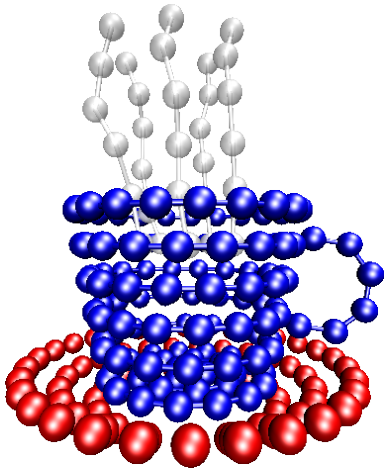


Tutorial 1: Lennard-Jones Liquid*

ESPResSo Basics

H.-J. Limbach M. Süzen K. Grass M. Sega
A. Arnold N. Gribova

March 8, 2012

Contents

*For ESPResSo 3.1.0

1 Introduction

Welcome to the basic ESPResSo tutorial!

In this tutorial, you will learn, how to use the ESPResSo package for your research. We will cover the basics of ESPResSo, i.e. how to set up and modify a physical system, how to run a simulation, and how to load, save and analyse the data.

The more advanced features and algorithms available in the ESPResSo package will be described in additional tutorials in the future.

2 Background

Today's research on Soft Condensed Matter has brought the needs for having a flexible, extensible, reliable and efficient (parallel) molecular simulation package. For this reason ESPResSo (Extensible Simulation Package for Research on Soft matter) [1] has been developed in Max Planck Institute for Polymer Research, Mainz by the Group of PD Dr. Christian Holm [2]. The Espresso package is probably the most flexible and extensible simulation package in the market. It is specially developed for coarse-grained molecular dynamics (MD) simulation of polyelectrolytes but not necessarily limited to this. It can be used even in simulating granular media for example. ESPResSo has been nominated for the Heinz-Billing-Preis for Scientific Computing in 2003 [3].

3 Tutorial Outline

In this short tutorial, you will be introduced to the ESPResSo package as smooth as possible with a minimal set of skills. We will guide you through the initial steps of working with ESPResSo and help you to examine a simple physical system, a Lennard-Jones liquid.

After a brief introduction to ESPResSo in Section 4, we provide you with a short tutorial to the Tcl programming language which is used to control simulations using ESPResSo in Section 5. In Section 6.2, we will introduce you to the problem system

studied in this tutorial and familiarise you with the necessary background knowledge. Please note, however, that it is beyond the scope of this tutorial to give a complete overview of the the area. We give a few references that can give you more detailed information regarding MD.

4 First steps

What is ESPResSo? It is not a coffee, indeed. It is an extensible, efficient Molecular Dynamics package specially powerful on simulating charged systems. In depth information about the package can be found in the relevant sources [1, 3] and a recent paper [2].

From the users point of view, ESPResSo is driven by Tcl(/TK)[1] [4], that the user can interact with the package core via command line interface (CLI) or scripts by using Tcl scripting language[2]. In a given ESPResSo script, some commands are interpreted by the scripting language (Tcl), while others by the core ESPResSo program written in C. However, all ESPResSo commands and directives in the script are transparent to the user, regardless of ifs implementation either on the C- or Tcl-level.

Note: This tutorial assumes that you already have a working ESPResSo installation on your system. If this is not the case, please go to `http://www.espresso.mpg.de/` *for information on how to obtain and install ESPResSo.*

Task 1 *To start ESPResSo, simply type* `Espresso` *in the command line shell. This will open the Command Line Interface (CLI) of ESPResSo. Upon issuing the ESPResSo command* ***code_info*** *you can see the options that are included in your ESPResSo binary*[a]*.*

[a] ESPResSo provides many different features, some which are mutually exclusive. This is why not all features are activated by default, but instead have to be explicitly requested at compile time of the executable. Please consult the user's guide for details on this.

Task 2 *Figure 1 shows the terminal output you should have obtained. What kind of output are you receiving from* ***code_info*** *command?*

On the screenshot you can see several incompiled features, i.e. the fake multiprocessing interface, the fast fourier transformation, the Lennard Jones and electrostatic potential. All features can be found explained in the ESPResSo User Guide.

[1] Tool command language

[2] In short, a *scripting language* allows the user to write instructions that are carried out by an interpreter without prior compiling. This enables the user to use ESPResSo for vastly different applications without the need of different specialised executable files.

