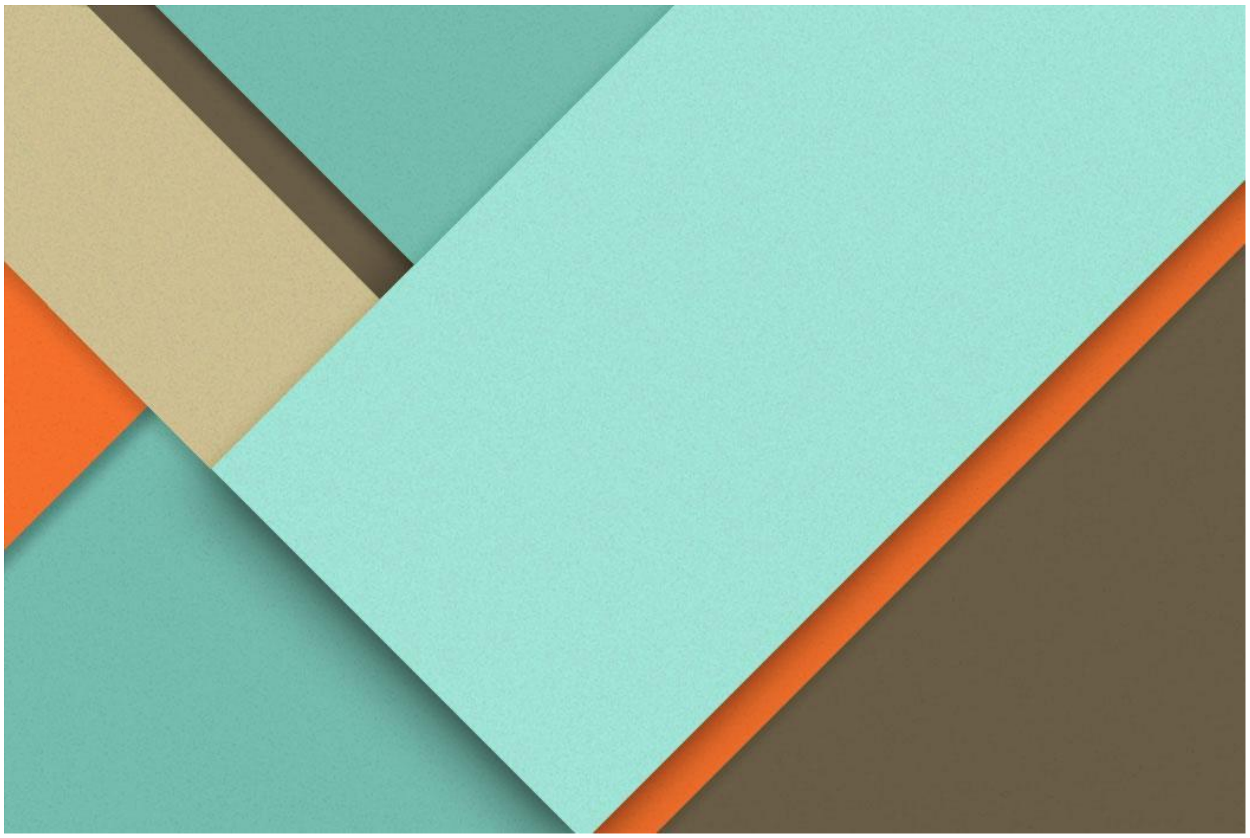


Digital 3D Protein Recognition & Matching

Computer Vision COMP508

Prepared by:

Asma Hakouz 0063315
Dzenaida Gicic 0063537
Dalyah Al-Jamal 0063583

Supervised by:

Prof. Yücel Yemez

Table of Contents

Introduction

Proteins interactions play a very critical role in their functions, which are important for drug design and discovery, as well as the body functions and malfunctions. Recognizing the type and the shape of the protein interacting regions (i.e., protein interfaces) is essential for figuring out if it could contribute in a certain chemical interaction.

In fact, two proteins or molecules can interact with each other if their interfaces have complementary 3D shapes, or in other words if their interfaces align structurally. Figure 1 shows an example of a protein-protein interaction through interfaces.

