Final Project in Programming for Physicists 2021-2022

Springs Oscillators chain

November 28, 2021

Abstract

We will simulate a one-dimensional open chain of point-like particles, connected to each other with springs (see fig 1), and plot graphs of the oscillatory motion and energy in time. We'll also examine the angular frequency extracted from the simulation and compare it to it's analytic solution. Finally, using this relatively simple model we will approximate the speed of sound in matter, with given properties.

1 Introduction

1.1 Multi-particles systems

When studying matter with one or more dimensions (not a point-like particle) and various properties, there are many ways in which we can describe the physical system. One way is to examine it as a continuous body, say the dynamics of a gas or liquid, and study the relations between its various properties, like temperature, heat capacity, volume, density etc. Another way is describing matter using its building blocks - a big system with many particles/sub units [1]. Solving the dynamics of such multi-particle systems analytically (using a pen and a piece of paper) is not an easy task, and often not doable. But using some simple assumptions, it can be a lot easier to simulate various aspects using a computer. In this simulation we'll look at a one-dimensional matter/chain, which is made out of many point particles, interacting with one another with springs. We will assume that the motion of the particles is horizontal only (parallel to the chain). If coding such complex system might seem intimidating, don't worry, we will break it into a few steps.

[1] This is the basic idea of statistical physics (which you'll learn more about in 3rd year and at advanced degrees).

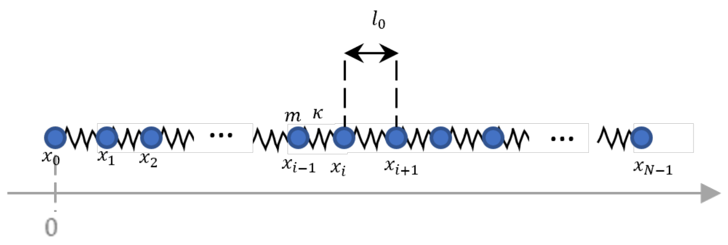

Figure 1: Illustration of a chain of N oscillators connected with $N-1$ springs.

1.2 Oscillations

Oscillatory motion can be described using periodic functions. The most basic/simple oscillation is of a harmonic oscillator, which you will encounter in classical mechanics course. In this subsection we'll only go over the important properties of oscillations, but you can find a more explicit description in the attached "`Oscillators_theory`" file.
An example for a simple harmonic oscillator is a mass connected to a spring which applies a force $F = -kx$ on the mass. At $x = 0$ there are no forces on the mass, and we call this the stability or equilibrium point. When the mass is displaced from this equilibrium point, it will begin a periodic motion. The displacement as function of time of a harmonic oscillator relative to its stability point is a sinusoidal function (see fig. 2).

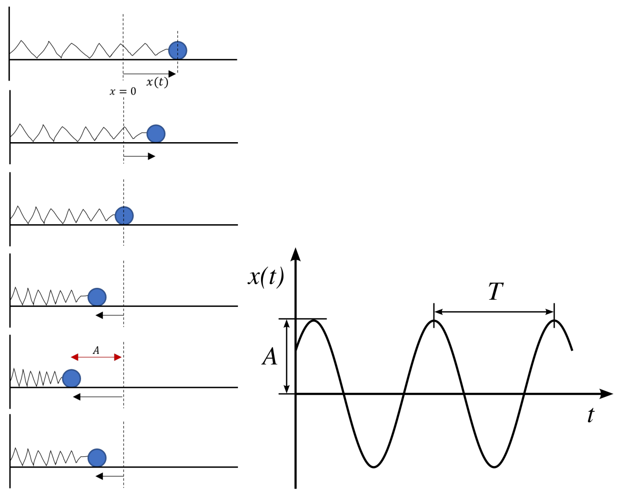

Figure 2: The displacement of a harmonic oscillator relative to its stability point as a function of time is described by a sinusoidal function.

