 4^{th} order Runge Kutta method from our choices of computational methods. We would ideally like the system to be as accurate as possible, however since our focus is on a n-pendulum system, we will in theory have many, many degrees of freedom - and therefore exponentially more calculations per step. We aim to choose a numerical method more efficient at handling large systems, so we must eliminate the Bulirsch-Stoer method. Therefore, we will choose to solve this system using the Leapfrog/Verlet method[2]. The Verlet method proceeds exactly as the Leapfrog method does but is used for second order ODEs, hence we shall proceed with the Verlet method.

2 Computational Approach

Note that for all proceeding calculations, we will be assuming small angle approximation (where appropriate), and also that the ropes connecting the masses in the n-pendulum system are massless and always taught.

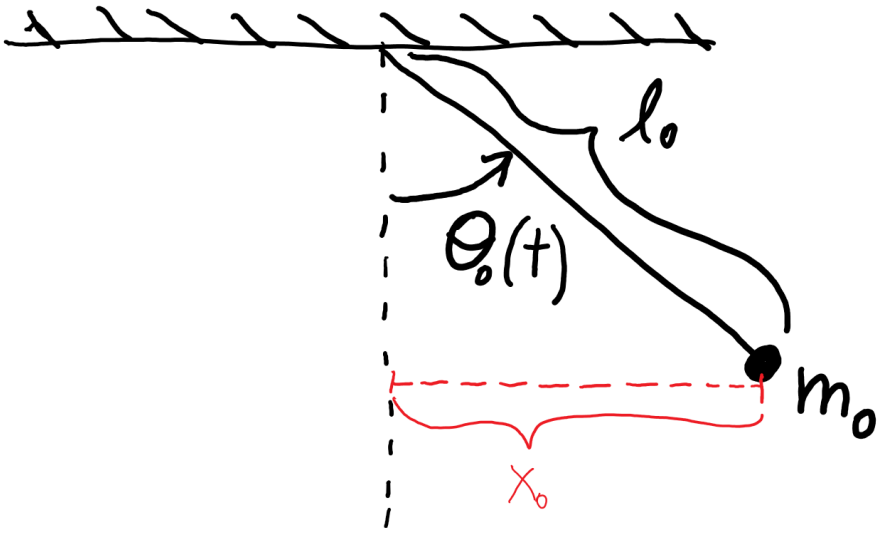

Figure 1: Diagram Illustrating $2D$ Pendulum

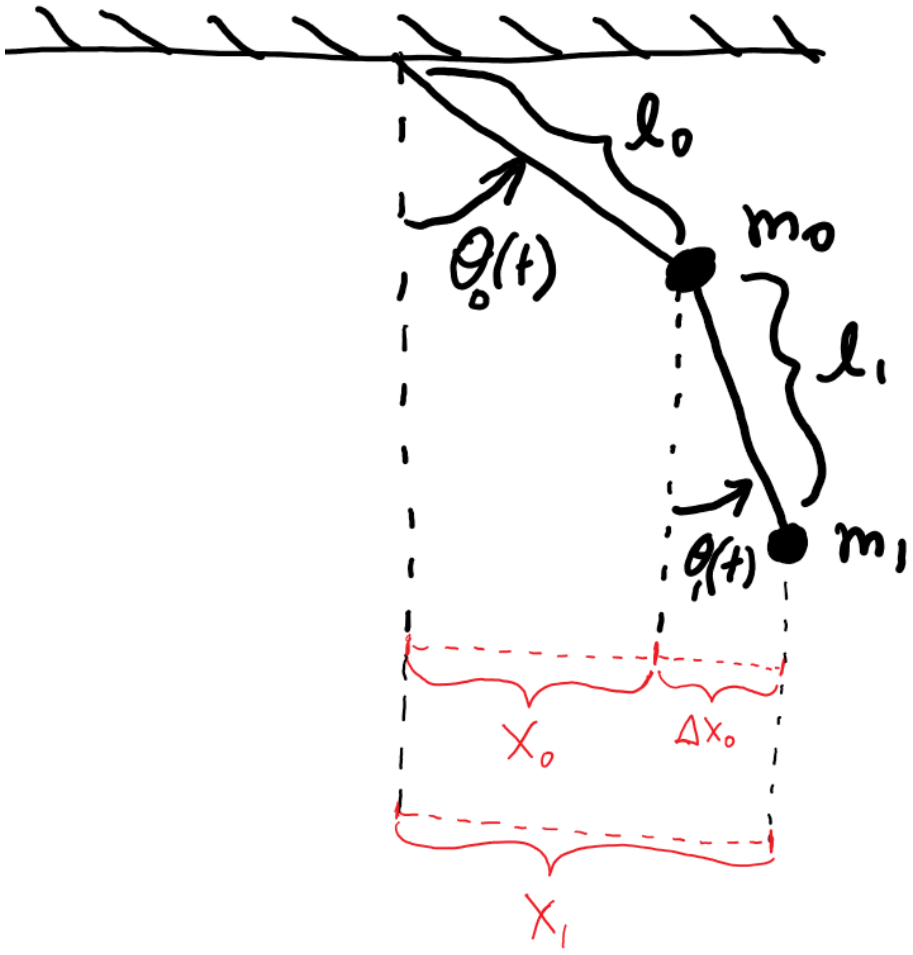

Figure 2: Diagram Illustrating $2D$ 2-Pendulum

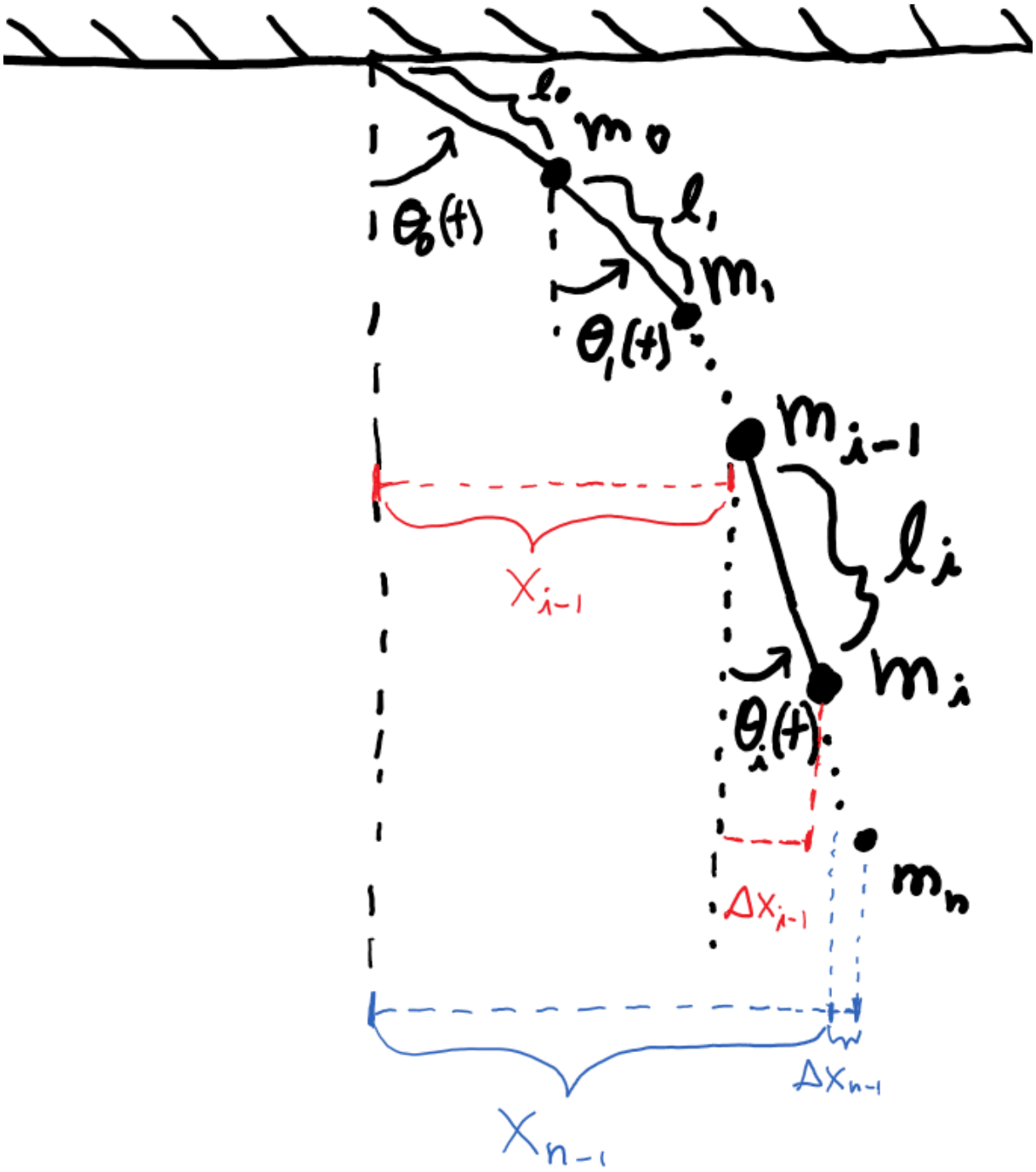

Figure 3: Diagram Illustrating $2D$ $n+1$-Pendulum

2.1 Newtonian Approach

The Newtonian method of solving the system requires a summation of forces on each pendulum in the system, each summation of forces will result in a second order differential equation of n variables; for a total of n-second order differential equations of n variables. We proceed by defining the position in x that each mass is away from equilibrium to be the sum of the Δx values of each mass before it (see figure 3). We proceed similarly to determine the position in y that each mass is vertically away from the vertical position of the pivot - set at the origin for simplicity. We find for the i^{th} mass:

$$x_i = \sum_{j=1}^{i} l_j \sin(\theta_j) \qquad\qquad y_i = \sum_{j=1}^{i} l_j \cos(\theta_j)$$

Wherein i is the index indicating the i^{th} mass, where $i \in \mathbb{Z}$, and l_i is the length of the rope between the i^{th} and $(i-1)^{th}$ masses. Note that the tension force on the $(i+1)^{th}$ mass $F_{T_{i+1}}$ due to all the masses before it is equal to the restoring force on the i^{th} mass F_i due to all the masses after it. With reference to Figure 3, it is clear to see that F_i is given by:

$$F_i = \sum_{j=i+1}^{n} m_j g \tan(\theta_i)$$

Wherein g is the gravitational constant, n is the total number of masses in the pendulum system, and m_j is the mass (units of weight) of the j^{th} mass. Hence, a force balance calculation on the i^{th} pendulum produces a recurrence relation differential equation for the equation of motion of the i^{th} pendulum:

$$F_{net_i} = m_i \ddot{x}_i = F_i - F_{T_{i-1}} = \sum_{j=i+1}^{n} m_j g \tan(\theta_i) - \sum_{j=i}^{n} m_j g \tan(\theta_{i-1})$$

After some algebra and appropriate approximations as presented in Reference 1[1], we arrive at the following recursive differential equation relationship:

