**Documentation**

In this documentation, the script numerical\_simulation\_of\_polysaccharide\_in\_3d.blend is explained in detail. If the script is executed with the correct function calls, the desired molecule is simulated depending on the choice of input parameters and functions. The programming language used to create the script is Python. To visualize the geometrical objects in their desired structure, the free and open-source 3D computer graphics software tool Blender is being used. Blender has a Python API and therefore the capability to process python code. For the explanation of each function of the script, the following uniform system is used:

Input - The input that is passed to the functions.

Processing - The data passed by the input is processed.

Output - The processed data is stored as output in a return value, passed on to another function as output.

Since there are some functions that call functions inside themselves there are exceptions where the system is not being used for the nested functions for the sake of readability.

## **Utilized software and tools:**

Python version 3.10.12

Blender version 3.6.2

## **Blender built in modules:**

bpy, bpy.data, bpy.ops, bpy.props, bpy.types, bpy.context, bpy.utils, bgl, gpu, blf, mathutils

Convenience Imports: from mathutils import \*; from math import \*

Convenience Variables: C = bpy.context, D = bpy.data

## **Description of the main script:**

**Import statements:**

import bpy   
import math  
import mathutils  
import random

**Cylinder creation:**

def create\_cylinders(cylinder\_location, direction\_vector, branch\_name):  
 rotation\_to\_align = get\_rotation\_from\_direction(direction\_vector)  
 bpy.ops.mesh.primitive\_cylinder\_add(  
 radius=normalized\_radius\*0.4,   
 depth=normalized\_depth\_of\_monomer\*0.5,   
 location=cylinder\_location,   
 rotation=rotation\_to\_align  
 )  
 *# Creates a system to name the created cylinder* bpy.context.active\_object.name = branch\_name + "\_cylinder\_" + str(len(bpy.data.objects))

Function name: create\_cylinders()

**Input:**

|  |  |
| --- | --- |
| Input parameter: | Description: |
| cylinder\_location | List / Tuple containing the 3D coordinates for each center point of every cylinder. |
| direction\_vector | List / Tuple containing a vector that points in the direction of the next cylinder, ultimately dictating the orientation of each cylinder. |
| branch\_name | String that contains the name of the respective branch. Each cylinder belongs to a named branch. |

**Processing:**

rotation\_to\_align = get\_rotation\_from\_direction(direction\_vector)

This line calls the function get\_rotation\_from\_direction() as a parameter and saves the results in the rotation\_to\_align variable. The rotation\_to\_align variable then contains three Euler angles. The Euler angles represent the rotations around the X, Y and Z axes that are needed to align the cylinder’s default orientation with the desired vector which in this script usually represents the direction from one cylinder to the next one. In Blender the default orientation for mesh objects is the Y axis.

**Example:**

If rotation\_to\_align = (pi/2, 0, 0) it means that to align the cylinder with the desired direction\_vector, it must be rotated by pi/2 or 90° around the X-axis and no rotations are required around the Y and Z axes. This calculation is performed by the function get\_rotation\_from\_direction() which is explained later in this section.

Right after, the function bpy.ops.mesh.primitive\_cylinder\_add() is called which visualizes a cylinder in Blender using the input parameters:

radius = Controls the radius of the cylinder.  
depth = Controls the height / depth of the cylinder.  
location = Controls at which point the cylinder is visualized.  
rotation = Controls the orientation of the Cylinder. The previously calculated rotation\_to\_align variable is passed in the rotation parameter.

For more flexibility these input parameters are calculated by multiplying normalized global variables with a desired value.

bpy.context.active\_object.name = branch\_name + "\_cylinder\_" + str(len(bpy.data.objects))

This line creates a unique name for every cylinder that is getting created. It also assigns the respective cylinder to its correct branch.

