Department of Physics, Chalmers, Göteborg
Göran Wahnström 2021-11-08

E1 Coupled harmonic oscillators

Oscillatory motion is common in physics. Here we will consider coupled harmonic oscillators. Fourier transformation of the time-dependence can be used to reveal the vibrational character of the motion and normal modes provide the conceptual framework for understanding the oscillatory motion.

You will be asked to solve a few computational problems. The first four is about the discrete Fourier transform and you will make use of the fast Fourier transform (FFT) technique, a computationally very efficient way of performing discrete Fourier transformations. You will then study the motion of coupled harmonic oscillators, by solving Newton's equation of motion using the velocity Verlet algorithm. You will determine the vibrational frequencies for a triatomic linear molecule, carbon dioxide, and finally you are asked to determine analytically the normal modes for the triatomic linear molecule.

You should solve the problems using the `C` language. For your convenience we provide you with a set of `C` programs as well as a plotting program in `Python`. In total, you can get 4 points in this exercise.

1 Coupled harmonic oscillators

Consider three harmonically coupled particles. The mass of each particle is m and the springs are identical with a spring constant κ. We denote the displacements from the equilibrium positions with q_i, i=1,2,3 and v_i for the corresponding velocities. The system is anchored to the walls and we introduce the coordinates q_0 and q_4 with $q_0 = q_4 = 0$.

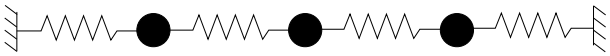

Figure 1: System of three coupled harmonic oscillators with fixed boundary conditions.

The Hamiltonian for the system is given by

$$\mathcal{H} = \sum_{i=1}^{3} \frac{m v_i^2}{2} + \sum_{i=0}^{3} \frac{\kappa}{2}(q_{i+1} - q_i)^2$$

and the corresponding equation of motion can be written as

$$m\frac{d^2}{dt^2}q_i(t) = \kappa\left[q_{i+1}(t) - 2q_i(t) + q_{i-1}(t)\right], \quad i = 1, 2, 3$$

This is a set of three coupled second order ordinary differential equations. These can be solved using a stepping procedure in time. One popular method is the velocity Verlet algorithm

$$q_i(t+\Delta t) = q_i(t) + v_i(t)\Delta t + \frac{1}{2}a_i(t)\Delta t^2$$
$$v_i(t+\Delta t) = v_i(t) + \frac{1}{2}[a_i(t) + a_i(t+\Delta t)]\Delta t$$

where Δt is the timestep and $a_i = d^2q_i/dt^2$ is the acceleration for particle i. In this case we have

$$a_i(t) = \frac{\kappa}{m}\ [q_{i+1}(t) - 2q_i(t) + q_{i-1}(t)]$$

One step in the velocity Verlet algorithm can be coded in an efficient way as

$$v_i(t+\Delta t/2) = v_i(t) + \frac{1}{2}a_i(t)\Delta t$$
$$q_i(t+\Delta t) = q_i(t) + v_i(t+\Delta t/2)\Delta t$$
`calculate new accelerations/forces`
$$v_i(t+\Delta t) = v_i(t+\Delta t/2) + \frac{1}{2}a_i(t+\Delta t)\Delta t$$

which only requires the storage of positions and velocities at one time point.

Fourier transformation

It is very useful to illustrate the vibrational motion in Fourier space. The transformation is done using the discrete Fourier transform. Some basic properties of the discrete Fourier transform can be found in Lecture notes "Molecular dynamics", appendix F.

To get used to this we first consider a few simple cases. Consider the time-dependent signal

$$h(t) = a\cos(2\pi f t + \phi)$$

1. The program E1code1 generates the signal $h(t)$ using $a = 1, \phi = 0, f = 2, \Delta t = 0.1$ and $N = 250$. Plot your result using Python. A help routine E1code2 is provided. Switch to $f = 1$, run the program and plot the new result for $h(t)$. Switch to $\phi = \pi/2$ and plot the result for $h(t)$. Which function do you obtain? (0p)

2. Consider next the Fourier transform of $h(t)$. E1code3 uses a a wrapper around gsl's fft routine provided through the file fft.c. Use E1code3 to determine the discrete Fourier transform of $h(t)$ using a=1, Φ=0, f=2, Δt=0.1, and N=250 and plot the corresponding power spectrum P_n for $n = 0, 1, \ldots, (N-1)$. Run the program and plot the result for P_n. Do you understand the spectrum? Compare with Fig. 2. (0p)

