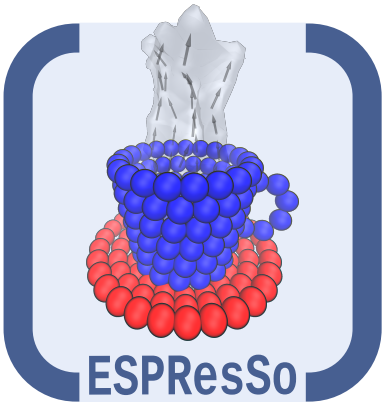


Tutorial: Object in fluid *

September 1, 2015

Contents

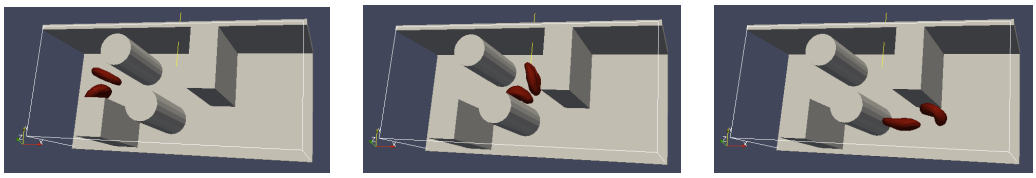
1 Introduction

This tutorial introduces some of the features of ESPResSo module Object in fluid (OIF). Even though ESPResSo was not primarily intended to work with closed objects, it appears very suitable when one wants to model closed objects with elastic properties,

*For ESPResSo 3.3.1

especially if they are immersed in a moving fluid. Here we offer a step by step Tcl tutorial that will show you how to use this module. The resulting code can be run using `Espresso <file>` from the ESPResSo source directory. It produces .vtk files that can then be visualized using Paraview.

The OIF module was developed for simulations of red blood cells flowing through microfluidic devices and therefore the elasticity features were designed with this application in mind. However, they are completely tunable and can be modified easily to allow the user model any elastic object moving in fluid flow.



2 Basic set up

In order to be able to work with elastic objects, one needs to configure ESPResSo with the following options in myconfig.hpp:
#define LB (or #define LB_GPU)
#define LB_BOUNDARIES (or #define LB_BOUNDARIES_GPU)
#define EXTERNAL_FORCES
#define MASS
#define CONSTRAINTS
#define BOND_ANGLE
#define VOLUME_FORCE
#define AREA_FORCE_GLOBAL

To create an elastic object, we also need a triangulation of the surface of this object. Sample sphere and red blood cell are provided in the directory scripts/input. User can create her own in gmsh, salome or any other meshing software. The required format is as follows:

The file `some_nodes.dat` should contain triplets of floats (one triplet per line), where each triplet represents the x, y and z coordinates of one node of the surface triangulation. No additional information should be written in this file, so this means that the number of lines equals the number of surface nodes. The coordinates of the nodes should be specified in such a way that the approximate center of mass of the object corresponds to the origin (0,0,0). This is for convenience when placing the objects at desired locations later.

The file `some_triangles.dat` should also contain triplets of numbers, this time integers. These refer to the IDs of the nodes in the `some_nodes.dat` file and specify which

three nodes form a triangle together. Please, note that the nodes' IDs start at 0, i.e. the node written in the first line of `some_nodes.dat` has ID 0, the node in the second line, has ID 1, etc.

We can start our script by specifying these files:

```
set fileNodes  "input/cell_nodes.dat"
set fileTriangles "input/cell_triangles.dat"
```

And continue with setting up some molecular dynamics parameters of the simulation engine:

```
setmd time_step 0.1
setmd skin 0.4
thermostat off
```

The skin depth `skin` is a parameter for the link–cell system, which tunes its performance, but will not be discussed here in detail. The one important thing a user needs to know about it is that it has to be strictly less than half the grid size.

Next we need to specify the simulation box:

```
set boxX 50
set boxY 22
set boxZ 20
setmd box_l $boxX $boxY $boxZ
```

and define the walls and boundaries. For clarity, these have been placed in a separate file `boundaries.tcl`, which we'll go over in the next subsection. The source code of this boundaries script is included using the command

```
source boundaries.tcl
```

but note, that the boundaries could have been specified directly at this point.