We can characterize oscillations by these properties (see fig 2):

- Amplitude A - the maxima of the oscillation. i.e, the furthest displacement from stability point.
- Period T - the time for a single oscillation. In other words, the minimal T such that $x(t) = x(t+T)$ **and** $\dot{x}(t) = \dot{x}(t+T)$ (where we note $\dot{x} \equiv \frac{dx}{dt}$).
- Frequency f - number of oscillations per unit of time. It is the inverse of the period:
$$f = \frac{1}{T} \tag{1.1}$$
For example, say we measure 120 oscillations within 60 seconds, then the frequency is $f = \frac{120}{60 \text{ sec}} = \frac{1}{2}\frac{1}{\text{sec}} = \frac{1}{2}$ Hz (where 1 Hz $= \frac{1}{\text{sec}}$ are Hertz units), and the time period $T = 2$ sec.
- Angular frequency ω is simply:
$$\omega = 2\pi f = \frac{2\pi}{T} \tag{1.2}$$
- Phase ϕ - a parameter of time shift, which we'll only mention. Mathematically, you can think of ϕ as moving the whole function left or right by some amount. In the physical sense, it is determined by the initial state of the system.

Examining a periodic function, the three parameters you'll actually see are A, ω and ϕ. For example, the displacement function of a single harmonic oscillator is:

$$x(t) = A\cos(\omega t + \phi) \tag{1.3}$$

where ω is determined by the system's parameters/properties (for example mass, spring constant), and A and ϕ are determined by the initial state of the system (initial displacement, for example).

Oscillations in general can be described as periodic functions, and not necessarily as a simple cosine or sine function like in eq. 1.3. A periodic function repeats its values at regular intervals (a period), and can be represented by some linear combination of sinusoidal functions (cosine and/or sine). See some examples at fig 3.

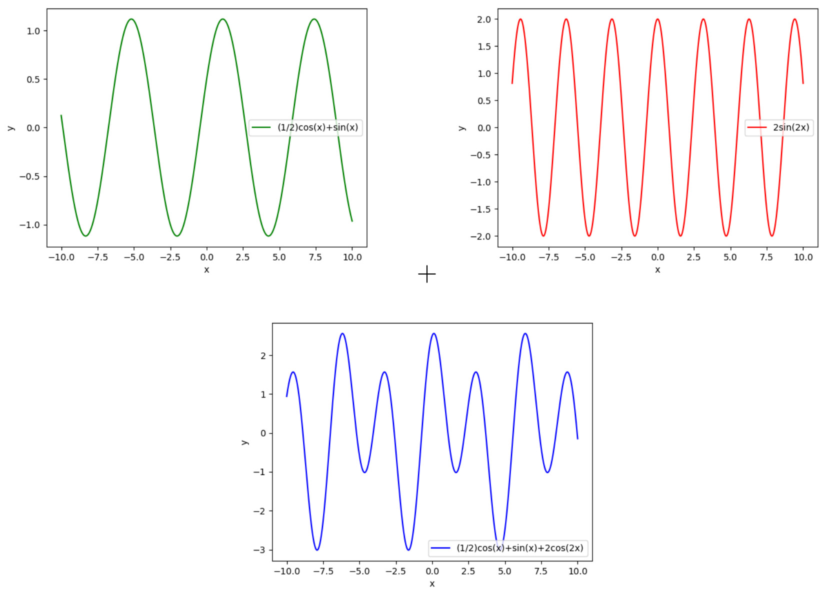

Figure 3: Three example graphs of periodic functions. The Blue function is a linear combination of the green and the red functions.

In our program we'll also see such cases where the oscillations are more complex then a simple cosine or sine function. Don't worry, we're only going to plot a graph of such cases :)

2 Program's Layout and Requirements

The program's goal is to simulate a chain of N oscillators where all oscillators have the same mass m and all the springs have the same spring constant κ and same length of rest l_0. N, κ, m and l_0 will be parameters for the program. The oscillators can only move in the axis of the chain (See fig 4).

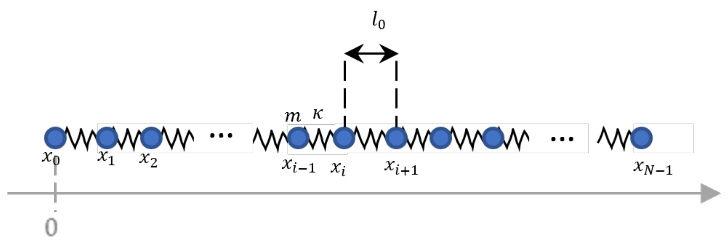

Figure 4: Illustration of a chain of N oscillators connected with $N-1$ springs.

