An Experimental Study of a Point Set Registration Algorithm

by

Hamid Homapour

M.Sc., Sharif University of Technology, 2013
B.Sc., Islamic Azad University, 2010

Project Submitted in Partial Fulfillment of the
Requirements for the Degree of
Master of Science

in the
School of Computing Science
Faculty of Applied Sciences

Approval

Name:	**Hamid Homapour**
Degree:	**Master of Science**
Title:	**An Experimental Study of a Point Set Registration Algorithm**
Examining Committee:	**Chair:** Thomas C. Shermer Professor
	Binay Bhattacharya Senior Supervisor Professor
	Ramesh Krishnamurti Internal Examiner Professor School of Computing Science
	Abraham P. Punnen External Examiner Professor Department of Mathematics Simon Fraser University Surrey
Date Defended:	**December 5, 2019**

Abstract

We study the problem of point set registration in the plane to measure the similarity between point sets in the plane. This problem plays an important role in a variety of applications, from computer vision to molecular biology. Here, in this project, we study, implement, test, and evaluate FPPMA which is one of the fastest algorithmic approaches in this context [43]. We then compare the results with another well-known technique in the computer vision community (GMMReg method [30]). Finally, we generalize FPPMA to solve the groupwise point set registration problem.

Keywords: point set registration; pairwise registration; groupwise registration

Dedication

To my parents - for their unwavering support.

Acknowledgements

I would like to thank my thesis advisor Professor Binay Bhattacharya of the School of Computing Science at Simon Fraser University. The door to Prof. Bhattacharya's office was always open whenever I ran into a trouble spot or had a question about my research or writing. He consistently allowed this paper to be my work but steered me in the right direction whenever he thought I needed it.

December 5th, 2019
Hamid Homapour

Table of Contents

List of Tables

List of Figures

List of Implementations

Chapter 1

Introduction

Pattern matching is important in various application areas, particularly in computer vision [35, 38, 41, 44], image registration [10, 32] and pattern recognition [8, 22]. It, also, has applications in other disciplines concerned with the form of objects such as astronautics [46], vehicle tracking [25], computational chemistry [20, 21], molecular biology [28, 38, 47], and medical imaging [27]. The applications often demand algorithms to find the similarity between a *pattern*, a set of objects, and an *image*, another usually a larger set of objects.

Finding correspondences is the primary goal of the point set registration algorithms to get an estimation of the transformation among two or more point sets. Proposed approaches, in this context, have some important challenges like deformation and the presence of noise in point sets. As examples of deformation, consider two differently taken images of an object with different points of view or two different images of an animal with different postures. Outliers and occlusion are examples of the presence of noise. Note that, the presence of the outlier is nearly like the presence of the occlusion, that is, in both situations, some points are present in one of the point sets while they are not in the other one. Moreover, large point sets and high dimensionality are the other challenges for methods in this context.

In the following, we first describe the basic definition of the nature of the problem followed by an overview of different proposed point pattern matching algorithms.

1.1 Problem Statement

Let P and Q be two point sets in 2-dimensional space; $P = \{p_1, p_2, ..., p_n\}$ and $Q = \{q_1, q_2, ..., q_m\}$, where all p_i and q_j are points in R^2. The goal is to find a similarity transformation, see [18, 24], T_{s,θ,t_x,t_y}, so that $T(P)$ matches Q. In transformation T_{s,θ,t_x,t_y}, s is a scaling factor, *theta* is a rotation angle, and t_x and t_y are translations along the x and y directions, respectively. Thus, for $(x, y) \in R^2$, we have:

$$T\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} t_x \\ t_y \end{pmatrix} + s\begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix}\begin{pmatrix} x \\ y \end{pmatrix}$$

Matching definition: Consider parameters $\rho \in [0,1]$ and $t \in R^+$ as the matching probability and the matching size, respectively. We define $T_{s,\theta,t_x,t_y}(P)$ matches Q, if there exists a subset P' of P and $|P'| \geq \rho n$, such that for each $p \in P'$ we have $|T_{s,\theta,t_x,t_y}(p) - q| < t$ for some $q \in Q$ [43].

Choose t: Let r be the radius of the minimum circle that contains point set Q. Considering uniform distribution of points, $\frac{r}{\sqrt{n}}$ is the average shortest distance of points in Q. A constant fraction (λ, i.e. matching factor) of this distance is a good candidate for t. Therefore, we have $t = \lambda.\frac{r}{\sqrt{n}}$.

