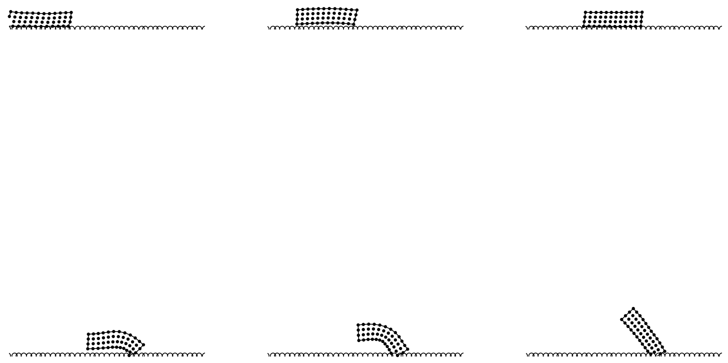
Patrik Norqvist, patrik.norqvist(at)umu.se
Department of physics
Umeå University

Computer exercise
Modelling with particle-spring systems
Fysikaliska modellers matematik 7.5 hp

Computer exercise:
Modeling with particle-spring systems



1 Introduction

In this exercise you should solve differential equations numerically. You might wonder why you need computers to do something that you spent lots of hours learning how to do analytically. The reason is that some problems are just not suited to be solved analytically. Systems with lots of coupled ordinary differential equations are very hard to solve analytically, and even if it sometimes is possible, the result may be very hard to interpret. By using Matlab you can let the computer do the tedious calculations, and also get a visual presentation (sometimes in real time that you can interact with)of the result that is easy to understand.[1] Another big advantage with numerical computations is that it is easy to combine many smaller modules that simulates different parts of the physics, to a computer program that can simulate complex physical situations – this is referred to as *multiphysics*.

The main problems we will study here are particle-spring systems representing small (microscopic) objects and – with minor changes in the code – larger (macroscopic) objects sliding on an irregular surfaces giving rise to friction and heating the object. Small modifications in your computer code enables you to model other physical objects, like elastic membranes.

The exercises should be solved and presented in a report (as specified in the text here) *individually*. This means: one person, one code, one report. But it is naturally allowed for you, we even encourage you, to discuss the problems and their solution with your class mates.

The deadlines are: **12 January 16:00** you should deliver the code for Exercise 3a) (the code should produce a stable simulation given a slight initial disturbance) – keeping this deadline is awarded with one bonus point. **17 January 09:00** the report exercise 2-4 (including numerical code) should be handed in – no award here, this deadline will simply be kept. Send the report as pdf, and the code in a seperate m-file. No need for all codes, one code for lab 2 and one for lab 4 is enough. All reports/results/codes are handed in via canvas.

The deadline for bonuslabs (lab 5-7),is 1 **February 17:00**.

[1]And some times even can be funny to look at! You have to be a pretty severe physics nerd to think an analytic solution is funny to look at ...

2 Theory

What good are springs? You may have noticed them in cars (dampers), in bed mattresses – and we've probably all been there, on our knees, cursing our curiosity while looking for that tiny spring shooting off when you take apart the last functioning ball pen. Besides from being useful in various mechanical constructions, springs are also very useful when making mathematical models of physical systems. The forces between the atoms in molecules (and in bigger objects, like cheese) are complicated but can for many applications be approximated by a spring force. This approximation makes it much more simple to study the propagation of sound or the microscopic nature of heat in matter (like cheese).

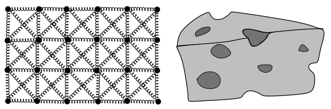


Figure 1: *Particle-spring systems can be used for modelling elastic objects (like a cheese).*

For many properties of macroscopic objects, such as the bounciness of cheese, you need not take into account all that happens on the microscopic level [2]. Instead you make a mathematical model of the object that describes only the macroscopic properties. You can for instance describe it as a point mass or a solid object. Many objects are deformable and elastic, however, and if you want to incorporate this property in the model one way to do it is (as you may have guessed already) to use springs. Alternatively you can use models for elastic materials that describe them as jelly-like fluids. These models are perfectly good, but when you are about to make computer simulations, particle-based models tend to be advantageous. You simply "divide" the object into a suitable number of entities (approximated by point particles) and "put" springs between the particles, see Figure 1 The sum of the particle masses should be equal to the mass of the object it is a model of and the configuration and properties of the springs should be chosen to give the desired elasticity properties. Next you write down the equations of motion for the particles and implement a numerical code that solves the system of equations.

2.1 The particle-spring system

In this section the equations that describe the dynamics of particles connected with springs are presented. The particles are treated as point masses and the springs have no mass.

2.1.1 One dimension

We start with a particle attached to a spring fasten in a static object, as in Figure 2. In one dimension the particle has position $x(t)$ and velocity $v(t) = \frac{dx(t)}{dt}$. The dynamics of the particle is given by Newton's law of motion together with Hook's law (that gives the spring force f):

$$m\frac{dv(t)}{dt} = f \tag{1}$$

$$f = -\kappa_s\left[x(t) - L\right] \tag{2}$$

Note that the force is zero when $x(t) = L$ (the *spring length*), positive (pushing the particle from the wall) when $x(t) < L$ and negative (pulling the particle) when $x(t) > L$. The strength of the spring is determined by the *spring constant* κ_s. The combined equation is well-studied. The general solution is $x(t) = x_0 \sin(\omega t + \varphi)$, where the constants x_0 and φ are determined by the initial conditions (initial position and velocity). This is a so called harmonic oscillation. The particle oscillates about the equilibrium point $x = L$ and the angular frequency of the oscillation

[2] A 300g cheese consists of roughly 10^{26} particles and no today existing computer (nor man or woman) is fit to solve the corresponding system of equations anyway.

is $\omega = \sqrt{\kappa_s/m}$. Note that the oscillation frequency depend on the spring constant and the particle mass.

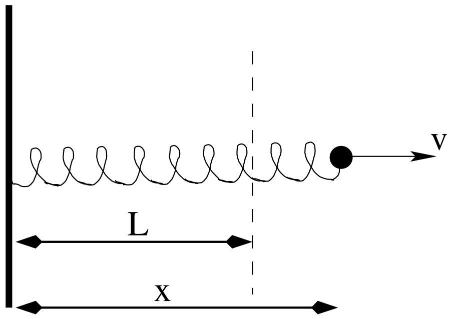


Figure 2: *A spring. If no force is applied the length of the spring is L.*

You may have noticed that real springs do not go on oscillating for all time. All real spring are damped to some extent. Damping is a form of friction, and depends on the rate at which the spring is contracted or stretched. The damping is included as an extra force term depending on particle velocity

$$f = -\kappa_s(x - L) - \kappa_d v \tag{3}$$

where κ_d is the *damping constant*. Observe the sign of the damping force – it acts in the opposite direction of the motion. Also in this case there exist analytical solutions, that are damped exponentially like $x(t) \sim e^{-\frac{\kappa_d}{2m}t}$ (combined with the oscillatory motion).

