FINAL PROJECT - CS723
Robert Slick
12/9/2020

Overview

For our solution to the problem of 3D point-set alignment, we provide an implementation of the Iterative Closest Point (ICP) algorithm for 3D point alignment. Our program, when run, will take in two helix-axis protein database (PDB) files, convert the residue coordinates to 3-D points and calculate the transformation matrix to transform one PCB into another. Our program works on any PCD but results are strongest when the two PCD's are transformations (similar geometry) of one another or are helix axis. To accommodate point set's of varying sizes, we have provided a filtering mechanism to pass through the denser of the two point clouds and remove points so that they are of equal size. These points are not removed from the resulting transformed PDB's, only the array of 3-D points used to generate a transform. Our program utilizes Biopython module in python3 to programmatically alter the coordinates associated with each residue in the original PDB. This parameters provided to the ICP were chosen from testing and remain optimized. As the results show below, the registration of helices and helix-axis are both strong and optimization could stand to render even tighter alignments. The program could be extended for full automated alignment of helices if a solution to compute helix axis' was provided to feed the axis' into the algorithm.

NOTE: Please visit: `https://colab.research.google.com/drive/1IYXW43aQeekV1W7goZX8X` `usp=sharing` where a helix-axis transformation along with the required modules are included for ease-of-viewing along with further instructions and code explanations.

Implementation Details

Below are the three methods used to calculate the best transformation matrix for two sets of 3-D points. This algorithm is agnostic to PCD's and takes two numpy arrays of 3-D points, A and B, to use in the transformation. A catching mechanism was put in for 3-D points of unequal lengths as can be seen on line 79. We filter the 3-D points to be of equal length prior to using icp. The Iterative Closest Point method finds best-fit transform that maps points A on to points B. The final homogeneous transformation that maps A on to B is found using Euclidean Distances for localization of points in conjunction with the k- nearest neighbors method. The knn method finds the nearest euclidean neighbor in dst for each point in src.

As can be seen in the Driver method beginning on line 264, the two helix-axis PDB's are passed to the method PDBToNpy which convert the 3-D coordinates associated with each residue. These numpy arrays are used as the agnostic 3-D point arrays which icp requires and it is on these that we filter out points and account for mismatches in size. The two numpy arrays are then passed into the method TransformAllPoints which facilitates the prepossessing of the numpy arrays and