$$\ddot{x}_i = g\left[(n-i)\left(\frac{x_{i+1}-x_i}{l_{i+1}} - \frac{x_i - x_{i-1}}{l_i}\right) - \left(\frac{x_i - x_{i-1}}{l_i}\right)\right]$$

2.2 Lagrangian Approach

From the time derivative of the aforementioned position in x and y of the i^{th} mass, we obtain:

$$\dot{x}_i = \sum_{j=1}^{i} \dot{\theta}_j l_j \cos(\theta_j) \qquad\qquad \dot{y}_i = -\sum_{j=1}^{i} \dot{\theta}_j l_j \sin(\theta_j)$$

From these, we can obtain the Kinetic Energy and Potential Energy as:

$$T = \frac{1}{2}\sum_{i=1}^{n} m_i v_i^2 = \frac{1}{2}\sum_{i=1}^{n} m_i \left(\dot{x_i}^2 + \dot{y_i}^2\right) = \frac{1}{2}\sum_{i=1}^{n} m_i \left[\left(\sum_{j=1}^{i} \dot{\theta_j} l_j \cos(\theta_j)\right)^2 + \left(\sum_{j=1}^{i} \dot{\theta_j} l_j \sin(\theta_j)\right)^2\right]$$

$$U = -g\sum_{i=1}^{n} m_i y_i = -g\sum_{i=1}^{n}\sum_{j=1}^{i} m_i l_j \cos(\theta_j)$$

Therefore, the Lagrangian determined using the Lagrangian method is presented as follows[1]:

$$L = \frac{1}{2}\sum_{i=1}^{n}\left(m_i\left[\left(\sum_{j=1}^{i} l_j\dot{\theta_j}\cos\theta_j\right)^2 + \left(\sum_{j=1}^{i} l_j\dot{\theta_j}\sin\theta_j\right)^2\right]\right) + g\sum_{i=1}^{n}\left(\sum_{j=1}^{i}\left[m_i l_j\cos(\theta_j)\right]\right)$$

For simplicity of the solution, let us assume small angle approximation ($\sin(\theta_i) \approx \theta_i$, $\cos(\theta_i) \approx 1 - \frac{\theta_i^2}{2}$). This assumption shall be retained for the remainder of the paper. With this assumption in mind, ignoring higher order terms and constants in the initial Lagrangian (keeping with small angle approximation), the Lagrangian becomes[1]:

$$L \approx \frac{1}{2}\sum_{i=1}^{n}\left(m_i\left[\left(\sum_{j=1}^{i} l_j\dot{\theta_j}\right)^2 - g\sum_{j=1}^{i} l_j\theta_j^2\right]\right)$$

2.3 Parallelizing n-Second Order ODEs of n variables (Identical Masses, Identical Lengths)

For simplicity of understanding, let us begin with the assumption that the length between each mass in the n-pendulum system is constant and equal ($l_i = l, \quad \forall l_i$), and with the assumption that the masses of each mass are identical ($m_i = m, \quad \forall m_i$). From the standard Euler-Lagrange equations and keeping such assumptions in mind, such a Lagrangian produces the following differential equation for θ_k[1]:

$$\frac{d}{dt}\frac{dL}{d\dot{\theta_k}} = \frac{dL}{d\theta_k}$$

$$\sum_{j=1}^{n}(n + max(k,j) + 1)\ddot{\theta_j} + (n - k + 1)\frac{g}{a}\theta_k = 0$$

Wherein θ_i is the angle between the z-axis and the string connecting the i^{th} and $(i-1)^{th}$ masses, i.e. $\theta_i = \frac{x_i - x_{i-1}}{l_i}$. Note that if we let $n = 1$, we get the differential equation for the single pendulum:

$$\ddot{\theta_1} = -\frac{g}{l}\theta_1$$

If we let $n = 2$, we end up with a system of 2 differential equations with 2 variables. In fact, for any n, we get a system of n differential equations with n variables. i.e. for $n = 2$:

$$2\ddot{\theta_1} + \ddot{\theta_2} = -2\frac{g}{l}\theta_1$$
$$\ddot{\theta_1} + \ddot{\theta_2} = -\frac{g}{l}\theta_1$$