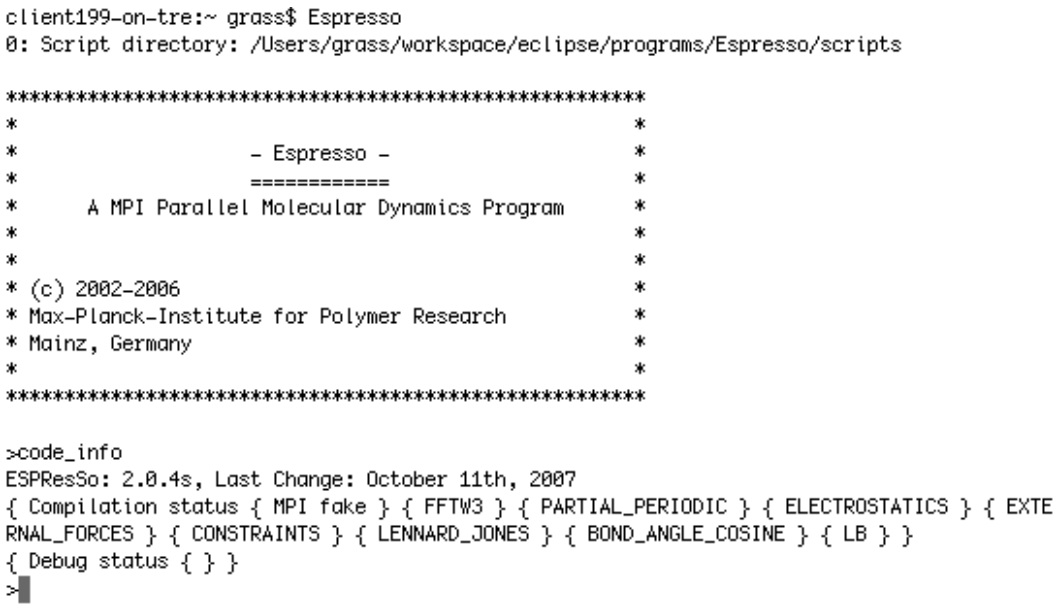


Figure 1: Example output of the ESPResSo Command Line Interface upon issuing the **code_info** command.

5 Short TCL Tutorial

Tcl (Tool Command Language) is a very powerful but easy to learn programming language. [3] Aside from writing any valid Tcl code, the user can write any valid ESPResSo command on the ESPResSo CLI. Here we will review the basic Tcl tutorial [5] in Espresso CLI. Now type `Espresso`. Once in the CLI, you can familiarise yourself with the basics of TCL.

5.1 Assignments and evaluation

A simple text output, or a print statement can be carried out with **puts** [4]

```
puts "Hello Espresso \n"
puts "This is line 1"; puts "this is line 2"
```

In order to declare a variable and/or assign a value to it, you use the assignment command, called **set**.

```
set X "This is a string"
```

[3] It is dynamically interpreted, which means that commands are read, interpreted, and executed by the computer one command line at a time.

[4] Valid TCL / ESPResSo commands are bold and blue, all following code can be direct used with ESPResSo.

```
set Y 1.24
puts $X
puts $Y
```

When assigning a value to a variable, the variable is accessed simply by its variable name (as in the first line, `X`). When a value is accessed, the variable name is preceded by a dollar sign (as in the third line, `$X`)
C-like backslash sequences can be used along put. For example, `\t` puts a tab, `\r` puts a line return and `\n` puts a carriage return. Comments are placed using a hash (#) sign.

```
set X 1.2 ;      # this is a comment
set Y 2.1 ;      # another comment
puts "\t tab \t another tab \n X=$X and Y=$Y "
```

To evaluate the result of mathematical expressions you can use the **expr** command. Note, that the entire statement is enclosed in square brackets.

```
set X  60
set Y  30
set Z [expr $X+$Y]
puts " X=$X and Y=$Y and X+Y=$Z"
set cosX [expr cos($X)]
puts "cos ($X) =  $cosX"
```

Most C-like operators and math functions are valid TCL syntax.

5.2 Comparisons and looping

Syntax of numeric comparison is as follows

```
set x 5
if {$x == 5} {puts "$x is 5"} else {puts "$x is not 5"}
```

You may also write this in different lines by using backslashes. As in many shell scripting languages such as Tcl, the backslash is used for line continuation only in the Espresso CLI, not when running a loaded script. However, there is one exception: even in a script, Tcl expects all parameters of a function, including flow control commands such as `if` or `while`, on the same line. Here you need continuation backslashes even in a script. However, a block with in curled braces, which is the most common in Tcl, does not require backslashes for continuation inside. Therefore, in a script, the above code might look like this:

```
set x 5
if {$x == 5} {
  puts "$x is 5"
} else {
  puts "$x is not 5"
}
```

but still, you need the backslashes in this style:

```
set x 5
if {$x == 5} \
  { puts "$x is 5" } \
else \
  { puts "$x is not 5" }
```

You can loop with standard **for** or **while** constructs. For example finding 10! with the **for** construct:

```
set factorial 1.0
for {set i 1} {$i <11} {incr i} \
  {set factorial [expr $factorial*$i]}
puts "10! is $factorial"
```

Or with a **while** construct

```
set factorial 1.0
set i 1
while {$i <11} {set factorial [expr $factorial*$i] ; incr i}
puts "10! is $factorial"
```

5.3 Lists

An ordered collection of entities can be assigned to a variable that makes it a **list**[5]. This is the basic data structure in TCL. Lists can be set similar to variables. To access the list data one can use **lindex** by using corresponding index value. Remember that in TCL the list indices start with 0 like in other scripting/programming languages .

```
set x "1 2 3"
puts "first element is [lindex $x 0]"
puts "second element is [lindex $x 1]"
puts "and the last [lindex $x 2]"
```

One can access all the elements by using **foreach** looping consruct as well

```
set i 0 foreach  j $x {\
  puts "$j is item number $i in list x"; incr i}
```

Also we can access list of lists

```
set y "{l00 l01} {l10 l11} {l20 l21}"
puts "first element of second list is [lindex $y 1 0]"
puts "second element of third list is [lindex $y 2 1]"
```

[5]This is similar - but not entirely equivalent - to arrays in computer languages such as C or C++.