the hand off to the icp method. What is returned into t3 on line 268 is a 4-D matrix consisting of the transformation and rotation matrices. We use this transformation matrix to transform the points using numpy in the CreatePDB method. This method also composes a BioPython structure with the new coordinate transformations and outputs a transformed PDB file that can be loaded into chimera or any desired visualization software. After performing the transformation matrix and transforming the helix axis', the program allows for the optional input of the two helix files corresponding to the two helix axis'. Using the transformation matrix applied in the prior steps, it will transform the points of the helix file and output a transformed helix in the same fashion as is done with the axis.

```python
1  import numpy as np
2  from sklearn.neighbors import NearestNeighbors
3  from numpy import load
4  from random import randint
5  from Bio import PDB
6  from numpy import asarray
7  from Bio.PDB import PDBParser
8  from Bio.PDB.PDBIO import PDBIO
9  from Bio.PDB.StructureBuilder import StructureBuilder
10 import numpy as np
11
12
13 class pdbTransform():
14     def __init__(self):
15         self.deletedIndex = []
16     # Accepts pdb, returns numpy array
17     def PDBToNPY(self, fpathin):
18         parser = PDB.PDBParser()
19         io = PDB.PDBIO()
20         struct = parser.get_structure('1ABZ',fpathin)
21         allcoords1 = []
22         for model in struct:
23             for chain in model:
24                 for residue in chain:
25                     for atom in residue:
26                         x,y,z = atom.get_coord()
27                         cSet = []
28                         cSet.append(x)
29                         cSet.append(y)
30                         cSet.append(z)
31                         allcoords1.append(cSet)
32         return allcoords1
33
34     def best_fit_transform(self, A, B):
35         '''
36         Calculates the least-squares best-fit transform that maps
    corresponding points A to B in m spatial dimensions
37         Input:
```

```
38          A: Nxm numpy array of corresponding points
39          B: Nxm numpy array of corresponding points
40          Returns:
41          T: (m+1)x(m+1) homogeneous transformation matrix that maps A on
      to B
42          R: mxm rotation matrix
43          t: mx1 translation vector
44          '''
45
46          assert A.shape == B.shape
47
48          # get number of dimensions
49          m = A.shape[1]
50
51          # translate points to their centroids
52          centroid_A = np.mean(A, axis=0)
53          centroid_B = np.mean(B, axis=0)
54          AA = A - centroid_A
55          BB = B - centroid_B
56
57          # rotation matrix
58          H = np.dot(AA.T, BB)
59          U, S, Vt = np.linalg.svd(H)
60          R = np.dot(Vt.T, U.T)
61
62          # special reflection case
63          if np.linalg.det(R) < 0:
64              Vt[m-1,:] *= -1
65              R = np.dot(Vt.T, U.T)
66
67          # translation
68          t = centroid_B.T - np.dot(R,centroid_A.T)
69
70          # homogeneous transformation
71          T = np.identity(m+1)
72          T[:m, :m] = R
73          T[:m, m] = t
74
75          return T, R, t
76
77
78      def nearest_neighbor(self, src, dst):
79          '''
80          Find the nearest (Euclidean) neighbor in dst for each point in
      src
81          Input:
82              src: Nxm array of points
```

```python
83              dst: Nxm array of points
84          Output:
85              distances: Euclidean distances of the nearest neighbor
86              indices: dst indices of the nearest neighbor
87          '''
88
89          assert src.shape == dst.shape
90
91          neigh = NearestNeighbors(n_neighbors=1)
92          neigh.fit(dst)
93          distances, indices = neigh.kneighbors(src, return_distance=True
    )
94          return distances.ravel(), indices.ravel()
95
96
97      def icp(self, A, B, init_pose=None, max_iterations=20, tolerance
    =0.001):
98          '''
99          The Iterative Closest Point method: finds best-fit transform
    that maps points A on to points B
100         Input:
101             A: Nxm numpy array of source mD points
102             B: Nxm numpy array of destination mD point
103             init_pose: (m+1)x(m+1) homogeneous transformation
104             max_iterations: exit algorithm after max_iterations
105             tolerance: convergence criteria
106         Output:
107             T: final homogeneous transformation that maps A on to B
108             distances: Euclidean distances (errors) of the nearest