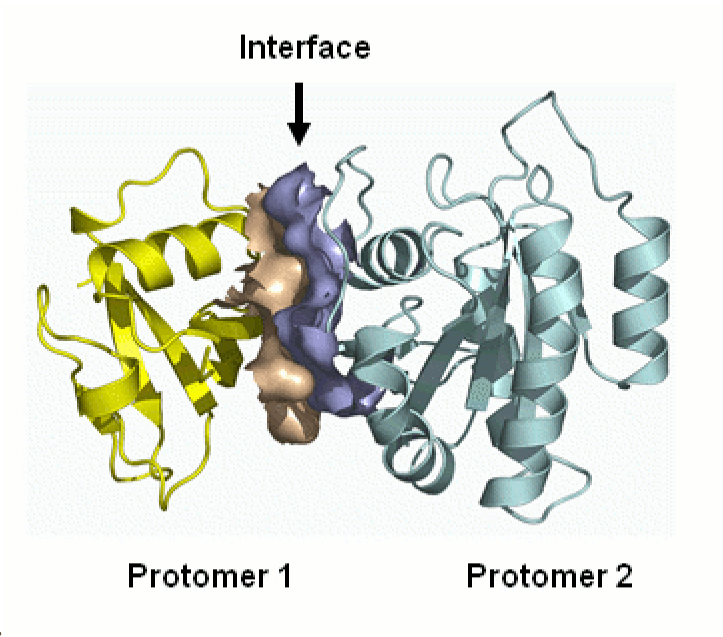


Figure 1- Visualization of Protein-Protein Interaction

Due to the rapid increase in the available proteins' structural data, computational methods were exploited to make use of the available data in creating and predicting and analyzing new structures. For example, from the field of computer vision, model-based recognition algorithms can be used to predict protein-protein interactions, where as shown in the figure 2 [5] below, a data set of experimentally known interface pairs (left and right partners) is processed as the models and a target protein is searched for a potential matching interface. Once a pair of proteins is found, where one contains the left interface and the other the right interface, it means that they can interact forming a complex that will have different functions and effects on the body than the original proteins.

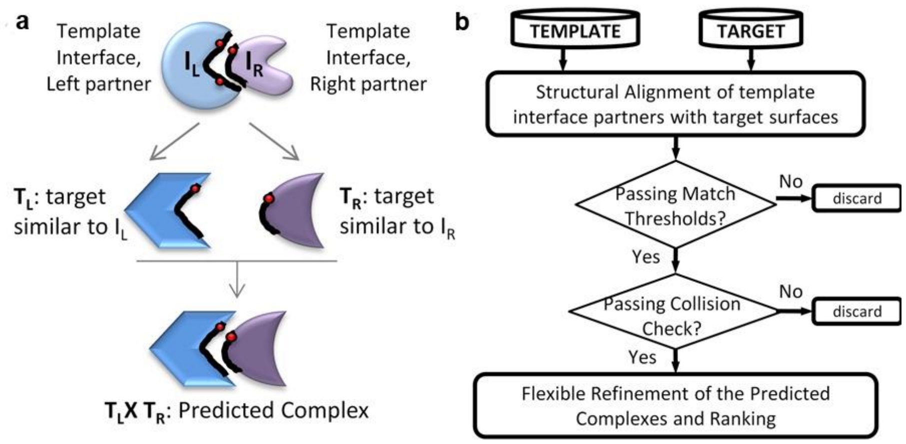

a
Template Interface, Left partner
Template Interface, Right partner
I_L
I_R
T_L: target similar to I_L
T_R: target similar to I_R
T_LX T_R: Predicted Complex
b
TEMPLATE
TARGET
Structural Alignment of template interface partners with target surfaces
Passing Match Thresholds?
No
discard
Yes
Passing Collision Check?
No
discard
Yes
Flexible Refinement of the Predicted Complexes and Ranking

Figure 2 -Identification of protein- protein interaction

There are several model-based recognition algorithms which can be used; however geometric hashing is one of the widely used ones because of its robustness, scale, rotation and translation invariance, in addition to it being parallelizable.

Geometric hashing technique was originally developed in the field of computer vision for object recognition in two dimensions 2D, then it was extended to handle higher dimensions.

In our project, firstly we implement the 2D version of the algorithm, then expanded it to 3D version where it could recognize interfaces in a protein.

The project mainly consisted of four main parts:

- Background reading and analysis.
- Data gathering and features extraction, focusing only on the features needed for matching interfaces structurally.
- The second part is implementing 2D Geometric hashing on 2D shapes.

- Finally, the main part of 3D recognition using geometric hashing is implemented and tested on some samples of the features extracted in the first step.

Importance of the project

This project aims to produce a program that demonstrates identifying the possible protein interaction usually done by biomedical researchers. In fact, this identification is useful also in drug design, diagnosing diseases, understanding protein functions and the overall body functioning.

Scope of the project

The main goal of this project was to apply geometric hashing algorithm for the purpose of protein interfaces recognition for predicting protein-protein interactions. The 2D version of the algorithm was firstly implemented to recognize 2D models before extending it to 3D version. Many computer vision concepts were applied during our implementation such as: feature extraction, 2D and 3D transformation matrices (including translation, rotation and scaling) and geometric hashing. Also, we used clustering as a data compression technique to eliminate the duplicates. More details about how each of the computer vision concepts contributed in this project are presented in later section in this report.

Tools and Languages

Python 2.7 language was used to implement the whole project. Pycharm software was used as an IDE for testing and debugging. In addition, some libraries were necessary to implement geometric features:

- Matplotlib
- Mpl_toolkits.mplot3d
- Random

Computer Vision Concepts Applied in the Project

The list below includes the important vision concepts used in the project, more details about how they were implemented will be presented under the implementation section.