2.1.2 Three dimensions

Next we generalize these equations to three dimensions. Take two particles, we call them a and b, connected with a spring. In what directions can the spring force act? The only possible directions for the spring force as well as the damping force is along the spring. This direction is given by the vector $\mathbf{r}$ that goes from $\mathbf{x}_a$ to $\mathbf{x}_b$, see Figure 3. We denote the spring force acting on particle a (due to spring between a and b) by $\mathbf{f}_{ab}$. The equation of motion for particle a becomes

$$m_a \frac{d\mathbf{v}_a(t)}{dt} = \mathbf{f}_{ab} \tag{4}$$

$$\mathbf{f}_{ab} = -\left[\kappa_s(|\mathbf{r}| - L) + \kappa_d \frac{\dot{\mathbf{r}} \cdot \mathbf{r}}{|\mathbf{r}|}\right] \frac{\mathbf{r}}{|\mathbf{r}|} \tag{5}$$

The vectors $\mathbf{r}$ (*relative distance*) and $\dot{\mathbf{r}}$ (*relative velocity*) are defined

$$\mathbf{r} = \mathbf{x}_a - \mathbf{x}_b \ , \quad \dot{\mathbf{r}} = \frac{d\mathbf{r}}{dt} = \mathbf{v}_a - \mathbf{v}_b$$

We take a closer look at the force $\mathbf{f}_{ab}$ in Eq. (5). Note that $\frac{\mathbf{r}}{|\mathbf{r}|}$ is a unit vector giving the direction of the force $\mathbf{f}_{ab}$. The spring force (the term $\kappa_s(|\mathbf{r}| - L)$) is a straight forward generalization of Eq. (2), but the expression for the damping force in Eq. (5) (the term $\kappa_d \frac{\dot{\mathbf{r}} \cdot \mathbf{r}}{|\mathbf{r}|}$) is maybe more unexpected. Why is the damping force not simply just $-\kappa_d \mathbf{v_a}$? Well, this would damp the motion of the particles even when there is no stretching or contraction in the spring. The desired expression for the damping force should instead involve the relative velocity *along* $\mathbf{r}$ and the force should also act in this direction. This gives precisely the expression given in Eq. (5).

What is the force acting on particle $b$then? Recall Newton's third law, the particle b feels a force equally as strong as a but opposite in direction: $\mathbf{f}_{ba} = -\mathbf{f}_{ab}$.

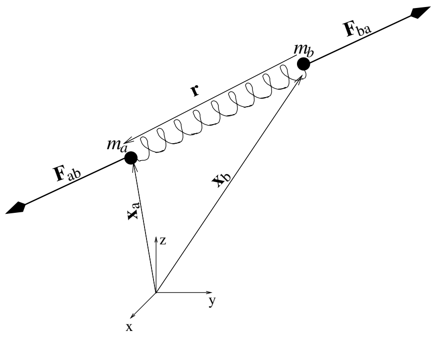


Figure 3: *Two particles connected with a spring with the notations used in the text.*

If we have a large set of particles and springs these should be included in the equations of motion, e.g. for particle a

$$m_a \frac{d\mathbf{v}_a(t)}{dt} = \mathbf{f}_a \tag{6}$$

$$\mathbf{f}_a = \sum_b \mathbf{f}_{ab} \tag{7}$$

where the summation is taken over all b:s that are connected with springs to a. If there are also external forces, like gravity ($m\mathbf{g}$), this should also be included in $\mathbf{f}_a$.

The energy of a physical system is always an interesting quantity and especially when doing numerical simulations. The energy for a spring is $\frac{1}{2}\kappa_s(|\mathbf{r}| - L)^2$. If the spring is relaxed ($|\mathbf{r}| = L$) it has zero energy, but if it is stretched or contracted there is energy stored in the spring. The total energy of a particle spring system is

$$E_{tot} = E_k + E_p + E_{spring}$$

where E_k is the sum of the particles *kinetic energy*, E_p is the sum of the particles *potential energy* (e.g. due to gravity) and E_{spring} is the sum of energy of the springs.

So, now we have the equations that govern a system of particles connected by springs. If we have Nparticles, then we have *3*Nordinary differential equations like Eq. (6) (recall that each equation has three vector components in 3D, thus the factor *3*). What really makes this system hard to solve analytically (and the solution hard to interpret) is that the equations are *coupled*. The equation for particle a involves also the position and velocity of any particle that it is connected to with springs. We shall next see how to solve the equations numerically.

2.2 Numerical methods for ordinary differential equations

Before the days of modern computers heavy computations where done manually by large teams of peoples, each team responsible for a small part of the computation cycle. Luckily those days are over. The numerical methods remains the same roughly, however, and here comes two of them.

2.2.1 Numerical integration of differential equations

An ordinary differential equation of the type

$$\frac{dx(t)}{dt} = f(t) \tag{8}$$

$$x(t_a) = X \tag{9}$$

where $f(t)$ is a given function and $x(t_a) = X$ is given initial data, has some solution $x(t)$. The solution – or rather a numerical approximation of the solution – can be found using *Euler's method*[3]. The numerical solution is a set of values $[x_a, ..., x_b]$ that correspond to the value of $x(t)$ at various points t (let's call this parameter time), $[t_a, ..., t_b]$. The function values are found – using the oldest trick in the book[4] – by approximating the derivative

$$\frac{dx(t)}{dt} \approx \frac{x(t + \Delta t) - x(t)}{\Delta t} \tag{10}$$

Observe that in the limit $\Delta t \to 0$ this is precisely the definition of derivative. In numerical simulations we do not take this limit all the way. Depending on whether the right hand side of Eq. (8) is evaluated at time t or time $t + \Delta t$, or any other choice, we end up with different numerical methods (with different properties). The method of numerically solving differential equations is also referred to as *numerical integration*.