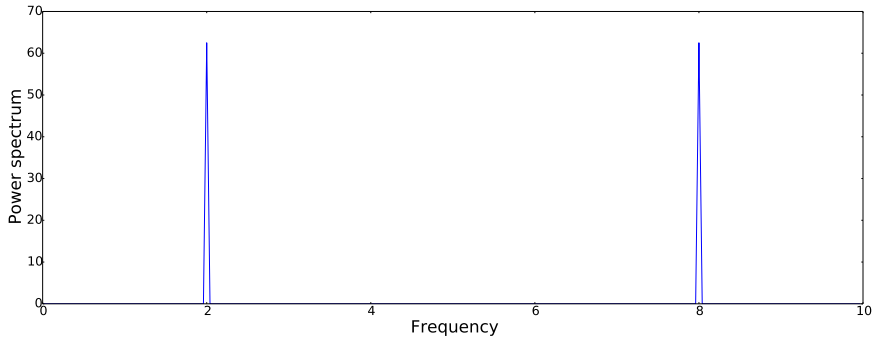

Figure 2: Power spectrum.

3. It is more convenient to plot P_n for $n = N/2, \ldots, (N-1), 0, \ldots, (N/2-1)$. Why? Modify E1code3 to perform this shift. A routine for this is provided in fft.c. Run the program and plot the result for P_n. Compare with Fig. 3. Do you understand the frequency scale on the x-axis? What are the minimum and maximum frequencies? Change to $N = 253$ and $N = 255$, respectively, and plot in both cases $h(t)$ and P_n. Can you comment on your results? (0p)

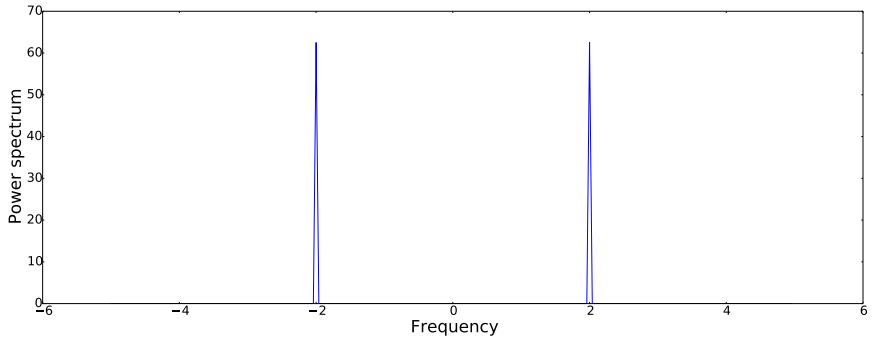

Figure 3: Power spectrum, shifted.

4. Consider now the signal

$$h(t) = a_1 \cos(2\pi f_1 t + \Phi_1) + a_2 \cos(2\pi f_2 t + \Phi_2)$$

with $a_1 = a_2 = 1$, $\Phi_1 = \Phi_2 = 0$, $f_1 = 2$, $f_2 = 6$, and $\Delta t = 0.1$, $N = 250$. What is the Nyqvist frequency in this case? Fourier transform $h(t)$ and plot P_n for $n = N/2, \ldots, (N-1), 0, \ldots, (N/2-1)$. Do you understand the result? What is the problem with the present signal

$h(t)$. Can you modify the sampling in time domain so that you get a more correct representation of the power spectrum? (1p)

Coupled oscillators

Next consider the the motion of three coupled harmonic oscillators with fixed boundary conditions (see Fig. 1). Each particle has the mass m and we will assume these to be carbon atoms. The spring constant is equal to $\kappa = 1.0$ kN/m. In atomic scale simulations it is not so convenient to use SI units. Other units are often used which are more appropriate for the considered length- and timescales. We will use the units "Atomic simulation units", introduced in appendix A

In Fig. 4 we show the result for the position of the three different particles as function of time. We then used the initial conditions $q_1(0) = 0.01$ Å and $q_2(0) = q_3(0) = v_1(0) = v_2(0) = v_3(0) = 0$. In Fig. 5 the corresponding time-dependence of the kinetic and potential energies are shown as well as the sum, the total energy. It can be seen that the total energy is very well conserved. We also show the corresponding power spectra in Fig. 6. Three clear peaks for positive and negative frequencies are seen. These corresponds to the normal mode frequencies (see below).

