Visualization of Ellipsoids

Bachelor’s Thesis in Physics

Presented by:
Holger Götz
May 14, 2018

Friedrich-Alexander-Universität Erlangen-Nürnberg

Supervisor: Prof. Dr. Klaus Mecke
Carried out at: Department of Physics, Umeå University
Supervisor in Umeå: Peter Olsson
# Contents

1. **Introduction**  
2. **Information about the simulations**  
   2.1. Shape of used particles  
   2.2. A simulation cycle  
3. **Quaternions**  
   3.1. Definition and properties  
   3.2. Describing rotations  
   3.2.1. Examples  
   3.3. Quaternions to rotation matrix  
4. **Blender**  
   4.1. The Python API  
   4.2. Animations  
   4.3. Colouring and texture  
5. **The visualization script**  
   5.1. The basic procedure  
   5.2. The input file  
   5.3. Applying of one particle configuration  
   5.3.1. Setting the location  
   5.3.2. Contact point highlighting  
   5.4. The render process  
   5.5. How to use this script  
6. **The Code**  
   A.1. Python files  
   A.2. Cpython files  
7. **References**
1. Introduction

The intention of this project is to create movies of particle simulations with Blender [1]. Where in this specific case the simulations are contact only simulations of frictionless ellipsoids within the context of the so called jamming transition. Within this document we first give more information about the simulations that are the base for the project. After that we introduce quaternions that are used by the simulations to describe the rotations of the particles. And finally we write about Blender, the program we used, and the actual script we created.

2. Information about the simulations

In order to better understand why specific things are done, this section is intended to provide more information about the simulations that shall be visualized. Though we encourage looking at e.g. [2] for further information.

The simulations are molecular dynamics ones of frictionless spheres or ellipsoids with contact only interactions. One special thing about them is that they are carried out at zero temperature. This means that the system, if not disturbed, will get stuck in a steady state with balanced forces after some time. The purpose of the simulations is to investigate the so called jamming transition. The jamming transition is a transition between the solid and the liquid state where both are equally disordered.

To prevent getting a crystal the simulations are done with two different sizes of particles such that there are equally as many small as big ones. To further prevent the system of getting into a steady state and to introduce the jamming transition the simulation is disturbed via shearing. This means that the simulation box is tilted with the time dependent tilt being described via the tilt variable $\gamma = t\dot{\gamma}$. $\dot{\gamma}$ here is the shear-rate that determines the change of $\gamma$ in time. For the purpose of the simulations the tilt variable is split into an integer part ($\gamma_1$) and a floating point part ($\gamma_0$) where $|\gamma_0| < 0.5$ so that one then jumps directly from $\gamma_0 = 0.5$ to $\gamma_0 = -0.5$. Lees-Edwards boundary conditions are used to introduce this tilt. This means that a particle at the position $(x, y, z)$ is identical to the particle at the position $(x \pm \gamma L_y, y \pm L_y, z)$, where $L_y$ is the size of the simulation box in the $y$ direction. For the $x$ and $z$ coordinates periodic boundary conditions apply. With this setup the shearing indeed can go on indefinitely, as the simulation has to only account for the $\gamma_0$ part. For this purpose the red marked part in figure [1] is shifted to the left side when $\gamma_0$ jumps from $+0.5$ to $-0.5$. With this in mind the visualization will use only $\gamma_0$, too.
2. Information about the simulations

![Figure 1: Top view of the simulation box with a tilt of $\gamma_0 = +0.5$ on the left and $\gamma_0 = -0.5$ on the right side. The unsheared box is indicated with green dots. Identical parts are marked with the same colour.](image)

2.1. Shape of used particles

As the simulated particles are ellipsoids, they are constructed via their axes. These are calculated by using two parameters $\alpha$ and $d$.

\[
\begin{align*}
  d_x &= d\alpha^{2/3}, \\
  d_y &= d\alpha^{-1/3}, \\
  d_z &= d\alpha^{-1/3},
\end{align*}
\]

Note that $d_x d_y d_z = d^3$ so that the volumes of the particles do not depend on $\alpha$. To account for the two different sizes of the particles $d$ is set to be $d_b = 1.4$ or $d_s = 1.0$ for the big and small ellipsoids respectively. For the description of the rotation we use quaternions. An example of one big and one small particle is shown in figure 2.

2.2. A simulation cycle

Before the simulations start the particles are placed in system with arbitrary positions and rotations. To initialize the starting positions and rotations of the particles for the actual simulation run the simulations are executed for a few iterations without recording anything. Alternatively the final configuration of a previously done simulation may be used.

In the actual simulation run then the positions and rotations are saved for each step so that one effectively obtains them as a function of time.

Typically the simulations are done with around 1024 particles and a shear-rate $\dot{\gamma}$ between $10^{-11}$ and $10^{-1}$. Mostly there are about 1000 to 10000 time steps with a length of about 0.05. Though to be able to visualize the simulations they are executed with a reduced number of only 64 particles.
3. Quaternions

As the simulations use quaternions to specify the rotations of the particles we here give a short introduction to them. This section is based on [3] and [4].

3.1. Definition and properties

Quaternions are an extension of the complex numbers by the additional imaginary units $j$ and $k$. These fulfil the following properties

$$i^2 = j^2 = k^2 = ijk = -1.$$

With this definition the imaginary units are anti commutative so that e.g. $ij = -ji$.

A general quaternion can then be written as

$$q = c_0 + c_i i + c_j j + c_k k \equiv q_0 + q_1 + q_2 + q_3.$$

Alternatively a quaternion may be represented in the form

$$q = q_0 + \vec{q} = [\vec{q}, q_0],$$

where $\vec{q} = c_i i + c_j j + c_k k$ is a vector within the associated three dimensional vector space spanned by the basis $\{i, j, k\}$.

For further descriptions we define two quaternions $q = [\vec{q}, q_0]$ and $q' = [\vec{q}', q_0']$. The following properties can then be found:
3. Quaternions

**Multiplication:** 
\[ qq' := [\vec{q} \times \vec{q}'' + q_0q_0' + q'0\vec{q}0' - \vec{q} \cdot \vec{q}'']. \]

**Addition:** 
\[ q + q' := [\vec{q} + \vec{q}' + q_0 + q_0']. \]

**Conjugate:** 
\[ q^* := [-\vec{q}, q_0], \quad (qq')^* = q^*q^*, \quad (q^*)^* = q. \]

**Norm:** 
\[ N(q) := qq^* = \sum_{i=0}^{3} q_i^2 \] with \( N(qq') = N(q)N(q'), \quad N(q^*) = N(q). \]

**Inverse:** 
\[ q^{-1} = q^*/N(q), \text{ if } N(q) \neq 0. \]

Where \( \times \) is the cross product and \( \cdot \) is the scalar product between three dimensional vectors.

### 3.2. Describing rotations

Quaternions are a useful way to describe rotation as e.g. they only require four numbers to be saved whereas e.g. a rotation matrix requires 9 (or at least 6) numbers [5]. Also the axis and angle of a rotation can easily be obtained by or converted into a quaternion. Specifically if we want to describe a rotation by the angle \( \theta \) around an unit vector \( \vec{u} \) we define an unit quaternion \( q \) as

\[ q = q_0 + \vec{q} = \cos(\frac{\theta}{2}) + \sin(\frac{\theta}{2})\vec{u}. \]

Then the vector

\[ \vec{w}' = q\vec{w}q^* \]

is the vector obtained from the rotation of the vector \( \vec{w} \) by the angle \( \theta \) around \( \vec{u} \).