2.1 Program's overview

The program should be able to run any number of oscillators, according to the initial parameters given to it. These parameters are described in the next section, and let us define the properties of the simulation we wish to run.

The chain's ends can either be both 'closed', both 'opened' or one 'opened' and the other is 'closed'. A 'closed' end means it can't move, and 'opened' end means it can. So for example if the chain is 'closed' on both ends, that means the first particle can't move, and also the last particle can't move.

In order to validate the simulation We'll analyze these following cases:

- One oscillator moving between two 'closed' ends.
- Two oscillators moving between two 'closed' ends.
- A general case of an $N > 100$ oscillators, where both ends are 'opened'.

You can read about the theory of these cases in the attached file "`Oscillators and waves theory.pdf`".

For any number of oscillators we'll plot a graph of the kinematic variables – $\Delta x(t)$, $v(t)$ and $a(t)$ – of only the middle oscillator, and a graph of the energies $E_{tot}(t)$, $E_k(t)$, $E_p(t)$ of the whole system. Further description later on.
For the first two cases (one 'closed' oscillator and two 'closed' oscillators) we can also use the exact value of the angular frequency ω, and compare it to a numerically approximated value ω_{approx}, which we'll calculate from the simulation. Further description will be later on.
Notice that for two oscillators there are two possible angular frequencies, which depend on their initial state (see attached file "`Oscillators and waves theory.pdf`").
For the last case (an open chain of $N > 100$ oscillators), we will approximate the velocity of sound wave numerically $C_{s,approx}$, and compare it to the exact value C_s (see attached file "`Oscillators and waves theory.pdf`"):

$$C_s = \sqrt{\frac{\kappa}{m}} l_0 \tag{2.1}$$

where m is a single oscillator mass, κ is a single spring's constant and l_0 is its rest length.

Let's break down the program into these concrete goals:

- Calculate the arrays $x(t)$, $\Delta x(t)$, $v(t)$ and $a(t)$ for all the oscillators (position, displacements, velocities, and accelerations at each point in time respectively) and a list/array of time t.
- Calculate the arrays $E_{tot}(t)$, $E_k(t)$, $E_p(t)$ – Total energy, kinetic energy and potential energy of the whole system at each point in time.

- Calculate numerically the angular frequency ω_{approx} (only of the oscillator in the middle) from the results of the simulation and compare it to the analytical solution ω, in the case of one and two confined oscillators.
- Calculate numerically the velocity of propagation, $C_{s,approx}$, of a wave/pulse from the results of the simulation and compare it to the analytical solution C_s, in the case of one and two confined oscillators.
- Plot 3 graphs of $\Delta x(t)$, $v(t)$ and $a(t)$.
- Plot 3 graphs of $E_{tot}(t)$, $E_k(t)$, $E_p(t)$.
- There are extra analysis questions you can answer for bonus in grade, regarding the resulting energies. See 'Bonus' subsection at the end of this section.

The last goal will be understood better after you read the next section ('The simulation's layout').

- We can analyze if the time interval Δt we use to advance the simulation is small enough. We can do that by examining the frequency's ω relative error (see eq 2.2) as a function of the time interval Δt respectively.
  - Calculate numerically couple of frequencies ω_{approx}, using a loop over a list of Δt. Meaning, for each value of Δt run a simulation and approximate the frequency respectively. So after the loop we'll have a list/array of frequencies and a list of the different time intervals respectively.
    For example `dts = [0.01, 0.02, 0.03]` and `w_approx = [2.02, 2.08, 2.12]` (The numbers are arbitrary. Just an example.)
  - Plot a graph of the function $\frac{\Delta\omega}{\omega}(\Delta t)$, where $\Delta\omega = |\omega - \omega_{approx}|$.