Euler's explicit method Given below is an algorithm for calculating the solution using *Euler's explicit method* (evaluating the right hand side at time t)

- The first step is to divide the time interval that is of interest, e.g. from time t_a to t_b, in a finite number (N) of time steps $[t_0, t_1, t_2, ..., t_N]$, where $t_0 = t_a$ and $t_N = t_b$. The distance between the points are $\Delta t = (t_b - t_a)/N$.
- The algorithm for computing the solution of Eq. (8) with Euler's explicit method can be summarized as:

  $$x_{n+1} = x_n + f_n \Delta t \tag{11}$$

  where the index x_n and f_n stands for the value of $x(t)$ and $f(t)$ at time $t = t_a + n\Delta t$. The time step should be chosen so small that if it is decreased further, the result doesn't change much. The calculation starts with $n = 0$ which gives $x_1 = x_0 + f_0 \Delta t$, where the initial data $x_0 = X$ is used. From this one can proceed with $n = 1$ that gives the value of x_2, and then set $n = 2$ etc (be glad you live in the age of modern computers).

A *second order* ordinary differential equation, for example

$$\frac{d^2x(t)}{dt^2} = f(t) \tag{12}$$

$$x(t_a) = X \quad , \quad \frac{dx(t)}{dt}_{|t=t_a} = V \tag{13}$$

can be solved by introducing an *auxiliary function*[5] $v(t) = \frac{dx(t)}{dt}$. The problem is then transformed to solving *two coupled* first order equations, one for $x(t)$ and one for $v(t)$. The algorithm, in this case, can be summarize as

$$x_{n+1} = x_n + v_n \Delta t \tag{14}$$

$$v_{n+1} = v_n + f_n \Delta t \tag{15}$$

and the initial conditions are $x_0 = X$ and $v_0 = V$. The procedure is then like above: calculate for $n = 0$, then $n = 1$ etc.

[3]There exists many other numerical methods for solving ordinary differential equations and they all have different properties (accuracy, computational speed, stability).

[4]The book referred to is *Physica Ultimata*, a book with all math and physics tricks and formulas you'll ever need beautifully condensed into twenty pages. Unfortunately the book only exists in the class of parallel universas known as the *best of worlds.* In our world this wisdom is spread over thousands of textbooks or just as a vague feeling – passed down from generation to generation of physicists – of *"-I should know this, dammit"*.

[5]We would probably call this function *velocity* if the equation is an equation of motion.

The Leap Frog method The Leap Frog method is another method for solving second order equations. Let us take the example (12)-(13) again, but written as:

$$\frac{dx(t)}{dt} = v(t) \tag{16}$$

$$\frac{dv(t)}{dt} = f(t) \tag{17}$$

$$x(t_a) = X \ , \quad \frac{dx(t)}{dt}_{|t=t_a} = V \tag{18}$$

Next we approximate the derivatives. Observe that the following is also a reasonable approximation

$$\frac{dx(t)}{dt} \approx \frac{x(t+\Delta t/2) - x(t-\Delta t/2)}{\Delta t} \tag{19}$$

In the limit of $\Delta t \to 0$ this also gives you the derivative – it is then mathematically equivalent to Eq. (10). When we don't take the limit $\Delta t \to 0$ all the way, there is a slight difference. They are both good approximations, but slightly differently good. The Leap Frog method is when you make use of this approximation on Eqs. (16)-(17) in the following way

$$x_{n+1} = x_n + v_{n+\frac{1}{2}}\Delta t \tag{20}$$

$$v_{n+\frac{1}{2}} = v_{n-\frac{1}{2}} + f_n\Delta t \tag{21}$$

with initial conditions $x_0 = X$ and $v_0 = V$. Look at Figure 4 to compare it with Euler's method! The position is updated every time step using information of the velocity between the time steps (i.e. at half integer time steps). The velocity is shifted one half time step in time and when the new velocity is calculated one uses information about the force between the velocity steps (i.e. at integer time steps). Before using this algorithm you must know the value of $v_{-\frac{1}{2}}$. This is accomplished with a half backwards Euler step: $v_{-\frac{1}{2}} = v_0 - f_0\Delta t/2$. The code may look like:

```
x_old=X;              % Initial conditions
f_old=...                 % Initial force calculation
v_old=V-f_old*DT/2;   % half backwards Euler step to get v_{-1/2}
for n=1:NS            % Main loop
    f=...             % Force calculation (may involve x_old and v_old)
    v_new=v_old+f*DT;        % Update velocity
    x_new=x_old+v_new*DT;   % Update position
    ...
end
```

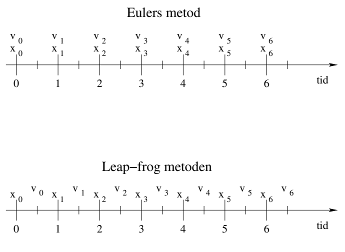

Eulers metod
v0 x0 v1 x1 v2 x2 v3 x3 v4 x4 v5 x5 v6 x6
0 1 2 3 4 5 6
tid
Leap–frog metoden
x0 v0 x1 v1 x2 v2 x3 v3 x4 v4 x5 v5 x6 v6
0 1 2 3 4 5 6
tid

Figure 4: *Comparing the explicit Euler and Leap Frog.*

Observe one thing. The force should be calculated at time n, that is $f_n = f_n(x_n, v_n)$ but typically we don't know the value of v_n since velocity is calculated at half integer time steps. There are two ways out of this. Either to actually calculate v_n at every step of the algorithm (using a half Euler step) or by approximating $f_n \approx f_n(x_n, v_{n-\frac{1}{2}})$. The later can be justified if the function f depends only weakly on v. This will be our case.

Stability So, we can choose between two methods (Euler's explicit method and Leap Frog). Why ever chose anything else than Euler, it seems to be the simplest? Well, the two methods have different *stability properties*. Both numerical methods has numerical errors (since we *approximate* the derivative). These errors typically fluctuate with time, but for certain methods applied to certain equations the errors may grow cumulatively (typically exponentially). An example of this is Euler's explicit method applied to particle-spring systems. The Leap Frog does not suffer from this.

