Recursive function definition:

# defining a recusive function to multiply all individual matrices for full alignment to 1st component

f = []

n = range(len(idv\_lh\_trans\_matrices))

def transformation\_lh\_matrix\_gen(idv\_lh\_trans\_matrices,n,f):

if (n == 0):

f.append(idv\_lh\_trans\_matrices[n]))

return idv\_lh\_trans\_matrices[n]

else:

previous\_matrix = transformation\_lh\_matrix\_gen(idv\_lh\_trans\_matrices,n-1,f)

current\_matrix = np.dot(previous\_matrix,idv\_lh\_trans\_matrices[n])

return current\_matrix

print("f = ",f[1])

import os

from stl import mesh

import matplotlib.pyplot as plt

from mpl\_toolkits.mplot3d import Axes3D

folder\_path = "C:\Users\whows\OneDrive - King's College London\Documents\BEng project\STL\Link02.stl"

stl\_file\_names = ["Link02.stl","Link03.stl","Link04.stl","Link05.stl","Link06.stl","Link07.stl","Link08.stl","Link09.stl","Link10.stl",

"Link11.stl","Link12.stl","Link13.stl","Link14.stl","Link15.stl","Link16.stl","Link17.stl","Link18.stl","Link19.stl",

"Link20.stl","Link21.stl"]

stl\_file\_paths = [os.path.join(folder\_path, file\_name) for file\_name in stl\_file\_names]

def load\_stl\_files(file\_paths):

meshes = [] # List to hold the mesh objects

for file\_path in file\_paths:

stl\_mesh = mesh.Mesh.from\_file(file\_path) # Loading the STL file

meshes.append(stl\_mesh) # Adding the mesh to the list

return meshes # Returning the list of meshes

def transformed\_meshes(meshes,trans\_matrices)

# Visualization Function

def visualize\_robot(meshes):

fig = plt.figure()

ax = fig.add\_subplot(111, projection='3d')

for stl\_mesh in meshes:

for i in range(len(stl\_mesh.vectors)):

triangle = stl\_mesh.vectors[i]

ax.add\_collection3d(plt.Polygon(triangle, edgecolor='k', alpha=0.3))

scale = stl\_mesh.points.flatten('F')

ax.auto\_scale\_xyz(scale, scale, scale)

ax.set\_xlabel('X')

ax.set\_ylabel('Y')

ax.set\_zlabel('Z')

ax.set\_title('Robot Visualization')

plt.show()

# loading in the stl files

robot\_components = load\_stl\_files(stl\_file\_paths)

from stl import mesh

import numpy as np

# load an stl files

stl\_mesh\_02 = mesh.Mesh.from\_file("Link02.stl")

stl\_mesh\_03 = mesh.Mesh.from\_file("Link03.stl")

stl\_mesh\_04 = mesh.Mesh.from\_file("Link04.stl")

# stl mesh of vertices

vertices = stl\_mesh.vectors

# Transformations regarding the STL files for operational use in Reinforcement Learning algorithm

import numpy as np

# array containing all indivdual left handed transformation matricies

idv\_lh\_trans\_matrices = []

# array containing all individual right handed transformation matricies

idv\_rh\_trans\_matrices = []

# array containing all left handed transformation matrices

lh\_trans\_matrices = []

# array containing all right handed transformation matrices

rh\_trans\_matrices = []

# Class for transformations matricies for STL file realignment

class TransformationMatrix():

# constructor initalising variables and calling matrix function

def \_\_init\_\_(self,alpha,theta,a,d):

# conversion to radians from degrees for alpha and theta angles

self.alpha = np.radians(alpha)

self.theta = np.radians(theta)

self.a = a

self.d = d

self.Transformation\_matrix = self.matrix\_generation()

# function for generating transformation matrix

def matrix\_generation(self):

matrix = np.array([

[np.cos(self.theta), -np.sin(self.theta) \* np.cos(self.alpha), np.sin(self.alpha) \* np.sin(self.theta), self.a \* np.cos(self.alpha)],

[np.sin(self.theta), np.cos(self.alpha) \* np.cos(self.theta), -np.sin(self.alpha) \* np.cos(self.theta), self.a \* np.sin(self.theta)],

[0.0, np.sin(self.alpha), np.cos(self.alpha), self.d],

[0.0, 0.0, 0.0, 1.0]