Reminder for relative error formula:

$$\frac{\Delta X}{X} = \frac{|X - X_{approx}|}{X} \tag{2.2}$$

for some general parameter X.

2.2 The simulation's layout

In this subsection will go over the calculation methods of the program.
First, consider **one oscillator**. Here is a Summarization of the analytic solution[2]. We can find the equation of motion of the oscillator using Newton's second law and Hooke's law:

$$\ddot{\tilde{x}}(t) = -\omega^2 \tilde{x}(t) \tag{2.3}$$

[2]See full description in the attached file "`Oscillators and waves theory.pdf`"

where $\tilde{x}(t) \equiv \Delta x(t) = x(t) - l_0$, and

$$\omega \equiv \sqrt{\frac{2\kappa}{m}} \tag{2.4}$$

The analytic solutions for eq 2.3 is:

$$\Delta x(t) = \Delta x_0 \cos(\omega t) \tag{2.5}$$

But what can we do when we can't easily solve the equation of motion? Instead, using a computer, we can advance eq 2.3 in time by small enough time steps Δt (which is chosen with caution). For a small enough Δt we can say that a and v are approximately constant. Then, if we know $a(t)$, $v(t)$, $x(t)$ at a specific moment t, we can calculate them a short time Δt later with simple equations of motion:

$$\begin{aligned} x(t+\Delta t) &= x(t) + v(t)\Delta t \\ v(t+\Delta t) &= v(t) + a(t)\Delta t \end{aligned} \tag{2.6}$$

Then calculate the new $a(t+\Delta t)$ (which depends on $x(t+\Delta t)$, and repeat the process for the next time interval.

Why do we use small time intervals? The process described above is of course only an approximation, but if we take small enough time intervals, it becomes a good approximation. Although time is continuous we can't make an array of points in time with infinitesimal intervals dt. Instead all the arrays (t, $a(t)$, $v(t)$, $x(t)$ and $\Delta x(t)$) are calculated using the small time interval interval Δt between each point in time, assumed to be small enough as to consider it infinitesimal $\Delta t \approx dt$.
There are couple of methods that realize this idea, and the one we'll use is called *Verlet leapfrog method*[3] - In this method, the position and acceleration are computed at time t, while the velocities are computed at time $t + \frac{\Delta t}{2}$. That is, first we have to advance the initial velocities by the time $\frac{\Delta t}{2}$. So we start with some initial state of the system and advance the whole system with small steps in time.

We can solve the equation of motion (eq 2.3) using the following:

- In each step compute the acceleration using newton's second law

$$a_i = \frac{1}{m}\sum F(x_i) \tag{2.7}$$

  where $\Sigma F(x_i)$ is the sum over all forces on the particle and a_i is the acceleration at step i.

- Then, update the position and velocity:

[3]See The Feynman Lectures on Physics (first volume, chapter 9), or Wikipedia for more information.

$$
\begin{aligned}
v_{\frac{1}{2}} &= v_0 + a_0 \frac{\Delta t}{2} \\
v_{i+\frac{1}{2}} &= v_{i-\frac{1}{2}} + a_i \Delta t \\
x_{i+1} &= x_i + v_{i+\frac{1}{2}} \Delta t
\end{aligned}
\tag{2.8}
$$

where x_i is position at step i, $v_{i+\frac{1}{2}}$ is the velocity at step $i+\frac{1}{2}$, and Δt is the size of each time step. The first velocity $v_{\frac{1}{2}}$ is calculated with half a step $\frac{\Delta t}{2}$, while each of the following velocities is calculated with full time interval Δt (See fig 5).

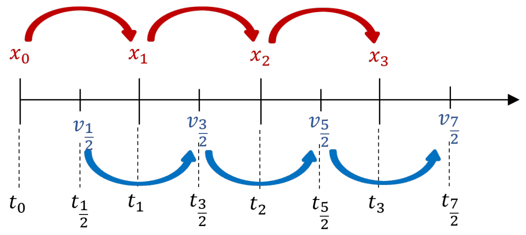

Figure 5: Illustration of the time advancement according to the "Verlet leapfrog method".