2.2.2 Leap Frog integration of particle-spring system

We we want to make a numerical simulation of a particle-spring system. The system is described by the Eqs. (6)-(7). Let us for the moment focus on one particle, say particle a. We assume it is attached to one particle only (particle b) with a spring. In the absence of gravity, the equation of motion for particle a is

$$m_a \frac{d\mathbf{v}_a}{dt} = \mathbf{f}_{ab}(\mathbf{x}_a, \mathbf{x}_b, \mathbf{v}_a, \mathbf{v}_b) \tag{22}$$

$$\frac{d\mathbf{x}_a}{dt} = \mathbf{v}_a \tag{23}$$

The problem involves springs so our choice of method is Leap Frog integration – Eqs. (20)-(21) in vector form

$$\mathbf{v}_a^{n+\frac{1}{2}} = \mathbf{v}_a^{n-\frac{1}{2}} + m_a^{-1}\mathbf{f}_{ab}^n \Delta t \tag{24}$$

$$\mathbf{x}_a^{n+1} = \mathbf{x}_a^n + \mathbf{v}_a^{n+\frac{1}{2}} \Delta t \tag{25}$$

Note that the force is a function of position and velocity $\mathbf{f}_{ab}^n = \mathbf{f}_{ab}(\mathbf{x}_a^n, \mathbf{x}_b^n, \mathbf{v}_a^n, \mathbf{v}_b^n)$ and we approximate $\mathbf{f}_{ab}^n \approx \mathbf{f}_{ab}(\mathbf{x}_a^n, \mathbf{x}_b^n, \mathbf{v}_a^{n-1/2}, \mathbf{v}_b^{n-1/2})$. Before we move on to the method of solving the entire system of particles and springs we stop and consider a more simple system in detail.

2.2.3 One dimensional system with two particles and one spring

The system including one spring and two particles (a and b) in one dimension has the following numerical integration scheme

$$v_a^{n+\frac{1}{2}} = v_a^{n-\frac{1}{2}} + m_a^{-1} f_{ab}^n \Delta t \tag{26}$$

$$v_b^{n+\frac{1}{2}} = v_b^{n-\frac{1}{2}} + m_b^{-1} f_{ba}^n \Delta t \tag{27}$$

$$x_a^{n+1} = x_a^n + v_a^{n+\frac{1}{2}} \Delta t \tag{28}$$

$$x_b^{n+1} = x_b^n + v_b^{n+\frac{1}{2}} \Delta t \tag{29}$$

The Eqs (26)-(27) they can collected in a vector as

$$\begin{pmatrix} v_a^{n+\frac{1}{2}} \\ v_b^{n+\frac{1}{2}} \end{pmatrix} = \begin{pmatrix} v_a^{n-\frac{1}{2}} \\ v_b^{n-\frac{1}{2}} \end{pmatrix} + \Delta t \begin{pmatrix} \frac{1}{m_a} f_a^n \\ \frac{1}{m_b} f_b^n \end{pmatrix} \tag{30}$$

$$\begin{pmatrix} x_a^n \\ x_b^n \end{pmatrix} = \begin{pmatrix} x_a^n \\ x_b^n \end{pmatrix} + \Delta t \begin{pmatrix} v_a^{n+\frac{1}{2}} \\ v_b^{n+\frac{1}{2}} \end{pmatrix} \tag{31}$$

Collecting the variables in vectors is a good thing when we are about to do numerical computations with MATLAB.

2.2.4 General system

We now go on to formulate the problem for the entire system of particles and springs. Say that we have N particles and N_S springs. It is essential now that we use vector notations in order to find a systematic way of solving the system. We collect positions, velocities and forces in $N \times 3$ matrices.

$$\begin{aligned}
\mathbf{X}^n &= [x_1^n, y_1^n, z_1^n; x_2^n, y_2^n, z_2^n; ...; x_{N-1}^n, y_{N-1}^n, z_{N-1}^n; x_N^n, y_N^n, z_N^n] \\
\mathbf{V}^n &= [vx_1^n, vy_1^n, vz_1^n; vx_2^n, vy_2^n, vz_2^n; ...; vx_{N-1}^n, vy_{N-1}^n, vz_{N-1}^n; vx_N^n, vy_N^n, vz_N^n] \\
\mathbf{F}^n &= [Fx_1^n, Fy_1^n, Fz_1^n; Fx_2^n, Fy_2^n, Fz_2^n; ...; Fx_{N-1}^n, Fy_{N-1}^n, Fz_{N-1}^n; Fx_N^n, Fy_N^n, Fz_N^n]
\end{aligned}$$

The variables in $\mathbf{X}^n$ and $\mathbf{V}^{n+\frac{1}{2}}$ and the forces in $\mathbf{F}^n$ are the ones that occur in Eq. (6). The numeration that ranges from 1 to N is the particle number (for which we have used the notation a earlier). Observe that the forces here is the *total* forces on a particle, just as in Eq. (7). The Leap Frog integration scheme for the entire system can with these variables be written as

$$\mathbf{V}^{n+\frac{1}{2}} = \mathbf{V}^{n-\frac{1}{2}} + \Delta t \mathbb{M}^{-1} \mathbf{F}^n \tag{32}$$

$$\mathbf{X}^{n+1} = \mathbf{X}^n + \Delta t \mathbf{V}^{n+\frac{1}{2}} \tag{33}$$

$$\tag{34}$$

The particle masses has been collected in a diagonal matrix $\mathbb{M}$ (see below). Recall that we make the approximation $f^n \approx f(x^n, v^{n-\frac{1}{2}})$, i.e. the force vector $\mathbf{F}^n$ depend on all particle positions $\mathbf{X}^n$ and velocities $\mathbf{V}^{n-\frac{1}{2}}$. The matrix $\mathbb{M}$ is

$$\mathbb{M} = \begin{pmatrix} m_1 & & & & & & & \\ & m_2 & & & & & & \\ & & m_3 & & & & & \\ & & & \ddots & & & & \\ & & & & m_{N-3} & & & \\ & & & & & m_{N-2} & & \\ & & & & & & m_{N-1} & \\ & & & & & & & m_N \end{pmatrix}$$

Observe that this particular form of the mass matrix depends on how the particles are ordered in the ordering in the matrices $\mathbf{X}^n$, $\mathbf{V}^n$ and $\mathbf{F}^n$. The mass matrix should have the form such that $\mathbb{M}\mathbf{V}^n$ is the vector of particle momentum (in the same particle ordering). The size of $\mathbb{M}$ is $N \times N$.

Now we have a numerical method that solves a particle-spring system. In each time step of the simulation we update the velocities by Eq. (32) and the positions using Eq. (33). This is very simple when we have MATLAB at our hands. The harder part is to find a systematic way to calculate the force $\mathbf{F}^n$. Recall that the system should first be initialized with making a backwards half Euler step for the velocity.

3 Exercises

In the remaining we will restrict ourself to two dimensions (2D) – this simplifies slightly and speeds up the computation. If you feel up to it, it is still a good idea to make the implementation in 3D (and set the forces, velocities and positions in the extra dimension to zero when doing the exercises below). It is then very simple to apply the same code on a 3D problem.

When doing numerical computations it is customary not to use ordinary SI-units (like seconds, meter etc.), but instead re-scaled units (*normalized units*). As will we do. For instance, length may be given as $L = 1$. We do not bother what these units means in reality (meters, inches, apples?) – you will have to practice this in some other course, here we just practice solving differential equations.