    neighbor
109             i: number of iterations to converge
110         '''
111
112         assert A.shape == B.shape
113
114         # get number of dimensions
115         m = A.shape[1]
116
117         # make points homogeneous, copy them to maintain the originals
118         src = np.ones((m+1,A.shape[0]))
119         dst = np.ones((m+1,B.shape[0]))
120         src[:m,:] = np.copy(A.T)
121         dst[:m,:] = np.copy(B.T)
122
123         # apply the initial pose estimation
124         if init_pose is not None:
125             src = np.dot(init_pose, src)
```

```
126
127          prev_error = 0
128
129          for i in range(max_iterations):
130              # find the nearest neighbors between the current source and
     destination points
131              distances, indices = self.nearest_neighbor(src[:m,:].T, dst
    [:m,:].T)
132
133              # compute the transformation between the current source and
     nearest destination points
134              T,_,_ = self.best_fit_transform(src[:m,:].T, dst[:m,indices
    ].T)
135
136              # update the current source
137              src = np.dot(T, src)
138
139              # check error
140              mean_error = np.mean(distances)
141              if np.abs(prev_error - mean_error) < tolerance:
142                  break
143              prev_error = mean_error
144
145          # calculate final transformation
146          T,_,_ = self.best_fit_transform(A, src[:m,:].T)
147
148          #return T, distances, i
149          return T
150     # Accepts two numpy arrrays
151     def TransformAllPoints(self, coord1,coord2):
152          #global deletedIndex
153          # Ensure they are the same size
154          #coord1 = list(load(fname1))
155          #coord2 = list(load(fname2))
156          coord1Copy =[] #coord1
157          coord2Copy =[]# coord2
158          for c in coord1:
159              coord1Copy.append(c)
160          for c in coord2:
161              coord2Copy.append(c)
162          #print("Point total of pdb1: " +str(len(coord1)))
163          #print("Point total of pdb2: " +str(len(coord2)))
164          if len(coord1Copy) != len(coord2):
165              print("\nResolving PDB size difference...")
166              if len(coord1) > len(coord2):
167                  diff = len(coord1) - len(coord2)
168                  # assume shape is same, density is diff
```

```python
169                     for i in range(diff):
170                         randInd = randint(0,len(coord1))
171                         del coord1[randInd]
172                         self.deletedIndex.append(randInd)
173                 else:
174                     diff = len(coord2) - len(coord1)
175                     # assume shape is same, density is diff
176                     for i in range(diff):
177                         randInd = randint(0,len(coord2))
178                         del coord2[randInd]
179                         self.deletedIndex.append(randInd)
180
181             #print("Point total of pdb1: " +str(len(coord1Copy)))
182             #print("Point total of pdb2: " +str(len(coord2Copy)))
183
184         # (A,B)
185         print("\nCalculating transformation matrix...")
186         T = self.icp(np.array(coord1),np.array(coord2))
187
188         coord1Adjust = []
189         for coord in coord1Copy:
190             tempList = coord
191             tempList.append(1)
192             coord1Adjust.append(np.array(tempList))
193         coord2Adjust = []
194         for coord in coord2Copy:
195             tempList = coord
196             tempList.append(1)
197             coord2Adjust.append(np.array(tempList))
198
199         # Print side by side
200
201         #print("Point total of pdb1: " +str(len(coord1Adjust)))
202         #print("Point total of pdb2: " +str(len(coord2Adjust)))
203
204         #np.dot(T,coord1[0])
205
206         print("Transformation matrix:\n"+str(T))
207         #print("Length of coord1 "+str(len(coord1Adjust)))
208         #print("Length of coord1 "+str(len(coord2)))
209         #print("Length of coord 1 adjust: "+str(len(coord1Adjust)))
210         #print("Length of coord 2 adjust: "+str(len(coord2Adjust)))
211         coord1 = np.array(coord1Adjust)
212         coord1T = []
213         #print(coord1Copy)
214         print(coord1T)
215         for coord in coord1Copy:
```

```python
216                #print(coord)
217                coord1T.append(np.dot(T,coord))
218
219          coord1T.reverse()
220          with open('ATrans_1.txt', 'w') as f:
221              for item in coord1T:
222                  f.write("%s" % str(item.tolist()[0])+","+str(item.
    tolist()[1])+","+str(item.tolist()[2])+","+str(item.tolist()[3])+"\n
    ")
223          B = np.array(coord2).tolist()
224          with open('B_1.txt', 'w') as f:
225              for item in B:
226                  f.write("%s\n" % str(item))
227          return (coord1T,T)
228      def CreatePDB(self,coordArray,fPath, ofile):
229          sloppyparser = PDBParser(PERMISSIVE = True, QUIET = True)
230          structure = sloppyparser.get_structure("MD_system", fPath)
231          print("\nGenerating PDB file...")
232          sb = StructureBuilder()
233          sb.set_header(structure.header)
234          # Iterate through models
235          for i in range(len(list(structure.get_models()))):
236              # Iterate through chains
237              models = list(structure.get_models())
238              counter = 0
239              for j in range(len(list(models[i].get_chains()))):
240                  chains = list(models[i].get_chains())
241                  #Iterate thgouth residues
242                  for k in range(len(list(chains[j].get_residues()))):
243                      #Iterate through
244                      residues = list(chains[j].get_residues())
245                      for l in range(len(list(residues[k].get_atoms()))):
246                          #Set coord for each
247                          for atom in structure[i][chains[j].id][residues
    [k].id].get_atoms():
248                              structure[i][chains[j].id][residues[k].id][
    atom.id].set_coord(np.array((float(coordArray[counter][0]),float(
    coordArray[counter][1]),float(coordArray[counter][2]))))
249                              #print(structure[i][chains[j].id][residues[
    k].id][atom.id].get_vector())
250                              counter += 1
251          io=PDBIO()
252          io.set_structure(structure)
253          io.save(ofile)
254          print("Transform file written to: "+ ofile)
255      def TransformHelices(self, fpath1, T):
256          c1 = self.PDBToNPY(fpath1)
```

```python
257
258          coord1T = []
259          for coord in c1:
260              coord.append(1)
261              coord1T.append(np.dot(T,coord))
262          coord1T.reverse()
263          self.CreatePDB(coord1T,"helix1.pdb","helix1Transformed.pdb")
264      def TransformFullHelices(self, fpath1, T):
265          c1 = self.PDBToNPY(fpath1)
266
267          coord1T = []
268          for coord in c1:
269              coord.append(1)
270              coord1T.append(np.dot(T,coord))
271          coord1T.reverse()
272          self.CreatePDB(coord1T,fpath1,"fullHelix1Transformed.pdb")
273          print("Full helix transformed and written to:
     fullHelix1Transformed.pdb")
274
275      def Driver(self,fpath1="helix1-axis.pdb",fpath2="helix2-axis.pdb"):
276          c1 = self.PDBToNPY(fpath1)
277          #print(c1)
278          c2 = self.PDBToNPY(fpath2)
279          t3 = self.TransformAllPoints(c1,c2)
280          self.CreatePDB(t3[0],fpath1,"helix1AxisTransform.pdb")
281          return t3
282          '''
283          while True:
284              print("Enter 1 use default helices, enter 2 to enter custom
     .")
285              choice = input('')
286              if choice == "1":
287                  self.TransformHelices("helix1.pdb",t3[1])
288              #elif choice == "2":
289          '''
290  def main():
291
292      print("**********Helix axis Transformation**********")
293      choice = input("ENTER 1 to use default test cases helix1-axis.pdb \
     nand helix2-axis.pdb.\nENTER 2 to input custom pdb files.\nOption: "
     )
294      transformObj = pdbTransform()
295      if choice == "1":
296          transform = transformObj.Driver()
297          print("**********Full helix Transformation**********")
298          helix1Path = input("Enter the helix pdb filepath associated
     with helix-axis1: ")
```

```
299            transformObj.TransformFullHelices(helix1Path,transform[1])
300       elif choice == "2":
301           try:
302               fpath1 = input("Enter fpath of helix-axis1: ")
303               fpath2 = input("Enter fpath of helix-axis2: ")
304               transform = transformObj.Driver(fpath1,fpath2)
305               print("**********Full helix Transformation**********")
306               helix1Path = input("Enter the helix pdb filepath
      assosciated with helix-axis1: ")
307               transformObj.TransformFullHelices(helix1Path,transform[1])
308           except:
309               print("File not found!")
310  if __name__ == "__main__":
311      main()
```

