

Lab assignment #6: Ordinary Differential Equations, Pt. I

Instructor: Nicolas Grisouard (nicolas.grisouard@utoronto.ca)

Due Friday, 23 October 2020, 5 pm

First, try to sign up for room 1. If it is full, sign up for room 2 (and ask your partner to join you there if they are in room 1). We have not needed a third room in a while, but can open one by request.

Room 1 Mohamed Shaaban ([Marker, m.shaaban@mail.utoronto.ca](mailto:m.shaaban@mail.utoronto.ca)),
URL: <https://gather.town/xg1R1VWH3wfgdCY/phy407-ms>
PWD: phy407-2020-MS

Room 2 Mikhail Schee (mikhail.schee@mail.utoronto.ca),
URL: <https://gather.town/ZdDgdR4j2gGZIqvc/MS-PHY407>
PWD: phy407-2020-MS

General Advice

- **Work with a partner!**
- This lab's topics revolve around computing solutions to ODEs, coupled or not, using fixed-steps Runge-Kutta and Verlet algorithms.
- Read this document and do its suggested readings to help with the pre-lectures and labs.
- Ask questions if you don't understand something in this background material: maybe we can explain things better, or maybe there are typos.
- Carefully check what you are supposed to hand in for grading in the section "Lab Instructions".
- Whether or not you are asked to hand in pseudocode, you **need** to strategize and pseudocode **before** you start coding. Writing code should be your last step, not your first step.
- Test your code as you go, **not** when it is finished. The easiest way to test code is with `print('')` statements. Print out values that you set or calculate to make sure they are what you think they are.
- Practice modularity. It is the concept of breaking up your code into pieces that as independent as possible from each other. That way, if anything goes wrong, you can test each piece independently.
- One way to practice modularity is to define external functions for repetitive tasks. An external function is a piece of code that looks like this:

```
def MyFunc(argument):
    """A header that explains the function
    INPUT:
    argument [float] is the angle in rad
    OUTPUT:
    res [float] is twice the argument"""
    res = 2.*argument
    return res
```

Place them in a separate file called e.g. `MyFunctions.py`, and call and use them in your answer files with:

```
import MyFunctions as f12 # make sure file is in same folder
ZehValyou = 4.
ZehDubble = f12.MyFunc(ZehValyou)
```

Computational Background

4th-order Runge-Kutta (RK4) In Lab #1, you have previously seen the Euler and Euler-Cromer methods for integrating systems of ODEs with initial values. A 4th-order method, which is probably the most widely used method for solving systems of ODEs, is the 4th order Runge-Kutta method (or RK4 for short).

The lecture only covers RK4 for one first-order ODE, but an extension to higher-order and coupled ODEs is relatively straightforward, as discussed in the text (§§ 8.2, 8.3). This method invokes calculating the RHS vector of $d\vec{r}/dt = \vec{f}(\vec{r}, t)$ at various intermediate points between steps. Full implementation requires coding the following 5 lines which are iterated over t values:

$$\vec{k}_1 = h\vec{f}(\vec{r}, t) \quad (1)$$

$$\vec{k}_2 = hf\left(\vec{r} + \frac{1}{2}\vec{k}_1, t + \frac{1}{2}h\right) \quad (2)$$

$$\vec{k}_3 = hf\left(\vec{r} + \frac{1}{2}\vec{k}_2, t + \frac{1}{2}h\right) \quad (3)$$

$$\vec{k}_4 = hf(\vec{r} + \vec{k}_3, t + h) \quad (4)$$

$$\vec{r}(t+h) = \vec{r}(t) + \frac{1}{6}(\vec{k}_1 + 2\vec{k}_2 + 2\vec{k}_3 + \vec{k}_4) \quad (5)$$

Verlet algorithm We will implement the Verlet method for a conservative system (i.e., the molecular dynamics simulation). The algorithm is described in the text on pp. 371-373. Starting from $a = F/m$, or equivalently,

$$\frac{d^2\vec{r}}{dt^2} = \vec{f}(\vec{r}, t) \quad (6)$$

and denoting the velocity as \vec{v} , the Verlet algorithm works as follows.