In the Hint section you find some hints that may be useful when doing the exercises, but don't look at them unless you are stuck on a problem.

3.1 Exercise 1: Preparatory exercise

Write a code that evolves the position $\mathbf{X}$ and velocity $\mathbf{V}$ for three particles with the force given below. This means that the problems involves no springs or spring forces to calculate, it is just an exercise to get the main program running. See the Hint section for suggested use of data types. The code should perform the following

- Setup the particles initial positions and velocities. Take $\mathbf{x}_1 = [-L/2, 0]$, $\mathbf{x}_2 = [L/2, 0]$, $\mathbf{x}_3 = [0, L/2]$ and $\mathbf{v}_1 = [v_{ini}, 0]$, $\mathbf{v}_2 = [0, -v_{ini}]$, $\mathbf{v}_3 = [-v_{ini}/2, -v_{ini}/2]$ with $L = 1$ and $v_{ini} = 0.01$.
- Compute and update the position $\mathbf{X}$ and velocity $\mathbf{V}$ for the particles during 1500 time steps with step size $\Delta t = 0.1$ and mass $m = 1$. Use the method described above – Eq. (32) and Eq. (33) – to compute the change in position and velocity each time step. Use the following given force expression:

  ```
  F = 0.5*(0.5-rand(NP,2))-0.1*V
  ```

  where `NP` is the number of particles, the 2 is due to *two* dimensions. `rand` produces a matrix of specified size of random numbers (with a Gaussian distribution) – this gives us a random force.
- Display the system graphically during the simulation. Plot the particles as points or circles and draw lines between them (from particle 1 to 2, 2 to 3 and 3 to 1). See the graphics hints.

Make sure that you write the code in an as general form as possible (make use of vectors) so that it is easy to change the number of particles. The code then forms a prototype for the coming exercises, and you just have to complement it with a variable handling the springs and some lines for the force computations.

3.2 Exercise 2: Simple particle-spring system

Finally we now have enough background to look at a real problem. In this exercise you should implement the described method for solving particle-spring systems. To test the implementation, make a simulation of the (almost) most simple problem with springs we can have – two particles connected by a spring, as in Figure 5. You can easily solve this analytically, but it is a good problem to test that your code is correct and more easy to understand the implementation. Write your code so that it is easy to add new springs and particles. You have then also done most of the work for the last two exercises.

Take the code from Exercise 1. This should be supplemented with a variable for the springs (see the Hint section for suggested use of data types) and force calculations based on the springs.

The system that you should test your code one is: Two particles, both with mass $m = 1$, connected by a spring with constant $\kappa_s = 10$ and to start with we set the damping to zero, $\kappa_d = 0$. Take a deep breath and set gravity to zero, $g = 0$ (we are now in outer space). Set the spring rest length to $L = 1$. At time $t = 0$, both masses are still, and at a distance 1.8 from each other, so set $\mathbf{x}_1 = [0, 0]$ and $\mathbf{x}_2 = [1.8, 0]$.

Questions you should answer:

a) How does the particles move as we let go of the system? Plot the moving masses (maybe you don't have to plot every single time step).

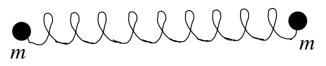


Figure 5: *The setup in Exercise 2 – with two particles and one spring.*

b) Calculate the energy quantities[6]: E_k, E_p, E_{spring}. Is the total energy constant? How large Δt can you use before your total energy changes more than 1% per oscillation? If your simulation moves too fast on your screen when you increase Δt put in a small pause each time you plot your result.

c) Set the spring damping to $\kappa_d = 0.5$. After how long time has the amplitude decreased to 10 % of the original value? Do you get the same value if you solve the equation analytically ($\ddot{r} + 2(\kappa_d/m)\dot{r} + 2(\kappa_s/m)r = 0$, where $r = |x_1 - x_2| - L$)? Do you get different times if you use different Δt?

d) Set the damping to zero again. Give both masses an initial velocity perpendicular to the x-axis. Assume that the spring at $t = 0$ lies along the x-axis. Give both masses the same speed $v = 5$, but with opposite directions, one in positive y-direction and the other in negative y-direction. Now the spring should rotate around the point ($x = 0.9$, $y = 0$). Calculate the angular momentum $\mathfrak{L} \equiv \sum m(xv_y - yv_x) = I\omega$ for the system relative the center of mass as a function of time. Plot the length of the spring (the distance between the particles) and angular frequency ω as a function of time. What is the equilibrium length of the oscillating spring? Why is it not equal to the spring rest length L?

3.2.1 Results for the report

Your report should include:

- ✓ Answers to questions (as a describing text, not just the plain answer).
- ✓ A plot of E_k, E_p, E_{spring} and E_{tot} versus time for both undamped and damped system.
- ✓ A plot of the angular momentum as a function of time. Plot spring length and angular frequency versus time in a single diagram.
- ✓ Code files to run to the simulation (.m format).

3.3 Exercise 3: Modeling a sliding microscopic 2D object

Here you will simulate a 2D object. We are going to study what happens with a small microscopic object that slide on a rough surface. It should be stressed that the model we use is very simplified (but gives good qualitative agreements with the real world).

In our simulation we will simulate the motion of each atom (molecule) of the object. Between two nearby atoms there is a force. This force can be a complicated function depending on relative distances between atoms, but in our simple model this force will be a simple linear spring. No important physics will be lost due to this. To get the object stable we need to connect our spring like in figure 6, otherwise it will collapse.

One big advantage with our model is that we can follow the vibrations of each single atom. This enable us to calculate the internal energy of the object. The internal energy can be seen as the temperature of the object. Normally when an object is dragged over a surface with friction we just say that we loose energy to friction without bothering with where the energy goes. In this simulation the object is actually heated (the particles begin vibrating) due to the interaction with the surface – and there is no magical loss of energy.

Your simulation should include:

[6] It is not necessary to calculate the energy at ever single time-step. Calculate and store the energy at, say every 10:th or 100:th time-step.

a) A two dimensional rectangular object with springs connecting particles like in Figure 6. Your box should have $Nr = 4$ rows and $Nc = 8$ columns, but write you code so it is trivial to change the size. The diagonal springs act to prevent the object from shearing. Take the particle masses to be $m = 1$, spring lengths $L = 1$ (vertical and horizontal) and $L = \sqrt{2}$ (diagonal springs), spring constant $\kappa_s = 100$, spring damping $\kappa_d = 0$ (but do include the damping terms in the code for later use), gravitational acceleration $g = 1$. Suggested time step[7]: $\Delta t = 0.002$.

