Scientific computing II, 2020

Final project: General instructions

Final project is a programming task somewhat larger than exercises. The purpose is to write a program that solves a certain problem in physics or computing. Choose one of the projects described in the following pages. You may also suggest a title of your own. Please, contact lecturer before you start working on it, so that we can ensure that it is suitable for this course. Note that you are not allowed to do the final project in groups or pairs. You should write a report of it with the order of 5 pages. The report should contain the following parts:

1. Introduction. Describe the problem and its background and motivation.
2. Methods. Describe (if applicable) the model, theoretical methods, and computational methods that you use.
3. Implementation of the methods. Describe the program written for the project: its data structures and procedures. If you use existing software libraries describe them. Give instructions on how to compile and use the program so that the reader is able to reproduce your results. *Do not attach the program source code to the report; it should be sent in a zip or tar archive (see below).*
4. Results. If applicable comment on the accuracy of the results and on the efficiency of the method/program.
5. Conclusions. Discuss for example the possible means to improve the methods and implementations.

Grading is is weighted roughly as (maximum 100 points)

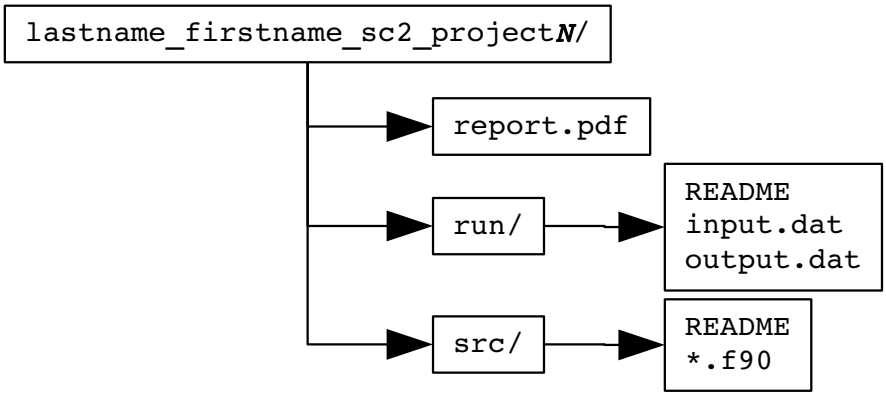
- Implementation: maximum 40 points
    - Does the code do what it's supposed to
    - Are all the features implemented
    - Error handling (this could be a part on it's own, maybe 5 points)
- Modularity: maximum 20 points
    - How is it divided into different files and modules
- Documentation: maximum 30 points
    - What does the program do
    - How do you run the program
    - What sort of inputs and outputs
- Ease of reading: -10-10 points

- Can be accomplished by comments and/or naming conventions
- *Minus for ugly code*

The report must be in PDF format. Put your name and student number along with the title of the project you chose on the front page of the report. You should also send the full source code, compilation instructions (possible `Makefile`) and sample input and output files. Submit your report and code to Moodle in one file as a `tar` of `zip` archive named as

`lastname_firstname_sc2_project`***`N`***`.tgz` or
`lastname_firstname_sc2_project`***`N`***`.zip`

where ***`N`*** is the project number (see below) and *containing a folder with the same name where all the files can be found*. This folder in turn, must contain the report (file `report.pdf`) and subfolders `src/` [source code and file `README` containing short compilation instructions] and `run/` [input file(s) and example output and file `README` containing short run instructions]. ***Failing to exactly comply with these instruction means lost points.***



Project topics[1]

1. Two-dimensional packing problem
2. Velocity filter simulation
3. Solving the heat conduction equation
4. Drunken sailor problem
5. Planetary motion simulator
6. Simple epidemic simulator
7. *Your own title* (contact lecturer before starting)

1 If you have a question it is a good idea to ask it on the course Moodle discussion forum. There may be other students having the same question. You may, however, email the teachers personally if you wish. Email: Firstname.Lastname@helsinki.fi.

1. Two-dimensional packing problem

The Monte Carlo method can be used in solving optimization problems. The idea is to generate new states of the system using the Metropolis algorithm while at the same time the system temperature is decreased. This is the so called simulated annealing (SA) method[2]. As the name says, it is analogous to the physical process of annealing where a piece of material is heated and then slowly cooled down to get rid of e.g. stresses in it.