We can also find the length of a list by **llength**, append an element by **lappend** and inserting an element by **linsert**:

```
set x "1 2 3 4" ;          # generate a list x
llength $x ;               # get the size of list x (number of elements)
lappend x 5 ;              # add a new member end of list
puts "x is {$x}" ;                 # print list again
set $x [linsert $x 3 3a] ;         # insert an element "3a" at index 3
puts "x is {$x}" ;                 # print list again
```

6 Adding a new Tcl command

In Tcl there is actually no distinction between commands (often known as 'functions' in other languages) and "syntax" [5]

```
proc sum {arg1 arg2}  { \
set x [expr {$arg1 + $arg2}]; \
return $x \
}
sum  1 4
puts [sum 1 4]
```

6.1 Writing to a file

It is useful to write the data into a file. For example

```
set file_handle [open "file.dat" "w"];
                # open a file called file.dat to write
                # and file channel is $file_handle

puts $file_handle "This will go into file!"
for {set i 0} {$i <10} {incr i} { \
   puts $file_handle "counting $i"  }

close $file_handle ;                    # close file channel
set file_content [exec cat file.dat] ;
                                        # exec runs shell commands
puts "$file_content"
exec rm file.dat ;                      # remove the file
```

For further and advanced language details please consult with official Tcl documentation [4].
So far, we have been typing all the commands line by line in the CLI. In practice, these lines are actually written in one text file, whose filename is usually ending with the extension `tcl`. The commands in that text file are then executed from the Linux command line with the command `Espresso filename.tcl`.

Task 3 *Write a Tcl procedure (custom Tcl command) to compute an arithmetic average*

$$\bar{x} = \frac{1}{N}\sum_{i=1}^{N} x_i$$

out of given list of real numbers respectively. Use rand() command to produce arbitrary number of real numbers between 0 and 1 to test your new command. Write your code into a file called task1.tcl and run as follows: `Espresso task1.tcl`*. Check your result with smaller data set that you can verify the correctness manually.*

Sample solution:

1. Define the new function and set the counting and result variables:

```
proc xsquare {arg1} {
set res 0
set i 0
```

2. Sum up the given values and printout the sum:

```
foreach j $arg1 { set res [expr $j+$res]; incr i };
puts $res;
```

3. count the elements, divide the sum an return the value:

```
set lang [llength $arg1]
set res [expr {$res / $lang } ];
return $res; }
```

4. to call your function with an array $x:

```
set y [xsquare $x];
puts $y;
```

6.2 System setup

An Espresso script is a tcl simulation script that drives the C-core of the package. It contains commands native to Tcl - like those we have already learned - plus special ESPResSo commands that execute procedures specific to MD calculations. In this section we will review some very basic commands that will help you to understand the sample introductory script. An actual script used for research is usually more complicated.

6.3 Lennard-Jones Potential

A pair of neutral atoms or molecules is subject to two distinct forces in the limit of large separation and small separation: an attractive force at long ranges (van der Waals

force, or dispersion force) and a repulsive force at short ranges (the result of overlapping electron orbitals, referred to as Pauli repulsion from Pauli exclusion principle). The Lennard-Jones potential (also referred to as the L-J potential, 6-12 potential or, less commonly, 12-6 potential) is a simple mathematical model that represents this behavior. It was proposed in 1924 by John Lennard-Jones. The L-J potential is of the form $V(r) = 4\epsilon[(\frac{\sigma}{r})^{12} - (\frac{\sigma}{r})^{6}]$ where ϵ is the depth of the potential well and σ is the (finite) distance at which the inter particle potential is zero and r is the distance between the particles. The $(\frac{1}{r})^{12}$ term describes repulsion and the $(\frac{1}{r})^{6}$ term describes attraction. The Lennard-Jones potential is an approximation. The form of the repulsion term has no theoretical justification; the repulsion force should depend exponentially on the distance, but the repulsion term of the L-J formula is more convenient due to the ease and efficiency of computing r^{12} as the square of r^{6}.

6.4 Units

Novice users must understand that Espresso has no fixed unit system. The unit system is set by user. Conventionally, reduced units are employed, in other words LJ units. [6]

6.5 Simulation Parameters

There are global parameters of the simulation system. Some of them are dynamic, that is to say we can change on the fly, others are read only[7]. One important ESPResSo command to address these parameters is **setmd**

```
setmd time_step 0.001;          # this sets integrator's
                                # time step to 0.00
setmd box_length 100.0 100.0 100.0;
                                # this sets cubic box L =100
set number_of_particles [setmd max_part];
                                # reads the number of particles
```

ESPResSo needs to know which integrator to use for dynamics. One can use NVE (particle Number, Velocity, Energy) or NVT (particle Number, Velocity, Temperature)(Langevin) as well as NPT-isotropic (particle Number, Pressure, Temperature) ensembles. Some examples how to use the thermostats

```
thermostat off
```

This implies to use NVE ensemble

```
thermostat langevin 1.0 0.5
```

Use a langevin thermostat (NVT ensemble) with temperature set to 1.0 and damping coefficient to 0.5

[6] If we have charges there is additionally a concept of Bjerrum length, consult Espresso original paper for more details.