**Output:**

This function does not return a specific Python value, it rather produces visual changes in the Blender scene by adding and naming cylinders.

def create\_cylinder\_between\_positions(pos1, pos2, branch\_name):  
 *# Calculates the direction* direction = []  
 for i in range(3):  
 direction.append(pos2[i] - pos1[i])  
  
 *# Calculates the midpoint* midpoint = []  
 for i in range(3):  
 midpoint.append((pos1[i] + pos2[i]) / 2)  
  
 create\_cylinders(midpoint, direction, branch\_name)

Function name: create\_cylinder\_between\_positions()

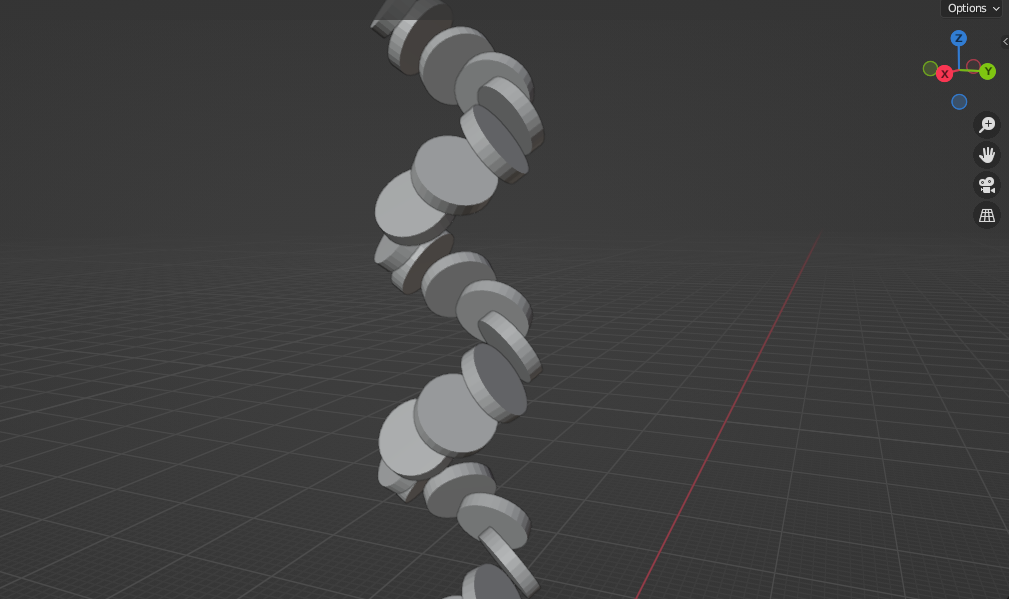
**Input:**

|  |  |
| --- | --- |
| Input parameter: | Description: |
| pos1 | 3D coordinates of a starting point in cartesian coordinates (X, Y and Z). |
| pos2 | 3D coordinates of an ending point in cartesian coordinates (X, Y and Z). |
| branch\_name | Name identifier for context information. |

**Processing:**

direction Stores a list where each element is a coordinate (X, Y and Z) and the final result of the calculation is a direction vector pointing from the start point to the end point.

One problem that occurred was, that the cylinders always pointed towards the center of the next cylinder, resulting in the following constellations of cylinders:

Figure 3.: Wrong orientation of the cylinders

To solve this problem and guarantee that the cylinders do not point towards the center point of the next cylinder but rather the outer surface of the next cylinder, the following calculations have been done:

midpoint = []  
for i in range(3):  
 midpoint.append((pos1[i] + pos2[i]) / 2)

The variable midpoint saves the middle position between two given points in cartesian coordinates. By setting this middle point as the center for each cylinder, each cylinder is aligned with pos2 but positioned halfway in the direction from pos1 to pos2. This ensures that the side surface of one cylinder is aligned with the side surface of the next cylinder and not the side surface of one cylinder with the bottom surface of the next one. Thus, the problem shown in the image is solved.

**Output:**

Like the function create\_cylinders(), this function also does not return a specific Python value, it rather processes data and calls another function with this processed data. It calculates the midpoint and direction between two points and calls the function create\_cylinders() directly with those values as parameters.