SA is a stochastic optimization method that has been mostly applied to discrete optimization problems (so called combinatorial problems) although nothing prevents its application to continuous problems where the space to be searched is $\mathbb{R}^N$.

In SA the function to be minimized ($f(\mathbf{x})$, so called *cost function*) can be thought to correspond to the potential energy of a physical system. We also need a control parameter c, that corresponds to the temperature of a physical systems. The state of the system is described by the variable $\mathbf{x}$ which in this case can be discrete, too.

The principle of the SA is to generate states that obey the Maxwellian distribution

$$P(\mathbf{x}) \propto \exp(-f(\mathbf{x})/c) . \qquad (1)$$

One can show that when the control parameter (or temperature) c approaches zero, the Maxwellian distribution of Eq. (1) approaches the minimum configuration

$$P(\mathbf{x}) \propto \delta(\mathbf{x} - \mathbf{x}_{\mathbf{min}}) \qquad (2)$$

where $\mathbf{x}_{\mathbf{min}}$ is the configuration where the function f has its minimum value and δ is the Dirac delta function.

The SA algorithm itself is the following

1. Set the control parameter to its initial value: $c \leftarrow c_0$.
2. Set the system to its initial configuration: $\mathbf{x}_i \leftarrow \mathbf{x}_0$.
3. Set: $i \leftarrow 1$.
4. Change the system configuration: $\mathbf{x}_i \rightarrow \mathbf{x}_i + \delta\mathbf{x}$.
5. Calculate the change in the cost function: $\delta f = f(\mathbf{x}_i + \delta\mathbf{x}) - f(\mathbf{x}_i)$.
6. Generate a random number evenly distributed in unit interval: $\xi \in [0,1)$

2 For more information on SA go to the home page of the course *Monte Carlo Simulations in Physics* at https://moodle.helsinki.fi/course/view.php?id=11741 .

7. If $\xi < \exp(-\delta f / c)$ accept the new configuration: $\mathbf{x}_{i+1} \leftarrow \mathbf{x}_i + \delta \mathbf{x}$
8. Set $i \leftarrow i+1$. If $i \leq i_{\max}$ go to step 4. Otherwise go to next step.
9. Decrease the control parameters: e.g. $c \leftarrow \alpha c$, $0 < \alpha < 1$.
10. If $c < c_{\min}$ stop. Otherwise set $i \leftarrow 1$ and go to step 3.

Here the steps 4–8 are the normal Metropolis Monte Carlo (MMC) algorithm. Steps 5 and 6 provide the acceptance of the new configuration with probability given in Eq. (1). The loop over different values of the control parameter is in steps 9 and 10.

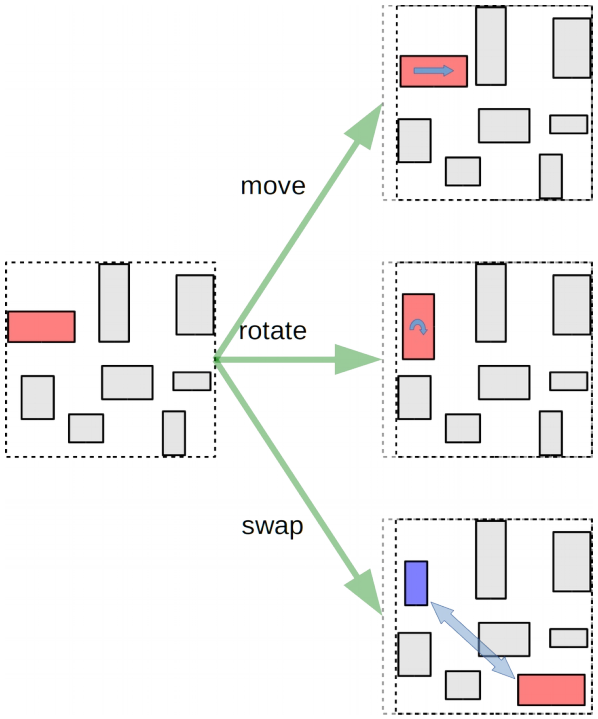
The purpose of this project is to use SA to solve a two-dimensional packing problem. You have an amount of rectangles on a plane and you should find a way to position them so that the total area covered by them (the footprint) is the smallest possible without any overlap of rectangles.