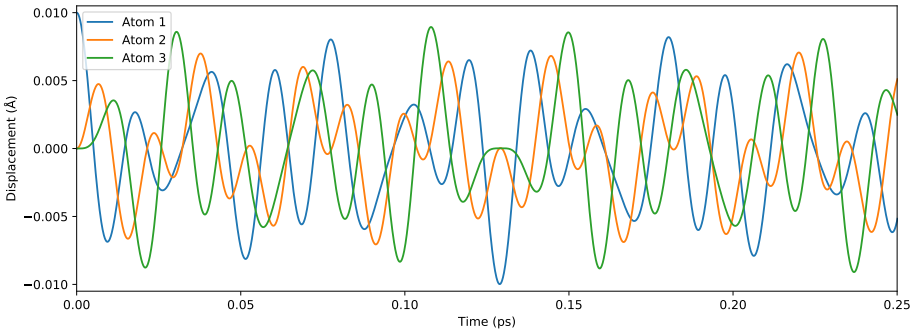

Figure 4: The time-dependent displacements for the three particle system.

5. The code E1code4 gives an implementation of the velocity Verlet algorithm for the motion of three identical coupled harmonic oscillators with fixed boundary conditions. Use this to determine the time evolution of the three harmonically coupled carbon atoms. Assume the initial condition $q_1(0) = 0.01$ Å, $q_2(0) = 0.005$ Å and $q_3(0) = -0.005$ Å. Set all initial velocities to zero, $v_1(0) = v_2(0) = v_3(0) = 0$. Plot your result. Determine also the time-dependence of the potential, kinetic and total energies. Do you conserve total energy? Vary the time-step

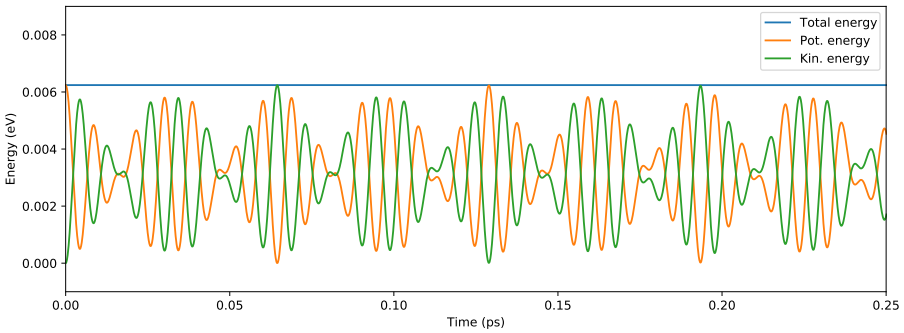

Figure 5: The time-dependence of the potential, kinetic and total energies for the three particle system.

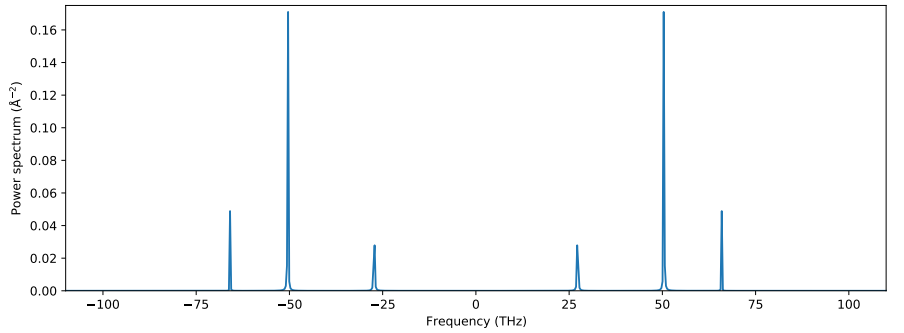

Figure 6: The power spectrum for the three particle system.

and investigate the energy conservation. Finally, determine the power spectrum. Do you obtain what you expect? You can compare your results with the corresponding results in Fig. 4, Fig. 5 and Fig. 6. (1p)

Next consider the motion of a three-particle system with free boundary conditions (see Fig. 7). The two end particles, each of mass m are bound to the central particle, with mass M, with two springs, each with spring constant κ. Consider the motion only in one dimension.