For $n = 3$:

$$3\ddot{\theta_1} + 2\ddot{\theta_2} + \ddot{\theta_3} = -3\frac{g}{l}\theta_1$$
$$2\ddot{\theta_1} + 2\ddot{\theta_2} + \ddot{\theta_3} = -2\frac{g}{l}\theta_2$$
$$\ddot{\theta_1} + \ddot{\theta_2} + \ddot{\theta_3} = -\frac{g}{l}\theta_3$$

And so on. We may re-arrange these equations such that we have a system of differential equations that is more familiar. i.e. for $n = 2$:

$$\ddot{\theta_1} = \frac{g}{l}\left(2\theta_1 - \theta_2\right)$$
$$\ddot{\theta_2} = \frac{g}{l}\left(-2\theta_1 + 2\theta_2\right)$$

for $n = 3$:

$$\ddot{\theta_1} = \frac{g}{l}\left(3\theta_1 - 2\theta_2\right)$$
$$\ddot{\theta_2} = \frac{g}{l}\left(-3\theta_1 + 4\theta_2 - \theta_3\right)$$
$$\ddot{\theta_3} = \frac{g}{l}\left(-2\theta_2 + 2\theta_3\right)$$

We may represent this system of equations in matrix notation, where $\mathbf{T}_n = (\theta_1, \theta_2, ..., \theta_n)$ such that $\ddot{\mathbf{T}}_n = (\ddot{\theta_1}, \ddot{\theta_2}, ..., \ddot{\theta_n})$, and $\ddot{\mathbf{T}}_n = -\frac{g}{l}\mathbf{M}_n$, where $\mathbf{M}$ is the matrix where the j^{th} column represents the j^{th} θ variable (θ_j), and the i^{th} row represents the coefficients of each θ_j, which when combined in a linear sum gives the differential equation for $\ddot{\theta_i}$. Note that here $\ddot{\mathbf{T}}_n \in \mathbb{R}^n, \mathbf{M}_n \in \mathbb{R}^n \times \mathbb{R}^n$. i.e.:

$$\left(\ddot{\mathbf{T}}_3 = \begin{bmatrix}\ddot{\theta_1}\\ \ddot{\theta_2}\\ \ddot{\theta_3}\end{bmatrix} = -\tfrac{g}{l}\begin{bmatrix}3 & -2 & 0\\ -2 & 4 & -1\\ 0 & -2 & 2\end{bmatrix}\right)$$

However, it is important to notice that this representation is a slight abuse of notation presented for readability. We cannot set a vector equal to a matrix, so we must adjust our parameters. We may instead present the $\mathbf{T}_n$ vector as a matrix with diagonal entries, where the j^{th} column represents the j^{th} $\ddot{\theta}$ variable $(\ddot{\theta_j})$, and the i^{th} row represents the coefficients of each $\ddot{\theta_j}^{th}$ ODE. In example:

$$\left(\ddot{\mathbf{T}}_3 = \begin{bmatrix} \ddot{\theta}_1 & 0 & 0 \\ 0 & \ddot{\theta}_2 & 0 \\ 0 & 0 & \ddot{\theta}_3 \end{bmatrix} = -\tfrac{g}{l} \begin{bmatrix} 3 & -2 & 0 \\ -2 & 4 & -1 \\ 0 & -2 & 2 \end{bmatrix}\right)$$

Or, more rigorously, this is the form of the differential equation the algorithm 'sees':

$$\left(\ddot{\mathbf{T}}_3 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} = -\tfrac{g}{l} \begin{bmatrix} 3 & -2 & 0 \\ -2 & 4 & -1 \\ 0 & -2 & 2 \end{bmatrix}\right)$$

The pattern here may not yet be apparent, but it becomes much clearer if we extend to $n = 5$:

$$\ddot{\mathbf{T}}_5 = -\frac{g}{l} \begin{bmatrix} 5 & -4 & 0 & 0 & 0 \\ -5 & 8 & -3 & 0 & 0 \\ 0 & -4 & 6 & -2 & 0 \\ 0 & 0 & -3 & 4 & -1 \\ 0 & 0 & 0 & -2 & 2 \end{bmatrix}$$

From this, we can notice that the system of equations can be represented as:

$$\ddot{\mathbf{T}}_n = -\frac{g}{l} \begin{bmatrix} n & -(n-1) & 0 & 0 & \dots & \dots & \dots & \dots & \dots & 0 \\ -n & 2(n-1) & -(n-2) & 0 & \dots & \dots & \dots & \dots & \dots & 0 \\ 0 & -(n-1) & 2(n-2) & -(n-3) & \dots & \dots & \dots & \dots & \dots & \vdots \\ 0 & 0 & -(n-2) & \ddots & \ddots & \dots & \dots & \dots & \dots & \vdots \\ 0 & 0 & 0 & \ddots & \ddots & \ddots & \dots & \dots & \dots & \vdots \\ \vdots & \vdots & \vdots & \vdots & \ddots & \ddots & -(n-i) & \dots & \dots & \vdots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & 2(n-i) & \ddots & \dots & \vdots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & -(n-i) & \ddots & \ddots & \vdots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \ddots & -(n-(n-1)) \\ 0 & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & 2(n-(n-1)) \end{bmatrix}$$

That is to say, if we refer to the matrix of coefficients of the θ_i as $\mathbf{M}$, $\mathbf{M}_{ij} = 2(n-i)$ if $j = i$, $\mathbf{M}_{ij} = -(n-i)$ if $j = i \pm 1$, and $\mathbf{M}_{ij} = 0$ for all other values of i, j. The only exception to this rule is the value in the first row, first column, which is always n. Henceforth, let us denote $\mathbf{T}_n = \mathbf{T}$ since n is simply the number of masses in the pendulum system. The reasoning will become apparent when explaining the Verlet method. **Note: The matrix of constants is M, the matrix denoted by T is a diagonal matrix with the values of θ_i along the i^{th} diagonals.** We may hence represent T as $T(t)$. Given that this matrix has a relatively simple pattern, it is easy to calculate an $n \times n$ matrix for any given value of n; so we shall design and assign a function `get_matrix()` to do so.

```python
def get_matrix(n):
    mat = np.zeros((n,n))
    for i in range(n):
        try:
            mat[i, i] = 2*(n-i)
            mat[i+1, i] = -(n-i)
            mat[i-1, i] = -(n-i)
        except IndexError:
            pass
    # Fixing errors for the matrix
    mat[0][0] = n
    mat[n-2][-1] = -1
    # n = 2 and n = 1 are special cases
    if (n > 2):
        mat[-1][0] = 0.
    return mat
```

2.4 The Verlet Method

Note that with the matrix in this form, we have a system of differential equations which can be represented as follows:

$$\ddot{\theta}_i = -\frac{g}{l}\left(-(n-(i-1))\theta_{(i-1)} + 2(n-i)\theta_i - (n-(i+1))\theta_{i+1}\right)$$

$$f_i(\mathbf{T}, t) = -(n-(i-1))\theta_{(i-1)} + 2(n-i)\theta_i - (n-(i+1))\theta_{i+1}$$

$$\frac{d^2}{dt^2}\mathbf{T} = -\frac{g}{l}\mathbf{f}(\mathbf{T}, t)$$

In fact, we have put our system of equations into a form which is perfect for the Verlet method, and which we have parallelized such that it is not much more computationally taxing to solve for than a system of 1 differential equation with 1 variable. We aim to solve this using the Verlet method. Noting that this is a second order system of ODEs, we introduce a new variable $\mathbf{v}$ as follows:

$$\mathbf{v} = \frac{d\mathbf{T}}{dt}$$

In doing so, we have turned the system of n second order ODEs into a system of $2n$ coupled first order ODEs[2]. From here, we first determine the value of $\mathbf{v}$ at one half-timestep for a chosen timestep h. We proceed as follows:

$$\mathbf{v}\left(t + \frac{1}{2}h\right) = \mathbf{v}\left(t\right) + \frac{1}{2}h\mathbf{f}(\mathbf{T}(t), t)$$