[7] For more information on read-only variables consult the user's guide.

6.6 Assigning Particle Properties

The power of the ESPResSo package lies in the flexible manipulation of particle data. Particles can be manipulated by the **part** command which recognises the unique particle id. Each particle must be a member of a group which is called *type*. Interactions among those types can be defined through *type* number with **inter** command. [8] For example to place a *particle id* 0 and *type* 0 at given position (x, y, z)

```
part 0 pos $x $y $z type 0
```

it is also possible to read the information on the given particle

```
part 0 print pos
```

which returns position vector of particle id 0.[9]

6.7 Assigning Interactions

LJ interaction among type 0 particles can be defined as follows

```
set lj1_eps      1.0
set lj1_sig      1.0
set lj1_cut      1.12246
set lj1_shift    0.0
set lj1_offset   0.0
inter 0 0 lennard-jones $lj1_eps $lj1_sig $lj1_cut $lj1_shift
  $lj1_offset
```

This [10] setting corresponds to following potential form

$$U(r) = 4\epsilon \left[\left(\frac{\sigma}{r - \text{offset}} \right)^{12} - \left(\frac{\sigma}{r - \text{offset}} \right)^{6} + \text{shift} \right]$$

6.8 Generating Data: Lennard-Jones Liquid Simulation

After we have shortly explained how you can use ESPResSo, we now come to the Lennard-Jones Liquid Simulation. Before we explain the script step by step, run the `lj_tutorial.tcl` with ESPResSo to get all generated files.

We include necessary functions with **source** lj_functions.tcl by using the external lj_functions.tcl file and print out the incompiled features of ESPResSo

[8] Note that, In most electrostatic algorithms, one does not need a type id for interaction specification.

[9] Note, that the words `pos`, `type`, and `print` are not variables but directives to the **part** command. See the ESPResSo user's guide for more details.

[10] As in the previous example, the word `lennard-jones` is not a variable but a directive of the command `inter`.

```
source lj_functions.tcl
puts ""
puts "============================"
puts "=lj_liquid_tutorial.tcl="
puts "============================"
puts ""
puts "Espresso Code Base : \n[code_info]\n"
cellsystem domain_decomposition  -no_verlet_list
```

6.8.1 System Setup

At first, we must configure the environment and set the needed parameters.

```
# System identification:
set name  "lj_liquid"
set ident "_s1"

# System parameters
###################
# we set 108 particles in our system

set n_part  108

# Interaction parameters
########################
# we must set our lennard-jones interaction parameters

set lj1_eps      1.0
set lj1_sig      1.0
set lj1_cut      2.5
set lj1_shift    [expr -(pow(1.0/$lj1_cut,12)-pow(1.0/$lj1_cut,6))]
set lj1_offset   0.0

# Integration parameters
########################
# we need some information about our system

thermostat off ;          # Simulation in NVE Ensemble

setmd time_step 0.001
set eq_tstep 0.0001
set tstep   0.001
set skin 0.1
setmd skin  $skin
set target_temperature 0.728

# we need some warmup information

set warm_steps   100
```

```
45 set warm_n_times 2000
46
47 # do the warmup until the particles have
48 # at least the distance min__dist
49
50 set min_dist      0.87
51
52 # some parameters for the integration self
53
54 set sampling_interval     1000
55 set equilibration_interval 1000
56
57 set sampling_iterations 200
58 set equilibration_iterations 200
59
60
61 # Other parameters
62 ################
63
64 set tcl_precision 8
65 #setting a seed for the random number generator
66 expr srand([pid])
```

As a second step we initialise the particles and interactions in our system.

```
67 # Particle setup
68 ##############
69
70 set density 0.8442      # we need the density of the particles
71
72 # now we setup the particle box
73 set box_length [expr pow($n_part/$density,1.0/3.0)+2*$skin]
74 puts " density = $density box_length = $box_length"
75 setmd box $box_length $box_length $box_length
76
77 # we set particles on random places within the Box
78 for {set i 0} {$i < $n_part} {incr i} {
79    set pos_x [expr rand()*$box_length]
80    set pos_y [expr rand()*$box_length]
81    set pos_z [expr rand()*$box_length]
82    part $i pos $pos_x $pos_y $pos_z q 0.0 type 0
83 }
84
85 # write initial particle data to file
86 writepdb data/config.pdb
87
88 # Interaction setup
89 ################
90 # we setup the lennard-jones potential
91 # as the only interaction between the particles
```

```
92
93 inter 0 0 lennard-jones $lj1_eps $lj1_sig $lj1_cut $lj1_shift
   $lj1_offset
```

Task 4 *Study the file* `lj_tutorial.tcl`. *This system mimics the case study 4 of section 4, in the book [6]. How can one define truncated-shifted potential in* `lj_tutorial.tcl`*? (keep in mind that Espresso has already a factor of 4 at shifted part with cut off* $r_c = 2.5$*)*

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

$$U(r)^{tr\text{-}sh} = \begin{cases} U(r) - U(r_c) & r_c > r \\ 0 & r_c < r \end{cases}$$