For the first step only,

$$\vec{v}\left(t + \frac{1}{2}h\right) = \vec{v}(t) + \frac{1}{2}h\vec{f}(\vec{r}(t), t), \quad (7)$$

then repeatedly apply the equations

$$\vec{r}(t+h) = \vec{r}(t) + h\vec{v}(t + \frac{1}{2}h), \quad (8)$$

$$\vec{k} = h\vec{f}(\vec{r}(t+h), t+h), \quad (9)$$

$$\vec{v}(t+h) = \vec{v}(t + \frac{1}{2}h) + \frac{1}{2}\vec{k}, \quad (10)$$

$$\vec{v}(t + \frac{3}{2}h) = \vec{v}(t + \frac{1}{2}h) + \vec{k}. \quad (11)$$

This last equation for $\vec{v}(t + \frac{3}{2}h)$ becomes $\vec{v}(t + \frac{1}{2}h)$ for the next iteration of Eqn. (8). Eqn. (10), which calculates $\vec{v}(t+h)$, is not required for the timestepping. It is useful if you want to diagnose the velocity at the same time as the $\vec{r}(t+h)$ for the purpose of, for example, calculating the energy.

Periodic Boundary Conditions One other thing we need for our molecular dynamics simulation is boundary conditions on the region we will simulate, i.e., the box surrounding the molecules. We don't really want the interactions of the molecules with the box to affect stuff but we also don't want the molecules moving really far apart from each other. This is usually handled with "periodic boundary conditions". Essentially, if a molecule reaches the boundary of the box, it doesn't interact with it, but as it leaves the box it just reappears on the opposite side of the box. This is a way of simulating a small subset of molecules embedded in a much larger region with molecules drifting in and out of the box of interest.

To do this, after updating the positions of all the particles at each time step, you need to check if any of them have exited the domain. If so, the particle should be moved back to the appropriate position inside the domain. The most compact way to do this is the following:

```
x = numpy.mod(x, Lx)
y = numpy.mod(y, Ly)
```

where x and y are the positions of a particular particle, and the domain is assumed to be a rectangle spanning $(0,0)$ and (L_x, L_y) .

The other change you need to make is a bit more subtle. Because of the periodic boundary conditions, the shortest distance between any two particles may not be along the line connecting them in the domain, but may actually be along a line passing through one of the domain boundaries. See Figure 1 for a visual explanation of this point. Here is a crude way to implement it at each time step, assuming that there are N particles in the domain:

- replicate the positions of the particles in the domain eight times. That is, pretend the domain is made of the 9 tiles represented in fig. 1, with $9 \times N$ particles.
- on a given particle in the innermost tile, compute the sum of the forces, due to the other $N-1$ particles in the innermost tile, *and* the sum of the forces, due to the $8N$ fictitious particles located in the other 8 tiles.

Strictly speaking, because of the periodic boundary conditions, there are actually infinitely many other particles that each particle is interacting with. However, because the Lennard-Jones potential decays very strongly with distance, it is sufficient to just take into account the 8 fictitious tiles surrounding the domain.

