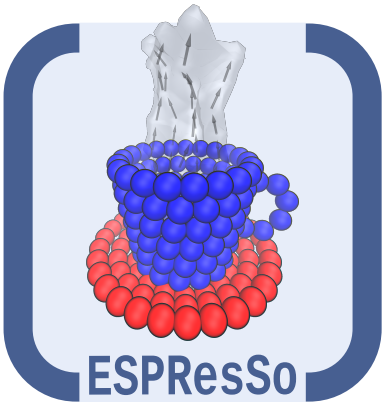


Tutorial 8: Visualization*

How to visualize your ESPResSo simulations while they are running

M. Kuron

January 26, 2018

1 Introduction

When you are running a simulation, it is often useful to see what is going on by visualizing particles in a 3D view or by plotting observables over time. That way, you can easily determine things like whether your choice of parameters has led to a stable simulation or whether your system has equilibrated. You may even be able to do your complete data analysis in real time as the simulation progresses.

Thanks to ESPResSo's Python interface, we can make use of standard libraries like Mayavi (for interactive 3D views) and Matplotlib (for line graphs) for this purpose. We will also use NumPy, which both of these libraries depend on, to store data and perform some basic analysis.

2 Simulation

First, we need to set up a simulation. We simulate a simple Lennard-Jones liquid in this tutorial using the script shown below. Visualization of more advanced features of

*For ESPResSo unknown

ESPResSo is also possible (for example, bonds) or will be possible soon (for example, constraints and lattice Boltzmann).

```
from __future__ import print_function
import espressomd._system as es
import espressomd
from espressomd import thermostat
from espressomd import integrate
from espressomd import visualization
import numpy
from matplotlib import pyplot
from threading import Thread

# System parameters
############################################################

# 10 000  Particles
box_l = 10.7437
density = 0.7

# Interaction parameters (repulsive Lennard Jones)
############################################################

lj_eps = 1.0
lj_sig = 1.0
lj_cut = 1.12246
lj_cap = 20

# Integration parameters
############################################################
system = espressomd.System()
system.time_step = 0.01
system.skin = 0.4
system.thermostat.set_langevin(kT=1.0, gamma=1.0)

# warmup integration (with capped LJ potential)
warm_steps = 100
warm_n_times = 30
# do the warmup until the particles have at least the distance min_dist
min_dist = 0.9

# integration
int_steps = 1000
int_n_times = 100
```

```
############################################################
#  Setup System                                            #
############################################################

# Interaction setup
############################################################

system.box_l = [box_l, box_l, box_l]

system.non_bonded_inter[0, 0].lennard_jones.set_params(
    epsilon=lj_eps, sigma=lj_sig,
    cutoff=lj_cut, shift="auto")
system.non_bonded_inter.set_force_cap(lj_cap)

# Particle setup
############################################################

volume = box_l * box_l * box_l
n_part = int(volume * density)

for i in range(n_part):
    system.part.add(id=i, pos=numpy.random.random(3) * system.box_l)

system.analysis.distto(0)
act_min_dist = system.analysis.mindist()
system.max_num_cells = 2744

############################################################
#  Warmup Integration                                      #
############################################################

# set LJ cap
lj_cap = 20
system.non_bonded_inter.set_force_cap(lj_cap)

# Warmup Integration Loop
i = 0
while (i < warm_n_times and act_min_dist < min_dist):
    integrate.integrate(warm_steps)
    # Warmup criterion
    act_min_dist = system.analysis.mindist()
    i += 1
```

```
#    Increase LJ cap
    lj_cap = lj_cap + 10
    system.non_bonded_inter.set_force_cap(lj_cap)

#########################################################################
#       Integration                                                     #
#########################################################################

# remove force capping
lj_cap = 0
system.non_bonded_inter.set_force_cap(lj_cap)

def main():
    for i in range(0, int_n_times):
        print("run %d at time=%f " % (i, system.time))
        integrate.integrate(int_steps)
main()

# terminate program
print("\nFinished.")
```

3 Live plotting

Let's have a look at the total energy of the simulation. We can determine the individual energies in the system using

```
print (system.analysis.energy())
```

to get

```
OrderedDict([('total', 1840.118038871784), ('ideal', 1358.743742464325),
```

Write that command right after the call to main() and see if you can get a similar result.