Using this first half timestep, we may determine the next value of each θ_i value at once using $\mathbf{T}(t)$:

$$\mathbf{T}(t+h) = \mathbf{T}(t) + h\mathbf{v}\left(t + \frac{1}{2}h\right)$$

Which we use to determine the matrix $\mathbf{k}$, used to determine $\mathbf{v}$ at the next timestep, and at the next half-timestep.

$$\mathbf{k} = h\mathbf{f}\left(\mathbf{T}\left(t+h\right), t+h\right)$$

$$\mathbf{v}(t+h) = \mathbf{v}\left(t + \frac{1}{2}h\right) + \frac{1}{2}\mathbf{k}$$

$$\mathbf{v}(t+\frac{3}{2}h) = \mathbf{v}\left(t + \frac{1}{2}h\right) + \mathbf{k}$$

We then repeat the process of evaluating the last 4 equations for a set number of timesteps[2]: we find the value of each θ_i at the next timestep using $\mathbf{v}$ determined at the most-recently determined half-timestep, calculating $\mathbf{k}$ using the equation for $\mathbf{f}$ evaluated at the newly determined θ_i values, and using $\mathbf{k}$ to determine $\mathbf{v}$ at the next timestep and at the next half-timestep.

When implementing this into our program, we start with initial conditions for the θ_i values and $\mathbf{v}$ values, organized in diagonal matrices as such:

$$\mathbf{T}(t=0) = -\frac{g}{l}\begin{bmatrix} \theta_1(t=0) & 0 & 0 & \dots & 0 \\ 0 & \theta_2(t=0) & 0 & \dots & 0 \\ 0 & 0 & \ddots & \ddots & \vdots \\ \vdots & \vdots & \ddots & \ddots & 0 \\ 0 & 0 & \dots & 0 & \theta_n(t=0) \end{bmatrix}$$

$$\mathbf{v}(t=0) = -\frac{g}{l}\begin{bmatrix} v_1(t=0) & 0 & 0 & \dots & 0 \\ 0 & v_2(t=0) & 0 & \dots & 0 \\ 0 & 0 & \ddots & \ddots & \vdots \\ \vdots & \vdots & \ddots & \ddots & 0 \\ 0 & 0 & \dots & 0 & v_n(t=0) \end{bmatrix}$$

Where $v_i = \frac{d\theta_i}{dt}$. We then proceed by evaluating $\mathbf{v}\left(\frac{1}{2}h\right) = \mathbf{v}\left(0\right) + \frac{1}{2}h\mathbf{f}(\mathbf{T}(0), 0)$, and $\mathbf{T}(h) = \mathbf{T}(0) + h\mathbf{v}\left(\frac{1}{2}h\right)$ trivially. When we evaluate $\mathbf{f}(\mathbf{T}(0), 0)$, we simply take the coefficients from our matrix of coefficients and multiply them by the corresponding θ_i values evaluated at that specific timestep. We then determine $\mathbf{k}$ trivially, and finally we determine $\mathbf{v}(t+h)$ and $\mathbf{v}(t+\frac{3}{2}h)$ using $\mathbf{k}$. We repeat the process of solving for these values for a given number of timesteps, and that concludes the main computation of our program.

2.5 Adaptive Timestep

Except, we have to note that since this is a chaotic system, some components will be moving at wildly different speeds than other components. Therefore, we have to implement an adaptive timestep to ensure accuracy. We proceed in the normal method for implementing an adaptive timestep. We first solve for two Verlet method processes (the aforementioned 4 equations) in timesteps of h. Let us denote the result as $\mathbf{T}^1$. We then solve for one Verlet method process (the aforementioned 4 equations) in a timestep of $2h$. Let us denote the result as $\mathbf{T}^2$. We take our error as:

$$\vec{\epsilon} = \frac{1}{30}\left(\mathbf{T}^1 - \mathbf{T}^2\right)$$

Where

$$\epsilon_i = \frac{1}{30}\left(\theta_i(t = t + 2h)^1 - \theta_i(t = t + 2h)^2\right)$$

And from this we can calculate:

$$\rho = \left[\sum_{i=0}^{n-1} (\epsilon_i)^2\right]^{-\frac{1}{2}} \delta h$$

Where δ is our target accuracy. If $\rho > 1$, then we know that the values of $\theta_i(t = t + 2h)^1$ and $\theta_i(t = t + 2h)^2$ are sufficiently close to each other such that they are accurate to our target accuracy. In this case, we keep the $\theta_i(t = t + 2h)^1$ values calculated and move on to the next timestep. Although, since $\rho > 1$, the timestep might be too small. Hence, we must increase the next timestep in an attempt to speed up the calculation as a timestep smaller than necessary may result in us doing more work than necessary at each step. We shall denote this new timestep by h'. However, if $\rho < 1$, this means that the values of $\theta_i(t = t+2h)^1$ and $\theta_i(t = t+2h)^2$ are not close enough to each other to meet our target accuracy. Hence, we need to decrease the timestep and repeat the process with the new timestep h'[2]. We continue this process until we find a timestep where $\rho > 1$, and proceed as mentioned.

h' is found identically in both situations as:

$$h' = h\rho^{\frac{1}{4}}$$

However, there are a few caveats to implementing this method towards solving the n-pendulum system using the Verlet method. Namely, when computed the ρ values start at < 1 and continue to approach 1, however they never reach 1. i.e. $\rho = 0.9999999$ after a number of repeated processes of trying smaller and smaller timesteps h. Since this causes the program not to converge, we must impliment an additional condition: if $\rho > 1 - \delta$, then we say that our $\theta_i(t = t + 2h)^1$ and $\theta_i(t = t + 2h)^2$ values are sufficiently close to each other and we can proceed to the next timestep.

Additionally, we must note that even though we are solving for $\mathbf{v}$ during each of our Verlet processes, we are not accounting for the error in $\mathbf{v}$ in the computation of each timestep. Hence, our $\mathbf{v}$ may not be accurate, even though our $\mathbf{T}$ will be. This is further explained in the discussions section.

In order to impliment this into our program, instead of setting a set number of timesteps of length h, we set a maximum timestep, and keep track of the sum of each timestep taken. In example, we will set an initial timestep h, and add it to a floating point object we label `time_passed`. After each successful timestep, we add the timestep h' to our `time_passed`, and we keep performing these calculations over and over until `time_passed >= time_final`; i.e. performing our calculations in a `while` loop.

2.6 Local Extrapolation

Note that even though we are saving our $\theta_i(t = t + 2h)^1$ values at each successful iteration of the adaptive step method, our saved values are not as accurate as, say, the 4^{th} order Runge Kutta Method. We may improve the accuracy of our calculations with little more effort by instead performing local extrapolation on our $\theta_i(t = t + 2h)^1$ values. That is to say, we redefine them by the following:

$$\theta_i(t = t + 2h)^1 = \theta_i(t = t + 2h)^1 + \frac{1}{15}\left(\theta_i(t = t + 2h)^1 - \theta_i(t = t + 2h)^2\right)$$

Which, for our program will look like:

$$\mathbf{T}^1 = \mathbf{T}^1 + \frac{1}{15}\left(\mathbf{T}^1 - \mathbf{T}^2\right)$$

And similarly:

$$\mathbf{v}^1 = \mathbf{v}^1 + \frac{1}{15}\left(\mathbf{v}^1 - \mathbf{v}^2\right)$$

In doing so, we increase the accuracy of our values to h^5 with an error to order h^6 at no extra work[2].