- ❖ 2D & 3D Transformation
- ❖ Pattern Recognition
- ❖ Dimensionality Reduction
- ❖ Clustering
- ❖ Scaling, Rotation and Translation Invariance

2D Geometric Hashing Algorithm

Geometric hashing algorithm has two main parts: preprocessing phase and recognition phase. In preprocessing phase, type of training for the machine is performed. Many defined models should be processed and hashed in the hashing table. While in recognition phase, a certain model is identified by searching for a match in the previous hashed table in the preprocessing phase.

Preprocessing Phase(Training/ Offline)

The pseudo code of the preprocessing phase is shown below.

```
For each unclustered model:
    Extract features (geometric coordinates)
    For each possible bases (2 distinct points):
        Calculate the transformation that places the 1st point at the origin and the
        2nd point on the positive x-axis.
        Apply the transformation to the remaining points to find the new 2D
        coordinates.
        Add an entry for each point's new coordinates in the temporary hash table
Merge the first unclustered sample to the big hash table and call it cluster representative.
Compare the rest of the sample by voting and mark the ones that vote to be the same as
representative as same and discard.
Repeat the process iteratively until all models clustered.
```

Recognition Phase

The pseudo code of the recognition phase is shown below.

```
For each possible pair in the testing scene:
        Calculate the hash table with the respect to the basis.
        Compare this table to the main hash table and perform voting.
Once voting is done based on the defined threshold decide if the model is identified or not.
```

Algorithm properties

Revelvance

Coordinates of the points specify the relevant region of the sample image, which is feature extraction on its own, therefore keeping focus only on the specified points in the 2D or 3D system reduces the time complexity of a search.

Invariance

Geometric Hashing as a method ensures that once a specific model has been saved in the hash table, its all possible pair based translations and scaling have been handled. In fact, no matter in which scale or position in 2D or 3D system the testing sample comes, it will be easily identified by a match model stored previously in the hash table.

Indexing based approach

Search on index of the interest is what decreases computational complexity and saves the memory. Instead of keeping a 2D or 3D array containing all the bins of the space in which the model is hiding, map-oriented search method is much easier and more efficient to perform.

Offline training

Since we have a fixed set of templates/models, in our case experimentally observed protein interfaces, performing the training of the algorithm, or in other words constructing the full hash table of the existing models requires only one-time computation.

Parallelism

Due to the offline training property testing can be performed parallelly and time complexity reduced accordingly.

Non-uniform Distribution

Due to the non-uniformity of the number of points per model, as well as different number of models having points at a specific bin of a hash table, or in other words specific coordinate in a space, space and time complexity reduction exists.

Occlusion Handling

Voting and adaptive threshold based on model related number of significant points methods allow a limited non-overlap of the testing and training models, therefore occlusions are partially accounted for.

Project Phases

I. Literature Review

As the first step, many researches were done to pick the suitable algorithm that can perform 3D object recognition. After we as a team decided to use Geometric Hashing algorithm for detecting protein atoms, “Geometric Hashing: An overview” was taken as reference for the team to understand the 2D case [2]. Discussions and meetings were held at the start to understand the algorithm and discuss how to extend it to 3D.

II. Data Gathering and Feature Extraction

Data (Proteins atoms and coordinates) are gathered from a universal Protein Database (PDB) [3].

In fact, only the centroids (C Alpha) of the amino acids in the protein molecule are the points we are caring about in recognizing protein interfaces. Thus, feature reduction was done to only consider C Alpha atoms in the recognition.