6. Apply the model to the triatomic molecule CO_2. The spring constant for carbon dioxide is $\kappa = 1.6$ kN/m. Derive the coupled set of equation of motions and solve them numerically using the velocity Verlet algorithm. Choose an appropriate time-step Δt based on conservation of the total energy. Fourier analyse the trajectories and try to determine

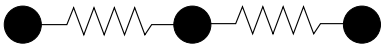

Figure 7: System of three coupled harmonic oscillators with free boundary conditions.

the normal mode frequencies. Compare with the experimental data for CO_2. (2p)

2 Normal modes

2.1 Three coupled oscillators

Consider again the three coupled harmonic oscillators in Fig. 1. The Hamiltonian, now expressed in terms of the momenta p_i and coordinates q_i, is

$$\mathcal{H} = \sum_{i=1}^{3} \frac{p_i^2}{2m} + \sum_{i=0}^{3} \frac{\kappa}{2}(q_{i+1} - q_i)^2$$

with the corresponding Hamilton's equation of motion

$$\begin{aligned} \dot{q}_i &\equiv \frac{\partial \mathcal{H}}{\partial p_i} = \frac{p_i}{m} \\ \dot{p}_i &\equiv -\frac{\partial \mathcal{H}}{\partial q_i} = \kappa(q_{i+1} - 2q_i + q_{i-1}) \end{aligned}$$

for $i = 1, 2, 3$. This can also be written on the matrix form

$$\frac{d^2}{dt^2}\mathbf{q}(t) = -\omega_0^2 \, \mathbf{K} \, \mathbf{q}(t)$$

where $\omega_0^2 = \kappa/m$ and

$$\mathbf{K} = \begin{bmatrix} 2 & -1 & 0 \\ -1 & 2 & -1 \\ 0 & -1 & 2 \end{bmatrix}$$

is the force constant matrix. If we make the ansatz

$$\mathbf{q}(t) = \mathbf{g} \cos(\omega t + \phi)$$

we get an eigenvalue problem

$$\mathbf{K} \, \mathbf{g} \; = \; \lambda \, \mathbf{g}$$

where $\lambda = \omega^2/\omega_0^2$. The corresponding eigenvalues and normalized eigenvectors are

$$\begin{aligned} \omega_1 &= \sqrt{2 - \sqrt{2}} \, \omega_0 \\ \omega_2 &= \sqrt{2} \, \omega_0 \\ \omega_3 &= \sqrt{2 + \sqrt{2}} \, \omega_0 \end{aligned}$$

and

$$\mathbf{g}_1 = \begin{bmatrix} 1/2 \\ 1/\sqrt{2} \\ 1/2 \end{bmatrix} \qquad \mathbf{g}_2 = \begin{bmatrix} 1/\sqrt{2} \\ 0 \\ -1/\sqrt{2} \end{bmatrix} \qquad \mathbf{g}_3 = \begin{bmatrix} 1/2 \\ -1/\sqrt{2} \\ 1/2 \end{bmatrix}$$

respectively. The transformation matrix is then given by

$$\mathbf{G} \equiv [\ \mathbf{g}_1\ \mathbf{g}_2\ \mathbf{g}_3\ ] = \begin{bmatrix} 1/2 & 1/\sqrt{2} & 1/2 \\ 1/\sqrt{2} & 0 & -1/\sqrt{2} \\ 1/2 & -1/\sqrt{2} & 1/2 \end{bmatrix}$$

This matrix is orthogonal and it diagonalizes the force constant matrix. We can now introduce normal coordinates

$$\mathbf{Q} = \sqrt{m}\ \mathbf{G}^T\ \mathbf{q}$$

with the corresponding inverse relation

$$\mathbf{q} = \frac{1}{\sqrt{m}}\ \mathbf{G}\ \mathbf{Q}$$

and in the same way for the momentum

$$\mathbf{P} = \frac{1}{\sqrt{m}}\ \mathbf{G}^T\ \mathbf{p}$$

and

$$\mathbf{p} = \sqrt{m}\ \mathbf{G}\ \mathbf{P}$$

More explicitly we have

$$\begin{aligned} Q_1 &= \frac{1}{2}(q_1 + \sqrt{2}q_2 + q_3) \\ Q_2 &= \frac{1}{\sqrt{2}}(q_1 - q_3) \\ Q_3 &= \frac{1}{2}(q_1 - \sqrt{2}q_2 + q_3) \end{aligned}$$

