Surfaces-based Protein Domains Retrieval (SHREC'21 Track4)

INTRODUCTION

This notebook presents a Step-by-Step implementation guidance for our novel 3D shape descriptor or retrieval method, called the APPFD-FK-GMM. We apply this retrieval method for the retrieval task of the SHREC 2021 dataset of Surface-based Protein Domains.

- Further details regarding this retrieval track can be found **here**.
- For full code implementation of this method, please go **here**.

OUR TEAM

1. Dr. Ekpo Otu (eko@aber.ac.uk)
2. Prof. Reyer(rrz@aber.ac.uk)
3. Prof. Yonguai (liuyo@edgehill.ac.uk)
4. Dr. David (dah56@aber.ac.uk)

- Please contact: Dr. Ekpo Otu for any issue or concern regarding this implementation.

In [ ]:

In [1]:
```
# Paths to DATASETS: QUERY and COLLECTION datasets, in the Shape-only category (i.e. .OFF triangular mesh files)
dataset_query = "c:/users/ekpo/desktop/query_data/"
dataset_collection = "c:/users/ekpo/desktop/collection_data/"
```

NOTE:

1. Total number of files (data) in the QUERY set is: **10**
2. Total number of files (data) in the COLLECTION set is: **554**

OUTLINE OF IMPLEMENTATION STEPS

We are going to follow these 4 simple guide towards this implementation.

- **1. Data Pre-processing.**
- **2. Computing Local APPFDs for 3D Surfaces.**
- **3. Computing Final APPFD-FK-GMM (i.e. Fisher Vector (FV)) Descriptor for A Single 3D Model.**
- **4. Matching Different 3D Models With APPFD-FK-FMM**

In [ ]:

1. Data Pre-processing

Given a 3D protein surface or model represented as a triangular mesh, and a fullpath to the model, we first sample $N = 3,500$ points from the surface of the mesh to form a point cloud representation of the 3D protein surface. The following line of code would load-in a 3D mesh from file, and sample N points to form the point cloud representation of the surface. The **Trimesh Python library** is needed to achieve this.

a. Load and Visualize Sample 3D Protein Surface Data (How-To)

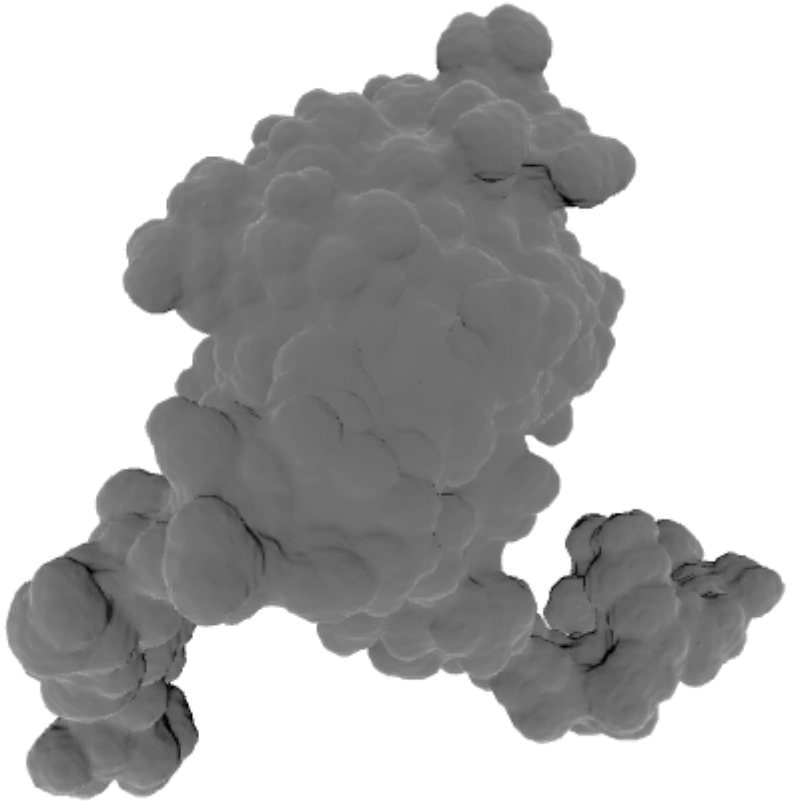
In [2]:

```
# Import necessary libraries, modules, and packages
%matplotlib inline

import matplotlib.pyplot as plt
import numpy as np
import trimesh
import ekpoUtilities as ekpo
# Import APPFD algorithm
import local_appfd_method as appfd

import time

# Full path to sample 3D model - .OFF ASCI format (i.e. 0_shape.off)
fullpath = "c:/gitEkpo/0_shape.off" # https://github.com/KoksiHub/APPFD_FK_GMM-Method-For-SHREC-2021-Surface-based-Protein-Domains

N = 3500

# Load-in the sample data (3D mesh)
mesh = trimesh.load_mesh(fullpath)
```

```
# Visualize mesh
mesh.show()
```

```
C:\Users\Ekpo\.conda\envs\Open3D_PyntcloudPy36\lib\site-packages\IPython\core\display.py:717: UserWarning: Consider using IPython.
display.IFrame instead
  warnings.warn("Consider using IPython.display.IFrame instead")
```