**Branch creation:**

def generate\_branch(num\_monomers, start\_position, start\_direction, branch\_name, theta=normalized\_theta):  
 monomer\_positions = [start\_position]  
 current\_position = mathutils.Vector(start\_position)  
 current\_direction = mathutils.Vector(start\_direction)  
  
 for i in range(num\_monomers):  
 move\_forward = current\_direction \* normalized\_distance  
 next\_position = current\_position + move\_forward  
 monomer\_positions.append(tuple(next\_position))  
 create\_cylinder\_between\_positions(tuple(current\_position), tuple(next\_position), branch\_name)  
 current\_position = next\_position  
 current\_direction = mathutils.Vector(rotate\_point(point=current\_direction,  
 rotation\_angles=(theta, phi, 0), center=(0,0,0)))  
  
 return monomer\_positions

Name of the function: generate\_branch()

**Input:**

|  |  |
| --- | --- |
| Input parameter: | Description: |
| num\_monomers | The total number of monomers out of which the branch is created. |
| start\_position | The first position where the branch starts to get created. |
| start\_direction | The initial direction in which the branch grows:  start\_direction = (1,0,0) means the branch initially grows along the x-axis.  start\_direction = (0,1,0) means the branch initially grows along the y-axis.  start\_direction = (0,0,1) means the branch initially grows along the z-axis. |
| branch\_name | Name identifier for context information. |
| theta | Theta controls the rotation around the YZ-plane relative to the start\_direction / the current\_direction. This means theta controls the rotation around the X-axis. |

It is important to note, that phi (φ) and theta (θ) both are relative to their current\_direction. In this script (0,1,0) is chosen as the start\_direction. For clear understanding of the angles in this case:

The direction of the cylinder and the direction of the entire branch is controlled via the two angles phi (φ) and theta (θ). If the branch has the start\_direction = (0,1,0) theta controls the rotation around the YZ-plane (in analogy with Euler angles you could say that relative to the direction of each cylinder, theta is the pitch or the rotation around the X-axis). Phi controls the rotation of the cylinder around the Y-axis. TODO: Add good image to illustrate phi and theta.

**Processing:**

|  |  |
| --- | --- |
| Line of code: | Functionality: |
| monomer\_positions = [start\_position] | Stores the previously mentioned starting position in a list. |

|  |  |
| --- | --- |
| Line of code: | Functionality: |
| current\_position = mathutils.Vector(start\_position) | The current\_position variable gets initialized with the start\_position and Blenders mathutils.Vector is being used for easier calculations. |

|  |  |
| --- | --- |
| Line of code: | Functionality: |
| current\_direction = mathutils.Vector(start\_direction) | The current\_direction gets initialized with the start\_direction and Blender mathutils.Vector is being used for easier calculations. |

for i in range(num\_monomers):  
 move\_forward = current\_direction \* normalized\_distance  
 next\_position = current\_position + move\_forward  
 monomer\_positions.append(tuple(next\_position))  
 create\_cylinder\_between\_positions(tuple(current\_position), tuple(next\_position), branch\_name)  
 current\_position = next\_position  
 current\_direction = mathutils.Vector(rotate\_point(point=current\_direction,  
 rotation\_angles=(theta, phi, 0), center=(0,0,0)))

|  |  |
| --- | --- |
| Line of code: | Functionality: |
| move\_forward = current\_direction \* normalized\_distance | The current\_direction which is a direction vector, gets multiplied with a scalar (regular number) to scale the vectors magnitude resulting in the correct length and direction of the new displacement vector which is saved in the variable move\_forward. |

|  |  |
| --- | --- |
| Line of code: | Functionality: |
| next\_position = current\_position + move\_forward | Add the displacement to the current position to get the next position which is saved in the variable next\_position. |

The next\_position then gets added to the monomer\_positions list as a tuple which is more efficient than using lists.

|  |  |
| --- | --- |
| Line of code: | Functionality: |
| create\_cylinder\_between\_positions(tuple(current\_position), tuple(next\_position), branch\_name) | create\_cylinder\_between\_positions() gets called with the calculated current\_position and next\_position. This function creates cylinders in between the two given points as described earlier. |

To ensure that the loop works for all monomers the current\_position and the current\_direction variables must be updated in every iteration. The current position can simply be set to current\_position = next\_position. The current\_direction must be calculated using the function rotate\_point(). The logic of this function is described in greater detail next. The return value is a vector that points in the desired direction which is dictated by the rotations around the three axes per cylinder center point. The current\_direction in this script will always point from one cylinder to the next one.