Now comes the initialization of OIF module using

```
oif_init
```

This command creates all the global variables and lists needed for templates and objects that will come afterwards. Elastic objects cannot be created directly. Each one has to correspond to a template that has been created first. The advantage of this approach is clear when creating many objects of the same type that only differ by e.g. position or rotation, because in such case it significantly speeds up the creation of objects that are just copies of the same template. The following

command creates a template

```
oif_create_template template-id 0 nodes-file $fileNodes \
triangles-file $fileTriangles stretch 3.0 3.0 3.0 \
ks 0.05 kb 0.01 kal 0.01 kag 0.01 kv 10.0
```

Each template has to have a unique ID specified using keyword `template-id`. The IDs should start at 0 and increase consecutively. Another two mandatory arguments are `nodes-file` and `triangles-file` that specify data files with desired triangulation. All other arguments are optional: `stretch` defines stretching in x, y, z direction and `ks, kb, kal, kag, kv` specify the elastic properties of the object (stretching, bending, local area conservation, global area conservation, volume conservation respectively). The keywords can come in any order.

Once we have the template, we can start creating objects:

```
oif_add_object object-id 0 template-id 0 origin 5 15 5 \
rotate 0 0 [expr $pi/2] part-type 0 mass 1
oif_add_object object-id 1 template-id 0 origin 5 5 15 \
rotate 0 0 0 part-type 1 mass 1
```

Each object has to have a unique ID specified using keyword `object-id`. The IDs should start at 0 and increase consecutively. Another three mandatory arguments are `template-id`, `origin` and `part-type`. `template-id` specifies which template will be used for this object. `origin` gives placement of object's center in the simulation box. And `part-type` assigns the particle type to all nodes of the given object. It is generally a good idea to specify a different `part-type` for different objects since it can be then used to set up interactions among objects. The optional arguments are `rotate` and `mass`. Rotate takes three arguments - angles in radians - that determine how much the object is rotated around the x, y, z axes. (Note: if you want to use the variable `$pi`, you need to specify it beforehand, i.e. `set pi 3.14159265359`). The optional keyword `mass` takes one value and this mass will be assigned to each surface node of the object.

The interactions among objects are specified using

```
inter 0 1 soft-sphere 0.005 2.0 0.3 0.0
```

where after `inter` come the particle types of the two objects and `soft-sphere` with four parameters stands for the "bouncy" interactions between the objects, once they come sufficiently close. (There are also other interaction types available in ESPResSo.) Similar interaction is defined with the boundaries:

```
inter 0 10 soft-sphere 0.0001 1.2 0.1 0.0
inter 1 10 soft-sphere 0.0001 1.2 0.1 0.0
```

Here 10 (the second number after keyword `inter`) is the the type of all boundaries and obstacles.

Finally, we specify the fluid, either by

```
lbfluid grid 1 dens 1.0 visc 1.5 tau 0.1 friction 0.5
```

or

```
lbfluid gpu grid 1 dens 1.0 visc 1.5 tau 0.1 friction 0.5
```

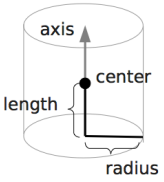
depending on the available computational resources. (The GPU computation can be two orders of magnitude faster than the CPU.)

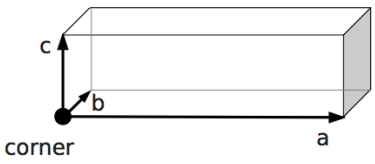
Specification of boundaries

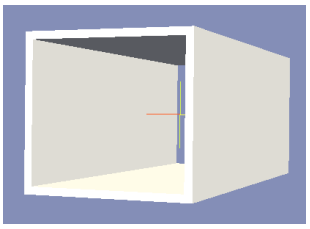
As was previously mentioned, all the boundaries and obstacles are conveniently grouped in separate file `boundaries.tcl`. This file contains two output procedures - one writes a rhomboid, another a cylinder into a .vtk file for later visualization. Below these procedures, one can specify the geometry of the channel. Here we only go over rhomboids and cylinders, but note that other boundary types are available in ESPResSo.