**Proof**

We define the quaternions \( v_0 = [\vec{v}_0, 0] \) and \( v_1 = [\vec{v}_1, 0] \) with \( \vec{v}_0, \vec{v}_1 \in \mathbb{R}^3, \|\vec{v}_0\| = \|\vec{v}_1\| = 1, \vec{v}_0 \times \vec{v}_1 \neq 0 \). We further define \( q = v_1v_0^* = [\vec{v}_0 \times \vec{v}_1, \vec{v}_0 \cdot \vec{v}_1] = [\vec{u} \cos \frac{\theta}{2}, \sin \frac{\theta}{2}] \) with \( \vec{u} = \frac{\vec{v}_0 \times \vec{v}_1}{\|\vec{v}_0 \times \vec{v}_1\|} \) and \( \theta = 2\angle(\vec{v}_0, \vec{v}_1) \), where \( \angle(\vec{v}_0, \vec{v}_1) \) is the angle between the vectors \( \vec{v}_0 \) and \( \vec{v}_1 \). We also remember from the definition in Sec. 3.1 that the multiplication with an unit quaternion does not change the norm.

Now let’s look at \( v_0'v_1 := (qv_0q^*)v_1^* \),

\[
(qv_0q^*)v_1^* = (qv_0(v_1v_0^*))v_1^* = qv_0v_0v_1^*v_1^* = -v_0v_1 = q(-1)(-1) = v_1v_0^*.
\]
which means that
\[ \vec{v}_1 \cdot \vec{v}_0 = ||\vec{v}_1|| ||\vec{v}_0|| \cdot \cos \frac{\theta}{2} = \vec{v}_0' \cdot \vec{v}_1, \]
and
\[ \vec{v}_1 \times \vec{v}_0' = -\vec{v}_1 \times \vec{v}_0. \]

So that \( \vec{v}_0, \vec{v}_0' \) and \( \vec{v}_1 \) are in the same plane and \( \angle(\vec{v}_0, \vec{v}_0') = \theta \). So \( \vec{v}_0' \) is obtained by rotating \( \vec{v}_0 \) by the angle \( \theta \) around \( \vec{v} \).

Now we look at \( v_1' v_0^* := (q v_0 q^*) v_1^* v_0^* \),
\[(q v_1 q^*) v_0^* = (q(q v_0) q^* (q v_0 q^*)^*) = q = v_1 v_0^*, \]
which means that
\[ \vec{v}_1' \cdot \vec{v}_0' = \vec{v}_1 \cdot \vec{v}_0, \]
and
\[ \vec{v}_0' \times \vec{v}_1' = \vec{v}_0 \times \vec{v}_1. \]

So that \( \vec{v}_0, \vec{v}_0', \vec{v}_1, \vec{v}_1' \) are in the same plane and \( \angle(\vec{v}_0, \vec{v}_1) = \angle(\vec{v}_0', \vec{v}_1') \). With that we can conclude, that again \( \vec{v}_1' \) is obtained from a rotation of \( \vec{v}_1 \) by \( \theta \) around \( \vec{v} \). Now we have covered every angle that is perpendicular to \( \vec{v} \). So we only need to look at \( v' := q v q^* \),
\[ q v q^* = [\vec{q} \times \vec{v} , \vec{q} \cdot \vec{v}] q^* \\
= \cos \frac{\theta}{2} \vec{q} \times \vec{v} = \vec{v} \times \vec{q} \]
\[ = [\vec{v} \times \vec{q} + \vec{v} \cdot \vec{q}] q^* \]
\[ = v q q^* = v. \]

Now we have shown, that every vector \( \vec{u}' \) defined with \( [\vec{u}', 0] := q [\vec{u}, 0] q^* \) is \( \vec{u} \) rotated by the angle \( \theta \) around \( \vec{v} \).

**3.2.1. Examples**

**Trivial Quaternions**

The quaternion \( q = i = \vec{e}_1 \) then can be used to apply a rotation of \( \pi \) around \( \vec{e}_1 \). The quaternions \( j \) and \( k \) can be interpreted similarly. Whereas \( q = 1 \) or \( q = -1 \) would represent a rotation of the angle 0, \( 2\pi \) or rather no rotation at all.
3. Quaternions

Rotation around the x-axis

For a rotation of the angle $\theta$ around the x-axis we accordingly define the quaternion to be

$$q = [\vec{e}_x \sin(\frac{\theta}{2}), \cos(\frac{\theta}{2})].$$

(1)

E.g. for $\theta = \frac{\pi}{2}$ this would be $[\vec{e}_x \sqrt{2} \frac{\sqrt{2}}{2}, \sqrt{2}] = \sqrt{2}[\vec{e}_x, 1]$ and for $\theta = \frac{\pi}{4}$ this would be $\approx [\vec{e}_x, 0.38, 0.92]$. For further illustration these rotations are applied to a cuboid shown in figure 3.

3.3. Quaternions to rotation matrix

For the purpose of creating a rotation matrix we look at the quaternion as if it were a 4-vector

$$[\vec{q}, q_0] \equiv \begin{pmatrix} q_x \\ q_y \\ q_z \\ q_0 \end{pmatrix}.$$
3. Quaternions

We first look at \( pv \) with \( v = [\vec{v}, v_0] \),

\[
qv = [\vec{q} \times \vec{v} + q_0 \vec{v} + v_0 q_{0} - \vec{q} \cdot \vec{v}]
\]

\[
= \begin{pmatrix}
q_y v_z - q_z v_y + q_0 v_x + q_x v_0 \\
q_z v_x - q_x v_z + q_0 v_y + q_y v_0 \\
q_x v_y - q_y v_x + q_0 v_z + q_z v_0 \\
q_0 v_0 - q_x v_x - q_y v_y - q_z v_z
\end{pmatrix}
\begin{pmatrix}
v_x \\
v_y \\
v_z \\
v_0
\end{pmatrix}
\]

\[
= \begin{pmatrix}
q_0 & -q_z & q_y & q_x \\
q_z & q_0 & -q_x & q_y \\
-q_y & q_x & q_0 & q_z \\
-q_x & -q_y & -q_z & q_0
\end{pmatrix}
\begin{pmatrix}
v_x \\
v_y \\
v_z \\
v_0
\end{pmatrix}
\]

\[
=: R_q v.
\]

So we have the rotation matrix for the left multiplication. Now we have to look at \( vp^* \) (remember: \( q^* = [-\vec{q}, q_0] \),

\[
vq^* = [\vec{v} \times \vec{q}^* + \vec{v} q_{0}^* + v_0 q_{0}^* - \vec{v} \cdot \vec{q}^*]
\]

\[
= \begin{pmatrix}
q_y v_z - q_z v_y + q_0 v_x + q_x v_0 \\
q_z v_x - q_x v_z + q_0 v_y + q_y v_0 \\
q_x v_y - q_y v_x + q_0 v_z + q_z v_0 \\
q_0 v_0 + q_x v_x + q_y v_y + q_z v_z
\end{pmatrix}
\begin{pmatrix}
v_x \\
v_y \\
v_z \\
v_0
\end{pmatrix}
\]

\[
= \begin{pmatrix}
q_0 & -q_z & q_y & -q_x \\
q_z & q_0 & -q_x & q_y \\
-q_y & q_x & q_0 & q_z \\
q_x & q_y & q_z & q_0
\end{pmatrix}
\begin{pmatrix}
v_x \\
v_y \\
v_z \\
v_0
\end{pmatrix}
\]

\[
=: R_{q^*} v.
\]
To get the total rotation matrix for the rotation represented by $q$ we have to multiply $R_q$ and $R_q^*$.

