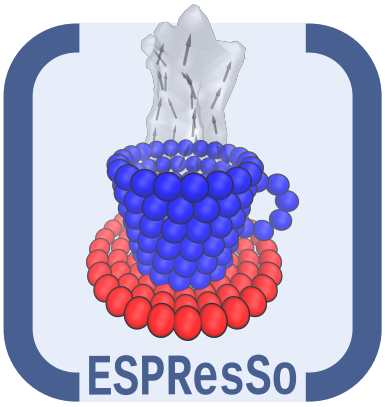


Tutorial 2: A simple charged system*

January 26, 2018

Contents

1 Introduction 2

2 Basic set up 2

3 Running the simulation 4

4 Writing out data 5

5 Analysis 6

6 Further ideas 8

7 Accelerating the simulation using a GPU 9

8 Using the MEMD algorithm 9

9 Partially periodic boundary conditions 10

*For ESPResSo unknown

1 Introduction

This tutorial introduces some of the features of ESPResSo by constructing step by step a simulation script for a simple salt crystal. We cannot give a full Tcl tutorial here; however, most of the constructs should be self–explanatory. We also assume that the reader is familiar with the basic concepts of a MD simulation here. The code pieces can be copied step by step into a file, which then can be run using `Espresso <file>` from the ESPResSo source directory.

2 Basic set up

Our script starts with setting up the initial configuration. Most conveniently, one would like to specify the density and the number of particles of the system as parameters:

```
# Simulation box setup
set n_part 200
set density 0.7
set box_l [expr pow($n_part/$density,1./3.)]
```

These variables do not change anything in the simulation engine, but are just standard Tcl variables; they are used to increase the readability and flexibility of the script. The box length is not a parameter of this simulation; it is calculated from the number of particles and the system density. This allows to change the parameters later easily, e. g. to simulate a bigger system.

The parameters of the simulation engine are modified using the `setmd` command. For example

```
setmd box_l $box_l $box_l $box_l
setmd periodic 1 1 1
```

defines a cubic simulation box of size `box_l`, and periodic boundary conditions in all spatial dimensions. We now fill this simulation box with particles

```
# Particle creation
set q 1
set type 0
for {set i 0} { $i < $n_part } {incr i} {
  set posx [expr $box_l*[t_random]]
  set posy [expr $box_l*[t_random]]
  set posz [expr $box_l*[t_random]]
  set q [expr -$q];
  set type [expr 1-$type]
  part $i pos $posx $posy $posz q $q type $type
}
```

This loop adds `n_part` particles at random positions, one by one. In this construct, only two commands are not standard Tcl commands: the random number generator `t_random` and the `part` command, which is used to specify particle properties, here the position, the charge `q` and the type. In ESPResSo the particle type is just an integer

number which allows to group particles; it does not imply any physical parameters. Here we alternate between creating positive charges of type 0 and negative charges of type 1.

Now we define the ensemble that we will be simulating. This is done using the `thermostat` command. We also set some integration scheme parameters:

```
# Thermostat setup
setmd time_step 0.01
setmd skin 0.3
set temp 1
set gamma 1
thermostat langevin $temp $gamma
```

This switches on the Langevin thermostat for the NVT ensemble, with temperature `temp` and friction coefficient `gamma`. The skin depth `skin` is a parameter for the link–cell system which tunes its performance, but shall not be discussed here.