Listing 1: Python sample code loaded from file

Sample outputs

The following is a visualization sequence of the algorithm. The images are captured using UCSF's Chimera for protein visualization. We use helix1-axis.pdb and helix2-axis.pdb with the intent to transform helix1-axis onto helix2-axis. Figure 1 shows the two helices in Chimera loaded with their initial residues coordinates. Because of their relative distance, the rendered visualization is difficult to interpret and thus we drew bounding elipses around each helix-axis. The subsequent figures show multiple views of the scene in which the transformed helix1-axis pdb was opened along with the helix2-axis. As is demonstrated below, the icp algorithm proves to be an effective method for the alignment of two helices. For reference, helix2-axis.pdb is shown in red and helix1-axis.pdb(transformed) is show in green.
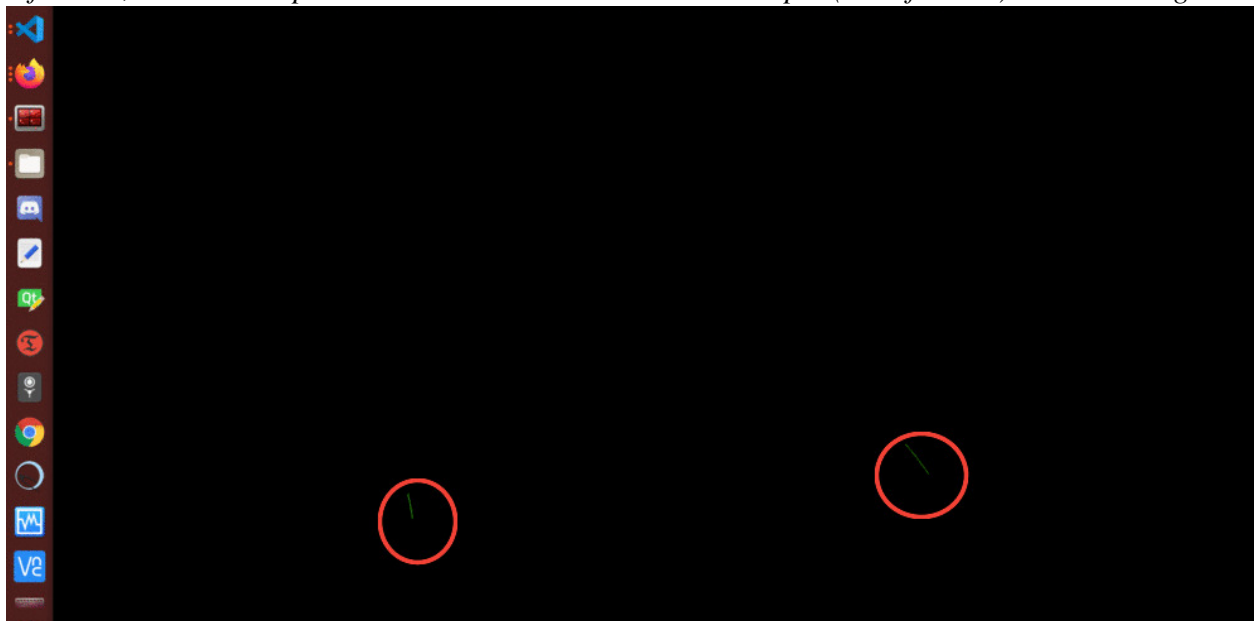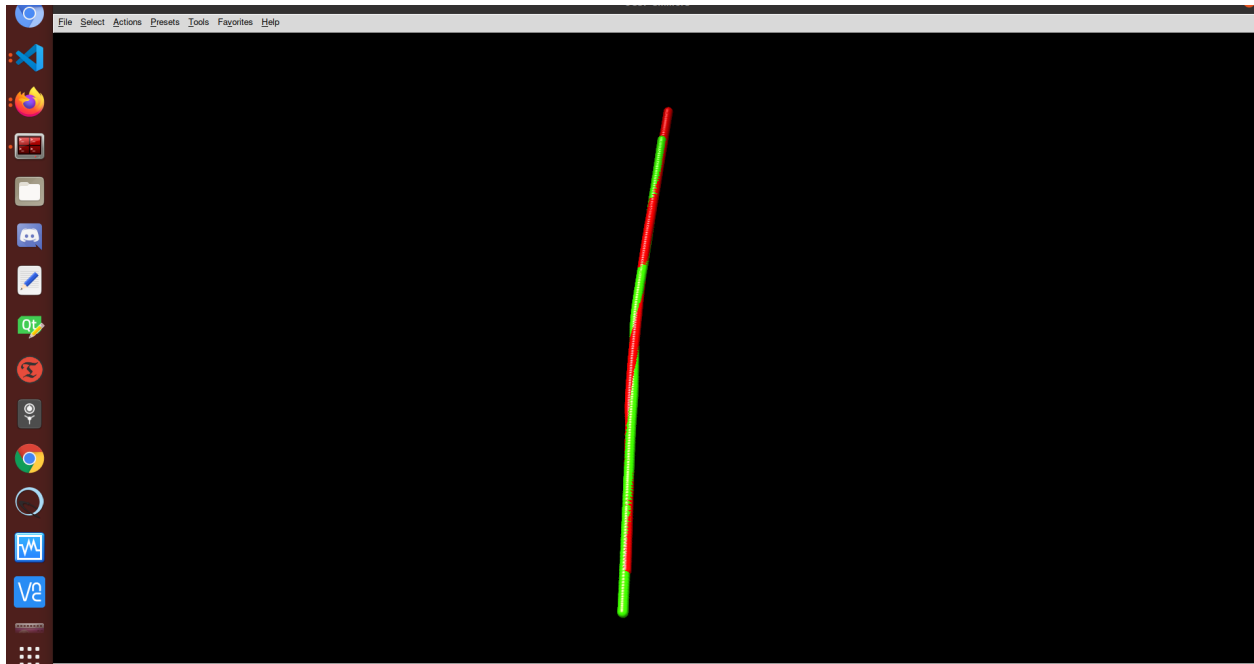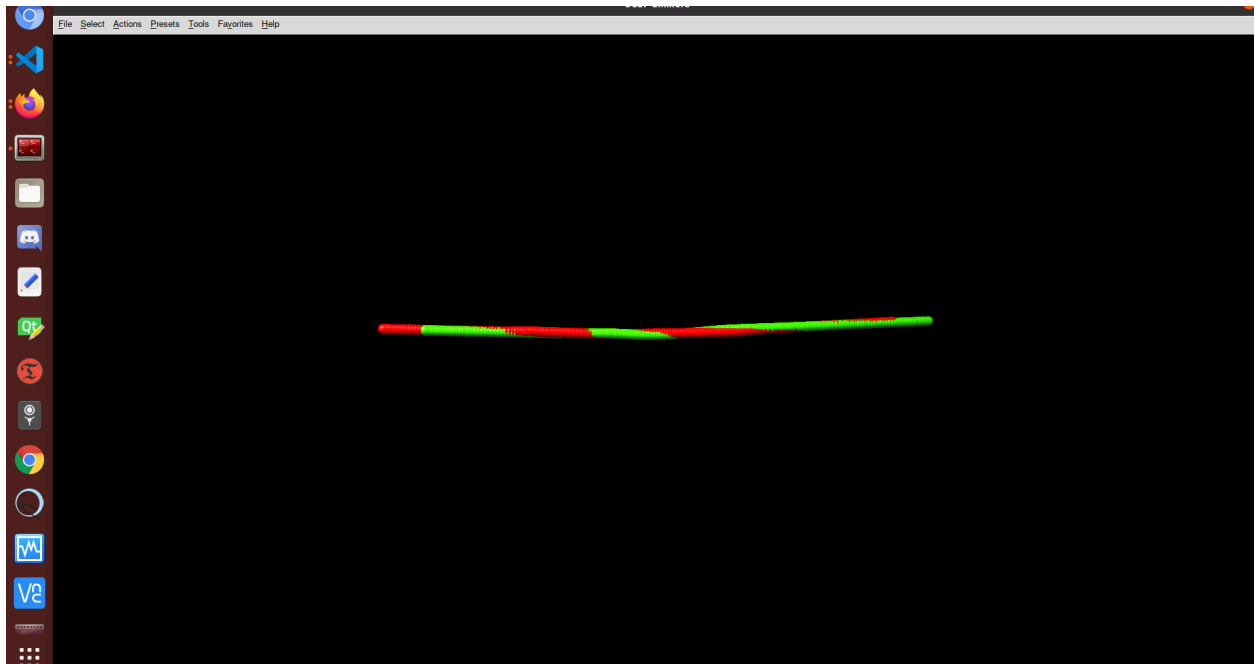
Figure 1: Pre Transform.