$$R_q R_q^* = \begin{pmatrix}
q_0 & -q_z & q_y & q_x \\
q_z & q_0 & -q_x & q_y \\
-q_y & q_x & q_0 & q_z \\
-q_x & -q_y & -q_z & q_0
\end{pmatrix}
\begin{pmatrix}
q_0 & -q_z & q_y & q_x \\
q_z & q_0 & -q_x & q_y \\
-q_y & q_x & q_0 & -q_z \\
-q_x & -q_y & -q_z & q_0
\end{pmatrix}
$$

$$= \begin{pmatrix}
q_0^2 - q_z^2 - q_y^2 + q_x^2 & -2q_0 q_z + 2q_x q_y & 2q_0 q_y - 2q_x q_z & 0 \\
2q_0 q_z + 2q_x q_y & q_0^2 - q_z^2 + q_y^2 - q_x^2 & 2q_0 q_y + 2q_x q_z & 0 \\
-2q_0 q_y + 2q_x q_z & 2q_0 q_z + 2q_x q_y & q_0^2 - q_z^2 - q_y^2 + q_x^2 & 0 \\
0 & 0 & 0 & q_0^2 + q_y^2 + q_z^2 + q_x^2
\end{pmatrix}
$$

$$= \begin{pmatrix}
N(q) - 2(q_y^2 + q_z^2) & 2(q_x q_y - q_0 q_z) & 2(q_x q_z + q_0 q_y) & 0 \\
2(q_x q_y + q_0 q_z) & N(q) - 2(q_z^2 + q_x^2) & 2(q_0 q_z - q_y q_x) & 0 \\
2(q_x q_z - q_0 q_y) & 2(q_y q_z + q_0 q_x) & N(q) - 2(q_z^2 + q_y^2) & 0 \\
0 & 0 & 0 & N(q)
\end{pmatrix}
$$

We now have the total rotation matrix $R = R_q R_q^*$ with $Rv = qv q^*$. For three-dimensional rotation of a vector we can of course only use the $3 \times 3$ part of $R$, as $v_0 = 0$ for a vector.

## 4. Blender

Blender is a 3D creation suit. This means that it provides a wide range of possibilities to create images, animations or movies or even the graphics for a video game. As such it includes a variety of tools and methods for modeling, rendering, animation, video editing and even more [1]. It also supports scripting and extending via Python and is released under GNU General Public License (GPL). It can run on multiple platforms such as Linux, macOS and Windows.

### 4.1. The Python API

For us the most important part is Blender’s Python API that may be used for every action available in Blender. Python scripts in Blender can be used as a plugin to extend the functionality of Blender as well as to represent libraries for other Python scripts, to set various Blender preferences or even to be executed on startup to define some GUI elements or operators. Additionally there is the possibility to execute the script once when it is needed. As we are not really interested in extending Blender but in creating movies with it we chose to make a script that can be executed as needed. For that purpose we have three possibilities to execute them:

**The Python console:** With the console you get a very interactive and direct access to the object’s data. This makes it a good way to get to know the Python API or test
4. Blender

single pieces of code. For the execution of larger code-snippets we do not think of it as a very good possibility.

The text editor area: The editor offers the possibility to open, edit and execute scripts directly from the Blender GUI. Especially the possibility to see the result of a script in the 3D-view without the need of rendering anything is very useful in some cases. Though it might be worth mentioning that the Blender GUI is frozen during the execution of the script.

The command line interface: This interface gives a very convenient way to execute a Python script via Blender without the need to open any graphical interface. This has the big advantage, that it can be executed without any monitor present. Also the render process might be faster as none of the rendered frames has to be shown directly. Additionally it is possible to specify any options via custom command-line options rather than editing the script itself.

The script discussed in this document is designed to work with the command line interface, but it still is possible to execute it from inside Blender via the Text Editor area. A Python documentation of all the available methods and functions can be accessed through the Blender website [1].

4.2. Animations

As we want to create movies rather than pictures of the simulations we have to make animations. For this purpose we use so called keyframes. A keyframe is defined as a marker which stores values of specific properties for a specific point in time. For us these properties will mainly be the location and rotation of the particles. Keyframes also allow automatic interpolations to be done by Blender if they are not defined in two consecutive frames. Interpolation here means that Blender creates estimated states of properties specified in the keyframes for the frames in between so that a change might appear smoother. If the use of these interpolations is not desired, just as in our case, Blender has be told not to do them. Though we normally define a keyframe every frame, so no interpolation would occur anyway.

For our purpose the data that is contained in the keyframes is limited to the location and quaternion of a particle, as the further properties do not change. Only for the objects that are used to highlight the contact points we also insert keyframes for the size and visibility.

In the latter case we use keyframes for the visibility in order not to have to remove and recreate the objects used for highlighting. This increases the speed of the movie creation process significantly as the creation of Blender objects takes considerably longer than defining keyframes.
4.3. Colouring and texture

We create the particles in two colours, red and blue, in order to make it easier to observe a specific particle. This is achieved by the use of materials. Blender materials can represent various properties of an object like shininess, reflectance or transparency. We however use it for the colouring purpose only. Additionally, we add a cloud texture in grey to the materials in order to make it easier to follow the rotations of the ellipsoids as this can be quite hard if there is no pattern on the surface. The standard cloud texture we use is created by Blender using Perlin noise which is a gradient noise named after the developer Ken Perlin. For illustration purpose three particles, one without material and texture, one with a material only and one with both, are displayed in figure 4.

![Figure 4: Three particles, one with no material (left), one with a red material (middle) and one with a material and a cloud texture (right).](image)

5. The visualization script

The visualization script is intended to be used for the visualization of moving particles (as described in section 2.1). As these simulations are done with contact only interactions the contact points between the particles might be of interest and are therefore also made visible. In order to function the script has to be executed via Blender or somehow else be supplied with the necessary libraries. The script was tested with Blender version 2.69.

5.1. The basic procedure

After the input file is read the script creates the visible objects in Blender that will represent the particles. A keyframe is then inserted for the rotations and locations of the particles for every frame. Also the contact points are determined and highlighted.
via the use of additional graphical objects. When everything is specified the frames are rendered and saved by Blender.

5.2. The input file

The input file should look like the configuration file defined in the instructions. This means it should contain the following for every time step, and thereby frame, to be displayed:

- $N$ (int): the number of particles.
- $n_{\text{var}}$ (int): the number of variables used to describe the configuration (in this specific case that would be $13N + 2$).
- $L_x, L_y, L_z$ (3 double): the size of the simulation box.
- $x, y, z, v_x, v_y, v_z, \omega_a, \omega_b, \omega_c, q_0, q_1, q_2, q_3$ (13N double): the locations, velocities, rotational velocities and the quaternion describing one particle. Please note that the angle part of the quaternion should be specified as $q_3$, whereas $q_0, q_1, q_2$ are the parts for the $x, y$ and $z$ axis respectively. Also note that the values of the (rotational) velocities are not relevant for the visualization but still have to be supplied.
- $\gamma_0, \gamma_1$ (2 double): The tilt-variable $\gamma$ split up in two parts, where $\gamma_1$ is the integer part and $|\gamma_0| \leq 0.5$.

Every even particle is small and every odd one is big (starting numbering with 0).

5.3. Applying of one particle configuration

After reading the configuration file the animation data has to be applied. Specifically this means that for every frame the locations of the particles have to be set and the contact points have to be highlighted. So in this section we describe how this is done for one frame.