Before we can really start the simulation, we have to specify the interactions between our particles. We use a simple, purely repulsive Lennard-Jones interaction to model a hard core repulsion. Additionally the charges interact via the Coulomb potential:

```
# Interaction setup
set sig 1.0
set cut [expr 1.12246*$sig]
set eps 1.0
set shift [expr 0.25*$eps]
inter 0 0 lennard-jones $eps $sig $cut $shift 0
inter 1 0 lennard-jones $eps $sig $cut $shift 0
inter 1 1 lennard-jones $eps $sig $cut $shift 0
puts [inter coulomb 10.0 p3m tunev2 accuracy 1e-3 mesh 32]
```

The first three `inter` commands instruct ESPResSo to use the same purely repulsive Lennard–Jones potential for the interaction between all combinations of the two particle types 0 and 1; by using different parameters for different combinations, one could simulate differently sized particles. The last line sets the Bjerrum length to the value 10, and then instructs ESPResSo to use the P^3M method for the Coulombic interaction and to try to find suitable parameters for an rms force error below 10^{-3}, with a fixed mesh size of 32. The mesh is fixed here to speed up the tuning; for a real simulation, one will also tune this parameter. The `puts` statement will print the parameters and timings that the tuning found to the screen. Tuning takes some time; if you run many simulations with similar parameters, you might want to save and reload the P^3M parameters. You can obtain the P^3M parameters by `inter coulomb`:

```
set p3m_params [inter coulomb]
puts $p3m_params
```

They are printed in a format suitable to feed them back to the `inter` command:

```
foreach f $prm_params {
  eval inter $f
}
```

3 Running the simulation

Now we can integrate the particle trajectories for a couple of time steps:

```
# Main integration loop
set integ_steps 200
for { set i 0 } { $i < 20 } { incr i } {
  set temp [expr [analyze energy kinetic]/((3/2.0)*$n_part)]
  puts "t=[setmd time] E=[analyze energy total], T=$temp"
  integrate $integ_steps
}
```

This code block is the primary simulation loop and runs 20×`integ_steps` MD steps. Every `integ_steps` time steps, the potential, electrostatic and kinetic energies are printed out. The latter one is printed as temperature, by rescaling by the number of degrees of freedom (3) multiplied by $1/2kT$. Note that energies are measured in kT, so that only the factor $1/2$ remains. Also note that $3/2$ should be written as 3/2.0; to ensure that Tcl does not perform integer division (resuting in 1 instead of 1.5).

Note, that in ESPResSo there are usually 3 translational degrees per particle. However, if `ROTATION` is compiled in, there are in addition 3 rotational degrees of freedom, which also contribute to the kinetic energy. You can get this number in the following way:

```
if { [regexp "ROTATION" [code_info]] } {
  set deg_free 6
} {
  set deg_free 3
}
```

or even

```
  set deg_free [degrees_of_freedom]
```

which does exactly the same.

Then all you need to do is to replace the hardcoded 3 by `$deg_free`.

However, if you run the simulation, it will still crash: ESPResSo complains about particle coordinates being out of range. The reason for this is simple: Due to the initial random setup, the overlap energy is around a million kT, which we first have to remove from the system. In ESPResSo, this is can be accelerated by limiting the maximum forces, i. e. modifying the Lennard–Jones force such that it is constant below a certain distance. Before the main integration loop, we therefore insert this equilibration loop:

```
# Warmup integration loop
set integ_steps 200
for {set cap 20} {$cap < 200} {incr cap 20} {
  inter forcecap $cap
  integrate $integ_steps
}
inter forcecap 0
```

This loop integrates the system with a force cap value of initially 20 and finally 200. The last command switches the force cap off again. With this equilibration, the simulation script runs fine. As a check that the equilibration loop works correctly, you

might want to copy the energy and force printing from the main integration loop. You can then observe how the temperature initially overshoots and is then relaxed to its target value by the thermostat.