To generate new configurations you may use operations depicted below:

(a) Movement of a rectangle

(b) Rotation of a rectangle by $\pm 90°$

(c) Swap the positions of two rectangles.



In practice, the operation to generate a new configuration is such that one of the move types (a)-(c) is chosen in random (with equal probabilities). Then one (or in case of swap, two) rectangle is chosen randomly. Then the move is performed. If it produces overlap it is rejected. Otherwise the MMC steps 5– 7 of the algorithm are performed.

The cost function $f(\mathbf{x})$ in this case is the footprint of the rectangles show as a dashed line in the

above figure. The configuration $\mathbf{x}$ is a vector of the 2D positions of all rectangles.

Your task is to write a program that generates a random set of rectangles (the initial configuration) and minimizes the footprint of the rectangles. The initial value for the control parameter (temperature) c and the value of the cooling parameter α you must find out by experimenting. Start with a moderate number of rectangles (around ten). When you have found suitable values you can simulate larger systems (hundreds). Note that the simulations may need to run for thousands of steps.

For generation of random numbers you can use the Mersenne twister implemented in the file `mtfort90.f90`[3].

It is a good idea to store the rectangles in a derived data type containing the position, width and height. The procedures you need are at least

1. Generate a random sized rectangle.
2. Check whether a rectangle with a size $w \times h$ can be positioned in (x, y) without overlapping other rectangles $\{w_i, h_i; (x_i, y_i)\}$.
3. Calculate the footprint of a set of rectangles $\{w_i, h_i; (x_i, y_i)\}$.

You also have to figure out how to visualize the results. Program `xgraph` that should be included in the most common Linux distributions can plot each rectangle separately (without plotting lines connecting them) if you write the corners of each rectangle in the following fashion

```
move x1 y1
x2 y2
x3 y3
x4 y4
move x5 y5
x6 y6
x7 y7
x8 y8
…
```

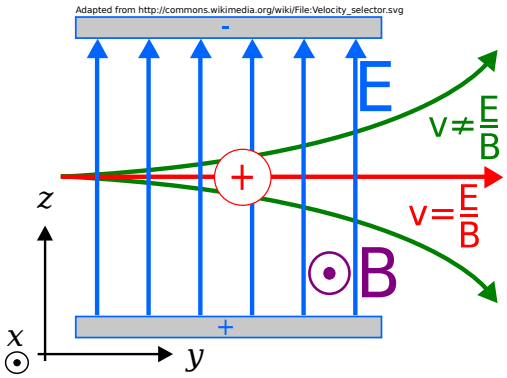
Here the keyword `move` 'raises the pen' and just moves to the point given after it.

In your report, show two to three examples of initial and final configurations of a set of rectangles. Plot also the footprint of these systems as a function of the simulation step.

3 http://www.courses.physics.helsinki.fi/fys/tilaII/files/mtfort90.f90 . An example main program is in http://www.courses.physics.helsinki.fi/fys/tilaII/files/mtfort90_test.f90 .

2. Velocity filter simulation

Velocity selector or a Wien filter[4] has perpendicular electric and magnetic fields (see the figure on the right):



$$\mathbf{E} = E\,\mathbf{k} \qquad (1)$$

$$\mathbf{B} = b\,\mathbf{i} \qquad (2)$$

$$\mathbf{v_0} = v_0\,\mathbf{j}\,. \qquad (3)$$

Only those charged particles that have the velocity

$$v = \frac{E}{B} \qquad (4)$$

pass straight through the filter, other particle bend up or down depending on their velocity and the sign of the charge.