5.3.1. Setting the location

The setting of the locations may be very easy. If the particles shall be displayed as specified in the input file the properties can simply be transferred to the displayed objects. But as the simulations may specify the particles to be outside of the cuboid-shaped (not tilted) simulation box due to the shearing, it might still be wanted to display the particles inside the box. In this case some further calculations have to be done in order to relocate some ellipsoids back into the simulation box. Additionally this then prevents jumps in the created movies that are introduced when the tilt variable suddenly
5. The visualization script

changes from $\gamma_0 = 0.5$ to $\gamma_0 = -0.5$. Figure 5 shows two consecutive frames where the described jump happens.

To implement this relocation we check if a particle is actually outside of the simulation box. If this is the case it should be relocated by $\pm L_{x,y,z}$, where $L_{x,y,z}$ is the size of the simulation box in the respective direction. As this may lead to unwanted back and forth jumping of a particle in two or more consecutive frames we introduce a small tolerance that a particle may deviate from the simulation box. To further also detect such jumps if they are introduced by the input file we observe the position of a particle in relation to the origin.

So before we relocate a particle we check if $x, y$ and $z$, the coordinates of the particle, are greater than zero in the current and in last frame. If then e.g. $x > 0$ in both frames or $x < 0$ in both frames the particle is moved by $\pm L_x$ along the x axis if $|x| > 0.5 \cdot L_x + \tau_x$, with $\tau_x$ the tolerance for the x-axis. The sign (plus or minus) and thereby the direction of the shift along the axis is determined by the sign of the coordinate. This means that if e.g. $|x| > 0.5 \cdot L_x + \tau_x$ and $x > 0$ the particle is shifted by $-L_x$ along the x-axis. If $x < 0$ the particle would be shifted by $+L_x$ along the x-axis.

But if $x > 0$ in one frame and $x < 0$ in the other frame the particle is relocated by $\pm L_x$ if $|x| > 0.5 \cdot L_x - \tau_x$ along the x-axis. Again the direction is determined by the sign of the coordinate. Of course $x$ is only an example here and the same is also done for the $y$ and $z$ coordinate of the particle.

If none of the described options apply the particle will not be relocated. In any case we then insert a keyframe for the object with the new location and quaternion so that the visible object is also changing its position.

5.3.2. Contact point highlighting

After the locations are set for one frame, the contact points are highlighted. Obviously the first thing that has to be done is to check if and where two objects are having contact. For this purpose we use a provided C-function that is wrapped with Cython and returns a struct (or rather a Python dictionary) with information about the contact point if it exists. Cython is a compiler that makes it possible to write C-extensions for Python.
5. The visualization script

So then if two ellipsoids are in contact we have to check if the returned points are actually valid for the visible objects. This is done in order not to highlight points where the contacting objects are not displayed next to each other. The check is necessary because the C-function that returns the contact point between two ellipsoids is aware of the boundary conditions. If indeed the visible objects have contact as well we set a red cylinder to appear at the intersection points so that its top and bottom surface is tangent to the surface of the ellipsoids. The scale of the cylinders is calculated, as if there were two spheres of radius $r_{\text{max}}$ touching. Where $r_{\text{max}}$ is the maximum of the semi-axes of the touching ellipsoids. Particularly the scale is then calculated with:

$$1.1 \cdot \sqrt{r_{\text{max}}^2 - d^2}$$

where $d$ is the distance between the intersecting surfaces calculated via the provided contact points on the surfaces of each of the cylinders. This may seem as a rather arbitrary choice but it yields the desired appearance.

It might be also noteworthy how we handle the cylinders that highlight the intersections. Every cylinder created once can and will be reused for any other contact point that may occur. For this purpose we save the object of every created cylinder in a list. For each frame then the contacting points are calculated and highlighted with an element from that list. The properties such as the location, rotation and even the size of this object will then be adjusted via keyframes. If the list does not contain enough objects to mark all the points new ones are created as needed. If on the other hand there are too many cylinders in the list the rest will be set to invisible for that frame, so that they don’t have to be removed and can be reused in later frames. We reuse the cylinders as creating and deleting new or old cylinders would make no sense and would take considerably longer.

5.4. The render process

Before now finally the render process starts, the camera is set to view top down onto the particles and focus them all. Also the lamp is set to light from the same direction as the camera. Shadows are turned off, as this leads to a better view of the particles that are not in the top plane.

When all is set the render process is finally started and Blender creates a movie that is saved to the desired location. For illustration of the result, figure 6 shows one frame of a created movie.

5.5. How to use this script

This script is intended to be used via the command-line-interface of Blender. A basic command would look like

```
bash$ blender --background --python ./script/render_particles.py \  -- Lat/0064_r7000_GDf100_KDk500_Ml100_EL120_idt20
```
5. The visualization script

Figure 6: Sample frame created by the described script.

where everything after "--" is passed as a command line argument to the Python script. The following (and more) command line parameters can be used:

- **-o ofilename** With this option an output file can be specified. Please note, that Blender might add some numbers describing the rendered frames. If -o is not defined the script uses the input filename as a base.

- **-of ofolder** With this option it is possible to define an output folder to save the generated movie in, if -o is not provided. If neither -af nor -o are specified the script will use the directory of the input file.

- **-nf number** Tells the script to apply only every numberth frame (default is 1).

- **-f** Specifies that the particles should be relocated in order to always be displayed in the simulation box.

- **-rp number** Sets the resolution to 0.01\cdot number\cdot(1920,1080) (default is 100).

- **-end number** Tells the script to stop after reaching frame number (default is 1000).

- **-h, –help** Shows a help message and displays all available options.

If the script is executed directly e.g. via ./render_particles.py the script will only read the configurations and then exit. Also a command how to use the script via Blender is printed out.

A second possibility is to execute the script from inside Blender via the Text Editor area. With this method however it is required to directly specify the command line
arguments within the script. Also Blender has to be started from the directory with the
custom libraries, as otherwise there might be some import problems. But as this is not
the intended usage, no further explanation is provided at this point.

A. The Code

A.1. Python files

The file render_particle.py, displayed in listing[1] is the main script. It imports methods
from utilities.py (shown in listing[2]) as well as from the library compiled with Cython
(see Appendix A.2) and uses them to create the movie. A short description of what each
function does is included as comments in the listings.