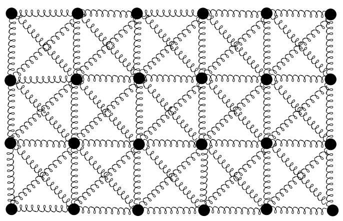


Figure 6: *A model of a microscopic 2D square. The diagonal springs acts to prevent shearing motion.*

b) A surface built up with 16 solid circles with radius $R = L$ distributed along the $x-$direction, Figure 7. The circle centers are placed with approximately the same distance R, but with a small random variation (`+0.1*R*randn`) and at the same height (same y-value). A particle collides with the surface if it is within one radius R distance from the center of a circle. In the collision the particle velocity is deflected as in Figure 8. When collision is detected just change the velocity from $\mathbf{v}$ to $\mathbf{v}_{new}$, which has the proper direction (observe that the particle doesn't loose any speed $|\mathbf{v}_{new}| = |\mathbf{v}|$, i.e the collision is elastic). To get the proper direction after the collision, assume that the incoming angle and the reflecting angle are identical[8], see Figure 8. The surface is treated as rigid (the circles are static and not affected by the collision).

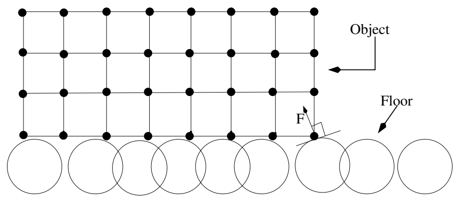


Figure 7: *A simple model of the surface. When a particle hits the surface we get a force F (doesn't have to be calculated). This force gives an impulse that change the direction of the particles velocity.*

c) The initial velocity should be $v = 5$ in the x-direction. This means that all particles have this velocity initially. The initial velocity in the y-direction is zero.

d) Calculate the energy quantities E_k, E_p, E_{spring} for the object during the simulation[9].

e) Calculate the velocity for the center of mass during the simulation. Does this velocity (the x-component) decrease linearly with time? If so, what does that say about the friction? What is the coefficient of friction? Repeat the simulation and analysis for initial velocities $v = 3$ and $v = 7$. Is the friction coefficient significantly velocity dependent?

[7]Compare this to the springs characteristic oscillation time scale $T \sim \sqrt{m/\kappa_s}$.

[8]To calculate the velocity after the collision it is smart to split your velocity into two parts, one parallel and one perpendicular to the normal of the circle at the point of the collision. Only the parallel part is changed at the collision. If you feel like you need help with this part, look in the hint section.

[9]It is not necessary to calculate the energy at ever single time-step. Calculate and store the energy at, say every 10:th or 100:th time-step.

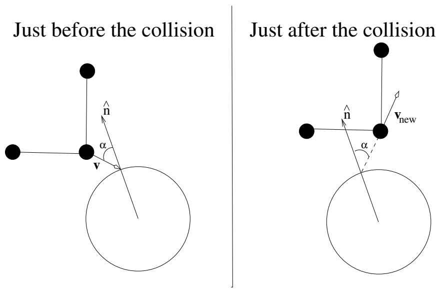


Figure 8: *How to get the proper velocity after the collision. $\hat{n}$ is the normal vector of the circle at the point of intersection. See also the hint section if you need further hints how to handle the collision.*

3.3.1 Results for the report

Your report should include:

- ✓ A plot of E_k, E_p, E_{spring} and E_{tot} versus time.
- ✓ A plot over velocity for the object versus time.
- ✓ The value of the friction coefficient for $v = 3$, $v = 5$, $v = 7$.
- ✓ Code files to run to the simulation (.m format).

3.4 Exercise 4: Modeling a sliding macroscopic 2D object

Before you scream, "Oh no, not another exercise", you should be aware of that you already have most of this program written. In fact if you wrote the previous program as recommended, you now just have to change a few numbers in the code.

This time you should make a simulation of a *macroscopic* object sliding over rough surface. How to go about that? A macroscopic object, like a cheese or a cube of jelly, consists of billions and billions of particles and even super computers cannot be used for that. What we can do is to change the model. We want to keep the elastic behavior at a macroscopic level need to drop the details at atomic level. We simply let the particles represent parts of the object and the adjust the springs so to model the macroscopic elasticity. Each spring is really extremely many mini-springs. How elastic the object gets, is decided by our choice of the stiffness of the springs in the model. Clearly we can reuse the code from the previous exercise – it is only the *meaning* of the springs and masses that has changed. Since we no longer have information on the microscopic level (the atom vibrations) we can let the springs be damped. This actually mimics the fact that in reality energy is converted into random atomic motion. Numerically, damping has the effect of stabilizing the simulation and as a result we can take larger time steps.

a) Use the same spring constant, mass etc. as above and set the damping constant to $\kappa_d = 5$. Repeat the simulation from Exercise 3 (initial velocity $v = 5$). This time the total energy will not be the same, since we no longer keep track of the heat in the object. The damping enables you to take larger time steps. Try[10] $\Delta t = 0.01$.

b) Run the simulation with $m = 1$, $g = 10$, $\kappa_s = 1000$, $\kappa_d = 50$, $L = 1$, $R = (3/4)L$, $v = 10$, $\Delta t = 0.005$.

c) **Optional.** Play around! See if you can get the object to turn over and start tumbeling. Try dropping it from above! Your imagination is the limit. Merry Christmas!

[10]Compare this to the springs characteristic time scale $T \sim \sqrt{m/\kappa_s}$.

3.4.1 Results for the report

Your report should include:

- ✓ A plot of E_k, E_p, E_{spring} and E_{tot} versus time.
- ✓ A plot over velocity for the object (center of mass) versus time.
- ✓ The value of the friction coefficient for $v = 5$.
- ✓ Code files if your solution is significantly different from Exercise 3.
- ✓ Optional. Snapshots from part c).

3.5 Additional programs for bonus points

Since this Matlab-exercise is a significant part of the whole course, it should give you a chance to help up your grade of the course.[11] Below follows three extra exercises. Bonus exercises 1-3 can give you one bonus point each. There is also one bonus point in award for keeping the *first* deadline. (Note that the bonus points helps you get higher grades, not pass the exam.) If you are not interested in bonus points, or do not want to spend time and effort you do not have to do these exercises. Furthermore, you do not have to do all three exercises, you can get bonus points from the exercises you do.