])

return matrix

# x = alpha. alpha: x-axis rotation. row 1 is left arm row 2 is right arm

x = [[90.0, 90.0, 0.0, 90.0, 0.0, 0.0, 90.0, 90.0, -90.0, -90.0, 90.0],

[90.0, 90.0, 0.0, 90.0, 0.0, 0.0, 90.0, 90.0, -90.0, -90.0, 90.0]]

# y = theta. theta: z-axis rotation (inital joint positions were used for y). row 1 is left arm row 2 is right arm

y = [[0.0, 90.0, 13.33, -13.33, 15.0, -65.0, -28.0, 90.0, 0.0, -90.0, -17.0],

[0.0, 90.0, 13.33, -13.33, -15.0, 65.0, 28.0, 90.0, 0.0, -90.0, 17.0]]

# a: x axis displacement

a = [0.0, 0.0, 155.0, 75.0, 180.0, 100.0, 0.0, 0.0, 35.0, 55.1, 0.0]

# d: y axis displacement

d = [0.0, 400.0, 116.0, 0.0, -24.0, 24.0, 86.0, 0.0, 150.5, 0.0, 58.65]

for i in range(len(x[0])):

t\_1 = TransformationMatrix(x[0][i],y[0][i],a[i],d[i])

t\_2 = TransformationMatrix(x[1][i],y[1][i],a[i],d[i])

idv\_lh\_trans\_matrices.append(t\_1)

idv\_rh\_trans\_matrices.append(t\_2)

n = range(len(idv\_lh\_trans\_matrices))

# defining a recusive function to multiply all individual matrices for full alignment to 1st component

f = []

n = len(idv\_lh\_trans\_matrices) - 1

def transformation\_lh\_matrix\_gen(idv\_lh\_trans\_matrices,n,f):

if (n == 0):

f.append(idv\_lh\_trans\_matrices[n].Transformation\_matrix)

return idv\_lh\_trans\_matrices[n].Transformation\_matrix

else:

previous\_matrix = transformation\_lh\_matrix\_gen(idv\_lh\_trans\_matrices,n-1,f)

current\_matrix = np.dot(previous\_matrix,idv\_lh\_trans\_matrices[n].Transformation\_matrix)

f.append(current\_matrix)

return current\_matrix

j = transformation\_lh\_matrix\_gen(idv\_lh\_trans\_matrices,n,f)

# Create a new plot

figure = pyplot.figure()

axes = figure.add\_subplot(projection='3d')

axes.add\_collection3d(mplot3d.art3d.Poly3DCollection(meshes.vectors))

# Auto scale to the mesh size

scale = meshes.points.flatten()

axes.auto\_scale\_xyz(scale, scale, scale)

# Show the plot to the screen

pyplot.show()

# Visualization Function

def visualize\_robot(meshes):

fig = plt.figure()

ax = fig.add\_subplot(111, projection='3d')

for stl\_mesh in meshes:

for i in range(len(stl\_mesh.vectors)):

triangle = stl\_mesh.vectors[i]

ax.add\_collection3d(plt.Polygon(triangle, edgecolor='k', alpha=0.3))

scale = stl\_mesh.points.flatten('F')

ax.auto\_scale\_xyz(scale, scale, scale)

ax.set\_xlabel('X')

ax.set\_ylabel('Y')

ax.set\_zlabel('Z')

ax.set\_title('Robot Visualization')

plt.show()

from stl import mesh

import numpy as np

# load an stl files

stl\_mesh\_02 = mesh.Mesh.from\_file("Link02.stl")

stl\_mesh\_03 = mesh.Mesh.from\_file("Link03.stl")

stl\_mesh\_04 = mesh.Mesh.from\_file("Link04.stl")

# stl mesh of vertices

vertices = stl\_mesh.vectors

# function for calculating quaternion

def quat\_calc(mat\_3X3):

# calculating the real part of the quaternion

qw = (1 + mat\_3X3[0,0] + mat\_3X3[1,1] + mat\_3X3[2,2]) \*\* 0.5

print("qw is: ",qw)

# caclulating the x/i imaginary part of the quaternion

qx = (mat\_3X3[2,1] - mat\_3X3[1,2]) / (4 \* qw)

# calculating the y/j imaginary part of the quaternion

qy = (mat\_3X3[0,2] - mat\_3X3[2,0]) / (4 \* qw)

# calculating the z/k imaginary part of the quaternion

qz = (mat\_3X3[1,0] - mat\_3X3[0,1]) / (4 \* qw)

# returning the quaternion

return [qw,qx,qy,qz]