Listing 1: render_particles.py

```python
#!/usr/bin/env python3
import sys, os
import math
import random
import array
from statistics import mean

# Add the current directory to the python-path if it is not already included
if os.path.dirname(os.path.abspath(__file__)) not in sys.path:
    sys.path.insert(0, os.path.dirname(os.path.abspath(__file__)))
    print("Added file-directory to pythonpath in order to find the custom library")

# Try some steps to import functions from the custom C-library
try:
    from module import check, check_get_overlap
except:
    sys.path.insert(0, os.path.abspath(".\"))
    try:
        from module import check, check_get_overlap
    except:
        raise ImportError("Could not import custom library. \nPlease consider starting Blender from " \n"the directory that contains it")

# import functions from utilities.py
from utilities import read_conf, get_alpha, get_semi_axes

# Specify some global variables.
# Some of them might be overwritten later.
main = 0
VERBOSE = 0
TOLERANCE = None
```
A. The Code

```python
render = 1

# Try to import libraries provided by Blender
try:
    import bpy
    from mathutils import Vector, Quaternion
    main = 0
    args = sys.argv[sys.argv.index("--") + 1:]
except:
    main = 1
    args = sys.argv[1:]
    print("ERROR: Not possible to import the module \"bpy\".\n"In order to render please execute \"blender \"
"--background --python {} -- {}\".\n
sys.argv[0], " ".join(sys.argv[1:])), file=sys.stderr)

# Parse Arguments
if len(args)<1:
    # Define Arguments here in order to
    # executed via the Text Editor Area
    print("not main")
    args = [ '../Lat/0064_r7000_GDf100_KDk500_M1100_EL120_idt20',
            '-r', '0', '--start', '420', '--end', '920', '-f']
    main = 0
    render = 0

    # Define the arguments
import argparse
parser = argparse.ArgumentParser(description='Create animations.')
parser.add_argument('file1', metavar='filename', type=str,
                    help='the input filename')
parser.add_argument('-o', dest='ofile', type=str,
                    help='output filename')
parser.add_argument('-of', dest='ofolder', type=str,
                    help='output folder - Note that this will be
                         neglected if \'-o\' is used')
parser.add_argument('-nf', dest='nframe',
                    type=int, default=1,
                    help='use only any \nframe\'th
                         \'particle-configuration\')
parser.add_argument('-vf', '--video-frame', dest='vframe',
                    type=int, default=1,
                    help='apply one particle configuration every
                         \'vframe\' video-frame. Obviously this
                         may lead to non-smooth movement')
parser.add_argument('-f', '--force', action='store_true',
                    help='force objects to be in the box')
parser.add_argument('-r', '---render', dest='render',
```

18
A. The Code

```python
import time

def get_sun():
    """This method returns a Lamp object if found"""
    if(len(bpy.data.cameras) == 1):
        return bpy.data.objects["Lamp"]
    raise LookupError("Lamp not found")

def get_camera():
    """
```

This method returns the camera object if found

```python
if(len(bpy.data.cameras) == 1):
    return bpy.data.objects["Camera"]
raise LookupError("Camera not found")
```

```python
def set_camera_pos(pos="normal", limits=(0,0,0), adjust_sun=True):
    ""
    This method can be used to set the position of the camera,
    as well as the light source
    ""
    camera = get_camera()
    if pos=="normal":
        pass
    elif pos == "north":
        camera.location.x = 0
        camera.location.y = 0
        camera.location.z = 4*limits[2]
        camera.rotation_euler.x = 0
        camera.rotation_euler.y = 0
        camera.rotation_euler.z = 0
    if adjust_sun:
        sun = get_sun()
        sun.location.x = camera.location.x
        sun.location.y = camera.location.y
        sun.location.z = camera.location.z
        focus_all()
```

```python
def focus_all():
    ""
    Selects every object in order to focus the camera on them.
    ""
    # Select objects that will be rendered
    bpy.ops.object.select_all(action='DESELECT')
    for obj in bpy.context.visible_objects:
        if not (obj.hide or obj.hide_render):
            obj.select = True
    bpy.ops.view3d.camera_to_view_selected()
    bpy.ops.object.select_all(action='DESELECT')
```

```python
def render_and_save(fname, frame_end, frame_start=0, \
    resolution_percentage=50, shadows=False, \n    args=None, transparent=0, image=False):
    ""
    Starts the render process and saves the movie
    to a file of format AVI with the H264 codec
    ""
    # Setting render options
    print("Setting render options...")
```
scene = bpy.context.scene

scene.frame_start = frame_start
scene.frame_end = frame_end

render = scene.render

if not image:
    render.image_settings.file_format = "AVI_JPEG"
    render.ffmpeg.format = "AVI"
    render.ffmpeg.codec = "H264"

    # setting shadows
    render.use_shadows = shadows
    if transparent:
        render.alpha_mode = "TRANSPARENT"

    # setting output
    render.use_placeholder = True
    render.use_file_extension = False
    if not "." in fname:
        if args:
            fname += "_nF{:02d}".format(args.nframe)
        if not image:
            render.filepath = fname + "_.avi"
        else:
            render.filepath = fname + "_{}.png".format(frame_end - 1)
    else:
        render.filepath = fname

    render.resolution_percentage = resolution_percentage
    render.resolution_x = int(render.resolution_x)

print("Starting render process")
if image:
    bpy.ops.render.render(write_still=True)
else:
    bpy.ops.render.render(animation=True)

def cleanup():
    
    """ Removes all currently existing spheres, cubes and cylinders. Very helpful if this script is executed from inside blender. """
    bpy.ops.object.select_all(action='DESELECT')
    for i in bpy.data.objects:
        if "Sphere" in i.name or "Cube" in i.name or "Cylinder" in i.name:
            bpy.data.objects[i.name].select = 1
    bpy.ops.object.delete()
def create_cylinder_mat():
    
    """
    Creates (if not already existant) and returns a material for
    the cylinders, that are used to display the contact points.
    """
    try:
        return get_cylinder_mat()
    except:
        pass

    cylinder_mat = bpy.data.materials.new(name="Cylinder")
    cylinder_mat.type="SURFACE"
    cylinder_mat.diffuse_color = (255,0,0)
    return cylinder_mat

def get_cylinder_mat():
    """
    Returns the cylinder material created by
    'create_cylinder_mat' if existing, otherwise an appropriate
    ERROR will occur.
    """
    return bpy.data.materials["Cylinder"]

def remove_all_cylinders():
    """
    Removes all objects that contain "Cylinder" in their name.
    Helpful if script is used and tested via the Blender GUI
    """
    bpy.ops.object.select_all(action='DESELECT')
    for i in bpy.data.objects:
        if "Cylinder" in i.name:
            bpy.data.objects[i.name].select = 1
            bpy.ops.object.delete()

def check_in_ellipsoid(vec, obj, objCoord=1):
    """
    Checks if a vector vec is contained in the ellipsoid obj.
    """
    vec = vec.copy()
    quat = Quaternion(obj.rotation_quaternion).copy()
    matrix = quat.to_matrix().copy()
    matrix.invert()
    loc = obj.location
    scale = obj.scale

    # If vec is in the coordinate system of the ellipsoid no
    # transformation has to be done
    if objCoord:
        return get_number(vec, scale)
vec = vec - loc
vec = matrix * vec

return get_number(vec, scale)

def get_number(vec, scale):
    ""
    Calculates the ellipsoid function and returns the value
    ""
    ret = 0
    for i in range(3):
        ret += (vec[i]/(scale[i]))**2
    return ret

def highlight_all_intersections_ellipsoid(particles, objects, 
    alpha, L, gamma, elements = [], 
    mat=None, frame=1, limits=None):
    ""
    This function uses the custom Python module in order to
    highlight the intersection between the Ellipsoids with a
    red cylinder.
    ""
    length = len(particles)

    n_elements = len(elements)
    n_used = 0

    # Iterate through all particles
    for i in range(length):
        for j in range(i+1, length):
            # define the actual position of the particle, as this
            # might have been changed by the update_pos method.
            part_i = []
            part_i.extend(objects[i].location)
            part_i.extend(particles[i][3:])

            part_j = []
            part_j.extend(objects[j].location)
            part_j.extend(particles[j][3:])

            # check if the two particles are touching
            overlap = get_intersection_ellipsoid_overlap(part_i, i, 
                part_j, j, alpha, L, gamma)
            if not overlap: continue

            element = None
            if n_used < n_elements:
                element = elements[n_used]
            n_used += 1

            # reshow the cylinders that will be reused this frames
            element = highlight_intersection_ellipsoid(objects[i], 
                objects[j], overlap, element, frame, i, j, limits)
if element:
    if n_used >= n_elements:
        elements.append(element)
        n_used += 1