The rhomboid is a 3D structure specified by one corner and three vectors originating from this corner. It can be a box and in that case the three vectors give the length, width and height. However, there is no requirement that the vectors are perpendicular (to each other or to the walls). It is a standard ESPResSo command.

Cylinder is specified by its center, radius, normal vector, and length. The length is the distance from center to either base, therefore it is only half the total "height". Note: the included `boundaries.tcl` script can only output cylinder with (0,0,1) normal.







Each wall and obstacle has to be specified separately as a fluid boundary and as a particle constraint. The former enters the simulation as a boundary condition for the fluid, the latter serves for particle-boundary interactions. Sample cylinder and rhomboid can then be defined as follows:

```
# obstacle cylinder1
set cX 16; set cY 17; set cZ 10;
set nX 0; set nY 0; set nZ 1;
set L 9
set r 3
set cylFile "output/cylinder1.vtk"
set n 20
output_vtk_cylinder $cX $cY $cZ $nX $nY $nZ $r $L $n $cylFile
constraint cylinder center $cX $cY $cZ axis $nX $nY $nZ radius $r \
length $L direction 1 type 10
lbboundary cylinder center $cX $cY $cZ axis $nX $nY $nZ radius $r \
length $L direction 1


# obstacle rhomboid1
set corX 25; set corY 1; set corZ 1;
set aX 5; set aY 0; set aZ 0;
set bX 0; set bY 20; set bZ 0;
set cX 0; set cY 0; set cZ 10;
set rhomFile "output/rhomboid1.vtk"
output_vtk_rhomboid $corX $corY $corZ $aX $aY $aZ $bX $bY $bZ \
$cX $cY $cZ $rhomFile
constraint rhomboid corner $corX $corY $corZ a $aX $aY $aZ b $bX $bY $bZ \
c $cX $cY $cZ direction 1 type 10
lbboundary rhomboid corner $corX $corY $corZ a $aX $aY $aZ b $bX $bY $bZ \
c $cX $cY $cZ direction 1
```

Note that the cylinder also has an input parameter `n`. This specifies number of rectangular faces on the side. The `direction 1` determines that the fluid is on the "outside".

To create a rectangular channel, there are two possibilities. Either the walls are specified as the `lbboundary wall` and `constraint wall` with normal and distance from origin. Alternatively, the channel can be built using four flat rhomboids as can be seen in the picture above.

Another way how to work with boundaries, is to set them up using `lbboundary` and `constraint`, make sure each boundary has a different type (the object-boundary interactions have to be modified accordingly) and then following command can be used for output of all of them at once:

```
lbfluid print vtk boundary "boundary.vtk"
```

The differences in visualization in these two approaches are discussed later in this

tutorial.

3 Running the simulation

One last thing needed before we can proceed to the main part of the simulation code, is to get the fluid moving. This can be done by setting the velocity of the individual `lbnodes` on one side of the channel and letting the flow develop, but this is not a very convenient setup because it has to be done at every integration step and the tcl-C communication slows down the computation. The alternative is to set up a wall/rhomboid with velocity. This does not mean that the physical boundary is moving, but rather that it transfers specified momentum onto the fluid. This can be done using the command

```
lbboundary rhomboid velocity 0.01 0 0 corner 0 1 1 a 1 1 1 \
b 0 [expr $boxY-1] 1 c 0 1 [expr $boxZ-1] direction 1
```

Now we can integrate the system:

```
set steps 200
set counter 0
while { $counter<150} {
    set cycle [expr $counter*$steps]
    puts "cycle $cycle"
    integrate $steps
    incr counter
}
```

The script will print out a cycle number every 200 MD steps.

4 Writing out data

We have already discussed how to output the walls and obstacles for later visualization, but we also need output for fluid and objects. The fluid output is done using the standard ESPResSo command

```
lbfluid print vtk velocity "output/fluid$cycle.vtk"
```

and for object output we have

```
oif_object_output object-id 0 vtk-pos "output/output_file.vtk"
```

This will save the positions of all surface nodes into the .vtk output file. The modified integration loop now looks like this:

```
while { $counter<150} {
    set cycle [expr $counter*$steps]
    puts "cycle $cycle"
    lbfluid print vtk velocity "output/fluid$cycle.vtk"
    oif_object_output object-id 0 vtk-pos "output/cell0_$cycle.vtk"
    oif_object_output object-id 1 vtk-pos "output/cell1_$cycle.vtk"
    integrate $steps
    incr counter
}
```

where each object has its own output file and a new file is written every `$steps` steps.