In matrix form the Hamiltonian can be written as

$$\mathcal{H} = \frac{1}{2m}\mathbf{p}^T\mathbf{p} + \frac{\kappa}{2}\mathbf{q}^T\mathbf{K}\mathbf{q}$$

which can be transformed to

$$\mathcal{H} = \frac{1}{2}\sum_{k=1}^{3}[P_k^2 + \omega_k^2 Q_K^2]$$

and the corresponding equation of motion takes the foorm

$$\frac{d^2}{dt^2}Q_k(t) + \omega_k^2\ Q_k(t) = 0\ \ ;\ \ k = 1, 2, 3$$

which explicitly shows that the normal modes are independent modes, uncoupled coordinates. The solution is given by

$$Q_k(t) = Q_k(0)\ \cos(\omega_k t) + \dot{Q}_k(0)\frac{1}{\omega_k}\ \sin(\omega_k t)$$

and, hence, the general solution can be written as

$$\mathbf{q}(t) = \frac{1}{\sqrt{m}}\mathbf{G}\mathbf{Q}(t)$$

Task

7. Consider again the three particle system with free boundary conditions in Fig. 7. Assume that the two end particles each have the mass m are bound to the central particle, with mass M, with two springs, each with spring constant κ. Determine analytically the normal mode frequencies. Compare with what you obtained in Task 6. (0p)

2.2 General case - N coupled oscillators

Consider now the general case with N coupled harmonic oscillators

$$\mathcal{H} = \sum_{i=1}^{N}\frac{p_i^2}{2m} + \sum_{i=0}^{N}\frac{\kappa}{2}(q_{i+1} - q_i)^2$$

The transformation matrix is given by

$$G_{ik} = \sqrt{\frac{2}{N+1}}\sin\left(\frac{ik\pi}{N+1}\right)$$

and the eigenfrequencies by

$$\omega_k = 2\sqrt{\frac{\kappa}{m}}\sin\left(\frac{k\pi}{2(N+1)}\right)$$

The transformed Hamiltonian takes the form

$$\mathcal{H} = \frac{1}{2}\sum_{k=1}^{N}[P_k^2 + \omega_k^2 Q_k^2]$$

and the normal modes

$$
\begin{aligned}
Q_k &= \sqrt{\frac{2}{N+1}} \sum_{i=1}^{N} \sqrt{m} q_i \sin\left(\frac{ik\pi}{N+1}\right) \\
q_i &= \sqrt{\frac{2}{N+1}} \sum_{k=1}^{N} \frac{Q_k}{\sqrt{m}} \sin\left(\frac{ik\pi}{N+1}\right)
\end{aligned}
$$

and

$$
\begin{aligned}
P_k &= \sqrt{\frac{2}{N+1}} \sum_{i=1}^{N} \frac{p_i}{\sqrt{m}} \sin\left(\frac{ik\pi}{N+1}\right) \\
p_i &= \sqrt{\frac{2}{N+1}} \sum_{k=1}^{N} \sqrt{m} P_k \sin\left(\frac{ik\pi}{N+1}\right)
\end{aligned}
$$

A System of units

SI system

The basic units in the SI system for length (L), mass (M), and time (T) are

L: 1 meter (m)
M: 1 kilogram (kg)
T: 1 second (s)

The units for *e.g.* frequency and energy are then derived from these: Hertz (1 Hz = 1 s^{-1}) and Joule (1 J = 1 $\mathrm{kg\,m^2\,s^{-2}}$), respectively, and cannot be chosen independently.

Atomic simulation units

In computational studies it is often convenient to use other system of units, which are more appropriate for the considered length- and time-scales. In atomic scale simulations the following units, "Atomic simulation units", [1] are often used. The starting point is the basic units

L: 1 Ångström (Å)
T: 1 picosecond (ps)
E: 1 electron volt (eV)