1.2 Variants of the Pattern Matching Problem

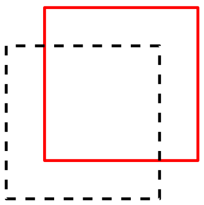
Pattern matching measures the similarity between two objects. Variants of this problem can be defined considering different criteria on the type of objects being matched, distance and similarity measures being used, types of valid transformations being considered, etc. First, let's define some of these variants:

- *Pairwise and Groupwise of objects*: Pairwise point pattern matching problem only considers two point sets and looks for only one transformation while groupwise point pattern matching problem considers more than two point sets and looks for multiple transformations for different sets simultaneously.

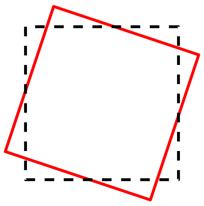
- *Type of objects*: Objects which are being matched are usually finite sets of shapes like points, line segments, or shapes given by polygons in two dimensions or polyhedral surfaces in higher dimensions. Most of the work has concentrated on point sets in two and three dimensions, Alt et al. [5].

- *Exact or approximate matching*: In the exact matching, the question is whether two sets can be matched exactly by some transformations. In practice, there are usually small errors (or noise) in the data which make an exact match unlikely. To overcome this problem, we can consider algorithms bringing each object in *pattern* within ϵ-*distance* of some object in *image*, for some $\epsilon > 0$ and distance function.

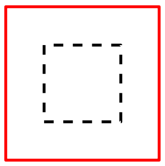
- *Fixed or dynamic pattern set*: Fixed pattern set means that the position of *pattern* objects remains unchanged and dynamic pattern set means that the position of *pattern* set can be changed using some transformations. The most important transformations (for two-dimensional space) are:
  - *Translation*: The simplest type of transformations are translations. The pattern set can be translated by some translation vector $t \in \mathbf{R}^2$, see Figure 1.1.a.
  - *Rotation*: The pattern set, also, can be rotated about a point (e.g. the rotation matrix about the origin is $\begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}$, where $\theta \in [0, 2\pi)$), see Figure 1.1.b.



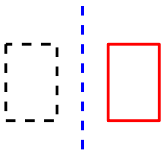
(a) Translation



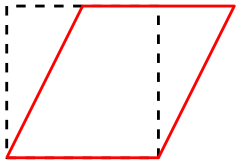
(b) Rotation



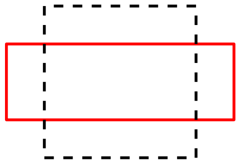
(c) Scaling



(d) Reflection



(e) Shearing



(f) Non-uniform Scaling

Figure 1.1: This figure shows some examples of transformations. Dashed and solid lines indicate original and transformed objects, respectively.

- *Scaling*: The linear transformation that stretches an object by a certain factor λ about the origin and is represented by the matrix $\begin{bmatrix} \lambda & 0 \\ 0 & \lambda \end{bmatrix}$ in two dimensions, see Figure 1.1.c.
- *Reflection*: In a reflection transformation, all the points of an object are reflected or flipped on a line called the axis of reflection or line of reflection, see Figure 1.1.d.
- *Rigid transformation or Euclidean transformation*: Rigid transformation is a combination of translation, rotation, and reflection. In most algorithms, the pattern only undertakes translation and rotation. The reflection can easily be included by running the algorithm twice, one for the original location of the pattern set and the other for the reflected pattern set, and return the better.
- *Homothety*: The Homothety is a combination of translation and scaling transformations.
- *Similarity*: The Similarity is a combination of rigid motions and scaling.
- *Non-rigid*: Typically these kind of transformation, involves nonlinear transformation. Non-rigid transformation include *affine* transformations such as scaling and shear mapping (see Figure 1.1.e), and can change the size but not the shape of the pre-image, see Figure 1.1.f.
- *Affine*: The affine transformation, which is nonrigid, includes transformations that preserve ratios of distances and collinearity (i.e. points which are on a line,

still be on a line after transformation). For example translation, rotation, reflection, similarity, contraction, expansion, homothety, dilation, anisotropic scaling, and skews.

Note: Roughly speaking, based on the difference in transformations, we can classify point set registration approaches into rigid and non-rigid transformations.

- *Distance function.* A distance function d on a set S is a nonnegative valued function $d : S \times S \to R^{\geq 0}$. Distance function has been used in calculating the similarity/dissimilarity between sets. For many pattern matching applications, it is desirable that d has some of the following properties:

  1. *Self-identity*: For all $x \in S$, $d(x, x) = 0$.
  2. *Positivity*: For all $x, y \in S$, $x \neq y$, $d(x, y) > 0$.
  3. *Symmetry*: For all $x, y \in S$, $d(x, y) = d(y, x)$.
  4. *Triangle inequality*: For all $x, y, z \in S$, $d(x, z) \leq d(x, y) + d(y, z)$.
  5. *Transformation invariant*: For a chosen transformation group G, for all $x, y \in S$ and $g \in G$, $d(g(x), g(y)) = d(x, y)$. This also implies that $d(g(x), y) = d(x, g^{-1}(y))$.