Part 1

Write a program in Fortran that calculates the motion of a charged particle throught a box having homogeneous magnetic and electric fields as shown in the above figure. Particles enter the filter at the center of the left end. Only those particles, which move in chosen speed should get through the system, while others should hit a wall of the box. Simulation should stop (or start with a new particle if there are particles left) when particle hits the wall or gets out from the other end. You should limit your program to Newtonian motion and the only force affecting the particles' trajectories is the Lorentz force

$$\mathbf{F} = q\left(\mathbf{E} + \mathbf{v} \times \mathbf{B}\right). \qquad (5)$$

The program should be able to complete the following tasks:

1. Read in the values of following parameters from a file:
    - The total number of particles to be simulated
    - A number to indicate which particle is which (1,2,3,4,5...)
    - Components of initial velocities of the objects, v_x, v_y and v_z
    - Masses of the objects
    - Charges of the objects

Note: You can choose a format of input file yourself, but it should be well documented so users can

4 http://en.wikipedia.org/wiki/Wien_filter

easily produce their own input files for your program. Your program should be able to read a file of arbitrary size.

2. Read in from command line [i.e. using `get_command_argument()`]:
    - Size of the time step
    - Magnetic field strength (you may choose to read 1 or 3 components)
    - Electric field strength (you may choose to read 1 or 3 components)
3. Write out output values to a file:
    - The number of objects that got through
    - Numbers to indicate which particles got through (e.g. 1,4,6,7...). These should match the indication numbers that were read from the file.
    - components of the final velocities for the particles that got through, v_x, v_y and v_z
    - x, y and z components of the final positions of the particles that got through
    - Masses of the objects that got through
    - Charges of the objects that got through.
4. Calculate the force affecting the objects from the Lorentz force of Eq. (5). From Newton's second law calculate the acceleration of the objects.
5. Simulate the motion of the particle using a discrete time step Δt. For the numerical integration of the equations of motion you may use for example the velocity Verlet algorithm or more simplified Euler method (see below). Do this to each object in your input file and after simulation (if the particle got through) save the required parameters to the output file so you get a list of particles that got through with their corresponding attributes (place, velocity, mass and charge).
   Note: You can implement a small hole in the other end of the box such as only those particles that hit the hole would go through a filter.

Notes:

- Use modules and functions/subroutines in a logical manner. Don't put all your code in one file.
- Comment and indent your code in a logical manner so your code is easy to read.
- Document clearly the units you are using for input and output values.
- Euler method of integration of the equations of motion is sufficient for this problem. Positions and velocities of particles at the next time step ($x_{t+\Delta t}$, $v_{t+\Delta t}$) can be calculated from those at the current step (x_i, v_i):

  $$a_t = F(v_t)/m$$

$v_{t+\Delta t} = v_t + a_t \Delta t$
$x_{t+\Delta t} = x_t + v_{t+\Delta t} \Delta t$ (similarly for $y_{t+\Delta t} = y_t + \ldots$ etc.)

Note that you have to choose the timestep Δt short enough so that the integration error does not grow too large. You can test this by setting $\mathbf{E} = 0$. In this case the particle trajectory should be a circle. If your timestep is too large you get a spiral instead of a circle. Euler algorithm is not the best one for this job (it's not the best for anything). If you are interested in better behaving algorithms you may check out the so called Boris pusher.[5]

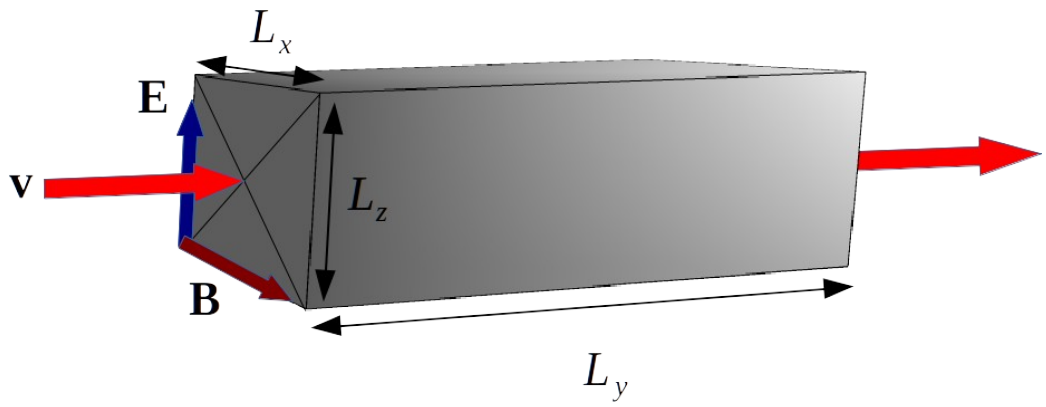
Part 2: Test your program

Create input files to test your program. Input files should include the number of particles in it and the required values for each particle (see part 1). For every task below provide suitable input files.