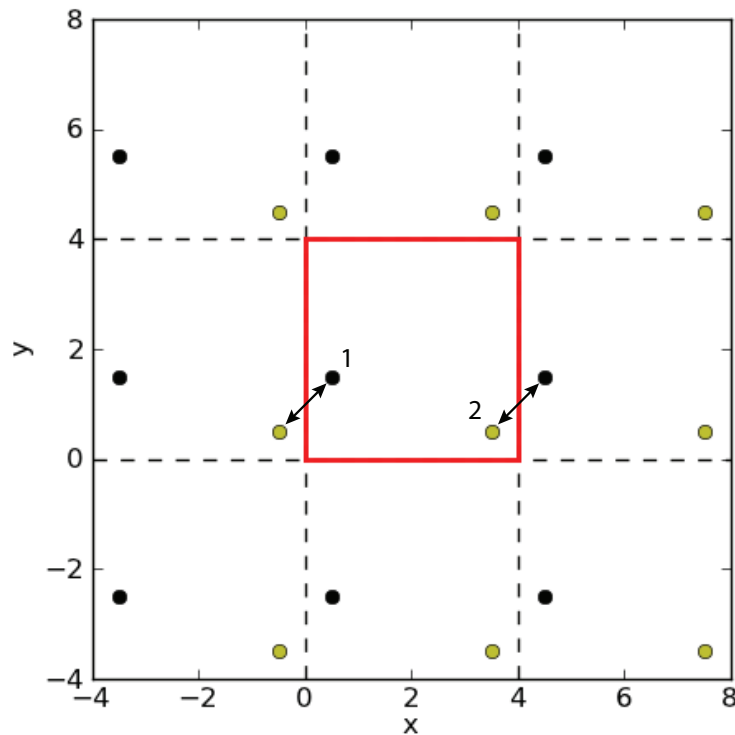


Figure 1: A representation of periodic boundary conditions for a molecular dynamics simulation with two particles. The red square indicates the domain of the simulation (assuming $L_x = 4$ and $L_y = 4$). There are two particles within the domain. Surrounding the computational domain are eight shifted “images” of the computational domain. For particle 1, the strongest interaction will be with the “image” version of particle 2, and vice-versa for particle 2. Thus, you must take into account these interactions when using periodic boundary conditions, not just the interactions across the actual computation domain.

Physics Background

Molecular Dynamics

- Most of the discussion below is from the text *Computational Physics* by Giordano and Nakanishi.
- Molecular dynamics techniques involve using a mechanical approach for studying multiparticle systems. We directly simulate the dynamics using the microscopic equations of motion.
- Essentially, you imagine a box containing a collection of molecules. These molecules move throughout the box as they collide with each other and with the walls of the box. To simulate this process, we employ Newton's second law to calculate the positions and velocities of all of the molecules as functions of time.
- The kinds of questions that can be addressed with this approach include the nature of the melting transition, the rate of equilibration after a sudden addition or loss of energy, and the rate at which molecules diffuse.
- The “molecules” in the system could be anything from atoms to droplets in an aerosol to particles in a flame to stars in a galaxy.
- In order to use Newton's second law, we need to calculate the forces acting on each particle. If we assume a conservative system (i.e. no outside forces), then the forces between any two particles can be given by an ‘interaction potential’ $V(r)$ where r is the separation of the particles. $V(r)$ depends on what kinds of particles are involved and the nature of the forces between them.
- For elements such as argon, the interactions at large distances are due to Van der Waals forces (i.e. a weak attraction arising from the transient electric dipole moments of the two atoms). This potential varies as r^{-6} and is attractive. When the atoms get close together, there is also a repulsive force due to the overlap of their electron clouds. This is commonly approximated by a term that varies as r^{-12} and is repulsive. Adding these 2 potentials yields the ‘Lennard-Jones potential’:

$$V(r) = 4\epsilon \left[\left(\frac{\sigma}{r} \right)^{12} - \left(\frac{\sigma}{r} \right)^6 \right], \quad (12)$$

where ϵ is the depth of the potential well.

In terms of energy, $V(r)$ is the potential energy of a system of *two* particles interacting at a distance r . The potential energy *per particle* is therefore $V(r)/2$.

Questions

1. Modeling space garbage (see Newman Exercise 8.8; 30% of the lab mark)

Do not do part (a), simply use the results as they are given in the text.

Do part (b), using RK4 to solve the equations.

Note: recall lab #1, and how to make sure that the axes have the same scale.

HAND IN THE PLOT AND THE CODE.**2. Molecular dynamics simulation, Part 1 (30% of the lab mark):**

- (a) Consider a 2-dimensional molecular dynamics simulation of $N = 2$ molecules under the influence of only the Lennard-Jones potential (Eqn. 12 above). Find expressions for the x and y components of the acceleration of one particle due to the presence of the other particle. You can use $\sigma = 1$ and $\epsilon = 1$ and also assume units where the masses of the molecules are 1.0.