(To find the solution look at line 26. Look at picture 2 to see a plot of the potential)

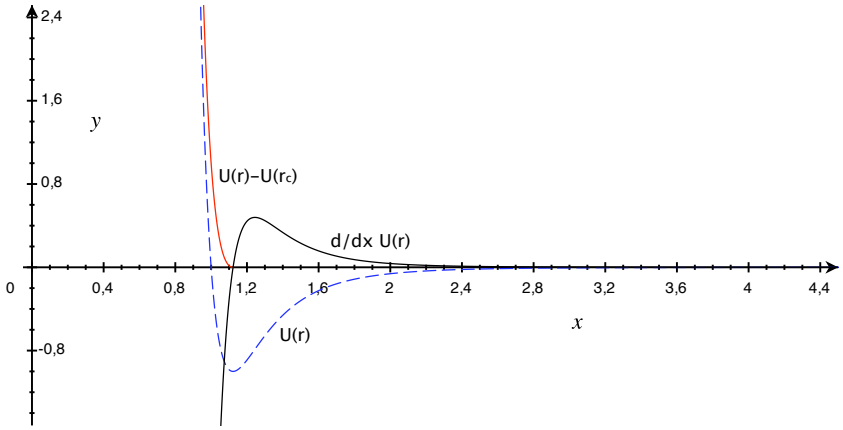


Figure 2: Lennard Jones Potential with $\epsilon = 1$ and radius $\sigma = 1$. If you use a large cutoff such as 2.5σ, the potential is practically zero at the cutoff. The red curve indicates the Weeks-Chandler-Andersen potential, which is obtained from the Lennard-Jones potential by cutting it off in its minimum at $r_c = \sqrt[6]{2}$ and shifting it up.

The **writepdb** command writes the atomic configuration in the PDB (Brookhaven Pro-

tein DataBase) format to the given file. This standard formatted file can easily be imported to standard applications like VMD.

As it was said in 6.6 we had to set up the interactions between all groups separately. After we have set the necessary environment we must warmup our system before we run the simulation. We set particles at random positions so some particles can overlap. In this situation ESPResSo will crash with an error: particle out of range. To take particles apart we cap forces by setting the Lennard-Jones force constant below a certain distance. Therefore we use the **inter** ljforcecap command. We do the procedure `$warm_n_times` times for `$warm_steps` steps and stop only if the minimal distance between particles is also larger than `$min_dist`, that was set earlier. To turn the capping off, we set ljforcecap to 0. Then we equilibrate our system until all relevant physical observables are fluctuating around their mean values. In the case of Lennard Jones it is enough to monitor energy. To equilibrate we rescale particles' velocities to reach the target temperature.

```
94 ############################################################
95 #  Warmup Integration                                      #
96 ############################################################
97
98 set act_min_dist [analyze mindist]
99 puts "Start with minimal distance $act_min_dist"
100
101 # open Observable file
102
103 puts "\nStart warmup integration:"
104 puts "At maximum $warm_n_times times $warm_steps steps"
105 puts "Stop if minimal distance is larger than $min_dist"
106
107 # set LJ cap
108 set cap 1.0
109 inter ljforcecap $cap
110
111 # Warmup Integration Loop, equilibrate particles
112 set i 0
113 while { $i < $warm_n_times && $act_min_dist < $min_dist } {
114     integrate $warm_steps
115
116     # Warmup criterion
117     set act_min_dist [analyze mindist]
118     puts -nonewline "run $i at time=[setmd time] (LJ cap=$cap) min
        dist = $act_min_dist\r"
119
120      # to force the printout immediately and don't wait for the
        buffer been printed
121     flush stdout
122
123     # Increase LJ cap
124     set cap [expr $cap+1.0]
```

```
125     inter ljforcecap $cap
126     incr i
127 }
128
129 inter ljforcecap 0;
130
131 puts "\n Warm up finished \n"
132
133 ### Thermalization
134 setmd time_step $eq_tstep
135 for { set i 0 } { $i < $equilibration_iterations } { incr i } {
136    integrate $equilibration_interval
137    set energies [analyze energy]
138    rescale_velocities  $target_temperature [setmd n_part]
139    puts -nonewline "eq run $i at time=[setmd time] \r"
140 }
```

After we have set the necessary environment and warmed up our system, we can now start with the actual simulation. To analyse our data after the simulation, we open some files for writing the data in.

```
142   puts "sampling "
143 setmd time_step $tstep
144
145 # files to save simulation datas
146 set en [open "data/energy.dat" "w"]
147 set blockfile [open "data/sim_info.dat" "w"]
148
149 puts $en "#"
150 puts $en "#"
151 puts $en "#  Pressure     Kinetic     Potential  Temperature "
152 puts $en "# "
153 for {set i 0} { $i < $sampling_iterations } { incr i} {
154      integrate $sampling_interval
155      save_sim $blockfile "id pos v f q type" "all"
```

Task 5 *Study the file* `lj_functions.tcl`*, specifically the procedure* `save_sim` *and how it is called in* `lj_tutorial.tcl` *file. Then run* `lj_tutorial.tcl` *and check* `data` *directory for the simulation data file. Inspect the simulation data file* `sim_info.dat`*.*
Run ESPResSo by typing `Espresso lj_tutorial.tcl` *on the Linux command line.*