Now we want to store the total energy over time in a NumPy array. To do that, modify the main function definition to be the following:

```
energies = numpy.empty((int_steps,2))
def main():
    for i in range(0, int_n_times):
        print("run %d at time=%f " % (i, system.time))
        integrate.integrate(int_steps)
        energies[i] = (system.time, system.analysis.energy()['total'])
```

Now we can do some analysis on the stored energies. For example, let us calculate the time-averaged energy:

```
print ("Average energy: %.6g" % energies[:,1].mean())
```

We can also plot the energy over time by adding

```
pyplot.xlabel("time")
pyplot.ylabel("energy")
pyplot.plot(energies[:,0],energies[:,1])
pyplot.show()
```

to the end of the script.

Of course, this plot only gets shown after the entire simulation is completed. To get an interactive plot, we update it from within the integration loop.

```
energies = numpy.empty((int_steps,2))
current_time = -1
pyplot.xlabel("time")
pyplot.ylabel("energy")
plot, = pyplot.plot([0],[0])
pyplot.show(block=False)
def update_plot():
    if current_time < 0:
        return
    i = current_time
    plot.set_xdata(energies[:i+1,0])
    plot.set_ydata(energies[:i+1,1])
    pyplot.xlim(0, energies[i,0])
    pyplot.ylim(energies[:i+1,1].min(), energies[:i+1,1].max())
    pyplot.draw()
def main():
    global current_time
    for i in range(0, int_n_times):
        print("run %d at time=%f " % (i, system.time))
        integrate.integrate(int_steps)
        energies[i] = (system.time, system.analysis.energy()['total'])
        current_time = i
        update_plot()
```

One shortcoming of this simple method is that one cannot interact with the controls of the plot window (e.g. resize the window or zoom around in the graph). This will be resolved using multiple threads when we combine the plotting with the 3D visualization in the next section.

4 Live visualization

In order to be able to interact with the live visualization, we need to move the main integration loop into a secondary thread and run the visualization in the main thread (note that visualization or plotting cannot be run in secondary threads).

First, let's revert to the main loop without plotting:

```
energies = numpy.empty((int_steps,2))
def main():
    for i in range(0, int_n_times):
        print("run %d at time=%f " % (i, system.time))
        integrate.integrate(int_steps)
        energies[i] = (system.time, system.analysis.energy()['total'])
```

Then, add the following line after the particle setup code:

```
mayavi = visualization.mayavi_live(system)
```

Now, go to the end of the main function definition and add

```
mayavi.update()
```

which sends the current simulation state to the visualizer. Now, go to the line where main() is called and replace it with the following code, which dispatches the function in a secondary thread, and then opens the visualizer window:

```
t = Thread(target=main)
t.daemon = True
t.start()
mayavi.run_gui_event_loop()
```

While the simulation is running, you can move and zoom around with your mouse or explore the buttons in the toolbar to see how the graphical representation can be changed.

5 Combined live visualization and plotting

Now let's merge the code from the preceding two sections so we can see the energy graph while viewing the 3D visualization of the particles. Do do that, we copy the pyplot-related lines from above:

```
current_time = -1
pyplot.xlabel("time")
pyplot.ylabel("energy")
plot, = pyplot.plot([0],[0])
pyplot.show(block=False)
def update_plot():
    if current_time < 0:
        return
    i = current_time
    plot.set_xdata(energies[:i+1,0])
    plot.set_ydata(energies[:i+1,1])
    pyplot.xlim(0, energies[i,0])
    pyplot.ylim(energies[:i+1,1].min(), energies[:i+1,1].max())
```

```
    pyplot.draw()
```

Then we merge the main function definitions from both the previous sections.

```
def main():
    global current_time
    for i in range(0, int_n_times):
        print("run %d at time=%f " % (i, system.time))
        integrate.integrate(int_steps)
        energies[i] = (system.time, system.analysis.energy()['total'])
        current_time = i
        mayavi.update()
        # update_plot() cannot be called from here
```

However, as we now have multiple threads, we cannot simply call update_plot() from the main function definition. Instead, we register it as a callback with the visualizer before we start up the visualizer GUI:

```
t = Thread(target=main)
t.daemon = True
t.start()
mayavi.register_callback(update_plot, interval=500)
mayavi.run_gui_event_loop()
```