III. 2D Model Recognition

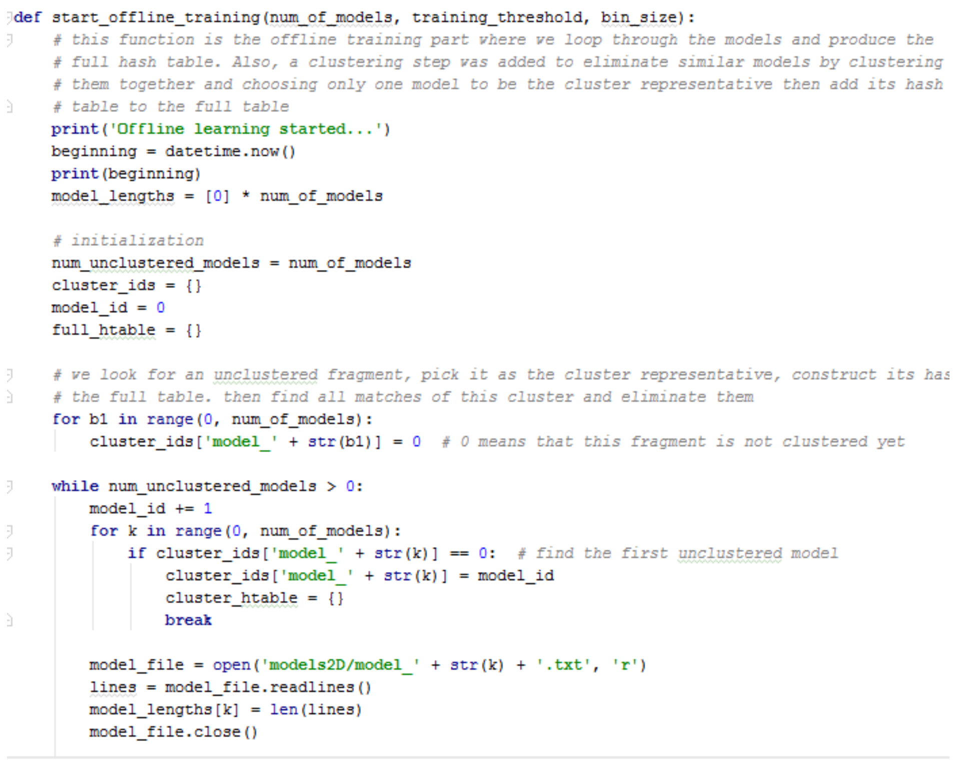
A. Implementation

The overall process consists of two main parts:

1- Training:

Training is the offline phase, since it can be done once and used repeatedly. In other words, training does not require re-calculation whenever a test scene is given. In this phase the models were processed following the geometric hashing technique described above and a full hash table was created to contain all models' entries (basis and model points after the transformation). An additional step was to cluster models so that we only pick a representative model of each cluster to add its entries to the full hash table.

Figure 3 shows part of the offline Training function that train the machine to cluster given models.

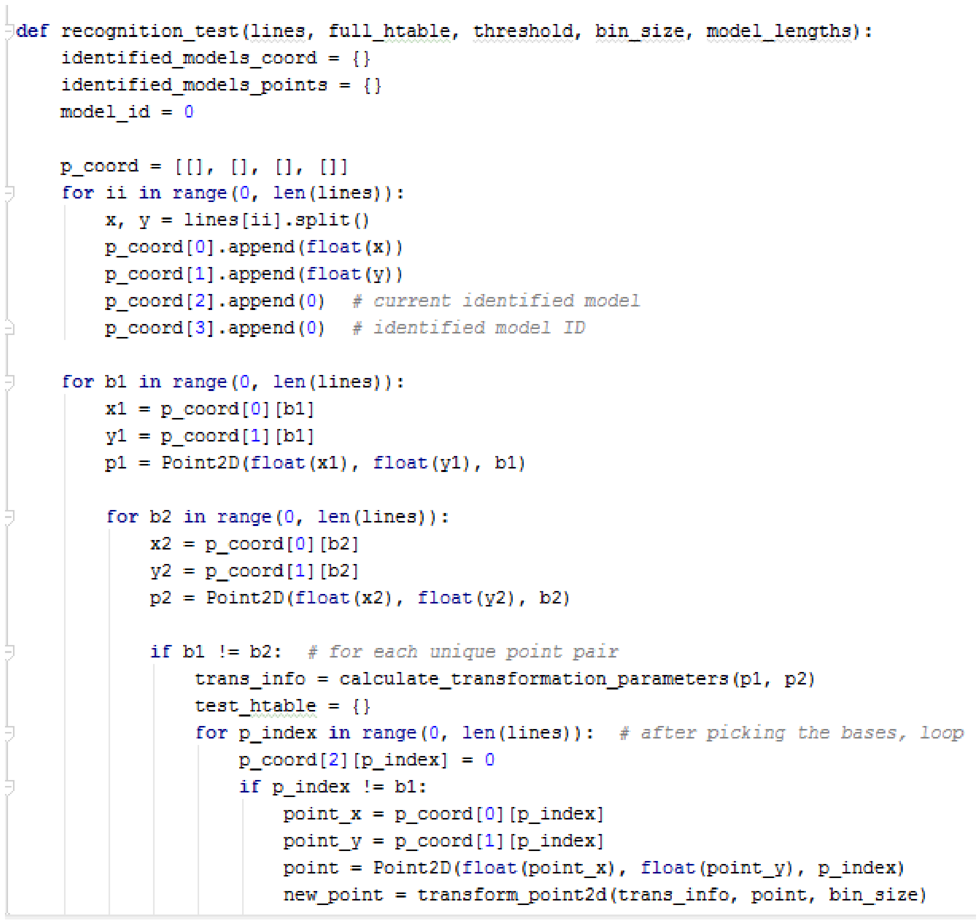


```
def start_offline_training(num_of_models, training_threshold, bin_size):
    # this function is the offline training part where we loop through the models and produce the
    # full hash table. Also, a clustering step was added to eliminate similar models by clustering
    # them together and choosing only one model to be the cluster representative then add its hash
    # table to the full table
    print('Offline learning started...')
    beginning = datetime.now()
    print(beginning)
    model_lengths = [0] * num_of_models

    # initialization
    num_unclustered_models = num_of_models
    cluster_ids = {}
    model_id = 0
    full_htable = {}

    # we look for an unclustered fragment, pick it as the cluster representative, construct its has
    # the full table. then find all matches of this cluster and eliminate them
    for b1 in range(0, num_of_models):
        cluster_ids['model_' + str(b1)] = 0  # 0 means that this fragment is not clustered yet

    while num_unclustered_models > 0:
        model_id += 1
        for k in range(0, num_of_models):
            if cluster_ids['model_' + str(k)] == 0:  # find the first unclustered model
                cluster_ids['model_' + str(k)] = model_id
                cluster_htable = {}
                break

        model_file = open('models2D/model_' + str(k) + '.txt', 'r')
        lines = model_file.readlines()
        model_lengths[k] = len(lines)
        model_file.close()
```