3.6 Bonus exercise 1: Modeling a 3D sliding macroscopic object

Do the same thing you did in 2D (Modeling a 2D sliding macroscopic object), but now for 3D. The code might be slow, so save the animation in an avi-file.[12]

3.7 Bonus exercise 2: A bouncing marble in a elastic net.

Let go of a marble (a solid sphere) and let it fall into an elastic net. Set up your net with springs similar to what you did for your 2D object before. At the border your net should be held fixed. Choose masses, gravity, and springs in a way that your animation is fun to look at. Hint: when the marble and the net collides, both the marble and net particle exchange momenta, or impulse. The impulse is directed along the normal to the marble surface at the collision point and the total momentum i conserved.

3.8 Bonus exercise 3: Particles in a box

In this exercise you should study the ideal gas law in air. The ideal gas law may be written in several different forms, the one most useful to us is the following,

$$PV = NkT \tag{35}$$

where P is the pressure of the gas, V the volume enclosing the gas, N the number of particles, k Boltzmanns constant and T the temperature of the gas. An ideal gas is a gas where the total volume of the particles is much smaller than the volume they occupy, air in room temperature and atmospheric pressure is almost an ideal gas.

The initial particle velocities comes from a normal distribution in velocity

$$f(\mathbf{v}) = c \exp\left(-m(v_x^2 + v_y^2 + v_z^2)/3kT\right)$$

where c is a normalization constant and $\mathbf{v} = (v_x, v_y, v_z)$.

[11]For informations regarding criteria for which grade you get see the information handed out in the beginning of the course.

[12]To save something as avi is very easy to do i Matlab, and is described in the hint section.

Your simulation should be able to handle particle-particle collisions and particle-wall collisions. All particle-particle collisions are perfectly elastic, which means that kinetic energy is conserved after a collision. The pressure of the gas may be calculated as,

$$P = \frac{F_{tot}}{A} = \frac{\frac{dp}{dt}}{A}. \tag{36}$$

where p is momentum and A is the area inside the box.

At the collision with a wall the perpendicular component of the velocity ($v_\perp$) changes sign, but the component parallel to the wall is the same, thus we have no friction and an elastic collision with the wall.

a) Check that the ideal gas law (35) holds for the air molecules in this very room (293 K, 1 atm) by calculating the pressure as in equation (36). You may assume air molecules are spheres with radius $4 \cdot 10^{-10}$ m and mass $4.8 \cdot 10^{-26}$ kg. Motivate why we can approximate all air molecules to have the same mass. Also plot the total kinetic energy of all the particles to make sure the collisions are elastic. **Hits**: For a given temperature and pressure, the concentration, N/V , of the gas is constant. Use 50 particles and calculate what volume you should use. Use SI units on all parameters to avoid prefix errors (1 length unit is 1 meter, 1 dt is one second and so on)

b) Put an initial temperature in you box and start with a cube with 64 times the volume of the cube in a). Let the box shrink with a (slow) constant speed until its the same as in a). How has the temperature changed? Plot temperature (average kinetic energy for the particles) as a function of the box size. Explain why the temperature is changing.

3.9 Hints

3.9.1 Suggested data types

Use the same (or very similar) notations for constants and variables as above and below. This simplifies when you are in need of help.

Particle vectors Use matrices `X` and `V`. This example shows how to use and refer to them.

```
N=4;
for a=1:N
particle_number=a;
    X(particle_number,1)=a*L;  % x-position
    X(particle_number,2)=0; % y-position
    V(particle_number,1)=V0;   % x-velocity
    V(particle_number,2)=0;     % y-velocity
end
```

The y-position of particle 3 is `X(3,2)`. The v_x-velocity of particle 3 is `V(3,1)`.

Spring It is practical to use a "struct" (as in C-language) to handle the information of the springs. This example of shows how this is done in the case of four particles and three springs. The "struct" variable here is `spring`. The information of the n:th spring is contained in `spring(n)`, which has five "elements" (the particle number of the particles connected by the spring, spring rest length and spring and damping coefficient).

```
NS=3; L=1; KS=10; KD=0.1;                    % number of springs, rest length etc
spring_number=0;
for m=1:NS                                            % loop over springs
```

```
    particle_number=m;
    spring_number=spring_number+1;
    spring(spring_number).from=particle_number;  % number of the ''from'' particle
    spring(spring_number).to=particle_number+1;  % number of the ''to'' particle
    spring(spring_number).length=L;                    % spring rest length
    spring(spring_number).KS=KS;                       % spring coefficient
    spring(spring_number).KD=KD;                      % damping coefficient
end
```

To get the particles (a and b) that are connected by spring number 2 we write

```
a=spring(2).from;
b=spring(2).to;
```

If we also want to know the spring rest length of spring number 2 it is given by `spring(2).length`. What is the position of particle a and b in this case? Given that position is stored in vector X, as described above, you need just refer to the correct element of this vector:

```
xa=X(spring(2).from,1);
ya=X(spring(2).from,2);
xb=X(spring(2).to,1);
yb=X(spring(2).to,2);
```

You can of course also store the spring information in an ordinary matrix (a $N_S \times 5$ matrix, to be specific). But a "struct" offers more flexibility and makes the code more clear to read.

3.9.2 How to set up the spring and particle system

It is very important that you are systematic when writing your code. One way to set up you model is:

1. Each particle is identified by its particle number (1 to N).
2. Assign positions to each particle. For particle n you should have `X(n,1)`$= x_n$ and `X(N,2)`$= y_n$.
3. Each spring is identified by a spring number (1 to N_S).
4. Declare the spring-particle connections. <u>One</u> way to do this is for a rectangular system, as in Figure 6, is:
    (a) Loop over all horizontal springs, row by row. For each spring, assign the particle numbers in `spring(m).from` and `spring(m).to` (or in a matrix, if you rather prefer that). Also assign the parameters for every specific spring (L, κ_s, κ_d).
    (b) Loop over all vertical springs, column by column. Assign particle numbers and parameters!.
    (c) Loop, row by row, over the diagonal springs pointing "up-and-right". Assign particle numbers and parameters! (Recall that these springs must be a factor $\sqrt{2}$ longer).
    (d) Loop, row by row, over the diagonal springs pointing "up-and-left". Assign particle numbers and parameters!

3.9.3 How to compute the force?

The most simple way is to loop over the springs, calculate the forces and add the contributions to the vector **F** .