2.7 Varying Masses, Varying Lengths

This is all good for the very specific case of all masses and lengths between masses being identical, but what if we want varying masses and lengths between them? Luckily, the situation is much the same - in fact it is computationally identical. The only difference is determining the matrix of coefficients. Since masses and lengths remain constant throughout the problem, we only need to determine this matrix once at the beginning of our program and we are allowed to use it throughout our program. Let us first refer to the Lagrangian with small angle approximation:

$$L \approx \frac{1}{2}\sum_{i=1}^{n}\left(m_i\left[\left(\sum_{j=1}^{i} l_j\dot{\theta}_j\right)^2 - g\sum_{j=1}^{i} l_j\theta_j^2\right]\right)$$

We then proceed as before, however not making the assumptions of all masses and lengths being identical this time:

$$\frac{d}{d\theta_k}L \approx -\frac{g}{2}\sum_{i=1}^{n} m_i\left(\sum_{j=1}^{i} 2l_j\theta_k\delta_{jk}\right) = -g\sum_{i=1}^{n}\left(\sum_{j=1}^{i} m_i l_j\theta_k\delta_{jk}\right)$$

Where δ_{jk} is the Kronecker delta. As a result, we can re-arrange this to find:

$$\frac{d}{d\theta_k}L \approx -gl_k\theta_k\sum_{i=k}^{n} m_i$$

Likewise:

$$\frac{d}{d\dot{\theta}_k}L \approx \frac{1}{2}\sum_{i=1}^{n}\left(2m_i\left(\sum_{j=1}^{i} l_j\dot{\theta}_j l_k\delta_{jk}\right)\right) = \sum_{i=1}^{n}\left(\sum_{j=1}^{i} m_i l_j\dot{\theta}_j l_k\delta_{jk}\right)$$

$$\frac{d}{dt}\frac{d}{d\dot{\theta}_k}L \approx \sum_{i=1}^{n}\left(\sum_{j=1}^{i} m_i l_j\ddot{\theta}_j l_k\delta_{jk}\right)$$

Which we may expand into:

$$\frac{d}{dt}\frac{d}{d\dot{\theta}_k}L \approx m_1\left(l_1\theta_1 l_k\delta_{1k}\right) + m_2\left(l_1\theta_1 l_k\delta_{1k} + l_2\theta_2 l_k\delta_{2k}\right) + m_3\left(l_1\theta_1 l_k\delta_{1k} + l_2\theta_2 l_k\delta_{2k} + l_3\theta_3 l_k\delta_{3k}\right) + \dots$$

$$\cdots + m_i\left(l_1\theta_1 l_k\delta_{1k} + l_2\theta_2 l_k\delta_{2k} + \cdots + l_i\theta_i l_k\delta_{ik}\right) + \cdots + m_n\left(l_1\theta_1 l_k\delta_{1k} + l_2\theta_2 l_k\delta_{2k} + \cdots + l_n\theta_n l_k\delta_{nk}\right)$$

Re-arranging:

$$\frac{d}{dt}\frac{d}{d\dot{\theta}_k}L \approx l_k[\delta_{1k}\theta_1 l_1(m_1 + m_2 + \cdots + {}_n) + \delta_{2k}\theta_2 l_2(m_2 + \cdots + {}_n) + \delta_{3k}\theta_3 l_3(m_3 + \cdots + {}_n) + \dots$$

$$\cdots + \delta_{ik}\theta_i l_i\sum_{j=max(i,k)}^{n}(m_j) + \cdots + \delta_{nk}\theta_n l_n m_n)]$$

Setting these equal, we get:

$$-gl_k\theta_k\sum_{i=k}^{n} m_i = \sum_{i=1}^{n}\delta_{ki} l_k l_i\ddot{\theta}_i\sum_{j=max(i,k)}^{n} m_j$$

So this is our equation of motion for one "row" in our matrix. But now if we set up the matrices as before, we will have the matrix of θ_i values being diagonal, and the matrix of $\ddot{\theta}_i$ values being non-diagonal. To re-arrange to the form we are familiar with such that we may solve using the Verlet method, we simply perform the following linear algebra calculation:

$$\ddot{\mathbf{M}}\mathbf{M}$$
$$\ddot{\mathbf{M}}^{-1}\ddot{\mathbf{M}} = \ddot{\mathbf{M}}^{-1}\mathbf{M}$$
$$\mathbf{I} = \ddot{\mathbf{M}}^{-1}\mathbf{M}$$

And now we have diagonalized our matrix of $\ddot{\theta}_i$ values, and hence have put our system back in the form apt for solving via the Verlet method. Once we have this matrix of constant coefficients, we simply draw from it each time we are evaluating $\mathbf{f}$ as mentioned previously and proceed as normal with the Verlet method. No attempt at finding a "pattern" for such a matrix of constants has been made as before, as now we can simply use these equations to create a function to calculate this matrix for us, as denoted below. It is a relativelly simple computation and only needs to be done once, so no optimization is necessary - in that, no "finding a pattern" for the matrix of constants and creating a function to construct a matrix of that pattern will significantly improve the speed of our program.

```
def get_weight_matrix(n, m, l, g):
    m_theta = np.identity(n)
    for k in range(n):
        m_theta[k,k] = -g*l[k]*np.sum(m[0:n-k])
    print("theta matrix: \n", m_theta)
    print('\n')

    m_theta_dd = np.identity(n)
    for k in range(n):
        for j in range(n):
            m_theta_dd[k,j] = l[j]*l[k]*np.sum(m[np.max((k,j)):n])
    print("theta_double_dot matrix: \n",m_theta_dd)
    print('\n')

    inv_m_theta_dd = np.linalg.inv(m_theta_dd)
    weight_matrix = inv_m_theta_dd@m_theta
    print("system of equations/weight matrix:\n", weight_matrix)

    return weight_matrix
```

2.8 Gathering Results & Animation

The rest of the program is based off of gathering the results that we've stored from each step in the Verlet method and plotting the resulting motion of the pendulum system. The `recover_position()`

function accepts a tuple of matrices $\mathbf{T}$ and $\mathbf{v}$ (where each matrix is representative of the values at one timestep) and returns a single matrix of values of dimension `len(h_t)`$\times n$ - where `len(h_t)` is the number of timesteps we have computed. Each i^{th} row in this matrix represents a given i^{th} timestep, and each j^{th} column in this matrix represents the value of θ_j at the i^{th} timestep. Following, the function calculates either the x_j & y_j or $\dot{x_j}$ & $\dot{y_j}$ values for the j^{th} mass at each timestep i using the aforementioned:

$$x_j = \sum_{k=1}^{j} l_k \sin(\theta_k) \qquad y_j = \sum_{k=1}^{j} l_k \cos(\theta_k)$$

$$\dot{x_j} = \sum_{k=1}^{j} \dot{\theta}_k l_k \cos(\theta_k) \qquad \dot{y_j} = -\sum_{k=1}^{j} \dot{\theta}_k l_k \sin(\theta_k)$$

Note that given that the timestep is adaptive, we do not know how many timesteps we have - amounts such as 10^2 and 10^8 are both as likely, it depends on the number of masses, the lengths between the massess, and the final time that we set. Hence, when we want to animate the motion, we must be selective of how many frames we want to use, and which frames we want to use. Hence, the next cell in the Jupyter Notebook calculates the number of frames to skip in between in order to match a total number of frames in the animation.

```
ideal_rate = len(h_t)
rate = 1
while (ideal_rate/rate > 800):
    rate += 1
```

In this case, the `rate` is calculated such that we will have `800` frames in our final animation.