The unit for mass (M) can then not be chosen independently but is given by

$$\mathrm{M} = \mathrm{ET^2L^{-2}}$$

[1] see http://lammps.sandia.gov/doc/units.html)

or

$$\text{M:}\quad \mathrm{m_{asu}} = 1\mathrm{eV}\,\frac{(1\mathrm{ps})^2}{(1\text{Å})^2} = 1.60218\cdot 10^{-23}\ \mathrm{kg} = 9649\ \mathrm{u}$$

where u=1.66054·10^{-27} kg is the atomic mass unit. In these units the mass m of a oxygen atom is

$$m = 15.9994u/\mathrm{m_{asu}} = 1.658\cdot 10^{-3}$$

B Example code

```c
/*******************************************************************************
 * E1code1
 *******************************************************************************
 * generate h(t) = a*cos(2*pi*f*t + phi)
 * and writes the result to a h_t.csv
 *
 * Compile me as:
 * clang E1code1.c -o <executable name> -lm
 */
/*******************************************************************************
 * Includes
 ******************************************************************************/
#include <stdio.h>   //fopen, fprintf
#include <stdlib.h>  //malloc
#include <math.h>    //cos
#include <stdint.h>  //uint64_t

/*******************************************************************************
 * Constants
 ******************************************************************************/
#define PI 3.14159


/*******************************************************************************
 * Helper functions
 ******************************************************************************/
/*
 * constructs the signal
 * @signal - array to be filled with signal values
 * @t - time array filled with discrete time stamps
 * @len_t - the length of the time array
 * @a - amplitude of signal
 * @f - frequency of signal
 * @phi - phase of signal
*/
double *generate_signal(double *signal, double *t, uint64_t len_t, double a,
            double f, double phi)
{
    for(int i = 0; i < len_t; i++){
    signal[i] = a * cos(2 * PI * f * t[i] + phi);
    }
    return signal;
}

/*
 * constructs time array
 * @array - array to be filled with time values
 * @start - start value
 * @len_t - number of times stamps in array
 * @dt - time step between two consecutive times
*/
void arange(double *array, double start, int len_t, double dt){
    for(int i = 0; i < len_t; i++){
    array[i] = start + i*dt;
    }
}

/*
 * constructs time array
 * @fname - File name
 * @time_array - array of time values
 * @signal - array with signal values
 * @n_points - number of points
*/
```

```c
void write_to_file(char *fname, double *time_array,
           double *signal, int n_points)
{
    FILE *fp = fopen(fname, "w");
    fprintf(fp, "time, signal\n");
    for(int i = 0; i < n_points; ++i){
        fprintf(fp, "%f,%f\n", time_array[i], signal[i]);
    }
    fclose(fp);
}

int main()
{
    int N = 250; double dt = 0.1;
    double a = 1; double f = 2; double phi = 0;
    double time_array[N];
    arange(time_array, 0, N, dt);

    double signal[N];
    generate_signal(signal, time_array, N, a, f, phi);
    write_to_file("signal.csv", time_array, signal, N);
    return 0;
}
```

```python
#!/usr/bin/env python
################################################################################
# E1code2
################################################################################

import numpy as np
import matplotlib.pyplot as plt

# skip_header skips the first
# row in data.csv
array = np.genfromtxt('signal.csv', delimiter=',', skip_header=1)

fig, ax = plt.subplots()
ax.plot(array[:, 0], array[:, 1])

ax.set_xlabel('time (arb.unit)')
ax.set_ylabel('signal (arb.unit)')
ax.grid()

fig.savefig('signal.pdf')
```

```c
/*******************************************************************************
 * E1code3
 *******************************************************************************
 * reads h(t) = a*cos(2*pi*f*t + phi)
 * runs the fft of h(t)
 *
 *
 * Compile me as:
 * clang -c fft.c -o fft.o -lgsl -lgslcblas
 * clang E1code3 fft.o -o <execuutable name> -lgsl -lgslcblas
 */

/*************************************************************
 * Includes
 *************************************************************/
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#include "fft.h"  // interface to fft routine

/*************************************************************
 * Macro defines
 *************************************************************/
#define N_POINTS 250