**Output:**

The return value is defined by the line return monomer\_positions. The monomer\_positions list is always updated with the next\_position (corresponding to the next position of a cylinder) in this line: monomer\_positions.append(tuple(next\_position)).

def rotate\_point(point, rotation\_angles, center=(0,0,0)):  
 vec = mathutils.Vector(point) - mathutils.Vector(center)  
 mat\_rot = mathutils.Euler(rotation\_angles).to\_matrix()  
 rotated\_vec = mat\_rot @ vec  
 rotated\_vec += mathutils.Vector(center)  
 return tuple(rotated\_vec)

Name of the function: rotate\_point()

**Input:**

|  |  |
| --- | --- |
| Input parameter: | Description: |
| point | This is the vector that is about to be rotated. Although the name suggests a point, it should be understood as a vector in 3D space. |
| rotation\_angles | These are Euler angles specified as (theta, phi, 0). In this context, theta and phi are used for rotations, and the third angle is set to zero. |
| center | This input parameter specifies the center point of rotation. |

**Processing:**

The logic of this function can be separated into three steps:

1. Translation to origin:

vec = mathutils.Vector(point) - mathutils.Vector(center)

The function starts by translating the input vector to the origin. This is achieved by subtracting the center from the point. The translated vector now originates from the origin.

2. Rotation using a matrix:

With the vector now centered around the origin, rotation can be applied directly using a rotation matrix without needing any complex adjustments. The Euler angles are converted into a 3x3 rotation matrix using mat\_rot = mathutils.Euler(rotation\_angles).to\_matrix(). This matrix represents the rotations about the individual x, y, and z axes. The matrix is then applied to the translated vector using matrix-vector multiplication: rotated\_vec = mat\_rot @ vec. The result gives back the rotated vector in the origin-centered reference frame.

3. Translation back to the original point:

After rotation, the rotated vector is translated back to its original reference frame by adding the center. This ensures the rotated vector gets back to its original position (but rotated) relative to other objects or points in the 3D space.

**Output:**

The function returns the final rotated vector, rotated\_vec, in its original frame of reference as a tuple. Essentially, this function calculates the rotations required to align the cylinders in a way that they line up with the side surfaces and form a helical or branch-like structure.

def recreate\_branch\_from\_positions(branch\_positions, branch\_name):  
 for i in range(len(branch\_positions) - 1):  
 create\_cylinder\_between\_positions(branch\_positions[i], branch\_positions[i+1], branch\_name)

Function name: recreate\_branch\_from\_positions()

**Input:**

|  |  |
| --- | --- |
| Input parameter: | Description: |
| branch\_positions | The branch\_positions contains a list of cartesian coordinate pairs (X, Y and Z) which in total form the respective branch. |
| branch\_name | The branch\_name contains the name of the respective branch. |

**Processing:**

This function visualizes the rotated branch by using the coordinates of the rotated branch and calling the create\_cylinder\_between\_positions() method. The idea behind this method is, that in Blender all objects remain in the scene until they are deleted. When rotating a branch, the data for the rotation is calculated based on the non-rotated branch but the non-rotated branch itself does not get repositioned nor deleted automatically. That’s why:

1: The non-rotated branch should be deleted.

2: The rotated branch should be visualized.

**Rotation methods:**

There are several rotation methods that perform different kinds of rotations and are described in this section.

def rotate\_branch\_in\_xy(branch, rotation\_angle\_xy\_plane):  
 if not branch:  
 return []  
  
 *# Temporarily shift everything so that the first point is at the origin* anchor = mathutils.Vector(branch[0])  
  
 shifted\_branch = []  
 for pt in branch:  
 shifted\_point = tuple(mathutils.Vector(pt) - anchor)  
 shifted\_branch.append(shifted\_point)  
  
 *# Rotate the shifted branch* rotated\_shifted\_branch = []  
 for pt in shifted\_branch:  
 rotated\_point = rotate\_point(pt, (0, 0, rotation\_angle\_xy\_plane))  
 rotated\_shifted\_branch.append(rotated\_point)  
  
 rotated\_branch = []  
 for pt in rotated\_shifted\_branch:  
 final\_point = tuple(mathutils.Vector(pt) + anchor)  
 rotated\_branch.append(final\_point)  
  
 return rotated\_branch

Name of the function: rotate\_branch\_in\_xy()