Out[2]:



b. Sample N = 3500 Points From Above Input Mesh (How-To)

In [3]:

```
# Sample N points from mesh to form point cloud
pointcloud = trimesh.sample.sample_surface(mesh, N)[0]
```

```
# We should get [N x 3] points, where N = 3500
print("Shape of sampled Point Cloud:\n ", pointcloud.shape)
print()
print("Point Cloud data:\n", pointcloud)
```

```
Shape of sampled Point Cloud:
  (3500, 3)

Point Cloud data:
 [[-11.01411438  -5.11773765  16.64533082]
 [  3.79560596  -1.86458588  -7.10680585]
 [ -2.51099351  15.39335587  -9.05124645]
 ...
 [ -1.07383515  -7.55779255  -1.7112427 ]
 [  9.57088572  -1.99868165  -2.97503393]
 [ 12.95843465 -10.51429113 -15.21703174]]
```

c. Scale Sampled Point Cloud *s.t.* Its RMS Distace From The Origin To Every Other Points Is 1 (How-To)

In [4]:

```
# Centre PointCloud on its centroid
pointcloud -= np.mean(pointcloud, axis = 0)

# Calculate the 'RMS scale' for the points (using NumPy)
n = len(pointcloud)

#Find the root-mean-square (RMS) value for pointCloud
rms = np.sqrt(np.sum(np.square(pointcloud)) / n)
scale = 1 / rms
pointcloud = np.multiply(scale, pointcloud)

# We should get [N x 3] RMS-scaled points, where N = 3500
print("Shape of RMS-scaled Point Cloud:\n ", pointcloud.shape)
print()
print("RMS-scaled Point Cloud data:\n", pointcloud)
```

```
Shape of RMS-scaled Point Cloud:
  (3500, 3)

RMS-scaled Point Cloud data:
 [[-0.67098692 -0.17271675  0.91713463]
 [ 0.1759955   0.01333419 -0.44127334]
 [-0.18468511  1.00033279 -0.55247781]
 ...
```

```
[-0.10249261 -0.31226588 -0.1326958 ]
[ 0.50628941  0.00566512 -0.20497326]
[ 0.70002665 -0.48135093 -0.90510511]]
```

d. Estimate Normal Vector To All Points In Point Cloud (How-To)

Given the RMS-scaled point cloud from ***Step-c*** as input, the function **normalsEstimationFromPointCloud_PCA(args)** would compute corresponding normal vector to every point on the surface of the input point cloud, using a simple eigen-decomposition technique.

> This fucntion is included in the **ekpoUtilities.py** script and accepts ONLY one parameter - $[N \times 3]$ array of point cloud, as input. This function then returns the input $[N \times 3]$ point cloud (P_s) and its corresponding $[N \times 3]$ normals, (N_s).