Finally, the `animate()` function creates the animation by plotting the position of each mass at any given time, and keeps the initial conditions plotted. This function is nearly identical to the animation function shown in class, with minor changes to fit the n-pendulum system. The size of each mass on the animation is changed to the value of it's mass, multiplied by 3 (3 was chosen arbitrarily to increase visibility of smaller masses, such as $m = 1$). The masses are plotted using the colour map `Viridis` to make it easier to visually distinguish between each mass to improve visibility for viewers with common forms of colour-blindness.

The program was written on a Windows machine, which does not have `FFMPEG` packages installed by default, hence there is a small section of code used to point to the locally stored `FFMPEG` executable such that the animations can be saved as `.mp4` files. If one runs the program on their windows machine and wishes the animation to be saved in `.mp4` format, they will need to change the path to point to the locally stored `FFMPEG` executable.[4] This line has been commented out in the final submission of the code by default if one wishes to run this on a Unix based OS. However, there are commented out lines of code using `PillowWriter` which save the animations as `.gif` files, in the event that one would prefer to save the animations in the `.gif` file format.

2.9 Energy Calculations

The Energy of the system can be determined in the regular manner. Note however, that if we do so we will not be finding the energy of our system - per se. Using the usual equations of energy as:

$$T = \frac{1}{2}\sum_{i=1}^{n} m_i \left[\left(\sum_{j=1}^{i} \dot{\theta}_j l_j \cos(\theta_j) \right)^2 + \left(\sum_{j=1}^{i} \dot{\theta}_j l_j \sin(\theta_j) \right)^2 \right]$$

$$U = -g \sum_{i=1}^{n} \sum_{j=1}^{i} m_i l_j \cos(\theta_j)$$

would be accurate, however note that these functions are not small angle approximated - which is how we proceeded with the entire calculation. Hence, we must small angle approximate the energy to determine the energy of our simulated system. We do this to get:

$$T = \frac{1}{2}\sum_{i=1}^{n} m_i \left(\sum_{j=1}^{i} \dot{\theta}_j l_j \right)^2$$

$$U = -\frac{g}{2} \sum_{i=1}^{n} \sum_{j=1}^{i} m_i l_j \theta_j^2$$

And calculate the total energy over each timestep accordingly. Note that for more chaotic systems (i.e. $n \geq 3$) the total energy vs time plots appear a bit disorganized (as shall be seen in the results section), hence we must account for this by calculating a rolling average[3] of the total energy over time to better understand the trend in Total Energy over time, and so we do this. These calculations and plots are computed and generated using the `get_energy()` function, trivially. This function is too long to be included in the report, but is well commented in the corresponding Jupyter notebook.

3 Results

3.1 Primary Results

Several results were produced for the system of n pendulums. Namely, the following 12 animations were produced to illustrate the capabilities of the program (Note: initial velocities for all results were set to 0 unless otherwise specified):

- System of 3 pendulums with identical masses, identical lengths, identical starting angles = $\frac{3\pi}{4}$
- System of 3 pendulums with identical masses, identical lengths, identical starting angles = $\frac{3\pi}{4}$ + 10degrees

- System of 3 pendulums with increasing masses, identical lengths, identical starting angles $= \frac{3\pi}{4}$
- System of 3 pendulums with identical masses, increasing lengths, identical starting angles $= \frac{3\pi}{4}$
- System of 3 pendulums with increasing masses, decreasing lengths, identical starting angles $= \frac{3\pi}{4}$
- System of 3 pendulums with decreasing masses, increasing lengths, identical starting angles $= \frac{3\pi}{4}$
- System of 3 pendulums with increasing masses, increasing lengths, identical starting angles $= \frac{3\pi}{4}$
- System of 3 pendulums with identical masses, identical lengths, differing starting angles
- System of 3 pendulums with increasing masses, increasing lengths, differing starting angles
- System of 3 pendulums with identical masses, identical lengths, identical starting angles $= \frac{3\pi}{4}$, non-zero initial velocity
- System of 3 pendulums with identical masses, identical lengths, identical starting angles $= \frac{3\pi}{4}$, large non-zero initial velocity
- System of 3 pendulums with differing masses, differing lengths, differing starting angles, non-zero initial velocity

All animations are available in the git, alongside the energy vs time plots. Note that the total energy varies with time in an odd manner. We would expect that energy over time varies sinusoidally when using the Verlet method, however such a pattern is not exactly apparent. Hence, rolling average[3] plots for the energy vs. timestep were also computed for each scenario, and are appropriately labelled. The plots - being .png files - were left in the git alongside the animations for clarity and organizational purposes, as to not lengthen the report.

3.2 Interesting Results

Here are some examples of results produced that illustrate interesting patterns, either in their movement and/or in their energy plots.

- System of 5 Pendulums with identical masses, identical lengths, identical starting angles $= \frac{3\pi}{4}$
- System of 9 Pendulums with identical masses, identical lengths, identical starting angles $= \frac{3\pi}{4}$

- System of 2 Pendulums with identical masses, different lengths, small starting angles = $(10°, 20°)$
- System of 27 Pendulums with identical masses, identical lengths, identical starting angles at $\frac{\pi}{2}$
- System of 4 Pendulums with varying masses, identical lengths, identical starting angles at nearly π
- System of 4 Pendulums with varying masses, varying lengths, varying starting angles

The program works correctly for any system of n pendulums and any starting angle in the range $(-\pi, \pi)$. It works for any mass amounts where each mass is > 0 and for any length amounts where each length is > 0. Finally, the program works for any initial velocities ≥ 0 that are reasonably not too large - these limitations are discussed in the followings sections.

4 Discussion

4.1 Accuracy of Simulation to Real Physics

4.1.1 General Motion

We should expect that the program works for masses and lengths > 0, as masses and lengths ≤ 0 are un-physical.

The most notable result from viewing the motion of the pendulum systems is that the animations appear slightly "floaty." This is in large part due to the adaptive step method. The animation function animates the motion of the pendulum by plotting the positions of all masses for a given timestep as a single frame, hence when animated together, the frames played at a constant rate will play back at "adaptive time" leading to motion perceived as "floaty". This effect is best seen in the 3 pendulum system with increasing masses and increasing lengths, and the 4 pendulum system with varying masses, lengths, and starting angles. In the former, it appears that the heaviest mass on the end of the longest piece of string is not obeying gravity. However, this particular instance of this effect is due to how the frame rate of the animation is restricted to match the motion of the fastest moving mass - in this case the smallest mass, mass 1. In reality, the entire system should be moving "in real time" at a much faster rate, and similarly for the 4 pendulum system with varying masses, lengths, and starting angles.

This could potentially be remedied by adapting the animation function to change the frame rate adaptivelly according to the timestep, however to do so in an implementation without skipping over the smaller timesteps (thereby rendering the work we've done in the adaptive step method useless) would require more advanced work in changing the playback frame rate of the video - which is beyond the scope of this project. Hence we shall concede that as long as the motion

appears physical aside from the "floaty" nature of the system with real time, the animations are sufficiently good.

Noting the motion of the different initial conditions, we clearly see very different behavior. Releasing the 3-pendulum system from the exact same starting conditions, but with varying pendulum lengths and/or varying masses results in highly different motion after even small amounts of time. In fact, we may also note that releasing a 3-pendulum system with identical masses and identical lengths results in motion which is notably different to an identical system released from an initial angle 10 degrees greater. This corroborates what we know about the n-pendulum system from real physics - it appears we have successfully simulated chaotic motion! In fact, it appears we have successfully demonstrated chaotic motion for a system of 3 pendulums with varying masses, lengths, and initial conditions!