#define dt 0.1

/*******************************************************************************
 * Helper functions
 *******************************************************************************/
/*
 * reads time_array and signal data from file
 * @fname - File name
 * @time_array - array of time values
 * @signal - array with signal values
*/
void read_data(char *fname, double *time_array, double *signal)
{
```

```
    FILE *fp = fopen(fname, "r");

    /* if file no found
     * error out and exit code 1
     */
    if(fp == NULL){
    perror("error:");
    exit(1);
    }

    /* skip header */
    fseek(fp, strlen("time, signal\n"), SEEK_SET);
    char line[128] = {0};
    char *token;
    int i = 0;
    while(fgets(line, sizeof(line), fp) != NULL){
    token = strtok(line, ",");
    time_array[i] = strtod(token, NULL);
    token = strtok(NULL, ",");
    signal[i] = strtod(token, NULL);
    i++;
    memset(line, 0, sizeof(line));
    token = NULL;
    }
    fclose(fp);
}

/*
 * constructs time array
 * @fname - File name
 * @time_array - array of time values
 * @signal - array with signal values
 * @n_points - number of points
*/
void write_to_file(char *fname, double *frequencies,
           double *spectrum, int n_points)
{
    FILE *fp = fopen(fname, "w");
    fprintf(fp, "time, signal\n");
    for(int i = 0; i < n_points; ++i){
        fprintf(fp, "%f,%f\n", frequencies[i], spectrum[i]);
    }
    fclose(fp);
}

/*************************************************************
 * Main routine
 *************************************************************/
int main(int argc, char **argv)
{
    double time_array[N_POINTS];
    double signal[N_POINTS];
    read_data("signal.csv", time_array, signal);

    /*
     * Construct array with frequencies
     */
    double frequencies[N_POINTS];
    for(int i = 0; i < N_POINTS; i++){
        frequencies[i] = i / (dt * N_POINTS);
    }

    /*
     * Do the fft
     */
    double fftd_data[N_POINTS];
    powerspectrum(signal, fftd_data, N_POINTS);
    /*
     * Dump fft and frequencies to file
     */
    write_to_file("powerspectrum.csv", fftd_data, frequencies, N_POINTS);
    return 0;
}
```

```
/*****************************************************************************
 * E1code4
 *****************************************************************************
 * Routine that runs the velocity verlet algorithm
 * Use as template to construct your program!
 */



/*
```

```
 * Calculate the acceleration
 * @a - vector that is filled with acceleration
 * @u - vector with the current positions
 * @m - vector with masses
 * @kappa - Spring constant
 * @size_of_u - the size of the position, acceleration and mass array
 */
void calc_acc(double *a, double *u, double *m, double kappa, int size_of_u)
{
    /* Declaration of variables */
    int i;

    /* Calculating the acceleration on the boundaries */
    a[0] = kappa*(- 2*u[0] + u[1])/m[0];
    a[size_of_u - 1] = kappa*(u[size_of_u - 2] - 2*u[size_of_u - 1])/m[size_of_u - 1];

    /* Calculating the acceleration of the inner points */
    for (i = 1; i < size_of_u - 1; i++){
        a[i] = kappa*(u[i - 1] - 2*u[i] + u[i + 1])/m[i];
    }
}

/*
 * Perform the velocity verlet alogrithm
 * @n_timesteps - The number of time steps to be performed
 * @n_particles - number of particles in the system
 * @v - array of velocity (Empty allocated array) - sizeof(q_n) = n_timesteps
 * @q_n - position of the n'th atom : sizeof(q_n) = n_timesteps
 * @dt - timestep
 * @m - vector with masses of atoms sizeof(n_particles)
 * @kappa - Spring constant
 */
void velocity_verlet(int n_timesteps, int n_particles, double *v, double *q_1,
             double *q_2, double *q_3, double dt, double *m,
             double kappa)
{
    double q[n_particles];
    double a[n_particles];
    q[0] = q_1[0];
    q[1] = q_2[0];
    q[2] = q_3[0];
    calc_acc(a, q, m, kappa, n_particles);
    for (int i = 1; i < n_timesteps + 1; i++) {
        /* v(t+dt/2) */
        for (int j = 0; j < n_particles; j++) {
            v[j] += dt * 0.5 * a[j];
        }

        /* q(t+dt) */
        for (int j = 0; j < n_particles; j++) {
            q[j] += dt * v[j];
        }

        /* a(t+dt) */
        calc_acc(a, q, m, kappa, n_particles);

        /* v(t+dt) */
        for (int j = 0; j < n_particles; j++) {