To estimate corresponding surface normals, N_s to a given RMS-scaled point cloud, P_s, the following two lines of code would do:

In [5]:
```
# import ekpoUtilities as ekpo
Ps, Ns = ekpo.normalsEstimationFromPointCloud_PCA(pointcloud)

print("Estimated Surface Normals:\n", Ns)
```

```
Estimated Surface Normals:
 [[ 0.39049315 -0.87027701 -0.30022163]
 [ 0.02764604  0.04041365  0.9988005 ]
 [-0.87835045  0.44767719  0.16758765]
 ...
 [ 0.48700884  0.19946234 -0.85031592]
 [-0.5042636   0.85694961 -0.10656261]
 [-0.2826217  -0.59847059  0.74963853]]
```

2. Computing Local APPFDs for 3D Protein Surfaces.

We provide a full implementation of local APPFD algorithm in **local_appfd_method.py** , which is intended to compute keypoints Augmented Point-pair Features Descriptor (APPFDs)for each Local Surface Patche (LSP) about a keypoint (K_p),for a given 3D protein surface - Mesh or Point Cloud.

The local_appfd_method.py script contains only ONE function - the keypoints_APPFD_6x35bins(args) which implements the local APPFD method. This alforithm takes in TWO main inputs (P_s and N_s) and FOUR other parameters, thus:

> - **i. pointsCloud:** N x 3 array, PointsCloud, P_s for a single 3D model.</break>
> - **ii. normals:** N x 3 array of Normal Vectors, N_s correspoinding to every points in the pointsCloud (i).

> - **iii. nSamples:** Number 'Random'/'Uniform' points to samples from 3D Triangular Mesh (i.e filename.obj). Default N = 3500
> - **iv. r (Float: Default = 0.27):** Radius param, used by r-nn search to determine the size of Local Surface Patch or Region.
> - **v. nBins = 35** #Number of bins for the 1-dimensional histogram of each of the Feature-dimension. Default = 15.
> - **vi. voxel_size (Float, Default = 0.15):** Parameter to be used by the Voxel Down-Sampling function of Open3D.

NOTE: The output from the **local_appfd_method.py / keypoints_APPFD_6x35bins(args)** algorithm/function is a $[K \times D]$ array of LSP descriptors (each for a single K_p or LSP) and for a single input 3D model (i.e. protein surface), where K is the total number of keypoints detected (or determined) as well as the number of extracted LSP for the 3D surface described, and D is the dimension of the computed local APPFD, which in this implementation, happens to be $6 \times 35 = 210$-dimension.

> Therefore, the output from the local APPFD algorithm above in a $[K \times 210]$ array of keypoints APPFD for each input 3D model.

The following lines of code computes local APPFD descriptors:

In [6]:
```
N = 3500
nBins = 35

# Note, the LARGER the value of 'r', the more the number of points in the 'Local Surface Patch(s)' extracted.
r = 0.50

# Voxel size: For voxel-grid downsampling algorithm.
voxel_size = 0.20

print("...Computing Shape-Descriptor for 3D Model: 0_shape.off")
print('\n')

startTime2 = time.time()
descriptors = appfd.keypoints_APPFD_6x35bins(Ps, Ns, N, r, nBins, voxel_size)
stopTime2 = time.time()
duration2 = stopTime2 - startTime2
print("Processing Time, Computing Local APPFD For: 0_shape.off:\t", str(duration2) + 'secs.')
print('\n')
```

```
...Computing Shape-Descriptor for 3D Model: 0_shape.off


Downsampled Cloud Size:          534
Processing Time, Computing Local APPFD For: 0_shape.off:         76.77373385429382secs.
```

LOCAL APPFDs - Confirmation:

NOW, let's inspect the computed local APPFDs for ***'0_shape.off'***. Below, we expect to see $[K \times D]$ array of output, Where $K = 534$ and $D = 210$.