  A function with properties $1 - 4$ is called a metric.

- *Dissimilarity measure*: A dissimilarity measure is a function defined on pairs of shapes indicating the degree of their discrepancy. Depending on the applications, different dissimilarity measures can be considered:

  – *Hausdorff distance*: The Hausdorff distance is the maximum of the distances from each object in one set to its nearest neighbor in the other set [2–4, 7, 14, 26, 29]. Perhaps the most studied similarity measure between point sets is the directed Hausdorff distance [7]. Aichholzer et al. [2] showed a $(1 + \pi/4)$-approximation algorithm for rigid motion and a $(3 + \pi/4)$-approximation algorithm for similarity transformation (translation, rotation and scaling) having the running time of $O(mn \log(mn) \log^*(mn))$. Other approximation algorithms were given by Goodrich et al. [26] who showed a 2-approximation algorithm with running time $O(nm \log n)$ for translations and a 4-approximation algorithm with running time $O(n^2 m \log n)$ for rigid transformations. Chew et al. [14] gave algorithms for minimizing Hausdorff distance for two sets of planar points under rigid motions. Their algorithm runs in time $O(m^3 n^2 \log^2(mn))$ for one-sided Hausdorff distance and in time $O((m+n)^5 \log^2(mn))$ for bidirectional Hausdorff distances. The algorithms have a decision subroutine to determine whether there exists a matching under some transformation which brings the pattern set in the ϵ-neighborhood of the image set.

  - *Euclidean Distance*: The Euclidean distance is defined by the sum of the Euclidean distances between the matched pairs [37,45]. In 2006, a graphical models-based algorithm presented by Caetano et al. [12] to get point pattern matching in Euclidean spaces of any dimension. This polynomial-time algorithm is optimal for complete matching between noiseless point sets. Later, McAuley et. al. [33] gave a new graph showing that improved performance and applied it to obtain point set matching. These algorithms are designed for affine transformations, but, they can not handle point sets under reflection transform. For this issue, Wang et. al. [45] showed an efficient algorithm to handle reflections.
  - *Bottleneck distance*. Bottleneck distance denotes the maximal distance between all matched pairs while matching is injective [6,7,19]. In contrast to the directed Hausdorff distance, the bottleneck distance has the advantage of being symmetric for equally-sized sets. Here are some state-of-the-art results on some of these variants. Benket et al. [7] gave some $1+\epsilon$-approximation for equally-sized point sets under similarity transformation. Erfat et al. [19] presented a scheme to find a translation approximating minimum bottleneck for equally-sized point sets in $\mathbf{R}^d$. The algorithm first achieves a preliminary constant factor approximation for the problem. Then, based on this approximated value of optimal solution introduces an $(1+\epsilon)$ approximation factor. The total time of the algorithm is $O((1+\frac{4}{\epsilon}\sqrt{d})^d \log \epsilon^{-1} d(1+\frac{1}{\epsilon})^d n^{1.5} \log n)$.
  - *Earth mover's distance(EMD)*: This similarity measures are defined for weighted objects and intuitively, can be seen as the least amount of work needed to move earth on the *pattern* objects to the holes in *image* objects [11, 15, 17, 31, 36, 39]. The earth mover's distance between two weighted point sets in the plane is a similarity measure whose potential use for measuring shape dissimilarity was first proposed by Mumford [36]. The problem of earth mover's distance under transformation has received considerable attentions. No algorithm is known that computes an exact solution for matching weighted point sets in the plane that minimize EMD under rigid motions. Cohen et al. [15] who introduced the problem, provided an iterative algorithm to find a local optimum for EMD under rigid transformation. Later, Cabello et al. [11], Klein et al. [31], and more recently Ding et al. [17] presented some approximate solutions, which are mainly of theoretical importance.

- *Partial/complete matching*: When the cardinalities of sets are equal, the problem is called complete matching. Otherwise, it is called partial matching in which a small set is matched to a subset of equal size of a bigger set.