4 Writing out data

For later analysis of the simulation results, one will usually like to write out simulation data to configuration files. For this purpose ESPResSo has commands to write simulation data to a Tcl stream in an easily parseable form. We add the following lines to the integration loop just after the `integrate` line in order to write out the system configuration files "config_0" through "config_19":

```
set f [open "config_$i" "w"]
blockfile $f write tclvariable {box_l density}
blockfile $f write variable box_l
blockfile $f write particles {id pos type}
close $f
```

The created files "config_..." are human–readable and look like

```
{tclvariable
    {box_l 6.58633756008}
    {density 0.7}
}
{variable  {box_l 6.58633756008 6.58633756008 6.58633756008} }
{particles {id pos type}
    {0 14.7658693713 29.5464807649 -17.5071728732 1}
    {1 26.702434508 -37.4986024417 114.617582522 0}
    [...]
}
```

As you can see, such a *blockfile* consists of several Tcl lists, which we call *blocks*, and it can store any data available from the simulation. Reading a configuration is done by the following simple script:

```
set f [open $filename "r"]
while { [blockfile $f read auto] != "eof" } {}
close $f
```

The `blockfile read auto` commands will set the Tcl variables `box_l` and `density` to the values specified in the file when encountering the `tclvariable` block, and set the box dimensions for the simulation when encountering the `variable` block. The particle positions and types of all 216 particles are restored when the `particles` block is read. Note that it is important to have the box dimensions set before reading the particles, to avoid problems with the periodic boundary conditions.

The blockfile mechanism is typically used for one of two main purposes, checkpointing and offline analysis. Checkpointing is useful for long-running simulations: If you have a simulation that runs for several days, you may want to have it periodically write its current state to a blockfile. That way, the simulation can be resumed if your computer crashes while the simulation is running. Offline analysis refers to analyzing physical

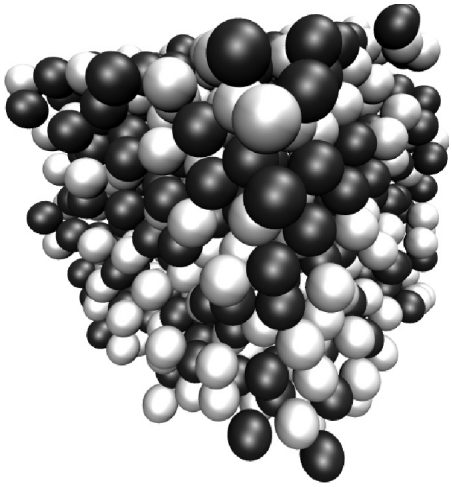


Figure 1: VMD Snapshot of the salt system

parameters of the simulation after the simulation has finished and will be discussed in the next section.

5 Analysis

Having written out blockfiles during the simulation, we can now investigate the system. As an example, we will create a second script which calculates the averaged radial distribution functions $g_{++}(r)$ and $g_{+-}(r)$. The radial distribution function for the current configuration can be calculated using the `analyze` command:

```
set rdf [analyze rdf 0 1 0.9 [expr $box_l/2] 100]
set rlist ""
set rdflist ""
foreach value [lindex $rdf 1] {
  lappend rlist [lindex $value 0]
  lappend rdflist [lindex $value 1]
}
```

The shown `analyze rdf` command returns the distribution function of particles of type 1 around particles of type 0 (i. e. of opposite charges) for radii between 0.9 and half the box length, subdivided into 100 bins. Changing the first two parameters to either "0 0" or "1 1" allows to determine the distribution of equal charges around each other. The result is a list of r and $g(r)$ pairs, which the following code snippet transposes up into a pair of lists, `rlist` and `rdflist`. To average over all configurations, we loop additionally over all blockfiles.

```
# Initialize the total RDF with zero
set cnt 0
for {set i 0} {$i < 100} {incr i} { lappend avg_rdf 0 }

foreach filename $argv {
  # Read the file
  set f [open $filename "r"]
  while {[blockfile $f read auto] != "eof" } {}
  close $f
  # Calculate the RDF
  set rdf [analyze rdf 0 1 0.9 [expr $box_l/2] 100]
  # Convert the list of (r,g) tuples into two lists
  set rlist ""
  set rdflist ""
  foreach value [lindex $rdf 1] {
    lappend rlist [lindex $value 0]
    lappend rdflist [lindex $value 1]
  }
  # Add this configuration's RDF to the total RDF
  set avg_rdf [vecadd $avg_rdf $rdflist]
  incr cnt
}
# Divide by the number of configurations to get the average
set avg_rdf [vecscale [expr 1.0/$cnt] $avg_rdf]
```