**Input:**

|  |  |
| --- | --- |
| Input parameter: | Description: |
| branch | A list containing cartesian coordinate pairs (X, Y and Z) which in total form the respective branch. |
| rotation\_angle\_xy\_plane | The angle that is used to rotate the branch around a fixed point in the XY-plane. |

**Processing:**

|  |  |
| --- | --- |
| Line of code: | Functionality: |
| anchor = mathutils.Vector(branch[0]) | The center point of the first cylinder in the branch which is the subject of rotation, is being saved in the anchor variable because it will serve as the rotation center. The tuple is also converted into a vector which is useful for performing mathematical operations on it. |

shifted\_branch = []  
 for pt in branch:  
 shifted\_point = tuple(mathutils.Vector(pt) - anchor)  
 shifted\_branch.append(shifted\_point)

By subtracting the anchor point from each monomer position in the branch the entire branch is effectively shifted so that the anchor point is at the origin (0,0,0). All other points keep their distances relative to the anchor point meaning that the entire branch is shifted and the anchor point which is also the first monomer of the branch is at (0,0,0).

*# Rotate the shifted branch* rotated\_shifted\_branch = []  
 for pt in shifted\_branch:  
 rotated\_point = rotate\_point(pt, (0, 0, rotation\_angle\_xy\_plane))  
 rotated\_shifted\_branch.append(rotated\_point)

In the next part of the function, the rotation is applied. Every monomer cartesian coordinates (X, Y, and Z) is rotated using the function rotate\_point(). The tuple (0, 0, rotation\_angle\_xy\_plane) represent Euler angles. The first and the second angle is 0 meaning no rotation about the X-axis and Y-axis are applied. The third angle is the angle that results in a rotation of the points in the XY-plane because it is around the Z-axis. For understanding, by rotating around the Z-axis the object is rotated in the XY-plane.

The rotated positions (in total forming the rotated branch) are then shifted back to the mother branch.

There is another function called rotate\_branch\_in\_xy() where the whole function is identical to this one except the rotations are applied around the X-axis resulting in a rotation of the branch in the XZ-plane.

**Output:**

The return value is a list containing the coordinates for all the points that represent the whole rotated branch as cartesian coordinates.

**Visualization helper functions:**

def clear\_mesh\_objects():  
 bpy.ops.object.select\_all(action='DESELECT')  
 bpy.ops.object.select\_by\_type(type='MESH')  
 bpy.ops.object.delete()

Name of the function: clear\_mesh\_objects()

**Input:**

No input

**Processing:**

In Blender all objects remain in the scene until they are deleted. When rotating a branch, the data for the rotation is calculated but the objects from the non-rotated branch are not moved. This Method deletes all mesh objects which the rendered visualization of the data is. Using this function at the beginning of the Blender script, also guarantees that if the script is executed multiple times, no objects from former script executions are overlapping with the rendered objects of the actual script execution.

**Output:**

There is no specific output. This function deletes objects and therefore manipulates the visual appearance of the Blender scene.

def clear\_specific\_branch(branch\_name):  
 if not isinstance(branch\_name, (str, tuple)):  
 print("Error: branch\_name should be a string or a tuple of strings!")  
 return  
  
 *# All objects in the whole Blender scene are deselected* bpy.ops.object.select\_all(action='DESELECT')  
  
 *# All the objects related to the respective branch are selected* for obj in bpy.data.objects:  
 if obj.type == 'MESH' and obj.name.startswith(branch\_name):  
 obj.select\_set(True)  
  
 *# All selected objects are deleted* bpy.ops.object.delete()

Function name: clear\_specific\_branch()

**Input:**

|  |  |
| --- | --- |
| Input parameter: | Description: |
| branch\_name | String that contains the name of the respective branch. Each cylinder belongs to a named branch. |

**Processing:**

A branch\_name is given as the function input parameter. All objects in the whole Blender scene are getting deselected. In the next step the objects related to the given branch\_name is selected and after that all the selected objects are being deleted. This achieves that all objects from a specific branch are deleted.

**Output:**

There is no specific output. This function rather deletes specific objects and therefore manipulates the visual appearance of the Blender scene.