1.3 Current Work

This project deals with the matching of geometric points, with an emphasis on techniques from computational geometry. Throughout this project report, unless otherwise stated, we consider matching for a *pattern* set P of size n and an *image* set Q of size m. The elements of the set are points in $\mathbf{R}^2$, and the *pattern* set can undertake translation, rotation, and scaling. We consider *Euclidean distance* as similarity and dissimilarity measures.

The rest of the paper is organized as follows. Chapter 2 describes FPPMA [43] and the details of its implementation. Chapter 3 shows the experimental results of FPPMA, and a comparison with the well-known GMMReg method. In Chapter 4, we use FPPMA to solve groupwise version of the problem.

Chapter 2

The Algorithm in Focus - FPPMA

The main purpose of this project is to implement, test, generalize, and evaluate the algorithm provided in [43] for 2D point pattern matching. We call this algorithm FPPMA (Fast Point Pattern Matching Algorithm). In the following, we give a brief description of FPPMA.

FPPMA comes with two main ideas. First, the idea is to select a random point in P, and hopefully, we can find its corresponding point in Q by a few random selections. Thus, by taking a random point (p) in P, the algorithm searches for a match (q) in Q (i.e. a local match). The second idea is to find a transformation T that matches the nearest neighbors of p to those of q, in the hope that T is a global transformation. The following is the pseudo-code for FPPMA.

Algorithm 1 Fast Point Pattern Matching Algorithm (FPPMA) [43]

```
1: procedure FPPMA
2: input:
3:     $P$, $Q$
4: pre-processing:
5:     For each point in $P$ and $Q$ find the $k$ nearest neighbors and store
   them in separate lists for each point set.
6:     Make a grid (₲) over the point set $Q$, and maintain a list of points
   for each cell of the grid.
7: main computations:
8:     for $i = 1$ to $n$ do
9:         for $j = 1$ to $m$ do
10:            Find a transformation $T$ (considering $p_i$, $q_j$)
11:            if $T$ is a local match then
12:                if $T$ is a global match then
13:                    return $T_{s,\theta,t_x,t_y}$
```

In this project, to implement FPPMA presented in [43], C++ is used as the programming language, and Python is used for illustration purposes. The C++ code consists of different parts: Input Reader, Nearest Neighbor Finder, Collision Detector, Road-map Maker, ShortestPath Finder, and Visualization. In the following, we will describe each part.

2.1 Input Reader.

Point set P is read from a comma-separated file (i.e. a CSV file) such that each line contains coordinates of a point. Therefore, *Input Reader* is for reading a CSV file, and converts and stores coordinate information into some arrays.

2.2 Transformation Finder Class

Transformation Finder, see A.1, is the main part of the implementation, that is, the implementation of FPPMA presented in [43]. In the following, explanations and detailed information about Transformation Finder are presented.

2.2.1 Proximity and Neighborhood

As mentioned before, the main idea of FPPMA is to find a local transformation for k nearest neighbors of a randomly selected point in P and a point in Q. For a neighborhood of a point, three different methods are used to select neighbors of a point. The three methods are k nearest neighbors for the first method, and Delaunay and Gabriel neighbors in Delaunay and Gabriel graphs for the two other methods, respectively. Thus, we implement three different versions of the algorithm, that is, FPPMA-kNN, FPPMA-dNN, and FPPMA-gNN to select k nearest, Delaunay, and Gabriel neighbors, respectively. In the following, a brief explanation of the neighborhood and some of its variants that are used in this project are given.

Proximity and location are basic concepts of computational geometry. The term proximity is related to the concept of closeness or neighborhood. Computing geometric structures with the proximity in mind is a requirement for most of the problems in this field. For example, Delaunay triangulation, Voronoi diagram, and similar graph structures like Gabriel graph or the relative neighborhood graph [16]. Moreover, nearest neighbors, geometric range searching, and related problems are another group of problems with the same requirement [34]. These structures and searching techniques have many applications, including data mining, document analysis, classification, and pattern recognition.

In the project, the three methods are implemented as a child class of *Base Finder* class, see B.1. The followings are a description of the methods and their implementation detail.

a. Nearest Neighbors - kNN

In the nearest neighbor (NN) problem, a set P containing n points is given in d-dimensional space. The task is to preprocess the set P and build a data structure. Then, for any given query point q, the algorithm is concerned with finding the point closest to the query point (generally k nearest points of P to q). Note that, different distance functions can be considered to compute the distance between two points. For example, Euclidean distance, Manhattan distance, and max distance. In the following, ANN library is introduced which is used in our project to be able to search nearest neighbors.

ANN **library [1]**: ANN is a library for approximate and exact nearest neighbor searching written in the C++ programming language. The work is done by David M. Mount of the University of Maryland and Sunil Arya of the Hong Kong University of Science and Technology.