1. Choose a particle of certain mass and charge. Find the velocity this particle need to have to go through the filter, which has the following parameters:

$L_y = 7.6$ cm (box length)
$L_x = L_z = 1.9$ cm (box width)
$E = 12$ kV/m
$B = 0.10$ T.

2. Test your code varying the charge and the mass of the particles (masses can be for example multiples of proton mass). Test your code using at least three different masses and two different charges. Do bigger particles get through easier (velocities farther away from the chosen velocity)? How does the charge affect the results?



Part 3: Visualize the trajectories.

In the Part 2, you've already researched what kind of particles would go through the filter described in task 1. Let's now visualize the trajectories.

1. Create an input file with two particles: the one that would go through and another one that wouldn't.

2. Write an additional subroutine that creates a separate output file for each particle with particle's

5 See e.g. https://www.particleincell.com/2011/vxb-rotation/

position (x,y,z) at every time step. Output files should have `.xyz` extension[6], e.g. `trajectory_1.xyz` and `trajectory_2.xyz`. First two lines of the file should have an information about the number of time steps and the box size. Example:

```
<number of time steps N>
# boxsize <Lx> <Ly> <Lz>
x1 y1 z1
…
xN yN zN
```

3. Visualize your trajectory files with any visualization tool that can read .xyz format. We recommend Ovito[7]. Add pictures of trajectories to your report.

4. *Optional(extra points)*: Study the effect of the sign of the charge on trajectories. Choose a particle of certain mass, charge and velocity that doesn't go through the filter(you can use the results of the previous problems). Visualize it's trajectory. Change the sign of particle's charge. Does the trajectory change? Provide two figures of the trajectories: with + and – sign of the charge.

6 https://en.wikipedia.org/wiki/XYZ_file_format
7 http://www.ovito.org

3. Solving the heat conduction equation

Heat conduction equation in two dimensions is

$$\frac{\partial T}{\partial t}=\alpha\left[\frac{\partial^2 T}{\partial x^2}+\frac{\partial^2 T}{\partial y^2}\right],\quad T=T(x,y,t),$$

where $\alpha>0$ is the heat conduction coefficient. This can be discretized in a lattice so that

$$x_i=hi,\ y_i=hi,\ i=1..N.$$

We can simplify the equation by setting $h=1$, and $\alpha=1$. One can show that time dependence of the temperature at point (x_i,y_j) can be calculated as [$T_{i,j}(k)=T(x_i,y_j,t_k)$]

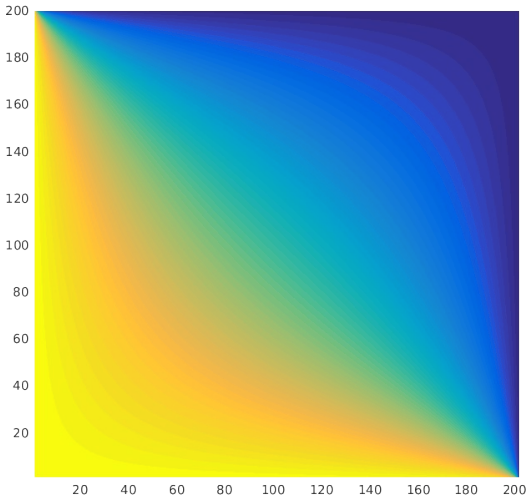
$$T_{i,j}(k+1)=T_{i,j}(k)-\gamma\left[4T_{i,j}(k)-T_{i+1,j}(k)-T_{i-1,j}(k)-T_{i,j+1}(k)-T_{i,j-1}(k)\right],$$

where $\gamma<1$, is related to the time step of the iteration.

Your purpose is by iterating the above formula over many time steps to reach a stationary state where the temperature in the grid points does not change anymore.

The temperatures of the four sides of the grid are set in the beginning and remain constant during the simulation. The initial temperature of the inner area, value of the constant γ, size of the grid N and number of time steps are also set in the beginning. The program should read the values from a file.