The **save_sim** statement calls the function defined in lj_tutorial.tcl . We tell the function what parameters to be saved and which particles we want to save (all). Before continuing

with our example we have a look at the **save_sim** procedure

```
proc save_sim {cfile parinfo range } {
 blockfile $cfile write variable all
 blockfile $cfile write tclvariable all
 blockfile $cfile write particles $parinfo $range
 blockfile $cfile write interactions
 blockfile $cfile write bonds
 blockfile $cfile write random
 blockfile $cfile write seed
 blockfile $cfile write bitrandom
 blockfile $cfile write bitseed
}
```

This procedure saves all variables available in our programme to $cfile which is in our example sim_info.dat. Have a look at the sim_info.dat to see what that means

```
{variable
{box_l 5.2387886 5.2387886 5.2387886}
{cell_grid 2 2 2}
{cell_size 2.6193943 2.6193943 2.6193943}
{dpd_gamma 0.0}
{dpd_r_cut 0.0}
...
}
{tclvariable
{density 0.8442}
{tstep 0.001}
{energies { energy -384.41147 } { kinetic 121.41453 } { 0  0 nonbonded -505.826 }}
{lj1_cut 2.5}
{blockfile file6}
...
}
{particles {id pos v f q type}
{0 13.08106 4.376436 2.4466876 1.2432192 ...}
{1 -2.7657422 -0.09919841 -1.0362237 -0.4642931...}
{2 4.4635587 5.4885591 5.460385 0.18670765 ...}
{3 -2.9920063 0.88519503 7.3518191 0.5820964 ...}
...
}
{interactions
{0 0 lennard-jones 1.0 1.0 2.5 0.0040792228 0.0 0.0 }
}
{bonds
{0 { } }
{1 { } }
...
}
{random
```

```
{1198928294 .... }
}
{seed
{16838}
}
{bitrandom
{0 147 2085679233 .... }
}
{bitseed
{16838}
}
...
```

What do we see? We find that there are five types of blocks which are repeated very often. The first block (variable) contains all ESPResSo variables, the second (tclvariable) all variables of the tcl script. In the third block we find the entire particle data we had told the **save_sim** function to save (id: particle id, pos: position, v: velocity, : force, q: charge, type). **save_sim** saves interactions, bonds, random, seed, bitrandom and bitseed. In our case the interaction is Lennard-Jones. The bonds block is empty because the particles are not bound to each other like, for example, in a polymer. The rest of the blocks contains information (random, seed, bitrandom and bitseed) to be able to restore the status of the random generator as it was while writing these blocks. What do we need that for? If we rerun the simulation we will get the same results after we have set our variables including the random generator status. Looking at the example script, we see that the **save_sim** function is called every time in the loop; that is why the blocks are repeated.

Task 6 *Study and run* `blockfile_read.tcl` *to see how to read offline data. Modify this script to print out simulation time and average particle positions* $x_{avg}, y_{avg}, z_{avg}$.

In the energy.dat file we print out the values for pressure, kinetic and potential energies, temperature obtained with the command **analyze** `energy`.

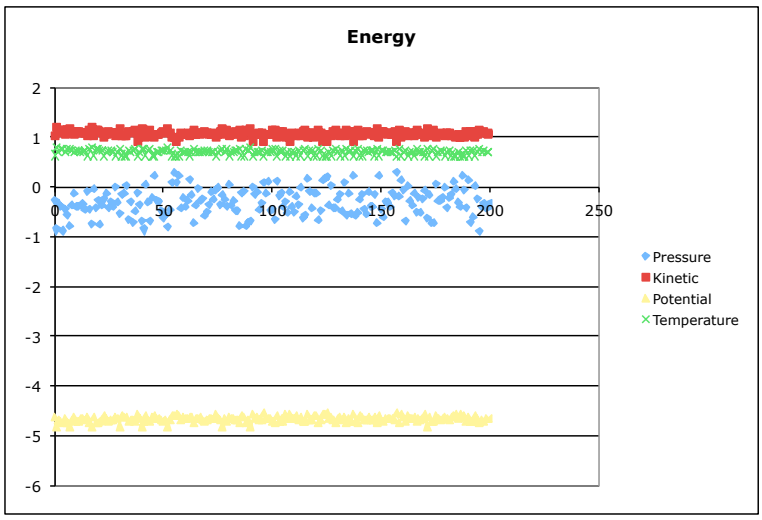
```
156     set energies [analyze energy]
157     set pressure [analyze pressure total]
158     set total [expr [lindex $energies 0 1]/$n_part]
159     set kinetic [expr [lindex $energies 1 1]/$n_part]
160     set potential [expr [lindex $energies 2 3]/$n_part]
161     set kinetic_temperature [expr [lindex $energies 1 1]/(1.5*[setmd
          n_part])]
162     set temperature $kinetic_temperature
163     puts $en "$i   $pressure   $kinetic   $potential   $temperature
          $total"
164     lappend apressure $pressure
```

```
165       lappend akinetic $kinetic
166       lappend apotential $potential
167       lappend atemperature $temperature
168       lappend atotal $total
169       puts -nonewline "integration step    $i / $sampling_iterations \r
            "
170 }
171 close $en
```

`kinetic temperature` here refers to the measured temperature obtained from kinetic energy and the number of degrees of freedom in the system. It should fluctuate around the preset temperature of the thermostat.