In [7]:
```
print("Shape of Local APPFDs for: 0_shape.off:\n", descriptors.shape)
print()
print("Local APPFDs for: 0_shape.off:\n", descriptors)
```

```
Shape of Local APPFDs for: 0_shape.off:
 (534, 210)

Local APPFDs for: 0_shape.off:
 [[0.16892169 0.43624434 0.64616644 ... 0.03198032 0.01189012 0.00041   ]
 [0.18683128 0.45555556 0.68312758 ... 0.01481481 0.00864198 0.00041152]
 [0.19116347 0.57673049 0.81178206 ... 0.04418262 0.0167894  0.00147275]
 ...
 [0.21100326 0.54498386 0.87572813 ... 0.03430421 0.01165048 0.002589  ]
 [0.14048532 0.35759896 0.53128988 ... 0.03192848 0.01277139 0.00766283]
 [0.1429019  0.40739578 0.69946724 ... 0.036979   0.01692259 0.00250705]]
```

In [ ]:
```

```

3. Computing Final APPFD-FK-GMM *i.e.* Fisher Vector (FV) Descriptor for A Single 3D Model.

Using the $[K \times D]$ local APPFD output from the ***local_appfd_method.py*** method.

i. The first step to computing a single/compact final 3D shape descriptor, which we call the APPFD-FK-GMM (a fisher-vector (FV) derived from locally computed APPFD for a single input 3D model, using Fisher-Kernel technique and Gaussian Mixture Model) is to:

> Compute LSP (i.e. $[K \times D]$) APPFDs for all database 3D models. The outcome of that would be an $M \times [K \times D]$ vertically-stacked LSP APPFDs. Where M is the total number of 3D models (protein surfaces) in the given dataset(database).
>
> For the SHREC 2021 Protein Domain Retrieval track, the ***Query*** dataset, for instance, contains 10 3D protein models. Therefore, we get $10 \times [K \times 210]$ local APPFDs, where the value, K varies with each 3D surface. For the ***Collections*** dataset which contains 554 protein models, we get $554 \times [K \times 210]$ local APPFDs would be returned.

ii. The second step in this implementation phase is to train/fit a Gaussian Mixture Model (GMM) with the $M \times [K \times D]$ vertically-stacked LSP APPFDs for all database 3D models. In our implementation, we adopt the ***GaussianMixture*** module from Scikit-Learn Python Library (i.e. from sklearn.mixture import GaussianMixture), using the ***diagonal*** Covariance Type and numbr of Gaussians = 10.

> **NOTE:** The fitted GMM must converge with the chosen number of Gaussians, while this value may vary for different domains of dataset. Therefore, different values must be tested until convergence is achieved. For this retrieval task, $nGaussians = 10$ converged.

To train/fit a GMM on a given $M \times [K \times D]$ data,the following code snippet would do:

In [ ]:
```
from sklearn.mixture import GaussianMixture
nGaussians = 10
gmm = GaussianMixture(nGaussians, covariance_type = 'diag', random_state = 0)
gmm.fit(M x [K x D])
```

iii. The third and final step to computing a single/compact final APPFD-FK-GMM 3D shape descriptor, for a single input 3D model (protein surface) is to compute a Fisher-Vector (FV) using the function ***fisher_vector(arg)***. This function takes the trained/fitted ***gmm*** object in step ii. and the locally computed $[K \times D]$ APPFDs for each 3D model as input.

> The ***fisher_vector(arg)*** function can be found in **fisher_vector.py** script.

In [ ]:
```

```

4. Matching Different 3D Models With APPFD-FK-FMM

For this research task, matching two different 3D protein models (both Shape-only and Shape+Electrostatics), for instance a i^{th}-Query and j^{th}-Collection models, has been reduced to finding the spatial (dis)similarity between the i^{th}-Query model APPFD-FK-GMM and j^{th}-Collection model APPFD-FK-GMM descriptors. This kind of matching is achieved using any spatial distance metric, such as the Cosine and Euclidean metrics.

> For our implementation, the Cosine metric returned better retrieval results. The results of comparig two APPFD-FK-GMM descriptors is a floating point number (similarity score), which is a value between 0.00 and 1.00. The smaller the value, the more similar the two models, whose APPFD-FK-GMM descriptors are compared are, and vice-versa.

In [ ]:
```

```