Initially, the sum of all $g(r)$, which is stored in `avg_rdf`, is set to 0. Then the loops over all configurations given by `argv`, calculates $g(r)$ for each configuration and adds up all the $g(r)$ in `avg_rdf`. Finally, this sum is normalized by dividing by the number of configurations. Note again the `1.0/\$cnt`; also here, this is necessary, since `1/\$cnt` is interpreted as an integer division, which would result in 0 for cnt > 1. `argv` is a predefined variable: it contains all the command line parameters. Therefore this script should be called like this:

```
Espresso <script> config_*
```

The printing of the calculated radial distribution functions is simple. Add to the end of the previous snippet the following lines:

```
set plot [open "rdf.data" "w"]
puts $plot "\# r rdf(r)"
foreach r $rlist rdf $avg_rdf { puts $plot "$r $rdf" }
close $plot
```

This instructs the Tcl interpreter to write the `avg_rdf` to the file `rdf.data` in gnuplot–compatible format. Fig. 2 shows the resulting radial distribution functions, averaged over 100 configurations. In addition, the distribution for a neutral system is given, which can be obtained from our simulation script by simply removing the command `inter coulomb ...` and therefore not turning on P^3M. To plot your own

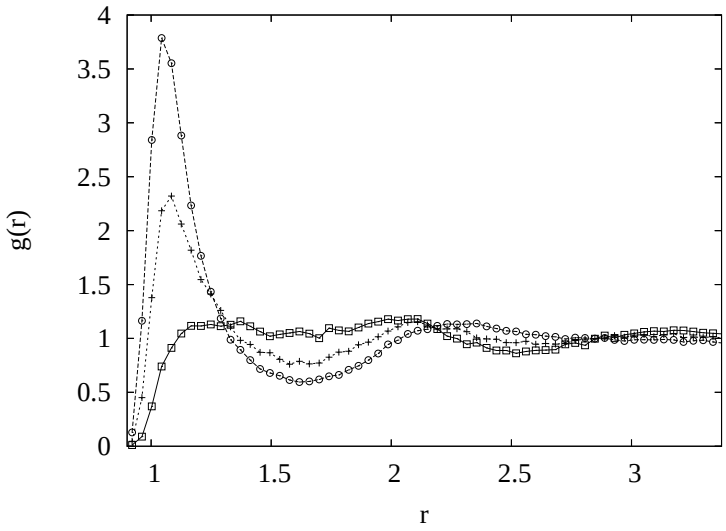


Figure 2: Radial distribution functions $g_{++}(r)$ between equal charges (rectangles) and $g_{+-}(r)$ for opposite charges (circles). The plus symbols denote $g(r)$ for an uncharged system.

results, run `gnuplot` in a Terminal window and type the following:

```
plot "rdf.data" using 1:2
```

If you have written out RDFs for multiple combinations of species, you can plot them into one graph like this:

```
plot "rdf00.data" using 1:2 w l title "g++", \
     "rdf11.data" using 1:2 w l title "g--", \
     "rdf01.data" using 1:2 w l title "g+-"
```

6 Further ideas

The code example given before is still quite simple, and the reader is encouraged to try to extend the example a little bit, e. g. by using differently sized particle, or changing the interactions. If something does not work, ESPResSo will give comprehensive error messages, which should make it easy to identify mistakes. For real simulations, the simulation scripts can extend over thousands of lines of code and contain automated adaption of parameters or online analysis, up to automatic generation of data plots. Parameters can be changed arbitrarily during the simulation process, as needed for e. g. simulated annealing. The possibility to perform non–standard simulations without the need of modifications to the simulation core was one of the main reasons why we decided to use a script language for controlling the simulation core.