    # Set all cylinders that are not used to highlight an
    # intersection this frame to hide.
    for i in range(n_used, n_elements):
        elements[i].hide_render = True
        elements[i].hide = True
        elements[i].keyframe_insert(data_path="hide", index=-1)
        elements[i].keyframe_insert(data_path="hide_render", 
                                   index=-1)

return elements

def highlight_intersection_ellipsoid(obj1, obj2, overlap, element, 
                                    frame=1, i=None, j=None, limits=None):
    ""
    Highlights one specific intersection if the visible objects
    are intersecting too.
    Note: This is checked in order to prevent markings that
    would appear on 'border'-particles.
    ""
    global VERBOSE

    e = check_in_ellipsoid(Vector(overlap['r_c1']), obj1, 0)
    f = check_in_ellipsoid(Vector(overlap['r_c2']), obj2, 0)

    # To check if the visible objects are touching, too
    if (e - 1 > 1e-6 or f - 1 > 1e-6):
        if VERBOSE >=1 and j!= None and i!= None :
            print("Not contacting between object {0} and {1} " \
                  "(at least in blender)".format(i, j))
        return None

    loc = tuple(overlap['r_c'])

    # Calculate the rotation of the cylinder, so that it is nicely
    # adjusted to the surface of the contacting points.
    d21 = Vector((overlap['r_c'][0]-overlap['r_c1'][0], 
                  overlap['r_c'][1]-overlap['r_c1'][1], 
                  overlap['r_c'][2]-overlap['r_c1'][2]))
    quat = d21.to_track_quat("Z", "X")
    dist = d21.length
    r_max = max(max(obj1.scale), max(obj2.scale))

    # This scale is a approximate of the radius required for the
    # cylinders. This is calculated, as if there were two Spheres
    # of r_max touching
scale = 1.3* math.sqrt((r_max*2-dist/2)*dist/2)

if scale<0.08*r_max:
    scale = 0.08*r_max

scale = [scale, scale, r_max*0.05]

mat = create_cylinder_mat()

return create_or_update_element(loc, scale, quat, mat, frame, 
    element, limits)

def get_intersection_ellipsoid(ia, ii, ja, jj, alpha, L, gamma):
    
    Returns the contact point between ia and ja or 0 if
    there is no contact.
    
    rc_contact = array.array('d', [0,0,0])
    if check(rc_contact, [0], ia, ii, ja, jj, alpha, L, gamma):
        return rc_contact
    return 0

def get_intersection_ellipsoid_overlap(ia, ii, ja, jj, 
    alpha, L, gamma):
    
    Returns a dictionary with more information about the
    contact between the particles.
    
    rc_contact = array.array('d', [0,0,0])
    overlap = check_get_overlap(rc_contact, [0], ia, ii, ja, jj, 
        alpha, L, gamma)
    return overlap

def create_or_update_element(loc, scale, quat, mat=None, frame=1, 
    cylinder = None, limits = None):
    
    Updates the object properties for the given frame. If no
    object is passed a new cylinder will be created.
    
    if not cylinder:
        # Create a new cylinder if non provided
        bpy.ops.mesh.primitive_cylinder_add()
        bpy.ops.object.shade_smooth()
        cylinder = bpy.context.scene.objects.active
        cylinder.rotation_mode = "QUATERNION"
        if mat: cylinder.data.materials.append(mat)

        for i in range(frame):
            cylinder.hide=True
            cylinder.hide_render = True
A. The Code

cylinder.keyframe_insert(data_path="hide", frame=i)
cylinder.keyframe_insert(data_path="hide_render", frame=i)

set_location(cylinder, loc, limits)
cylinder.scale = scale
cylinder.rotation_quaternion = quat

cylinder.hide=False
cylinder.hide_render = False

# This is necessary for blender to realize that there are # some changes this frame
cylinder.keyframe_insert(data_path="location", index=-1, frame=frame)
cylinder.keyframe_insert(data_path="rotation_quaternion", index=-1, frame=frame)
cylinder.keyframe_insert(data_path="scale", index=-1, frame=frame)
cylinder.keyframe_insert(data_path="hide", index=-1, frame=frame)
cylinder.keyframe_insert(data_path="hide_render", index=-1, frame=frame)

return cylinder

def create_wireframe(x, y, z):
    ""
    Creates a cube displayed as a wireframe with a scale of
    (x,y,z) around the origin
    ""
    bpy.ops.mesh.primitive_cube_add()
bpy.ops.object.shade_smooth()
cube = bpy.context.scene.objects.active
cube.location = (0,0,0)
cube.scale = (x/2, y/2, z/2)

    mat = bpy.data.materials.new(name="Wire")
    mat.type="WIRE"
    mat.diffuse_color = (0, 0, 0)
cube.data.materials.append(mat)

cube.draw_type = "WIRE"
return cube

def create_sphere(loc = (0,0,0), scale = (1,2,3), q=(1,2,3,5), mat=None, limits=None):
    ""
    Creates and returns a sphere with the desired properties
    ""
A. The Code

```python
bpy.ops.mesh.primitive_uv_sphere_add()
bpy.ops.object.shade_smooth()
sphere = bpy.context.scene.objects.active

sphere.scale = scale

set_location(sphere, loc, limits)
sphere.rotation_mode = "QUATERNION"
sphere.rotation_quaternion = q

if mat:
    sphere.data.materials.append(mat)

return sphere

def create_cloud_mat(name, color, mat_grey=False):
    if mat_grey:
        mat = bpy.data.materials.new(name=name)
        mat.type="SURFACE"
        #Adding a texture to the material
tex = bpy.data.textures.new(name="Clouds", type="CLOUDS")
        slot = mat.texture_slots.add()
        slot.color = color
        slot.texture = tex
    else:
        mat = bpy.data.materials.new(name=name)
        mat.type="SURFACE"
        mat.diffuse_color = color
        #Adding a texture to the material
tex = bpy.data.textures.new(name="Clouds", type="CLOUDS")
        slot = mat.texture_slots.add()
        slot.diffuse_color_factor = 0.9
        slot.blend_type="MIX"
        slot.textures.add() = tex
        slot.texture_coords = "ORCO"

    return mat

def range1(start, end, step):
    """Returns a list with numbers beginning at start up to and
       not including end that are separated by step. Every number
       is converted to integer before appending."
    """
    ret = []
    curr = start
    while curr<end:
        ret.append(int(curr))
        curr += step
    return ret
```
def create_particles(particles, d_b = (1,1,1), d_s = (1,1,1), \\
                     limits=None):
    ""
    Creates spheres according to the data contained in
    particles. Also takes care of creating a material for the
    spheres with a cloud texture.
    ""

    # Creating a material for the Spheres
    sphere_mat = create_cloud_mat("Sphere", color=(0,0,2))
    sphere_mat2 = create_cloud_mat("Sphere2", color=(1,0,0))

    # Determine the index of particles that are displayed with
    # a different colour.
    different_particles = range(0, len(particles), 3.5)

    last_pos = []
    ellipsoids = []

    # Iterate through the particles
    for i in range(len(particles)):
        sys.stdout.write("Creating particle {} of {} ... \r".format(i+1, len(particles)))
        sys.stdout.flush()

        p = particles[i]