Figure 3 - Offline Training 2D

2-Recognition:

Once we have a test scene we go through possible bases pair combinations and look for a matching model based on voting technique and index-based search using the entries in the full hash table. Figure 4 shows part of 2D recognition test function used to identify new test scene model.



```
def recognition_test(lines, full_htable, threshold, bin_size, model_lengths):
    identified_models_coord = {}
    identified_models_points = {}
    model_id = 0

    p_coord = [[], [], [], []]
    for ii in range(0, len(lines)):
        x, y = lines[ii].split()
        p_coord[0].append(float(x))
        p_coord[1].append(float(y))
        p_coord[2].append(0)   # current identified model
        p_coord[3].append(0)   # identified model ID

    for b1 in range(0, len(lines)):
        x1 = p_coord[0][b1]
        y1 = p_coord[1][b1]
        p1 = Point2D(float(x1), float(y1), b1)

        for b2 in range(0, len(lines)):
            x2 = p_coord[0][b2]
            y2 = p_coord[1][b2]
            p2 = Point2D(float(x2), float(y2), b2)

            if b1 != b2:  # for each unique point pair
                trans_info = calculate_transformation_parameters(p1, p2)
                test_htable = {}
                for p_index in range(0, len(lines)):  # after picking the bases, loop
                    p_coord[2][p_index] = 0
                    if p_index != b1:
                        point_x = p_coord[0][p_index]
                        point_y = p_coord[1][p_index]
                        point = Point2D(float(point_x), float(point_y), p_index)
                        new_point = transform_point2d(trans_info, point, bin_size)
```

Figure 4 - 2D Recognition Test

*Complement code implementation is attached with the document.

B. Testing

To test the algorithm, we manually constructed a trivial data set containing seven initial models, which were clustered into 2 main clusters; square (model id # 0) and triangle (model id # 1).

After that, we provided a test scene containing several scaled, rotated and translated versions of the triangle and the square and they were all identified as yellow dots as shown in the figure 5 below.