7 Accelerating the simulation using a GPU

ESPResSo can make use of an Nvidia graphics processing unit (GPU) for accelerating simulations. Of course, we expect that results of the GPU-based implementation of P^3M to be exactly identical to those of the traditional CPU implementation. So copy the results from the previous task to a different file name (e.g. `rdf_p3m_cpu.data`) for comparison. Additionally, we'd like to find out exactly how much faster the GPU implementation is. So first wrap your previous simulation script with a time measurement, run it again and note down the execution time.

```
set starttime [clock seconds]

# previous script code goes here

set endtime [clock seconds]
puts "Simulation took [expr $endtime-$starttime] seconds"
```

To switch to the GPU-based implementation of P^3M, go to the line of your simulation script that contains `inter coulomb` and replace `p3m` with `p3m gpu`. Then run the simulation again and note down the execution time, which should be a lot shorter now. Plot the RDFs from both the CPU and the GPU simulation into one graph and check whether the results agree.

8 Using the MEMD algorithm

ESPResSo provides a variety of different electrostatics solvers. So far, we have been introduced to P^3M, but in this section we will try something new. A fairly recent addition to the family of electrostatics solvers is the "Maxwell Equations Molecular Dynamics" (MEMD) algorithm. To use it, two parameters have to be set: A mesh size, and a method parameter called `f_mass`, which can be estimated by a formula given in the ESPResSo user guide. In this example, a good estimate for the mesh size is 12, but this can be varied, affecting speed and accuracy of the algorithm.

The MEMD algorithm relies on a very specific spatial distribution of the particles across processors, and will therefore currently not work with Verlet lists. Since those are switched on by default, they will have to be turned off manually by adding the following line before the `setmd` commands:

```
cellsystem domain_decomposition -no_verlet_list
```

Besides this, you will have to replace the initialization of the P^3M interaction (`inter coulomb`...) with an appropriate one for MEMD:

```
set memd_mesh 12
set f_mass [expr 100.0*pow( ([setmd time_step]*$memd_mesh/$box_l),2.0)]
inter coulomb 10.0 memd $f_mass $memd_mesh
```

In the example script included within the ESPResSo code, there is already an `if`-construct provided and you can switch between the methods at the very top of the script.

You can copy the results from your completed tasks so far to a different filename (e.g. `rdf_p3m.data`) for comparison. Then, just run the simulation and analysis again, and compare the speed and resulting radial distribution function of the two methods.

9 Partially periodic boundary conditions

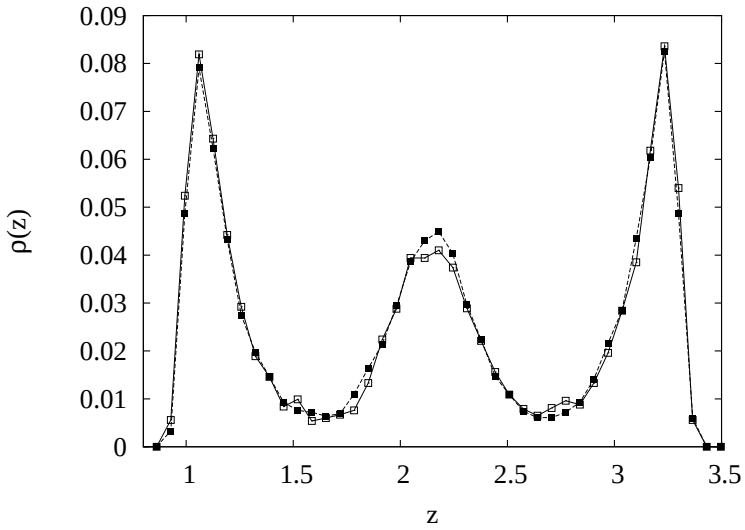


Figure 3: Distribution of positive charges $\rho_+(z)$ (open squares) and of negative charges $\rho_-(z)$ (closed squares) under confinement along z.