NOTHING TO SUBMIT.

- (b) Write pseudocode and then a program that updates the position of the 2 particles using the Verlet method. Use a time step of $dt = 0.01$ and run the simulation for 100 time steps. Set all initial velocities to zero. Plot the trajectories of both particles on the same plot for the following sets of initial conditions:

- i. $\vec{r}_1 = [4, 4], \vec{r}_2 = [5.2, 4]$
- ii. $\vec{r}_1 = [4.5, 4], \vec{r}_2 = [5.2, 4]$
- iii. $\vec{r}_1 = [2, 3], \vec{r}_2 = [3.5, 4.4]$

Ensure that you are getting the proper behaviour in each case (e.g. repulsion when the particles are close, attraction when they are far).

HAND IN YOUR PSEUDOCODE, NOT YOUR CODE, AND HAND IN TRAJECTORIES FOR EACH OF THE THREE INITIAL CONDITIONS.

Note: In your trajectory plots, don't use solid lines to join the points, instead use a marker like a dot. You can do this in your plot command as follows:

`plot(x, y, 'o')`

- (c) One of the above initial conditions leads to oscillatory motion for both the particles. Which case is it, and **can you explain why?**

HAND IN A SHORT WRITTEN ANSWER.

Hint: think about energy conservation.

3. Molecular dynamics simulation, Part 2 (40% of the lab mark):

- (a) Now update your pseudocode and your code from Q2 so that it can simulate the motion of an arbitrary number of particles. Here, we will consider a simulation with $N = 16$. In this case, you have to calculate the net force on each particle ' i ' as the sum of the interaction forces between particle ' i ' and all other particles. As a first step, your code for this question should do the following:

- Set the initial positions of the particles to be evenly spaced in a square domain of side length L . You can adapt the code snippet below to set the initial conditions:

```
N = 16
Lx = 4.0
Ly = 4.0
```

```

dx = Lx/sqrt(N)
dy = Ly/sqrt(N)

x_grid = arange(dx/2, Lx, dx)

y_grid = arange(dy/2, Ly, dy)

xx_grid, yy_grid = meshgrid(x_grid, y_grid)

x_initial = xx_grid.flatten()
y_initial = yy_grid.flatten()

```

- Assign the initial velocities of all N particles to zero.
- For each time step, for each particle i :
 - Calculate the distance r_{ij} to particle j from particle i for all j .
 - Calculate the acceleration due to the total interaction potential for i .
 - Use the Verlet method to update the position and velocity of particle i .

Run your code for $T = 1000$ time steps with $dt = 0.01$. Create a plot showing the trajectories of all 16 particles.

HAND IN YOUR PSEUDOCODE, NOT YOUR CODE, AND A PLOT OF THE TRAJECTORIES OF ALL 16 PARTICLES.

- (b) Compute the energy of the system (adding both the kinetic and potential energy) at each time step. You should find that the energy is conserved to within about 1% as the simulation progresses. If energy is not being conserved, you know something is definitely wrong!

HAND IN CODE AND EITHER PRINTED OUTPUT OR A PLOT DEMONSTRATING ENERGY CONSERVATION.

- (c) Now implement periodic boundary conditions. You will need to make two changes to your code. Briefly, you need to make particles that exit the domain re-enter at the appropriate spot. You also need to make the changes to how the forces between particles are calculated, as indicated in the computational background.

Using the same initial conditions as above, now run your code with periodic boundary conditions. Run the simulation for $T = 1000$ time steps, again with $dt = 0.01$. Describe what is happening over the course of the simulation, and compare with the result of the previous question.

HAND IN YOUR CODE, A PLOT OF THE TRAJECTORY OF THE PARTICLES, AND THE WRITTEN EXPLANATIONS.