            v[j] += dt * 0.5 * a[j];
        }

        /* Save the displacement of the three atoms */
        q_1[i] = q[0];
        q_2[i] = q[1];
        q_3[i] = q[2];
    }
}
```

```
/*
fft.h

Header file for fft.c

Created by Martin Gren on 2014-10-22.
*/

#ifndef _fft_h
#define _fft_h

extern void powerspectrum(double *, double *, int);

extern void fft_freq(double *, double, int);

extern void powerspectrum_shift(double *, int);
```

```

extern void fft_freq_shift(double *, double, int);

#endif
```

```
/*
    fft.c
    Program with powerspectrum from (fast) discrete Fourier transform  using GSL
    Created by Martin Gren on 2014-10-22
*/
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <gsl/gsl_fft_real.h>
#include <gsl/gsl_fft_halfcomplex.h>
#include <complex.h>

/* Makes fft of data and returns the powerspectrum in powspec_data */
void powerspectrum(double *data, double *powspec_data, int n) /* input data, output
     powspec_data, number of timesteps */
{
    /* Declaration of variables */
    int i;
    double complex_coefficient[2*n]; // array for the complex fft data
    double data_cp[n];

    /*make copy of data to avoid messing with data in the transform*/
    for (i = 0; i < n; i++) {
        data_cp[i] = data[i];
    }

    /*Declare wavetable and workspace for fft*/
    gsl_fft_real_wavetable *real;
    gsl_fft_real_workspace *work;

    /*Allocate space for wavetable and workspace for fft*/
    work = gsl_fft_real_workspace_alloc(n);
    real = gsl_fft_real_wavetable_alloc(n);

    /*Do the fft*/
    gsl_fft_real_transform(data_cp, 1, n, real, work);

    /*Unpack the output into array with alternating real and imaginary part*/
    gsl_fft_halfcomplex_unpack(data_cp, complex_coefficient,1,n);

    /*fill the output powspec_data with the powerspectrum */
    for (i = 0; i < n; i++) {
        powspec_data[i] = (complex_coefficient[2*i]*complex_coefficient[2*i]+
     complex_coefficient[2*i+1]*complex_coefficient[2*i+1])/n;
    }

    /*Free memory of wavetable and workspace*/
    gsl_fft_real_wavetable_free(real);
    gsl_fft_real_workspace_free(work);
}


/* Shifts the input powspec_data to center the 0 frequency */
void powerspectrum_shift(double *powspec_data, int n) /* input data, timestep, number of
     timesteps */
{
    /* Declaration of variables */
    int i;

    /* make copy of fft_data as reference for the shift */
    double powspec_cp[n];
    for (i = 0; i < n; i++) {
        powspec_cp[i] = powspec_data[i];
    }

    /* make shift */
    for (i = 0; i < n; i++) {
        if (n % 2) /*if n odd*/ {
            if (i<=(n-2)/2) {
                powspec_data[i] = powspec_cp[(i+(n+1)/2)];
            }
            else {
                powspec_data[i] = powspec_cp[(i+(n+1)/2)%(n)];
            }
        }
        else {
            if (i<n/2) {
                powspec_data[i] = powspec_cp[i+n/2];
            }
```

```
77              else {
78                  powspec_data[i] = powspec_cp[(i+n/2)%(n)];
79              }
80          }
81      }
82  }
83
84  /* Makes a frequency array fft_freq with frequency interval 1/(dt*n) */
85  void fft_freq(double *fft_freq, double dt, int n) /* output frequency array, timestep, number
        of timesteps */
86  {
87      /* Declaration of variables */
88          int i;
89      /* Fill the output aaray with frequencies */
90      for (i = 0; i < n; i++) {
91          fft_freq[i] = i/dt/n;
92      }
93  }
94
95  /* Makes a frequency array fft_freq with frequency interval 1/(dt*n) with a centered 0
        frequency */
96  void fft_freq_shift(double *fft_freq, double dt, int n) /* output frequency array, timestep,
        number of timesteps */
97  {
98      /* Declaration of variables */
99          int i;
100     /* Fill the output aaray with shifted frequencies */
101     for (i = 0; i < n; i++) {
102         if (n % 2) /*if n odd*/ {
103             if (i<=(n-2)/2) {
104                 fft_freq[i] = (-(n-1)/2+i)/dt/n;
105             }
106             else {
107                 fft_freq[i] = (i-(n-1)/2)/dt/n;
108             }
109         }
110         else {
111             if (i<n/2) {
112                 fft_freq[i] = (-n/2+i)/dt/n;
113             }
114             else {
115                 fft_freq[i] = (i-n/2)/dt/n;
116             }
117         }
118     }
119 }
```