Task 7 *Plot the time evolution of pressure and energy, which are written into the* `data` *directory in the file* `energy.dat`.

The result plot for `energy.dat` should be similar to this one:



As we can see the system is in equilibrium because pressure, potential and kinetic energy per particle and calculated current temperature fluctuate around their mean values.

6.9 Simple Error Analysis on Time Series Data with `uwerr`

Espresso provides a build-in time series analysis tool called **uwerr**. In our simulation we don't know if values of the same variable that we got from two adjacent samples are correlated or not and sampling too seldom will lead either to long runs of the simulation or to bad statistics. On the other hand, sampling too often leads to strong correlations between the samples, which will make us underestimate the statistical errors in our measurements using usual formulas e. g. for the standard deviation. **uwerr** is used to determine the mean and its standard error of total energy per particle for arbitrary numerical time series based on the article by Wolff [7]. Unlike the standard formulas, it can be used even with strongly correlated samples.

Here to obtain, for example, the error for the total energy we submit as a first argument for **uwerr** the array of all measured values of total energy `$atotal`, and then the total number of samples `$sampling_iterations`. 1 as the third argument means that we ran only one full measument (complete simulation). In general, the mean value could be also obtained from several simulations, providing `$atotal` then as a matrix, not as an array, the total number of samples for every simulation should be the same and the third argument for **uwerr** would be the number of simulations. **uwerr** returns a string, the first value of which is a calculated mean value and the second is its error.

```
172   puts "--Reporting Energies and Temperature"
173   set error [uwerr $atotal $sampling_iterations 1 ]
174   set value [lindex $error 0]
175   set verror [lindex $error 1]
176   puts "      Total Energy: $value  $verror"
177   set error [uwerr $akinetic $sampling_iterations 1 ]
178   set value [lindex $error 0]
179   set verror [lindex $error 1]
180   puts "      Kinetic Energy: $value  $verror"
181   set error [uwerr $apotential $sampling_iterations 1 ]
182   set value [lindex $error 0]
183   set verror [lindex $error 1]
184   puts "      Potential Energy: $value  $verror"
185   set error [uwerr $atemperature $sampling_iterations 1 ]
186   set value [lindex $error 0]
187   set verror [lindex $error 1]
188   puts "      Temperature : $value  $verror"
189   set error [uwerr $apressure $sampling_iterations 1 ]
190   set value [lindex $error 0]
191   set verror [lindex $error 1]
192   puts "      Pressure : $value  $verror"
193 exit
```

The last line here is the last line of the script terminating the process.

Task 8 *Inspect what* `analyse pressure total` *command returns. Make a similar error analysis for total pressure.*

Hint: `analyse pressure` (without `total`) returns the pressure and corresponding contributions to it.

6.10 Other Useful Scripts

The radial distribution function (RDF) describes the distibution of particles around the center of a fixed particle, as a function of the particle-particle distance. This of course assumes that the particle distribution is isotropic around the particles.

Task 9 *Run* `rdf.tcl`*, inspect the code and plot the RDF from* `data/rdf.dat`*. Try different parameters for the* `analyze rdf` *command, such as the bin size. What do you observe?*

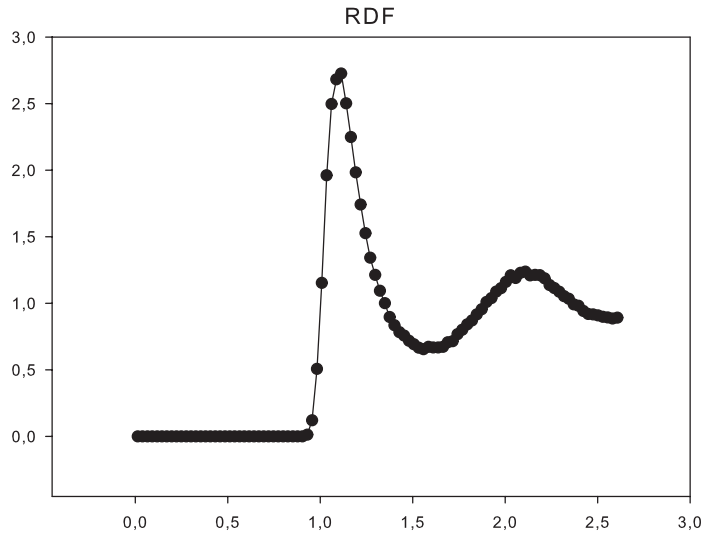


Figure 3: The rdf.dat plot should be similar to this one

Our RDF is structureless, which means we have liquid. If the bin size is increased, the number of points in the graph will also increase, but due to poorer sampling the curve will not be smooth anymore.

The velocity autocorrelation function (VACF) is an averaged time dependent correlation function of all particles' velocities.