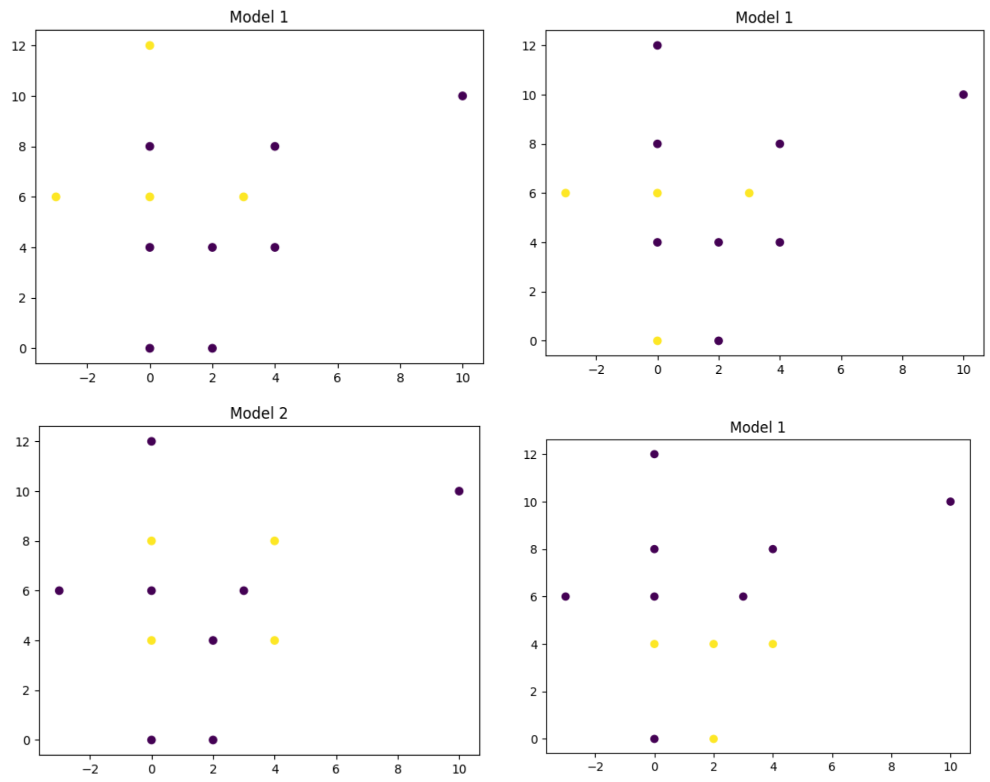


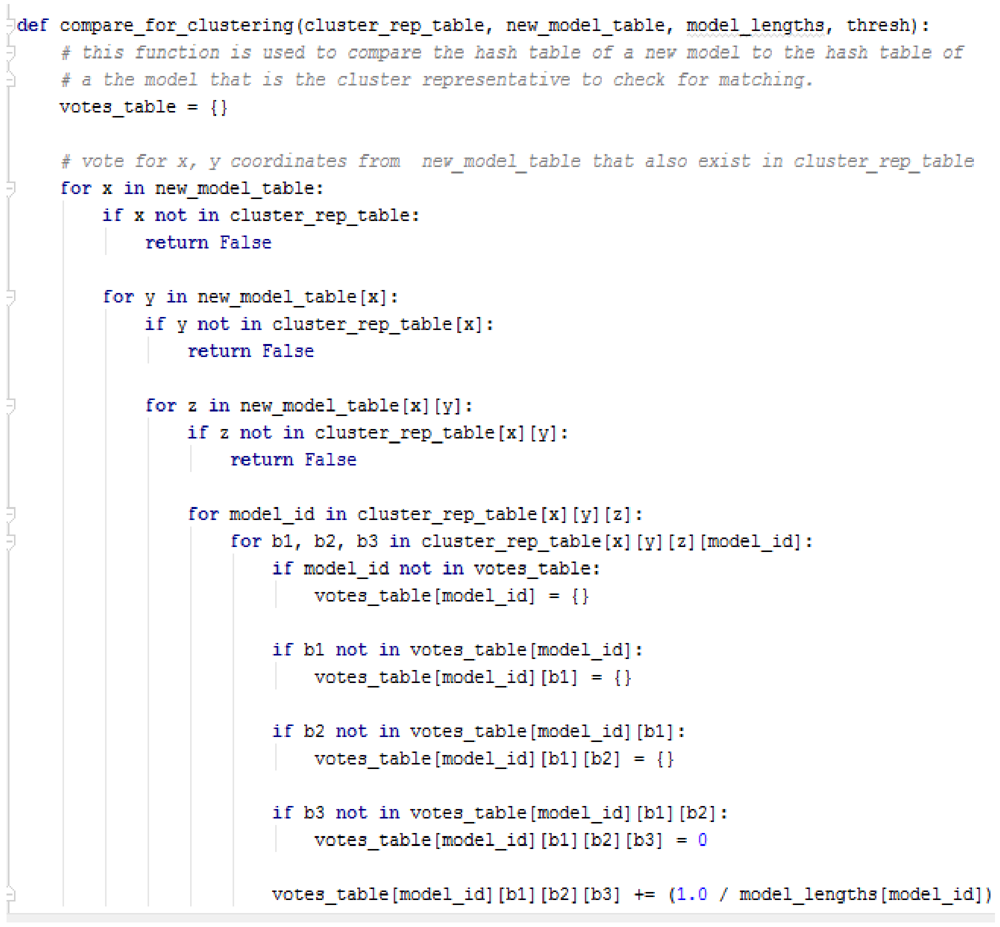
Figure 5 - 2D Test Models

IV. 3D Model Recognition

A. Implementation

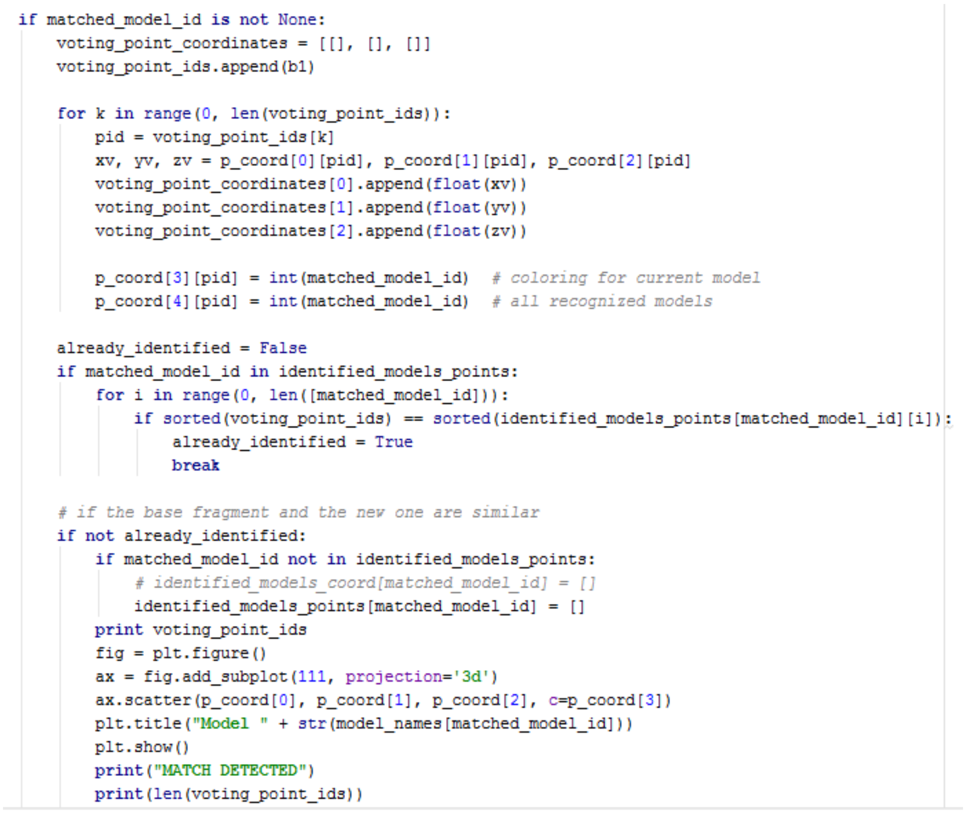
Basically, same methodology used in 2D recognition was used in 3D. However, to extend the algorithm for handling 3D objects, we had to apply some modifications in many stages and parts of the process. For example, instead of picking 2 points in the 2D case as bases for our reference axis, we pick 3 points to form our reference plane; after transformation the first point should be centered at the origin of the new coordinate system, the second point on the positive x-axis while the third point should lie on the xy plane. Then, we can directly define the new z-axis as the direction perpendicular to that plane following the right-hand rule to decide on the positive direction.

Figure 6 shows part of compare clustering function that was used in both 2D and 3D cases while figure 7 shows part of how the voting were performed to identify the model.



```python
def compare_for_clustering(cluster_rep_table, new_model_table, model_lengths, thresh):
    # this function is used to compare the hash table of a new model to the hash table of
    # a the model that is the cluster representative to check for matching.
    votes_table = {}

    # vote for x, y coordinates from  new_model_table that also exist in cluster_rep_table
    for x in new_model_table:
        if x not in cluster_rep_table:
            return False

        for y in new_model_table[x]:
            if y not in cluster_rep_table[x]:
                return False

            for z in new_model_table[x][y]:
                if z not in cluster_rep_table[x][y]:
                    return False

                for model_id in cluster_rep_table[x][y][z]:
                    for b1, b2, b3 in cluster_rep_table[x][y][z][model_id]:
                        if model_id not in votes_table:
                            votes_table[model_id] = {}

                        if b1 not in votes_table[model_id]:
                            votes_table[model_id][b1] = {}