We also want to calculate the energies as function of time - at each step i. We know the total energy, kinetic energy, and potential energy of a single oscillator at step i (at a given time/frame) respectively are:

$$
\begin{aligned}
E_{tot_i} &= E_{k_i} + E_{p_i} \\
E_{k_i} &= \frac{1}{2} m v_i^2 \\
E_{p_i} &= \frac{1}{2} k \Delta x_i^2
\end{aligned}
\tag{2.9}
$$

Notice that in order to calculate E_{k_i} we have to use the velocities at a whole time steps v_i, which we can approximate using time regression:

$$
v_i = v_{i+\frac{1}{2}} - a_i \frac{\Delta t}{2} \tag{2.10}
$$

Meaning, only after $v_{i+\frac{1}{2}}$ is already calculated, we use it to go back with half a step.

How can we approximate the angular frequency ω_{approx} ? After the simulation is done and we have all the arrays we need, we can then numerically calculate ω_{approx} if we find the average period T_{avg} of the oscillatory motion, by:

$$\omega_{approx} = \frac{2\pi}{T_{avg}} \tag{2.11}$$

How can we approximate the average period T_{avg} ? First, a single period can be computed by the distance between two nearby maxima[4] (see fig 2), which is simply the time difference between these two points:

$$T_{approx} = |t_{max1} - t_{max2}| \tag{2.12}$$

Second, assuming we simulated more than one period of oscillation, we can compute couple of T's in a list/array, and average all of them. So for example, if we simulated 3 oscillations overall (i.e. you'll see 3 peaks when you plot $\Delta x(t)$), we'll have a list/array of 3 periods, which we can average.

And **the analytic solution to** ω for one, two and more oscillators can found in the attached file "`Oscillators and waves theory.pdf`". You are not required to calculate ω for more then two oscillators between two walls though.

Now, consider **two or more oscillators**. Each variable becomes a list/array. So now for N oscillators eqs 2.7 and 2.8 become:

$$\begin{aligned} a_{i,n} &= \frac{1}{m}\sum F(x_{i,n-1}, x_{i,n}, x_{i,n+1}) \\ v_{\frac{1}{2},n} &= v_{0,n} + a_{0,n}\frac{\Delta t}{2} \\ v_{i+\frac{1}{2},n} &= v_{i-\frac{1}{2},n} + a_{i,n}\Delta t \\ x_{i+1,n} &= x_{i,n} + v_{i+\frac{1}{2},n}\Delta t \end{aligned} \tag{2.13}$$

for the $n-th$ oscillator. If you find the indices confusing, it simply means we have 2-dimensional array, where one dimension represents the evolution in time, and the other represents the number of oscillator along the chain.
First finish writing a simulation for one oscillator with eq 2.8, and then the transition to arrays will be much easier. It is highly recommended to work with the *numpy* library.

Notice: the forces applied on the oscillators depend on the distance between the oscillators, so think carefully both when and how to calculate the forces.

[4]In general we can take any two nearest repeating points in the motion. That is, where both $x(t) = x(t+T)$ <u>and</u> $\dot{x}(t) = \dot{x}(t+T)$. But it will be much easier to identify two close local maxima.

The energies of all the oscillators at step i (at a given time/frame) are:

$$\begin{aligned} E_{tot_i} &= E_{k_i} + E_{p_i} \\ E_{k_i} &= \sum_{n=0}^{N-1} \frac{1}{2} m v_{i,n}^2 \\ E_{p_i} &= \sum_{n=0}^{N-1} \frac{1}{2} k \Delta x_{i,n}^2 \end{aligned} \tag{2.14}$$

The approximation of the angular frequency ω_{approx} in the case of two oscillators is no difference of the single oscillator case, Also using eqs 2.11, 2.12. But, we can calculate ω_{approx} using this method, <u>only</u> for a specific initial state of the system. These specific conditions will be given as an input file, as will be described in the next subsection.

In all of the following subsections we'll go through the requirements and technicalities of the program.