Task 10 *Run* `vacf.tcl`*, inspect the code and plot the VACF from* `data/vacf.dat`*. The VACF $C(t)$ can be computed directly:*

$$C(t) = \langle \mathbf{v}_i(0)\mathbf{v}_i(t) \rangle$$

which can be estimated by

$$C(t) = \frac{1}{N} \sum_{i=0}^{N} \mathbf{v}_i(0)\mathbf{v}_i(t)$$

where N is the number of particles.
Try to modify `vacf.tcl` *by using* `vecsub` *and* `veclen` *and the tcl math functions.*

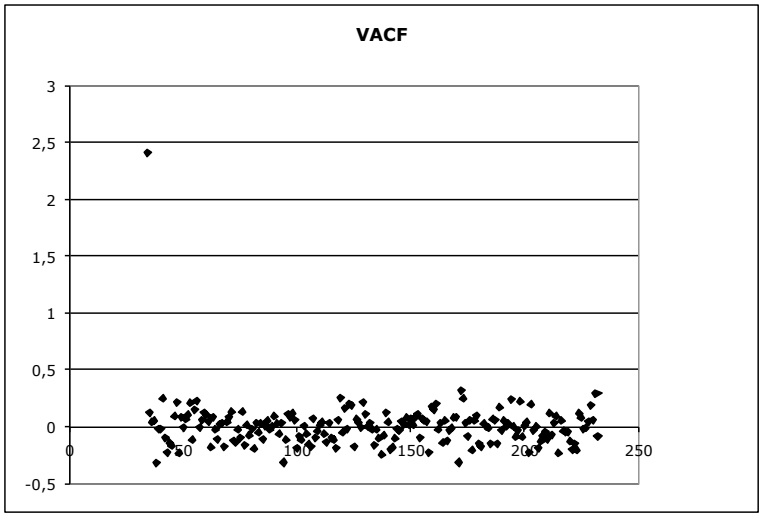


Figure 4: The vacf.dat plot should be similar to this one

We can see when time is bigger than 50 the particles have already 'forgot' about their initial velocities. Sampling more in the time interval [0;50] will show the decay of VACF there.

The mean square displacement (MSD) is the average squared distance that a particle travelled during a given time.

Task 11 *Run* `msd.tcl`*, inspect the code and plot the MSD from* `data/msd.dat`*. The MSD can be simply computed by:*

$$\langle \Delta r(t)^2 \rangle = \frac{1}{N} \sum_{i=0}^{N} \Delta \mathbf{r}_i(t)^2$$

or

$$\langle \Delta r(t)^2 \rangle = \frac{1}{N} \sum_{i=0}^{N} |r_i(t) - r_i(0)|^2$$

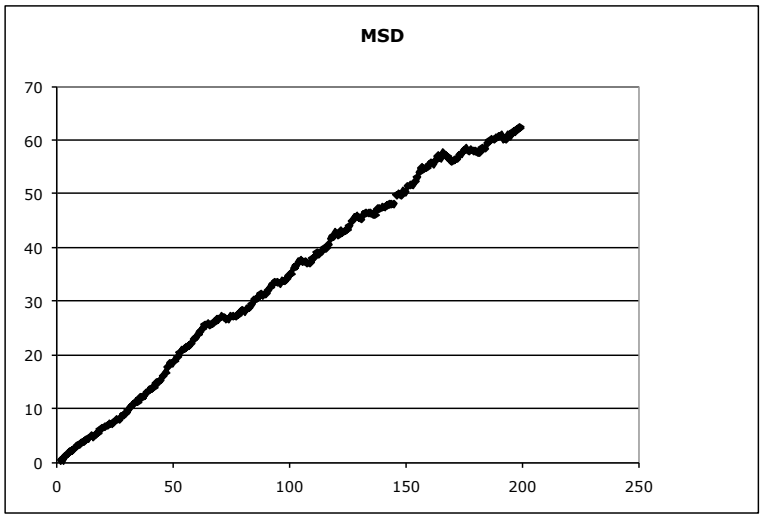


Figure 5: The msd.dat plot should be similar to this one

On short time scales a particle does not collide with others (ballistic regime) so the distance travelled should be proportional to the time and therefore the MSD should increase quadratically. On bigger time scales, a particle performs sort of a random walk due to many interactions with other particles. This regime is called diffusive; in this regime the MSD increases linearly with time. The coefficient of proportionality is the diffusion coefficient: $D = \frac{1}{2dt}\langle \Delta r(t) \rangle^2$, where d is the dimensionality of the problem and t the travelling time.

If `msd.dat` is plotted in log-log scale, then ballistic and diffusive regimes will be visible even better (better sampling won't harm).

References

[1] http://www.espresso.mpg.de/.

[2] HJ Limbach, A. Arnold, and B. Mann. ESPResSo; an extensible simulation package for research on soft matter systems. *Computer Physics Communications*, 174(9):704–727, 2006.

[3] A. Arnold, BA Mann, HJ Limbach, and C. Holm. ESPResSo–An Extensible Simulation Package for Research on Soft Matter Systems. *Forschung und wissenschaftliches Rechnen*, 63:43–59, 2003.

[4] http://www.tcl.tk.

[5] http://www.tcl.tk/man/tcl8.5/tutorial/tcltutorial.html.

[6] Daan Frenkel and Berend Smit. *Understanding Molecular Simulation*. Academic Press, San Diego, second edition, 2002.

[7] U. Wolff. Monte carlo errors with less errors. *Comput. Phys. Commun.*, 156:143–153, 2004.