At each time step of the simulation:

1. Set the vectors **F** to zero (with correct dimensions).
2. Loop over the springs
   (a) For each spring read off the "from" and "to" particle numbers (a and b) and spring parameters (L, κ_s, κ_d).
   (b) Get the position and velocity of particle a and b. Store position and velocity in temporary variables $\mathbf{x}_a$, $\mathbf{v}_a$ and $\mathbf{x}_b$, $\mathbf{v}_b$.
   (c) Compute the spring force $\mathbf{f}_{ab}$ (recall $\mathbf{f}_{ba} = -\mathbf{f}_{ab}$).
   (d) Add results from **c.** to **F** at the correct positions. Be aware that you *add* and *don't overwrite* (different springs may result in contributions at same position)!
3. Apply the force.

3.9.4 The change in particle velocity colliding with the surface

A few things needs to be considered when you should change the velocity direction. We can see this as a reflection where incoming and outgoing angle, with respect to the normal of the surface, are the same. (Figure 8 can be helpful to look at.)

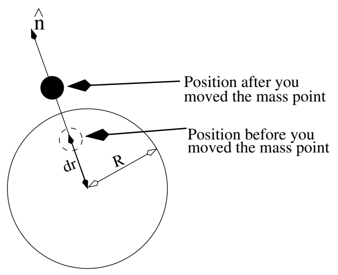
- We can divide our velocity into one part parallel to the normal vector $\hat{\mathbf{n}}$ and one perpendicular to $\hat{\mathbf{n}}$, thus we can write $\mathbf{v} = \mathbf{v}_\perp + \mathbf{v}_\|$. After the collision the parallel part parts is reversed, while the perpendicular part is the same:

  $\mathbf{v_{new}} = \mathbf{v}_\perp - \mathbf{v}_\|$. Combining these we get $\mathbf{v_{new}} = \mathbf{v} - \mathbf{2v}_\|$.

  $\mathbf{v}_\|$ can be computed as $\mathbf{v}_\| = (\mathbf{v} \cdot \hat{\mathbf{n}})\hat{\mathbf{n}}$.

  To get $\hat{\mathbf{n}}$ we must make a vector with direction from the center of the circle to the point of collision. The length of $\hat{\mathbf{n}}$ is of course 1. Let $\mathbf{dr} = \mathbf{x_a} - \mathbf{c}$ be a vector from the center of the circle (which is given by $\mathbf{c}$) to the collision point $\mathbf{x_a}$ (see figure 8). You get $\hat{\mathbf{n}}$ as

$$\hat{\mathbf{n}} = \frac{\mathbf{dr}}{|\mathbf{dr}|}$$



$\hat{n}$
Position after you moved the mass point
Position before you moved the mass point
dr
R

Figure 9: *Notations used in the text. Note that the difference between R and dr are very exaggerated.*

3.9.5 How to avoid to get a mass point inside a circle

A particle is under the ground surface (made up by circles) when you detect that it is closer to the center of a circles than the circle radius. Say that our mass point is at distance dr from the center of the circle with radius R. The condition for collision is then $dr \leq R$. (See Figure 9).

One way to simulate an instant collision is to move our particle out of the circle, so that it is the same distance outside the circle as it was inside. This way we can simulate an collision that occurred in between two time steps. That is, we should move it $2(R - dr)$ is the $\hat{n}$-direction.[13]

3.9.6 MATLAB

A short MATLAB guide is posted on the course home page. This contains a part about graphics that should be useful (source code can also be down load at the course home page). Here follows a few tips and hints:

Updating graphics. Avoid making new plot every time step, that will slow down the simulation and look ugly. Make the plot once, assign a *handle* and update the graphics using the handle and the command `set`. See the graphics example in the short MATLAB guide on the home page.

Point plot. You can use `plot(X;Y,'.','MarkerSize','5')`.

Plotting a circle. Use `rectangle` and set the *curvature* to form a circle.

Plotting a objects contour. Use `line`.

Slow code 1. Function calls in MATLAB slows down the code (so avoid them if computational speed is important).

Slow code 2. Building vectors takes time, so if you are storing the value of a quantity E every time step **don't do** it like `E=[E,value]` – **do** it like `E(n)=value` where you before the simulation loop created E with the correct size, like `E=zeros(number_time steps,1)`.

Slow code 3. Many loops take time (especially loops in loops) so try to avoid them, if possible. Try to apply the possibility in MATLAB to *vectorize* the code. The calculation

$$C = \sum_{n=1}^{N} \sin(A_n)\sqrt{1 + B_n^2}$$

can be done by looping n from 1 to N, calculating $\sin(A_n)\sqrt{1 + B_n^2}$ and add to the previous result (this is how you would do it in C, probably). The same result, but much faster, is obtained through the vectorized computation `sin(A(n))*sqrt(1+B.^2)'`. It is actually surprising how often it is possible to vectorize code.

3D graphics. Use `plot3` instead of plot. `line` can be used in 3D. Surfaces can be displayed using `surf` (you may want try some simple examples first using `meshgrid`, see the MATLAB help).

Record an animation. Animations are useful to display the simulation at a higher rate (if the computation is time consuming) or if you want to export the simulation result to a computer that doesn't have MATLAB installed. The following lines show how to record an animation.

```
% Initialize animation file before main loop
figure; % Create a new figure, if needed.
set(gcf,'Position',get(0,'Screensize')); % Maximize the window for quality
MOVE = VideoWriter('ExampleVideo.avi'); % Create an video struct, "MOVE.",
                                  % with the output file ExampleVideo.avi
MOVE.Quality = 100; % Set the quality (0-100)
MOVE.FrameRate = 25; % Set frames per seond to PAL standard (animation
speed)
```

[13]The very clever student now might argue that the point of collision is not **exactly** at the point where $\hat{n}$ intersect with the circle. This is true, but with small time steps this error is possible to neglect. However, if you want to correct it you are free to do so.

```
open(MOVE);
set(gca,'nextplot','replacechildren');
...
% Begin main time loop
... % Update positions etc. and plot
writeVideo(MOVE,getframe); % Get a snapshot of the active figure frame
...
% End of main time loop
...
close(MOVE); % Close and save the avi-file
```

The produced avi-file, `ExampleVideo.avi`, can be viewed with a standard multimedia player, such as *Windows Media Player* or *Real Player*.

Instructions for the report

The report is not a full report, but it should present your results and your interpretations of them.

1. Answer all the questions in the text. You can divide into sections per exercise.
2. Try to answer all the questions in each exercise using fluent text without writing down the actual questions.
3. Remember that velocity and position has an offset of half a time-step. Adjust that before calculating the total energy.
4. Have units on all axis in your figures. Make sure the readings at 0 match your initial conditions.
5. Explain carefully how you get the friction coefficient, both with equations an a figure with mass center velocity in the x-direction versus time.