Figure 2: Top View.

Figure 3: Side View.

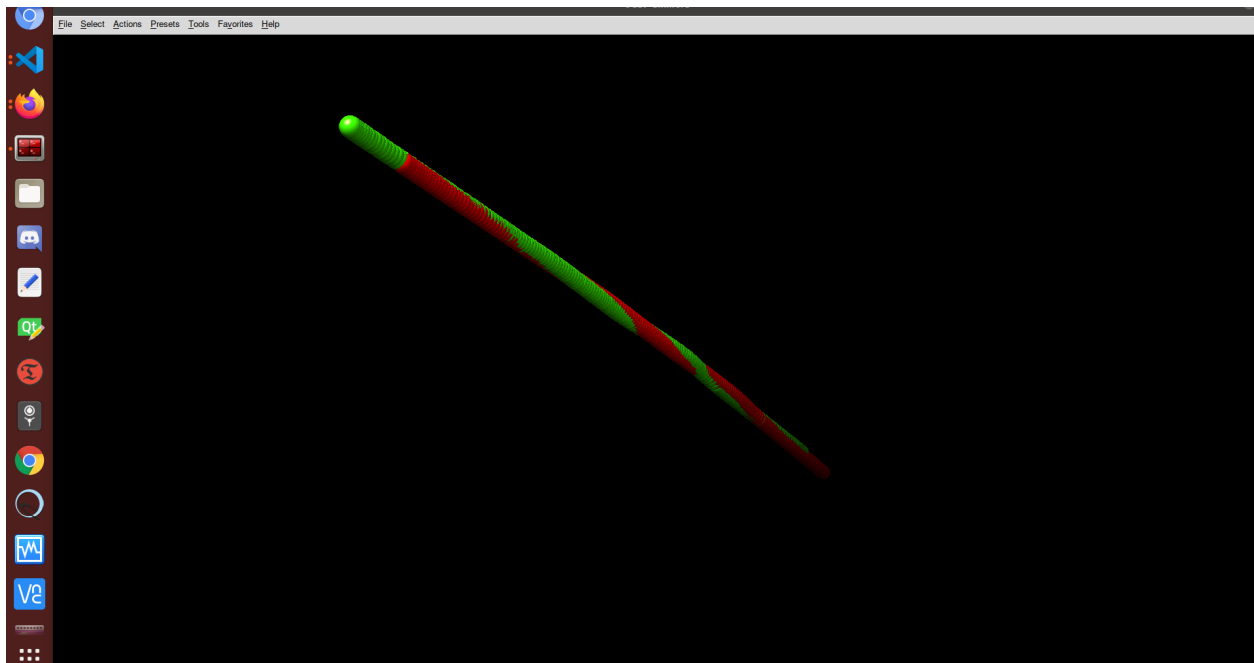**Figure 4:** Head-On View.

Figure 5: Diagonal View.

Running instructions for helix axis transformation. (Default files.)

Please ensure the proper modules are installed, for full list and interactive demo please see: `https://colab.research.google.com/drive/1IYXW43aQeekV1W7goZX8XrZkBU4yzGAI?usp=sharing` *where dependencies are*

listed in the top cell.

1) Run 'python3 ICPTransformation.py'

2) ENTER 1 to use default test cases helix1-axis.pdb and helix2-axis.pdb, included already in the folder.

3) View the transformation matrix in the terminal and helix1AxisTransform.pdb

4) If desired, open Chimera and load helix2-axis.pdb. Then open helix1AxisTransform.pdb for visual verification.

5) In the terminal enter the helix pdb filepath associated with helix-axis1. In this case, enter 'helix1.pdb'. This will apply the transformation matrix to each residue coordinate in helix1.pdb as it did to helix1-axis.pdb in step 2.

6) Enter 'helix1.pdb' to apply the transform to each residue coordinate in helix1.pdb.

7) The transformed helix is in fullHelix1Transformed.pdb. Open in chimera along with helix2.pdb for viewing.

Running instructions for helix axis transformation. (Custom files.)