It is important to acknowledge that the motion appears to be non physical upon inspection when viewing the 3 pendulum system of identical masses and varying lengths. The motion of such a system is prompted by the assumption that the strings holding the pendulum system together are mass-less and constantly taught. Here we see a shortcoming with the system; it does not portray systems of low mass and largely varying lengths accurately. Fortunately, this is a different kind of pendulum problem than is attempted to explore in this investigation; such an issue would be much more prominent when plotting the motion of a rigid pendulum system - where the system is composed of a series of rods with mass connected by friction-less pivot points at their ends.

We should also note that for pendulum systems where $n \neq 3$, we achieve the same desired results. Notably, the Interesting Results section illustrates that our program does work for different values of n, and produces interesting results. Namely, noting the system of 27 pendulums, we notice that the behavior of the pendulum begins to resemble a string - in fact this is what we would expect physically as well as a string can be approximated as an infinite series of increasingly small masses connected together by mass-less strings. However, the motion of this pendulum system begins to break down from this "string-like" approximation at later times, as it begins to behave like a system of a large number of pendulums (wiggling/bouncing behavior).

The animation for the double pendulum system should illustrate the strength of the program; simulating the motions of the pendulum for small angles. This is expected, as throughout the project we have assumed small angle approximation. The systems of 5 and 9 pendulums more so stand to illustrate that the program works for different values of n. The values 5 and 9 were chosen arbitrarily, and it is encouraged to run the program given any chosen number of parameters and initial conditions, with the constraints as indicated at the end of the results section.

4.1.2 Large initial velocity "Bounce-Back"

Note that when we start off with a sufficiently large initial velocity, the pendulum system completes a full rotation, but then instead of following through the motion of the rotation, the system

"bounces back" and continues the motion in a spring-like fashion instead of in a pendulum-like fashion. This is most notable in the 3-pendelum system with large nonzero initial velocity. Here we see one of the largest flaws with our entire system - small angle approximation. Although this was good for most approximations - in fact it is good even for relatively large angles, as long as specific parameters are satisfied - the small angle approximation leads to nonphysical motion in such edge cases.

Note that when we assumed small angle approximation, we assumed that the system would proceed as a harmonically oscillating system. This is easiest to see with the 1-pendulum system. The ODE for a 1-pendulum system solved with small angle approximation results in a harmonic oscillator; which is fine for a single pendulum system, but does not lend itself well to illustrating a chaotic system. Hence, when we attempt to solve a double pendulum system using small angle approximation, we are assuming that it follows some sort of harmonically oscillatory motion - not chaotic motion. Likewise for a 3-pendulum, and so on. The addition of more masses, and changing the lengths and masses themselves does make for more chaotic behavior as the program solves the system of equations, however the equations it is solving assumes underlying harmonic oscillatory behavior of each mass at some level - hence this motion is not "truly" chaotic - it is more apt to call it "pseudo-chaotic". In fact, the small angle approximation results in unreliable re-creations of motion of very specific pendulum systems with wildly varying lengths and masses - as can be seen in the 4 pendulum system with varying masses, but identical lengths and starting angles. However it is important to note that such nonphysical behavior is an exception and not the norm, as all other n-pendulum system examples provided illustrated mostly physical behavior.

4.2 Conservation of Energy

Note that energy was computed using small angle approximation, as that is the energy of our system. The "true" energy plots were also computed and are included in the same directories as the other energy plots and animations, however these simply serve to illustrate how they do not accurately represent the energy of the n-pendulum system simulated by this program, since we have been working with small angle approximation throughout. The plots have been labelled and titled accordingly.

As can be seen in the energy vs timestep plots accompanying each animation, there are 3 interesting properties:

1. Large "spikes" in Kinetic Energy at intervals of time
2. Periodic motion in Potential Energy
3. Almost-periodic motion in Total Energy

An example of the energy over time plot for the 2-pendulum system is given below to provide an example of the energy calculation in this report. All energy vs time and rolling average energy vs

time plots are included in the git, in the directories with their corresponding animations. They were not included in the report in order to prevent lengthening the report. Note that the final time for the Energy vs Time plots and Rolling Average Energy vs Time plots was changed (usually increased) from the final time provided in the animation in order to better illustrate the trends in energy vs time for each system. The animations were not produced to these (usually much larger) final times in the interest of time and brevity.

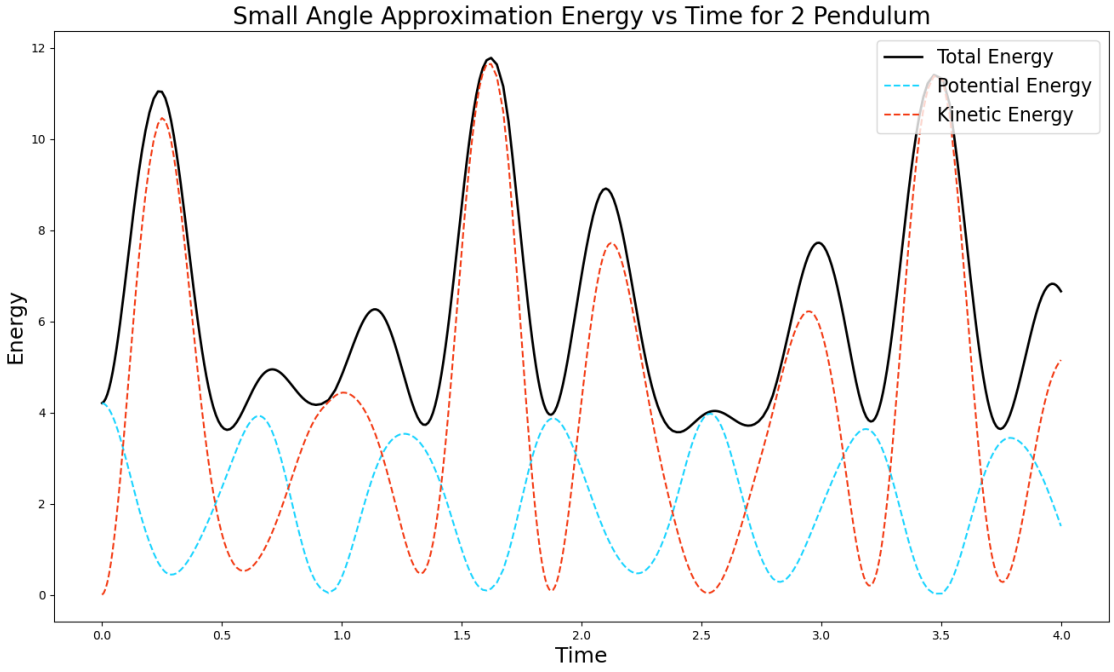

Figure 4: Plot of Energy vs Time for Double Pendulum system with identical masses, different lengths, and initial angles of 10°&20°, respectively

Note that in the computation of the adaptive stepsize for the Verlet method, energy in the angular velocity was not taken into account, but energy in the position was taken into account. This explains why the potential energy is periodic in nature; the potential energy is determined using only the masses, lengths, and angles over time; therefore if we expect potential energy to be accurately determined, we expect potential energy to change in a periodic fashion as illustrated by the Verlet method - and in fact, this is what we observe. However, the kinetic energy of the system is significantly large at certain points for certain plots. This is likely due to the fact that error in the angular velocity is not taken into account when choosing an adaptive step size; the adaptive step size is chosen to minimize the error in angle, and not in angular velocity, leading to some points in time where the angular velocity is far greater (or far lower) than what it should

be within our desired accuracy - resulting in occasional spikes in angular velocity over time. Since kinetic energy is calculated from angular velocity, angles, and masses, this directly affects the kinetic energy and subsequently we observe this spiking behavior in the kinetic energy. The error in the angular velocity also results in angular velocity that is significantly greater in magnitude than the magnitude of potential energy at times.