The program should print the temperature values at the end of the iteration in such a format that can be read in as a matrix by `matlab`/`octave`. You can plot the results for example with `matlab`/`octave` using the `surf` function which eventually should resemble the figure shown below. Examine how different values of γ, size of the grid N and number of time steps affect the simulation.



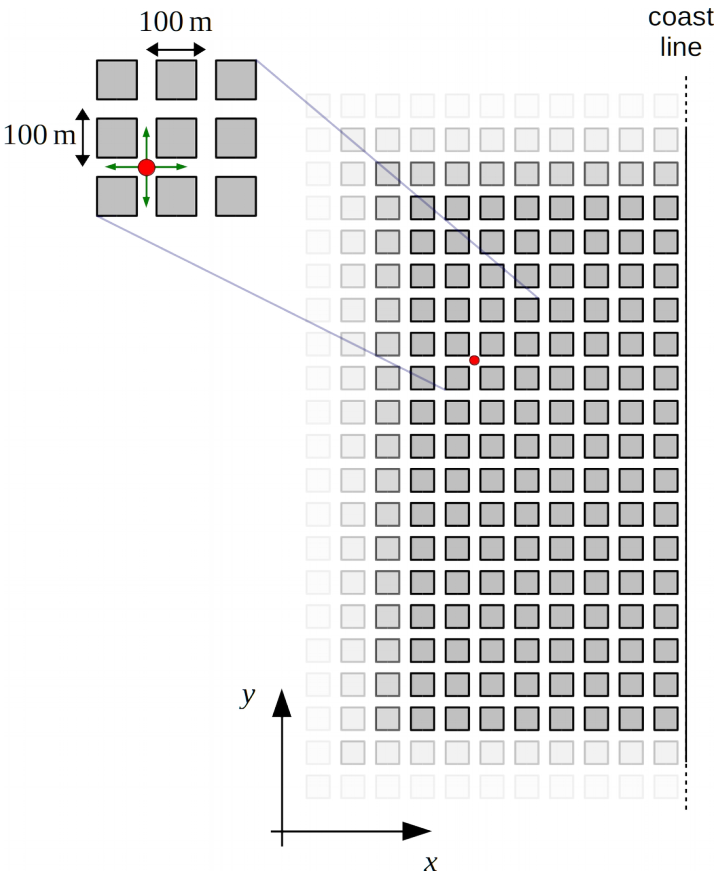
4. Drunken sailor problem

Drunken sailor is random walking from bar to the port in an infinite city with an infinite coast line. City consists of square blocks with 100 m sides (along x - and y -axes). Coast line is straight along the y -axis. Sailor walks at speed of 100 m/min and chooses her walking direction at random at every crossroad. Sailor finds her ship and her mates if she gets to the shore at any point. The ship leaves in the morning (in 10 hours) and the sailor has a finite life span and dies of old age if she doesn't get to the shore in 50 years.

Questions:

1. If the bar is 10 blocks away, how likely it is that
   a) The sailor gets to the shore on time before the ship leaves?
   b) The sailor dies of old age before getting to the shore?

Use large enough statistics to check that your results are accurate.



Having learned from her past mishaps with rum, this time, when the sailor is thrown out of the bar, she starts to leave a trail of empty bottles in her wake, so that she can avoid crossing her old path.

2. How does it affect the sailor's chance to get to the ship before morning, when she's doing self-avoiding walk (SAW) through the city?
   a) Modify your program for SAW and test it by plotting some example trajectories. Unfortunately there is a big flaw in her plan, as self-avoiding walkers have a big risk of walking into a dead end (imagine a snake in 'matopeli'). If the sailor gets completely blocked by her old path, we can assume that she just gives up trying to get to the ship, sits down, and entertains herself with leftover rum for the rest of the night.
   b) Calculate what is the probability this time to get to the shore in 10 hours starting again 10 blocks away. Now the walk also ends in failure if the sailor gets stuck in a dead-end. Report also the probability of this happening.
   Note that this time you have to record the walk into an array. You can just create a large-enough 2D array based on how far the sailor can move in 10 hours and the bar-shore distance.

Program requirements:

Input parameters for the program (given as command line parameters) are

1. Number of random walks to simulate
2. Parameter to tell whether to print the trajectories of all walks to a file.
3. Parameter telling whether the sailor is doing normal random walk or SAW.