        # Set the size and material of the particle
        if i%2:
            scale = d_b
        else:
            scale = d_s
        if i in different_particles:
            mat = sphere_mat2
        else:
            mat = sphere_mat

        sphere = create_sphere(loc = (p[0], p[1], p[2]), \\
                                scale=scale, q = (p[-1], p[-4], \\
                                p[-3], p[-2]), mat=mat, \\
                                limits=limits)

        last_pos.append([sphere.location[0]>0, \\
                           sphere.location[1]>0, \\
                           sphere.location[2]>0])
        ellipsoids.append(sphere)

        print("Creating particle {} of {} ... done.".format(i+1, \\
                                 len(particles)))

    return ellipsoids, last_pos
def set_location(obj, loc, limits=None, lastpos=None, i=-1, \\
gamma=None, dim=3, tolerance=-1):
    
    """
    Sets the location of a provided obj to loc and adjusts loc
    if it is outside the given limits (and tolerances).
    If lastpos is provided it also controls that there are no
    jumps before a particle reaches the limits (with tolerances).
    """
    
    p_number = i
    global VERBOSE, TOLERANCE
    if not TOLERANCE and limits:
        # Set TOLERANCE to some specific value if not yet defined
        factor = 0.075
        factor_x = 0.02
        TOLERANCE = (factor_x*limits[0], factor*limits[1], \\
        factor*limits[2])
        if VERBOSE >1:
            print(" Tolerance : {}
Limits : {}".format(TOLERANCE, \\
            limits))
        if tolerance == -1:
            # Set tolerance to TOLERANCE if tolerance is not provided.
            tolerance = TOLERANCE

    if gamma:  
        gamma = gamma[0]

    loc = list(loc)
    if "Cylinder" in obj.name:
        obj.location = loc
        return
    
    if limits:
        for i in range(dim):
            if not lastpos or p_number==-1:  
                if i==1:
                    if gamma==None:
                        continue
                    continue
            loc[i] = check_tolerance(loc[i], limits[i], \\
            tolerance[i])
    else:
        last_state = lastpos[p_number][i]
        state = loc[i]>0
        if state == last_state:
            m = +1
        else:
            m = -1
        last_loc_i = loc[i]
        loc[i] = check_tolerance(loc[i], limits[i], \\
        m*tolerance[i])
        lastpos[p_number][i] = loc[i]>0
        if (i==1 and loc[i] != last_loc_i):
if loc[i] > last_loc_i:
    m = 1
else:
    m = -1
loc[0] += 2*gamma*limits[i]*m
loc[0] = check_tolerance(loc[0], limits[0], 0)
last_pos[p_number][0] = loc[0] > 0

obj.location = loc

return lastpos

def check_tolerance(pos, limit, tolerance):
    """Returns pos with the applied limit and tolerance."""
    if pos > limit + tolerance:
        pos -= 2*limit
    elif pos < -(limit + tolerance):
        pos += 2*limit
    return pos

def update_pos(particles, obj, limits=None, gamma=None, 
                last_pos=None):
    """Updates the position of the objects according to particles
    and inserts a keyframe."""
    for i in range(len(particles)):
        part_i = particles[i]

        obj_i = obj[i]

        last_pos = set_location(obj_i, (part_i[0], part_i[1], 
                                        part_i[2]), limits, i=i, 
                                        gamma=gamma, lastpos=last_pos)

        obj_i.rotation_quaternion = (part_i[-1], part_i[-4], 
                                     part_i[-3], part_i[-2])

        # Required so that blender realizes the change for this
        # frame.
        obj_i.keyframe_insert(data_path="location", index=-1)
        obj_i.keyframe_insert(data_path="rotation_quaternion", 
                              index=-1)

        return last_pos

if not main:
    try:
        bpy.ops.object.mode_set(mode='OBJECT', toggle=False)
    except RuntimeError:
        pass

    # Remove all visible spheres, cubes and cylinders if executed
from inside the Blender GUI.

cleanup()

alpha = get_alpha(args.file1)

with open(args.file1, "rb") as f:
    
    # Read the configurations from a file.
    
    n_conf = 0
    confs = []
    conf.gamma = []

    while True:
        conf = read_conf(f)
        if conf == None:
            break

        if VERBOSE >=2:
            print("read {} of {} variables".format(ncounted, n_var))
            print("gamma_0={}, gamma_1={}".format(gamma_0, gamma_1))

        n_conf +=1
        confs.append(conf[0])
        conf.gamma.append(conf[1])
        x, y, z = conf[2]

        if VERBOSE >=2:
            print()

    print("Read {} configurations!".format(n_conf))

    if VERBOSE>=1:
        print("alpha = {}".format(alpha))

    
    # Calculate the scale of the ellipsoids.
    
    if main:
        
        # If the programm is executed from outside of blender and
        # can't import bpy or mathutils and is still running it will
        # end here.
        
        pass
    else:

        wireframe = create_wireframe(x,y,z)

        # To deactivate the relocation of the particles inside the box,
        # set limits=None.
        
        if args.force:
            limits = (x/2,y/2,z/2)
        else:
            limits = None
particles = confs[0]
obj, last_pos = create_particles(particles, d_b = d_b, \
    d_s = d_s, limits=limits)

if len(confs) <= 1:
    print("Not enough confs found. Exiting...")
sysexit(0)

number_of_frame = args.start_frame
if args.end_frame == -1:
    args.end_frame = len(confs)
elements = []
gamma_last = 0
for i in range(args.start_frame, len(confs)):
    if (i+1)%args.nframe != 0:
        continue
    sys.stdout.write("Applying configuration {} of {} ... \r".format(i+1, n_conf))
    sys.stdout.flush()

    # Set the position of the objects for this frame
    bpy.context.scene.frame_set(number_of_frame)
    last_pos = update_pos(confs[i], obj, limits=limits, \
        gamma=conf_gamma[i], last_pos=last_pos)

    # Set the cylinders to highlight the contacting points
    highlight_all_intersections_ellipsoid(confs[i], obj, \
        alpha, [x,y,z], conf_gamma[i][0]+conf_gamma[i][1], \
        frame=number_of_frame, limits=limits)
    number_of_frame += args.vframe
    if number_of_frame >= args.end_frame:
        break

    print("Applying configuration {} of {} ... done.").format(i+1, n_conf)

    """
The following code is necessary in order to not get
unwanted animations, if a configuration is not applied to
every frame.
    """
if args.vframe > 1:
    for obj in bpy.data.objects:
        if obj.animation_data == None:
            continue
        fcurves = obj.animation_data.action.fcurves
        for fcurve in fcurves:
            for kf in fcurve.keyframe_points:
                kf.interpolation = 'CONSTANT'
set_camera_pos(pos="north", limits=(x/2, y/2, z/2))

if args.render:
    render_and_save(args.ofile, number_of_frame, 
        args.start_frame, args.resolution_percentage, 
        args=args, transparent=args.transparent, 
        image=args.image)

if args.measure_time:
    end_time = time.time()
print("Time passed: {:.03f}s".format(end_time-start_time))

import struct
import math

def get_int(byte):
    
    Converts a byte-sequence into an integer.
    
    return struct.unpack('i', byte)[0]

def get_double(byte):
    
    Converts a byte-sequence into a double.
    
    return struct.unpack('d', byte)[0]

def read_conf(f, v_set=0, sphere=0):
    
    Reads in a configuration file.
    
    global VERBOSE

    if sphere:
        n_pp = 9
    else:
        n_pp = 13
    n_start = f.read(4)
    if n_start == b'':
        # Check if the end of the file is reached.
        return None
    n_particles = get_int(n_start)
    n_var = get_int(f.read(4))

    if n_particles*n_pp+2 == n_var:
        if v_set and VERBOSE>=1:
            print("Configuration has {:d} particles described" \
                  "by {:d} variables".format(n_particles, n_var))
        else:
            # If there is a mismatch between the expected number and
            # given number of variables the reading process is broken
            # up at this point.
A. The Code

print("Perhaps there is something wrong...")
print("Breaking up read process...")
return None