One of the strengths of ESPResSo is the possibility to simulate charged systems with partially periodic boundary conditions. Before you proceed, check your *myconfig.hpp* file to make sure that the `PARTIAL_PERIODIC` feature is enabled.

As an example of a system in partially-periodic boundary conditions, we will modify our script to simulate our simple salt in a slit pore. Remove the `cellsystem`, `setmd box_l` and `setmd periodic` lines left over from your previous simulation simulation script and replace them with these lines:

```
set box_lxy [expr sqrt(2)*$box_l]
set box_lz  [expr 0.5*$box_l]
setmd box_l $box_lxy $box_lxy [expr $box_lz + 1.0]
setmd periodic 1 1 0
constraint wall normal 0 0  1 dist 0 type 2
constraint wall normal 0 0 -1 dist [expr -$box_lz - 1.0] type 2
```

The final two lines add two confining walls at the top and the bottom of the simulation box. The wall is given by its normal vector (pointing up and downwards here), and its distance from $(0, 0, 0)$. Therefore this defines one wall at $z = 0$, and a second one at $z =$`box_lz`$+1$. Finally, the type of the wall is used like a particle type to define the interaction of the walls with the particles.

The addition of 1.0 to the slit width is due to the fact that we will use a Lennard-Jones potential to model the wall; this means that particles of diameter 1.0 cannot come closer than 1.0 to the wall. In a way, the wall itself therefore has a thickness of 0.5. To compensate for this, we simply make the box bigger by 1.

We also need to choose the initial positions of our particles to match the new box dimensions. Replace the appropriate part of your simulation script with this random drawing of particle positions:

```
set posx [expr $box_lxy*[t_random]]
set posy [expr $box_lxy*[t_random]]
set posz [expr ($box_lz-1.0)*[t_random] + 1.0]
```

When defining the interactions, we now also need to add the interactions between the particles and the walls, which is the same as between the particles:

```
inter 0 2 lennard-jones $eps $sig $cut $shift 0
inter 1 2 lennard-jones $eps $sig $cut $shift 0
```

For the electrostatic part, we also need to choose a different algorithm, as P^3M can only handle fully periodic boundary conditions. We choose the MMM2D method:

```
cellsystem layered 3
inter coulomb 1.0 mmm2d 1e-4
```

which replaces the `inter coulomb 10.0 p3m ...` code. Note the decreased Bjerrum length — the confined system would take too long for a tutorial to equilibrate with Bjerrum length 10.0. Still, equilibrating the system is now more difficult. First, we cannot simply ramp up all Lennard-Jones interactions anymore; otherwise, particles will penetrate the walls and break the confinement. Second, we need to ramp up the electrostatic interaction more carefully now. The following code replaces the equilibration loop previously used. It gradually increases the Bjerrum length to the target value of 1.0, and caps only the Lennard-Jones interactions between the particles. The latter can be done by switching on individual force capping, and setting a force cap radius for the particle-particle interactions:

```
inter forcecap individual
for {set i 1} {$i < 10} {incr i} {
    set rad [expr 1.0 - 0.5*$i/10.0]
    set lb [expr 1.0 * $i / 10.0]
    inter 0 0 lennard-jones $eps $sig $cut $shift 0 $rad
    inter 1 0 lennard-jones $eps $sig $cut $shift 0 $rad
    inter 1 1 lennard-jones $eps $sig $cut $shift 0 $rad
    inter coulomb $lb mmm2d 1e-4
    integrate $integ_steps
}
inter forcecap 0
inter coulomb 1.0 mmm2d 1e-4
```

Finally, when writing out, we should update the set of Tcl-variables to represent the asymmetric box, and write out `box_lxy` and `box_lz` instead of just `box_l`.