Output of the program should consist of

1. Distance walked in km (average, minimum, maximum)
2. Time walked in relevant units, e.g. 2 years, 121 days, 3 hours and 44 minutes (average, minimum, maximum). Print only units that are needed!
3. Fraction of walkers who got to the ship on time, fraction of walkers who died before getting to the shore (normal random walk only), fraction of walkers who ended up in a dead end (SAW only)
4. Trajectory file (if needed) with coast as the y -axis (`xgraph` readable; see the hint on page 5)

You need a good random number generator for this project. An example is in file `mtfort90.f90`[8]. It contains module `mtmod` including function `igrnd(` n_0 `,` n_1 `)` that produces an random integer in the interval $[n_0, n_1]$ (limits included).

8 http://www.courses.physics.helsinki.fi/fys/tilaII/files/mtfort90.f90

5. Planetary motion simulator

The object of this final project is to produce a program that can simulate a planetary system. The program will then be used to simulate the Solar System. The simulation can be done using the rather simple Velocity Verlet algorithm[9] where the particle positions, velocities and accelerations at the next time step (x_{i+1} , v_{i+1} , a_{i+1}) are calculated from those at the current step (x_i , v_i , a_i):

$$x_{i+1} = x_i + v_i \Delta t + \frac{1}{2} a_i \Delta t^2 \quad \text{(similarly for } y_{i+1} = y_i + \text{... etc.)}$$
$$a_{i+1} = F(x_{i+1})/m$$
$$v_{i+1} = v_i + \frac{1}{2}(a_i + a_{i+1})\Delta t \ .$$

Write a program in Fortran that calculates the motion of N celestial objects in each others gravity fields. You should limit your program to Newtonian motion[10] and the only force you are required to use is the gravitational force.

In the most sensible coordinate system and in the absence of external forces your system (the objects) should only move relative to each other. Document clearly the units you are using for input values and output values. Always output the object positions at the beginning and end of the simulation.

The program should be able to complete the following tasks. (Hint: these tasks can be implemented one at a time and tested thoroughly before moving on to the next task.)

1. Read from a file the values for the necessary parameters (at least the ones listed here):
   (a) The number of objects
   (b) Masses of the objects
   (c) Initial positions and velocities of the objects
      i. They can be read in as vectors, or separate x -, y - and z -components
   (d) To run the simulation you need two of the following (the third can be determined from the others)
      i. length of timestep
      ii. total length of simulation
      iii. number of steps
   (e) Your documentation should specify clearly how the input file should be formatted for the program to be able to read it

9 https://en.wikipedia.org/wiki/Verlet_integration#Velocity_Verlet
10 https://en.wikipedia.org/wiki/Newton's_law_of_universal_gravitation

2. It is good to every now and then check how the simulation is proceeding. You can do this by printing to the screen info about it. This should be done also at the beginning and end of the simulation. During the simulation these should be printed every m steps. m should be read from the input file. At least these should be included:

   (a) The number of objects

   (b) Time and number of steps from the beginning of the simulation

   (c) Number of iteration steps written to file

   (d) Current positions of the objects (with some way to know which object is which)

3. Calculate the forces affecting the objects using Newton's law of universal gravitation. From the forces calculate the acceleration for each object.

4. Simulate the motion of the objects using the velocity Verlet algorithm and the input parameters read from the file. After every iteration save the positions of the objects, appending them to the same output file.

5. When doing actual simulations it is not sensible to save the positions at every time step, because the amount of data would be huge and you really don't need all of it. For instance the amount of data needed to make a series of sensible plots (or an animation of those plots) is a few tens of positions (maybe 30-50) per orbit. For the simulation to be accurate at all, you need the actual algorithm to calculate the positions much more often, so you only save the positions after a certain amount of steps (every 1000 steps for instance). Change the program so that it saves the positions only every *k* steps. *k* is read from the input file.

Data visualization

It is also important to be able to visualize the results of a simulation. Plots (and animations) of the data can be done pretty simply using for instance Matlab/Octave or Python. Produce plots (or if you can, animations) of the positions of the objects during the simulation. You can also use the same visualization program Ovito as is instructed in the project ‘Velocity filter’.