                        if b2 not in votes_table[model_id][b1]:
                            votes_table[model_id][b1][b2] = {}

                        if b3 not in votes_table[model_id][b1][b2]:
                            votes_table[model_id][b1][b2][b3] = 0

                        votes_table[model_id][b1][b2][b3] += (1.0 / model_lengths[model_id])
```

Figure 6- Compare Clusters



```
if matched_model_id is not None:
    voting_point_coordinates = [[], [], []]
    voting_point_ids.append(b1)

    for k in range(0, len(voting_point_ids)):
        pid = voting_point_ids[k]
        xv, yv, zv = p_coord[0][pid], p_coord[1][pid], p_coord[2][pid]
        voting_point_coordinates[0].append(float(xv))
        voting_point_coordinates[1].append(float(yv))
        voting_point_coordinates[2].append(float(zv))

        p_coord[3][pid] = int(matched_model_id)  # coloring for current model
        p_coord[4][pid] = int(matched_model_id)  # all recognized models

    already_identified = False
    if matched_model_id in identified_models_points:
        for i in range(0, len([matched_model_id])):
            if sorted(voting_point_ids) == sorted(identified_models_points[matched_model_id][i]):
                already_identified = True
                break

    # if the base fragment and the new one are similar
    if not already_identified:
        if matched_model_id not in identified_models_points:
            # identified_models_coord[matched_model_id] = []
            identified_models_points[matched_model_id] = []
        print voting_point_ids
        fig = plt.figure()
        ax = fig.add_subplot(111, projection='3d')
        ax.scatter(p_coord[0], p_coord[1], p_coord[2], c=p_coord[3])
        plt.title("Model " + str(model_names[matched_model_id]))
        plt.show()
        print("MATCH DETECTED")
        print(len(voting_point_ids))
```

Figure 7- Voting to recognize the 3D Model

*Complement code implementation is attached with the document.

B. Testing

First, we applied the code for a simple data set containing only one model which resembles to a house shape (i.e., a pyramid on top of a cuboid). The identified model is shown in Figure 8 as yellow points.