The culmination of these two errors lead to the total energy being dominated by the kinetic energy for most systems, and results in the total energy almost taking on the form of the kinetic energy entirely. It is difficult to see the pattern in the total energy over time due to the spikes in kinetic energy, hence we introduce a rolling average to observe the trend in the total energy over time[3]. In fact, when we do so we see that overall the total energy is following somewhat oscillatory motion over time (see Figure 5) - which is exactly what we expect for the Verlet method.

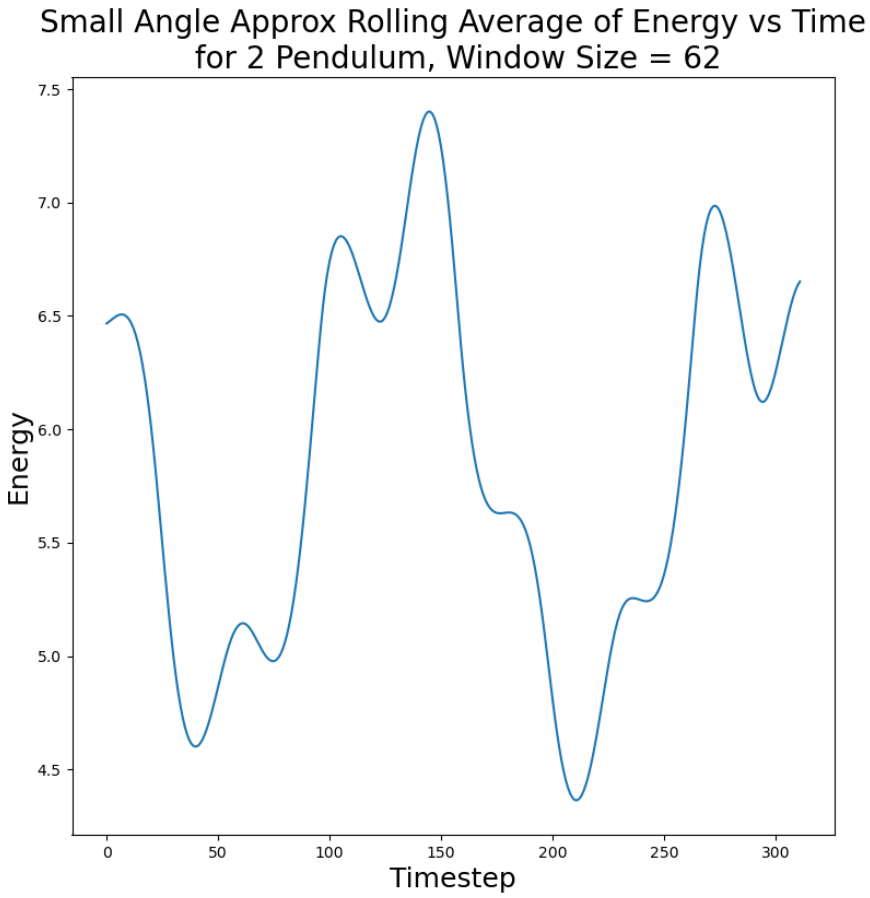

Figure 5: Plot of Rolling Average of Energy vs Time for Double Pendulum system with identical masses, different lengths, and initial angles of 10°&20°, respectively

Note that, for odd-mass systems the energy over time does not seem as "cleanly oscillatory" as it does for even-mass systems. Here we see another issue caused by small-angle approximation; since we are essentially predicting the motion of "chaotic oscillator" systems linked together, and noting that the total energy is calculated as a sum of the energies of each mass, we have that there is some interference between the oscillations of the energy over time between each mass. Therefore, if we Fourier-transform the total energy over time plot and obtain the n-leading sin waves that make up such a plot, each of those n-sin waves would be the total energy over time of one of the masses in the system. For odd-mass systems - and in fact, for systems with wildly different masses and lengths - this results in periodic total energy plots which do not have a clear periodic structure. For even-mass systems - actually, for so-called "neat" systems with similar masses and lengths and relatively small initial angles - such interference results in plots which have more clear periodic structure.

4.3 Potential Improvements & Future Work

There are a few key points which could be improved upon to produce a more robust program. Primarily, small angle approximation should not be used in the computation. However, there is a very large caveat which comes with this suggestion for improvement: there is no exact closed form solution for an n-pendulum system of differential equations. There does exist a family of solutions which can be solved for, however if one aims to solve such a system using matrices as we have done in this project one would need to account for a variety of issues that comes with representing this system in matrix form. Computing the sin and cos of specific angles at specific locations within the matrices would need to be done carefully, division by zero errors would have to be taken into account for specific angle values, etc. These processes would slow down the program considerably and be very complex, and hence were not implemented in the interest of time, and in the interest of illustrating what a small-angle approximated system of n-pendulums could accomplish.

A relatively easy improvement to the system to implement would be including error in angular velocity during the adaptive timestep. In fact, this was actually attempted for this project, however it was abandoned for the final version as the implementation resulted in the timestep decreasing to order magnitude $\times 10^{-17}$ nearly every time; resulting in very many, very long computations to save very little results with barely any motion in time.

Finally, a catch for angles beyond the range of $(-\pi, \pi)$ may be implemented after the initial conditions have been defined. For example, one may take $\theta_i' = \pi - (\theta_i \% (\pi))$, where θ_i is the angle implemented outside the range of $(-\pi, \pi)$, and θ_i' is the corresponding "true" value of the angle the i^{th} string makes with the y-axis. This was not implemented in the final version as such an implementation does not work with all angles for a system that is solved using small angle approximation.

5 Conclusion

We find that the Verlet method with an adaptive timestep is useful for solving the equations of motion of a system of differential equations for an n-pendulum system in time, assuming small angle oscillations. However, we note that such a system is limited by the small angle approximation in it's accuracy. Therefore, there are distinct limitations to the kinds of n pendulums that can be reliably illustrated. There is error in the magnitude of angular velocity at certain timesteps (and hence in the kinetic energy calculation), however the overall trend of oscillating energy over time is expected and found in the rolling average of total energy over time using this method. Overall, this method for solving an n-pendulum system is sufficient for small-angle pendulum systems of most ranges of masses and lengths, however it struggles at accurately producing motion of some large-angle and large-initial-velocity pendulum systems.

6 Citations & References

1. Rubenzahl, Ryan, and S. G. Rajeev. "Small Oscillations of the n-Pendulum and the 'Hanging Rope' Limit as n Approaches Infinity." *Ryan Rubenzahl, University of Rochester Physics and Astronomy, University of Rochester*, 2 Jan. 2017, http://www.pas.rochester.edu/ rrubenza-/projects/RR_PHY235W_TermPaper.pdf.

2. Newman, Mark. *Computational Physics.* Mark Newman, 2012.

3. Chou, Y.-L. "17.9 Method of Moving Averages." *Statistical Analysis: With Business and Economic Applications*, Holt, Rinehart and Winston, Jamaica, New York, 1969, pp. 556–558.

4. "How to Save Matplotlib Animations: The Ultimate Guide." *HolyPython*, Holy Python, 1 Oct. 2021, https://holypython.com/how-to-save-matplotlib-animations-the-ultimate-guide/.