Analysis

For a such a strongly confined system, the radially averaged distribution function is inappropriate. Instead, it would be more interesting to study the distribution of the particles along the slit width. ESPResSo does not provide such a function, however, we can easily write one using the `bin` command, which simply allows us to create a histogram based on a set of values given as a Tcl list. Therefore, we first create the list of z-coordinates, and then bin them:

```
set bins 20
set data ""
for {set p 0} {$p <= [setmd max_part]} {incr p} {
  lappend data [lindex [part $p pr pos] 2]
}
set rho [bin -linbins 0.5 [expr $box_lz + 0.5] $bins $data]
```

Here, `bins` is a variable that you should set to the desired number of bins (20 should be fine). Otherwise, this code can replace the analyze rdf command. The `bin` command can also be used to calculate the coordinates of the bins:

```
bin -linbins 0.5 [expr $box_lz + 0.5] $bins -binctrwdth
```

will return a list of the centers and widths of the bins, which can be used in analogy to `rlist` in the previous analysis code. Of course, it would be interesting to study the distribution of positive and negative charges separately; for this, you need to duplicate the averaging code, and use two data sets to append the z-coordinates to, depending on the particle type:

```
if {[part $p pr type] == 0} {
  lappend data0 [lindex [part $p pr pos] 2]
} {
  lappend data1 [lindex [part $p pr pos] 2]
}
```

The resulting distribution of charges is shown in Fig. 3. You can see a layering effect of the confinement on the charge distributions.

Charging the walls

So far, the distributions of particles of type 0 and 1 are more or less the same, as one would expect. However, we can change this by introducing charged walls. That is done using the `constraint plate` command:

```
set sigma [expr -0.25*$n_part/($box_lxy*$box_lxy)]
constraint plate height 0 sigma $sigma
constraint plate height [expr $box_lz + 1.0] sigma $sigma
```

Note that you need to have both the `constraint plate` and the `constraint wall` commands in your simulation script. This adds two plates at the bottom and top of the simulation box. The charged walls (the capacitor *plates*) are necessarily perpendicular to the z-axis, since for all electrostatics methods for 2d-periodic systems, ESPResSo assumes that the z-axis is the non-periodic axis.

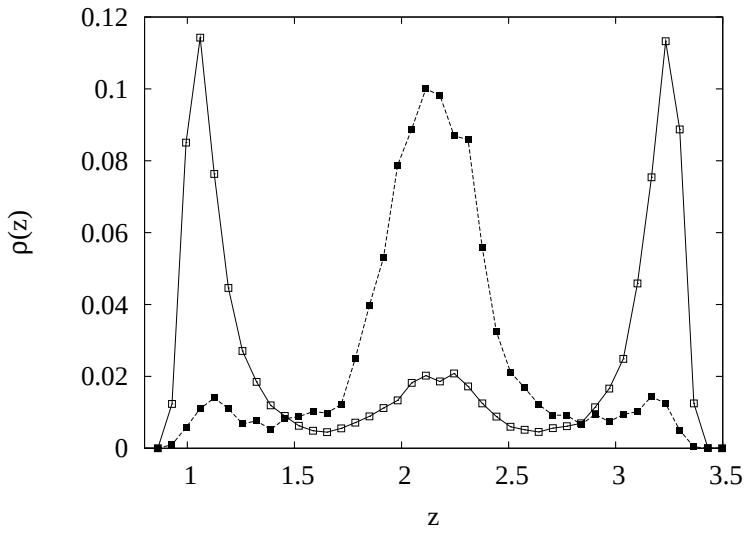


Figure 4: Distribution of positive charges $\rho_+(z)$ (open squares) and of negative charges $\rho_-(z)$ (open squares) under confinement along z. The two confining walls are negatively charged, pushing away the negative charges.

Note that this adds two walls with a total charge of `n_part`/2, which would make the overall system charged. So to maintain charge neutrality, we need to only create half of the charges (i.e. 100) with alternating charges as before, while the rest is created with positive charge. For this system, you should obtain distributions as shown in Fig. 4. The distribution of charges differs strongly between the two types of charges; while the positive charges layer at the walls, the negative charges accumulate in the center of the system.