5 Visualization

For visualization we suggest the free software Paraview. All .vtk files (boundaries, fluid, objects at all time steps) can be loaded at the same time. The loading is a two step process, because only after pressing the Apply button, are the files actually imported. Using the eye icon to the left of file names, one can turn on and off the individual objects and/or boundaries. It is also possible to do this when one imports all the boundaries from a single .vtk file (created using command `lbfluid print vtk boundary "boundary.vtk"`), however only when each boundary had been assigned a unique type number and is then selected in the bottom left menu by this number.

Fluid can be visualized using Filters/Alphabetical/Glyph (or other options from this menu. Please, refer to the Paraview user's guide for more details).

Note, that Paraview does not automatically reload the data if they have been changed in the input folder, but a useful thing to know is that the created filters can be "recycled". Once you delete the old data, load the new data and right-click on the existing filters, you can re-attach them to the new data.

It is a good idea to output and visualize the boundaries and objects just prior to running the actual simulation, to make sure that the geometry is correct and no objects intersect with any boundaries. This would cause "particle out of range" error and crash the simulation.

6 Other available OIF commands

The OIF commands that we have covered so far are

```
oif_init
oif_create_template
oif_add_object
oif_object_output
```

Here we want to describe the rest of the currently available OIF commands and note that there are more still being added. We would be pleased to hear from you about any suggestions on further functionality.

`oif_info` prints out information about all global variables, currently available templates and added objects.

`oif_mesh_analyze` takes two mandatory arguments: `nodes-file nodes.dat` and `triangles-file triangles.dat`. Their required format is discussed at the beginning of this document, in Basic set up section. The three optional arguments: `orientation`, `repair` and `flip` determine what will be done with the mesh. `orientation` checks whether all triangles of the surface mesh are properly oriented, `repair` corrects the orientation of those that are not and `flip` flips the orientation of all triangles in the triangulation.

`oif_object_analyze` has only one mandatory argument, `object-id 0`, and the optional arguments specify what function will be performed. `origin` outputs the location of the center of the object, `pos-bounds b-name` computes the six extremal coordinates of the object. More precisely, runs through the all mesh points and remembers the minimal and maximal x-coordinate, y-coordinate and z-coordinate. If `b-name` is the name of one of these: `z-min, z-max, x-min, x-max, y-min, y-max` then the procedure returns one number according to the value of `b-name`. If b-name is `all`, then the procedure returns a list of six numbers, namely Xmin, Xmax, Ymin, Ymax, Zmin, Zmax. `volume` outputs the volume of the object, `surface-area` outputs the surface of the object and `velocity` outputs the average velocity of the object by calculating the average velocity of object's mesh points.

`oif_object_set` also has only one mandatory argument `object-id 0`. The optional arguments are: `force valX valY valZ`, which sets the force vector (valX,valY,valZ) to all mesh nodes of the object. Setting is done using ESPResSo command `part $i set ext_force $valX $valY $valZ`. Note, that this command sets the external force in each integration step. So if you want to use the external force only in one iteration, you need to set zero external force in the following integration step. `origin posX posY posZ` moves the object in such a way that the origin has coordinates posX posY posZ.

`mesh-nodes "mesh-nodes.dat"` deforms the object in such a way that its origin stays unchanged, however the relative positions of the mesh points are taken from file mesh-nodes.dat. (The file mesh-nodes.dat should contain the coordinates of the mesh points with the origin's location at (0,0,0).) The command also checks whether number of lines in the mesh-nodes.dat file is the same as the corresponding value from oif_nparticles. `kill-motion` stops all the particles in the object (analogue to the `kill_motion` command in ESPResSo).

More information can be found in the OIF documentation and on our website www.cell-in-fluid.weebly.com.