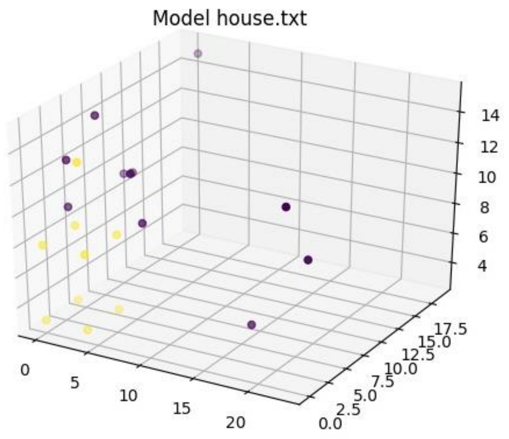
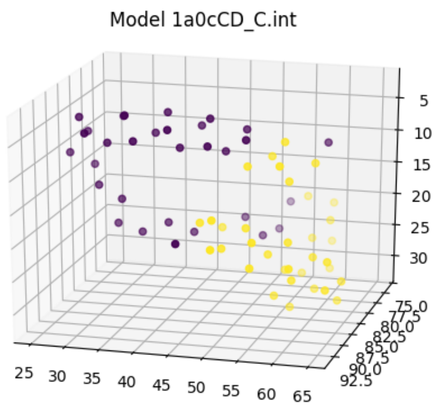


Figure 8- 3D Test Case

Then, we do some protein testing to figure out if the program is able to recognize protein interfaces within the protein. So, to train the machine to recognize certain protein interfaces and store them in the hash table, we used PIFACE website [6] which have the protein interfaces that can contribute in chemical interactions. We trained the program with 10 different interfaces. Then, we used Protein Data Bank [3] to get our test samples and reduced their sizes as the recognition was taking so much time. The program was able to recognize the interfaces in each test case. Figure 9 shows the result from the first test case. Figure 10 shows the corresponding original protein drawn using Chimera software [4].



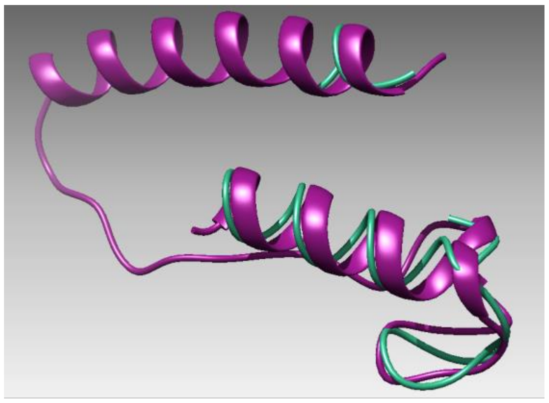


Figure 9- Identified Interfaces

Figure 10 - original Protein

Enhancements & Future Work

There are still some programming details that can be improved so that complexity decreases and the efficiency increases. Firstly, during the clustering phase of the training, instead of repetitive calculation of the hash tables for the unclustered training samples, lazy and smart or dynamic programing could be used, which implementation might require change of programming language we used, nevertheless further research is required. For example, instead of checking the bases pairs one by one in order, random checking could be performed (similar to RANSAC idea), so the program could recognize the interfaces faster. Secondly, in case of using more powerful computational device, parallel programming techniques could be implemented. Lastly, voting could be performed in a local neighborhood of the basis.

Conclusion

The aim we had once we made the project proposal was to apply the geometric hashing technique to recognize 3D objects which we successfully achieved it. Implementation of the 2D Geometric Hashing algorithm and extending it to 3D while it successfully works in both cases is achieved. Together with the implementation of already learned computer vision concepts we have learned and implemented something new. Computer vision concepts open our minds to such good solutions to many problems in different disciplines like: drug design and protein interaction. We are so thankful having the opportunity to study computer vision concepts because of their high demands in solving many problems nowadays.

References

1. Protein Interaction [Digital image]. (n.d.). Retrieved from http://noxclass.bioinf.mpi-inf.mpg.de/help.php 2. Wolfson, H. J., & Rigoutsos, I. (1997).
2. Geometric hashing: An overview. IEEE computational science and engineering, 4(4), 10-21.
3. Bank, R. P. (n.d.). PDB. Retrieved January 10, 2018, from https://www.rcsb.org/pdb/home/home.do
4. UCSF Chimera [Computer software]. (n.d.). Retrieved from https://www.cgl.ucsf.edu/chimera/
5. Tuncbag N, Gursoy A, Nussinov R, Keskin O. Predicting Protein-Protein Interactions on a Proteome Scale by Matching Evolutionary and Structural Similarities at Interfaces Using PRISM, Nature Protocols, 2011, 6(9):1341-54
6. PIFACE. (n.d.). Retrieved January 12, 2018, from http://prism.ccbb.ku.edu.tr/piface/index.php