Once you've gotten your program ready, you should run simulations of the following scenarios:

1. The biggest planet in the Solar System is Jupiter. Use real values for the input parameters listed earlier to simulate the relative motion of these two objects (Sun and Jupiter). Adjust your time step so that you get a good value for the time it takes for Jupiter to orbit once around the Sun (Jupiter's orbital period). If the time step is too large you might not get a good approximation for the orbital period of Jupiter, though decreasing the time step will not indefinitely increase the accuracy of the simulation.

   (a) What sort of time steps did you try at first? How big a time step can you use and still get a reasonable approximation for the orbital period. Is this approximation shorter or longer than Jupiter's real orbital period.

   (b) When you simulate the system for a time of one orbital period of Jupiter has your “Jupiter” moved over or under one full period of motion? By how much? You can

estimate this approximately from the 2D animation image. Report the time step you used for this approximation.

(c) What is the approximate radius of motion and period of motion for the Sun?

2. Your program can now be used to approximate some common time periods that are important for humans. In each case use the two most appropriate Solar System objects for the following time period approximations and report the values you get. What are appropriate values for the simulation time step in these cases:

   (a) a year

   (b) a month.

3. So far you have experimented with only two bodies. Now let's try the whole Solar System, specifically the Sun and planets (and, no, Pluto is not a planet!). You don't have to use the current actual positions of the planets, but you need to get the distances from the Sun (orbit radius) and initial velocities correct, otherwise you might not get a stable system and planets will be flying all over the place.

   (a) If you use the same time step as in the first scenario, how good are the approximations of the orbital periods of the inner planets?

   (b) How much (if at all) do you need to shorten the time step to get a good approximation for Mercury?

6. Simple epidemic simulator

In this project we take a collection of random walkers and see how an epidemic develops in a group of individuals.

Part 1

Generate a population of random walkers that walk in a $N \times N$ grid. They should have following properties:

1. Their positions (site) can be expressed as two integers (x,y).
2. Each time step they have equal probability to move one step left, right, up or down from their current location [i.e either $(x+1,y)$, $(x-1,y)$, $(x,y+1)$ or $(x,y-1)$].
3. If they move out from the boundary of the $N \times N$ grid, they emerge back from the opposite side of the grid (periodic boundaries; *hint: use Fortran* `mod` *function.*
4. Are always in one of the three states: healthy, sick or immune

Now you should have a program that has M random walkers randomly walking inside a periodic $N \times N$ grid.

Part 2

Implement the disease.

At every timestep:

1. Move every random walker one step to a random direction (random direction for each agent individually).
2. Every healthy random walker that is in a same site with at least one sick random walker, changes their status to sick with a probability p_c.
3. Every sick random walker heals (changes its status to immune) with a probability p_h.

Part 3

Test your code.

1. Test how population density (amount of random walkers / grid area) affects the spread of the disease.
2. Test how p_c affects the spreading of the disease.
3. How does vaccinating the starting population affect the spread of the disease (start with some portion of the random walkers already in immune status).

Time units are arbitrary so put the p_h to something constant such as 0.01. Start with a few random walkers with sick status.

Part 4:

Visualize your results.

Plot at least these:

1. How many are sick at each timestep.
2. How many are immune at each time step (also indicates total amount of contagions)

Also visualize the configuration of the grid. You can use the xyz file format:

1st line: number of random walkers
2nd line: comment (any text; for example time step and grid size)
3rd … *M*+2 line: random walker information: position (x,y), status (healthy, sick immune coded as an integer)

One file can have multiple time steps, just repeat the previous format (note that amount of walkers comes again before the comment line for the next timestep).

Example for two timesteps for 5 walkers:

```
5
#Step 1
1 1.0 2.0  1
1 3.0 3.0  1
1 5.0 2.0  2
1 2.0 2.0  2
2 1.0 2.0  1
5
#Step 2
2 2.0 2.0  1
1 2.0 3.0  3
1 5.0 3.0  2
1 2.0 1.0  2
2 1.0 3.0  1
```

You can also add a third component for the position (e.g. add 0.01 for every walker) to see that multiple walkers are indeed in the same point in the grid (but would now appear on top of each other).

We recommend `ovito`[11]for visualizing these xyz files.

11 `ovito.org`. Download Ovito Basic