# Read the limits/ size of the bounding box.
x = get_double(f.read(8))
y = get_double(f.read(8))
z = get_double(f.read(8))

ncounted = 0
if v_set and VERBOSE>=2:
    print("The bounds are: Lx={}, Ly={}, Lz={}".format(x,y,z))
particles = []
# Read all particles for this configuration.
for _ in range(n_particles):
    p = []
    for _ in range(n_pp):
        p.append(get_double(f.read(8)))
        ncounted += 1
    particles.append(tuple(p))

# Read and save gamma for this configuration.
gamma_0 = get_double(f.read(8))
ncounted += 1
gamma_1 = get_double(f.read(8))
ncounted += 1
gamma = (gamma_0, gamma_1)
if sphere:
    # skip a few bytes
    clist = get_int(f.read(4))
ncontacts = get_int(f.read(4))
f.read((8*clist+2*4)*ncontacts)
return particles, gamma, (x,y,z)

def get_alpha(filename, suppress=0):
    """
    Gets and returns alpha by analysing the filename.
    """
    if "EL" in filename:
        # Determine alpha.
        alpha1 = filename[filename.find("EL")+2: \
                   filename.find("_", filename.find("EL")+2)]
        alpha = float(alpha1)/(10**(len(alpha1)-1))
        return alpha
    if suppress:
        return None
    else:
        raise LookupError("Alpha not found")

def get_semi_axes(alpha, db=0.7, ds = 0.5):
    """
    Returns the sizes of the semi-axes of the ellipsoids.
    """
A. The Code

```python
a23 = math.pow(alpha, 2/3.0)
a13 = math.sqrt(a23)
d_b = (db*a23, db/a13, db/a13)
d_s = (ds*a23, ds/a13, ds/a13)
return d_b, d_s
```

A.2. Cpython files

As the functions for getting the contact points between two ellipsoids were already implemented in a C-library used for the simulations we access them from Python. There are several methods how to use C-code from inside Python. For our purpose we use **Cython** [6], a compiler that makes it possible to write extensions for python in C. The file `module.pyx`, shown in listing [3] contains redeclarations of the used C-functions and data types as well as some Python functions that call the desired C-functions and convert their results into python objects.

With the file `setup.py`, displayed in listing [4] we can then compile the library. This is done by executing the command `python setup.py build_ext --inplace`. After that a file named similar to `module.cpython-34m.so` will appear. If the folder of this shared object library is now included in the Python-path (it is easiest to just copy the file to the directory of the script itself) it is possible to import the library with the statement `import module`.

```
from cpython cimport array
import cython

""
A few redefinition of structs and functions from ellipsoids.h and other files so that they are accessible in python
"

cdef extern from "../bin/ellipsoids.h":

struct Overlap:
    # DESCRIBES THE INTERACTIONS BETWEEN PARTICLES
    # defined per couple of particles
    # Communication between functions
    int overlapping # boolean; 1=overlapping / 0=not overlapping
    # Variables for the computation of forces
    double mu # rescaling factor for the ellipsoids to be externally
        # tangent refreshed only if overlap.overlapping
    double r_c[3] # contact point
    double r_c1[3] # contact point on particle 1
    double r_c2[3] # contact point on particle 2
    double v1c2[3] # local velocity of particle 1 at its point of
        # contact with particle 2
    double v2c1[3] # local velocity of particle 2 at its point of
        # contact with particle 1
```

Listing 3: module.pyx
# variables for measurements

```c
# coordinates of the contact point on the
# surface of particle 1 in particle 1 frame
double r_c1_in1[3]
```

```c
# coordinates of the contact point on the
# surface of particle 2 in particle 2 frame
double r_c2_in2[3]
```

```c
# error checking
double err # error on the determination of the root of
    # polynomial H_{AB}
```

### struct Forces:

```c
# DESCRIBES THE EVOLUTION OF THE MOVEMENTS OF THE PARTICLES
# Forces
double el[3] # elastic forces
double dis[3] # dissipative forces
# Moments
double M[3] # moments of forces
# Quaternions
double wdot[3] # first derivative of the rotation vector in the
    # body frame
```

```c
cdef extern int check_contact ( double *rc_contact , double *mu , 
    double *ia , int ii , double *ja , int jj , 
    double alpha , double *L, double gamma )
```

```c
cdef extern Overlap * check_contact_get_overlap ( double *rc_contact , 
    double *mu , double *ia , int ii , double *ja , 
    int jj , double alpha , double *L, double gamma )
```

```python
@cython.locals(rc_contact=array.array)
def check(rc_contact, mu, ia, ii, ja, jj, alpha, L, gamma):
    ""
    This method calls the check_contact method
    ""
    rc_contact_arr = rc_contact
cdef array.array mu_arr = array.array('d', mu)
cdef array.array ia_arr = array.array('d', ia)
cdef array.array ja_arr = array.array('d', ja)
cdef array.array L_arr = array.array('d', L)

    return check_contact(rc_contact_arr.data.as_doubles, 
        mu_arr.data.as_doubles, ia_arr.data.as_doubles, ii, 
        ja_arr.data.as_doubles, jj, alpha, 
        L_arr.data.as_doubles, gamma)
```

```python
@cython.locals(rc_contact=array.array)
def check_get_overlap(rc_contact, mu, ia, ii, ja, jj, alpha, L, 
    gamma, force_overlap=0):
    ""
    This method calles a modified check_contact method that also
    ""
```
returns an Overlap-struct

rc_contact_arr = rc_contact
cdef array.array mu_arr = array.array('d', mu)
cdef array.array ia_arr = array.array('d', ia)
cdef array.array ja_arr = array.array('d', ja)
cdef array.array L_arr = array.array('d', L)

overlap = check_contact_get_overlap( \
rc_contact_arr.data.as_doubles, \
mu_arr.data.as_doubles, ia_arr.data.as_doubles, \
ii, ja_arr.data.as_doubles, jj, alpha, \
L_arr.data.as_doubles, gamma)

if overlap.overlapping or force_overlap:
    return create_dict_from_overlap(overlap)
return 0

cdef create_dict_from_overlap(Overlap *overlap):
    """
    Creates a python dictionary from the C-Overlap-struct
    """
    ret = {}

    ret['overlapping'] = overlap.overlapping
    ret['mu'] = overlap.mu

    r_c = []
    r_c1 = []
    r_c2 = []
    v1c2 = []
    v2c1 = []
    r_c1_in1 = []
    r_c2_in2 = []
    for i in range(3):
        r_c.append(overlap.r_c[i])
        r_c1.append(overlap.r_c1[i])
        r_c2.append(overlap.r_c2[i])
        v1c2.append(overlap.v1c2[i])
        v2c1.append(overlap.v2c1[i])
        r_c1_in1.append(overlap.r_c1_in1[i])
        r_c2_in2.append(overlap.r_c2_in2[i])
    ret['r_c'] = r_c
    ret['r_c1'] = r_c1
    ret['r_c2'] = r_c2
    ret['v1c2'] = v1c2
    ret['v2c1'] = v2c1
    ret['r_c1_in1'] = r_c1_in1
    ret['r_c2_in2'] = r_c2_in2
    ret['err'] = overlap.err
References