Please ensure the proper modules are installed, for full list and interactive demo please see: `https://colab.research.google.com/drive/1IYXW43aQeekV1W7goZX8XrZkBU4yzGAI?usp=sharing` *where dependencies are listed in the top cell.*

1) Run 'python3 ICPTransformation.py'

2) ENTER '2' and input the first and second helix axis files. The first will be transformed into the second.

3) View the transformation matrix in the terminal and helix1AxisTransform.pdb

4) If desired, open Chimera and load second helix axis file entered in step 2. Then open helix1AxisTransform.pdb for visual verification of the transformation.

5) In the terminal enter the helix pdb filepath associated with helix-axis1. In this case, enter 'helix1.pdb'. This will apply the transformation matrix to each residue coordinate in helix1.pdb as it did to helix1-axis.pdb in step 2.

6) Enter the file path of the helix associated with helix axis 1 from step 2 to apply the transform to each residue coordinate in the full helix file.

7) The transformed helix is in fullHelix1Transformed.pdb. Open in chimera along with helix2.pdb for viewing.

NOTE: The two files fullHelix1Transformed.pdb and helix1AxisTransform.pdb are overwritten each run.

TASK 1 Questions

Given two vectors in 3-dimensional space, give the mathematical formula to align the two vectors.

To find the rotation matrix R that will rotate vector A onto vector B, perform the following calculations:
1) Let v = A X B
2) Let s = $\|\|v\|\|sin(ab\_ang)$
3) $Let c = a * b * cos(ab\_ang)$
4) $Rotation matrix is given as: R = I + [v]_x + [v]_x^2(1-c)s^2$
$where v\_x is the skew-symmetric cross product of v.$

Discuss the main idea of two approaches to align two objects, each having multiple points. As an example, each protein often has more than 100 amino acids. Each object in a 3D image often has more than 1000 points.

The problem of aligning two or more point clouds is an exemplification of aligning two points with multiple objects. A point cloud is represented as a set of 3-Dimensional points and is often captured by devices such as LiDar. This task is commonly known as point cloud registration. One way to calculate transormations between two sets of points is as follows:
1) First, compute the center of gravity of each cloud.
2) Translate each point with a translation vector given by the center of gravity of its cloud (center each cloud on the origin).
3) Calculate rotation 4) Compute rotation by calculating the singular decomposition of the point cloud.
4) Apply transformation matrix to each point.
NOTE: This assumes matching has already taken place.

A second application of 3d point-set alignment is image homography. Two images of a scene are related by a homography when two images are that of a same plane or two images were acquired by rotating the camera about its optical axis. In such a case the object in question is a match and determining the transformation matrix is what is needed for alignment. To find a homography one can use a system of linear equations or else use the opencv function findHomography if 4 points are known.

Discuss existing programs that are available for alignment of two sets of 3D points. Indicate the software available and examples of how to use the program to perform alignment of two sets of points.

Two open source tools that come to mind for alignment of 3-d point sets are openCV and Teaserplusplus. Teaserplusplus is a C++ implementation of iterative closeset point and performs alignment on already matched point clouds. It is an emerging open source tool considered close to SoA. Examples can be found here: `https://github.com/MIT-SPARK/TEASER-plusplus`

that show its capabilities to work with 3D depth data. Iterative closest point was discussed in the question above. OpenCV provides a rich set of tools for image processing and here is an example of their functionality being employed to solve for a homography alignment as discussed above: `https://docs.opencv.org/master/d1/de0/tutorial_py_ feature_homography.html`.

References

- *GitHub*, `https://github.com/rmslick/HelixAxisRegistration.git`

- *Colab,* `https://colab.research.google.com/drive/ 1IYXW43aQeekV1W7goZX8XrZkBU4yzGAI?usp=sharing`

- *Homograph,* `https://www.learnopencv.com/image-alignment-feature- based-using-opencv-c-python/`

- *Mathy Stuff,* `https://math.stackexchange.com/questions/180418/ calculate-rotation-matrix-to-align-vector-a-to-vector-b-in- 3d`