2.3 Program's input (10 points)

- The program will read all the parameters from a supplied input file `input_parameters.json`. This file will contain the parameters of the run:
  - Number of frames - `frames_num`
  - A small time interval (used to advance to simulation) - `dt`
  - Spring's length of rest - `l_rest`
  - Spring's force constant - `k`
  - The mass of a particle - `mass`
  - The angular frequency ω (for specific initial states) - `omega`
    Its value might be `None/null` in
  - Number of oscillators - `osc_num`
  - Two boolean parameter - `first_open` and `last_open`.
    These variables choose between open/close/semi-open chain. In other words, they will let us decide if the first and/or last particles are statics (can't move) or not, respectively. For example,
    `first_open=True`
    `last_open=False`
    means the first particle can move, but the last particle can't move.
- Consider that `dt` are a list of floats, and `frames_num` is a list of integers. In each run of the simulation use one of each list together, so we'll be able to run the simulation for each time interval, and later plot $\Delta\omega(\Delta t)$. In other words, if for example `dt = [0.1, 0.2, 0.3]` and `frames_num`

= [100, 200, 300], then we will run the simulation first with 0.1 time interval and 100 frames, second we will run the simulation with 0.2 time interval and 200 frames and so on.

- The program will read the initial state of the particles from the supplied input file `input_initState.json`:
  - Initial coordinates - `x_init`
  - Initial velocities - `v_init`
- In the moodle there are couple of such files for different cases of the simulation, and the program should be able to run all of them. You can assume they are all written in the same format.
- You'll see the files have numbers, that's because they come in couples:
  - `input_parameters1.json` and `input_initState1.json`
  - `input_parameters2.json` and `input_initState2.json`
  - `input_parameters3.json` and `input_initState3.json`
  - `input_parameters4.json` and `input_initState4.json`
  - `input_parameters5.json` and `input_initState5.json`

  Each files couple should be used as inputs together.
- All parameters files are in `.json` format for your convenience, so you can use the *json* library to read the files into your program with only few lines of code. But you don't have to use *json*, and you can read the files as you wish.

2.4 Coordinates compute (45 points)

- The program will calculate an array of all the particles coordinates at each given frame/time.
- Each array will be written to a file `framei.txt` (where "i" is the number of frame). A frame describes the system state at a given time. For example, if the number of frames is `frames_num` = 1000 and the number of oscillators is `osc_num` = 5, then there will be 1000 arrays of 5 coordinates, and there will be 1000 files named `frame1.txt`, `frame2.txt` ... `frame1000.txt`. An example of a file is attached. Please make sure you use the exact same format - the separation between numbers is always '\t'.
- **Alternatively** you can write one file `frames.txt`, where there `frames_num` lines and `osc_num` columns.

2.5 Program's functions (25 points)

- The program will have a function `calc_E(dx, v)` that calculates and returns the following arrays/lists:
  - Array of the total kinetic energy at each frame/time - $E_k(t)$
  - Array of the total potential energy at each frame/time - $E_p(t)$
  - Array of the total overall energy at each frame/time - $E_{tot}(t)$
- The program will have a function that calculates and returns the average/mean angular frequency of an oscillator `mean_frequency(dx, t, dt)`.
  - The function's inputs are a list/array of the oscillator's displacements relative to their stability point `dx` at each time/frame, a list/array of all discrete points in time `t` and the time interval `dt`.
  - The function's output is the mean angular frequency of the oscillator's periodic movement.
  - In order to find the peaks/local maxima of the oscillator's displacement function, you can import the function `find_peaks()` from `scipy.signal`, and use it to find the specific indices of the array:
    `indices, x_peaks = find_peaks(x, height=x[0])`
  - If the average of T is 0, the program should print a message like: "`T_avg = 0 for dt = ...`", where instead of "`...`" write the value of the given `dt`, to point where the problem is.
- The program will also write $\omega_{analytic}$, ω_{approx} and their relative error (see eq 2.2), in this order, for each simulation in a file "`frequencies.txt`". That is, for each `dt`, and `frames_num` given there will be written a line of the 3 mentioned results.
- The program will have a function `calc_wave_vel(v, t)` that calculates the velocity of wave in the chain **only** in cases where `osc_num>100`
  - The function's inputs are a list/array of all velocities at each time/frame of **only** the last oscillator in the chain `v`, a list/array of all discrete points in time `t`.
  - The function will calculate the time difference between the beginning (assume $t_0 = 0$) and the moment the last oscillator began to move. For a very specific initial condition we'll assume this is the time that wave went through the material and reached the end.
  - The function we'll approximate (naively) and return the wave velocity:

$$C_{s,approx} \approx \frac{L}{\Delta t} = \frac{(N-1)l_0}{\Delta t} \tag{2.15}$$

where Δt is the time difference calculated and L is the length of the chain at complete rest. Here N is number of oscillators (`osc_num`) and l_0 is the length of rest of each spring (`l_rest`).

- The program will also write C_s, $C_{s,approx}$ and their relative error $\frac{\Delta C_s}{C_s}$ (See eqs 31 in the "`Oscillators and waves theory.pdf`", and 2.15 and 2.2 respectively), in this order, for each simulation in a file "`v_wave.txt`". An example of a file is attached. Please make sure you use the exact same format - the separation between numbers is always '\t'.

- All the functions mentioned above must be in your program, but you can write more functions if you wish.

2.6 Program's plots (20 points)

- Considering **only the middle oscillator** in the chain, the program will plot the three functions of $\Delta x(t), v(t), a(t)$ (using the arrays calculated in the simulation) and save the plot to a file "`kinematics.png`". You can use the `subplots()`, `savefig()` functions from the *matplotlib.pyplot* library.

- The program will plot the three functions of $E_{tot}(t)$, $E_k(t)$, $E_p(t)$ and save the plot to a file "`Energy.png`".

- The program will read all $\frac{\Delta\omega}{\omega}$ (relative error) from the "`frequencies.txt`" file, and plot a function of $\frac{\Delta\omega}{\omega}(\Delta t)$. The plot will be saved to a file "`freq_vs_dt.png`".

2.7 Bonus (10 points)

Write an answer, as a comment in the end of you code, to the following questions:

- Describe what result would we expect to have in the energies graph?

- Does the result satisfy our expectations? If not, why?

2.8 Animation - for fun

In the moodle you can find another python file called "`Animation.py`", and by using the results of your simulation you can use it to visualize the motion of the oscillators. "`Animation.py`" has a function that reads all of the coordinate from your "`frame_i.txt`" files, and animate them. In order to run it you'll need:

- Download "`Animation.py`" to the same folder as your "`frame_i.txt`" files are (or your single "`frames.txt`" file is).

- Install the package called *pygame*.

- Make sure the format of your files are as explained in the program's requirements section.

- Run `Animation.py`. (You can also import it as a module to your own code, and use the function *play()*)

3 General Instructions

3.1 Work Instructions

- Read this project description thoroughly and make sure you understand all of its requirements before you start writing your code.
- Try to break it into relatively simple tasks, and handle each task separately. It is easier to find errors in a small section of the code, dealing with a limited task.
- Use meaningful variable names and meaningful function names to enhance the legibility of the program.
- Document the program using comments. For example, it is recommended to explain briefly at the beginning of each function its purpose and the meaning of its main variables. Please note that points will be reduced due to insufficient documentation.
- Testing your program under various parameters and initial conditions is essential in order to obtain a valid, working project.

3.2 Guidance by advisors

- You are allowed to ask any question regarding study material from the course.
- You are not allowed to ask about compilation errors.
- None of the advisors are allowed to see your program code.
- Namely, you cannot show us the project until it is submitted. If one of your sub-procedures does not work properly (not compilation errors), you may phrase a question so that we can advise you how to solve the problem: you are to construct a code called temp.py, containing a simple program which consists of the sub-function you are trying to run, a main code that calls it, and supply an input file if it requires one. Send the file to one of us and explain the problem.

3.3 Administrative Details

- The project is personal. It is not allowed to work in pairs or groups.
- The final version of the programs should be submitted via Moodle.
- Submission date: 18.2.22 20:00

- Make sure that the final version of your project can be compiled and run with python 3.
- You will be orally examined to verify the validity of the program and your understanding of the project. The exam will take place in a time and location that will be announced soon.

Good luck!