ANN does some preprocessing over the given point set (say P), and creates a *KD-Tree* data structure. Then, for any query point $q = (x, y)$, it efficiently reports the nearest or generally k nearest points of P to q. ANN is an in-memory approach, that is, data sets should be small enough so that the search structure can be stored in the main memory. Both preprocessing time and space are linear to the number of points and the dimension. Also, any class of distance functions of Minkowski metrics can be used in ANN. For this project, we used Euclidean distance to find the nearest neighbors of a point.

In this project, ANN is used to find k nearest neighbors of each point in P and Q. In the preprocessing time, we find k nearest neighbor of each point of P and Q, and maintain them in a separate list for easy access. *NNFinder*, see B.2, is a child class of *BaseFinder*, and it deals with the problem of finding nearest neighbors in our implementation.

In the rest of this report, we use FPPMA-kNN to refer to the implementation of FPPMA using nearest neighbors.

b. Delaunay Neighbors - dNN

For a given set P of discrete points in a plane, a Delaunay triangulation (DT) is a triangulation in which for any triangle in $DT(P)$, no point of P is inside the circumcircle of the vertices of the triangle. In such a triangulation, the minimum angle of all the angles of the triangles is maximized. Therefore, Delaunay triangulation gives us a *fat* triangulation. It is worth to mention that Delaunay triangulation is completely related to the Voronoi diagram. It corresponds to the dual graph of the Voronoi diagram for P.

Bowyer-Watson algorithm: In our implementation, to compute the Delaunay triangulation, we used the Bowyer-Watson algorithm. It can be used to compute DT of a finite set of points, as well as the Voronoi diagram of the points (i.e. the dual graph of DT). Actually, the algorithm is incremental. In each iteration, one point is added to a valid Delaunay triangulation of the points which are already added. By adding a new point (p_i), any triangles that p_i lies in its circumcircles, are deleted. Then, the affected area is a star-shaped polygonal hole, and it is then re-triangulated using the new point [40].

In the implementation, *DNFinder*, see B.3, is a child class of *BaseFinder*, and it deals with the problem of finding neighbors based on the Delaunay triangulation.

In the rest of this report, we use FPPMA-dNN to refer to the implementation of FPPMA using Delaunay neighbors.

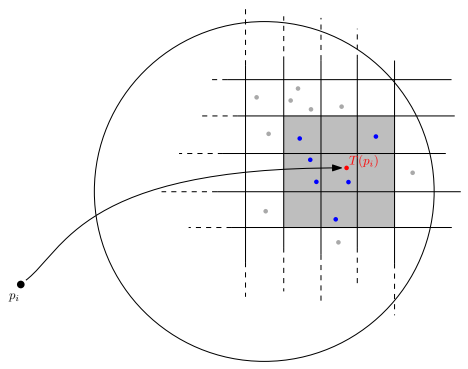


Figure 2.1: Using the grid strategy to facilitate searching to find a possible match.

c. Gabriel Neighbor - gNN

For a set P of points in the Euclidean plane, the Gabriel graph shows one notion of proximity or closeness of those points. More formally, two points p and q in P form an edge of the graph if they are distinct, and no point of P is inside the closed disc of which line segment pq is a diameter [23]. Note that, the Gabriel graph is a subgraph of the Delaunay triangulation [16]. Computing the Gabriel graph, when the Delaunay triangulation is given, can be done in linear time. Also, it is worth to mention that the Euclidean minimum spanning tree, the relative neighborhood graph, and the nearest neighbor graph are a subgraph of the Gabriel graph.

Gabriel Graph Computation: Simply, the Gabriel graph computation is based on the euclidean Delaunay triangulation and keeps only edges whose circumcircle does not contain any other input point than the edge extremities. The implementation can be found in B.4. *GNFinder*, see B.4, is a child class of *DNFinder*, and it deals with the problem of finding neighbors based on the Gabriel graph.

In the rest of this report, we use FPPMA-gNN to refer to the implementation of FPPMA using Gabriel neighbors.

2.2.2 Grid Search Method

Another part of the Transformation Finder class, see A.1, is a strategy to look up for a neighborhood of a given position fast. When we want to find a possible match for a transformed point $T(p_i)$ in the target point set (i.e. Q), we need a fast strategy to look for the potential match. We need to check whether there is a point in Q in a short distance ($\leq t$) of any coordinate $T(p_i) = (x, y)$. In this regard, the algorithm does some preprocessing to create a grid ($\mathbb{G}$) over the target point set Q. By creating the grid, considering c as an arbitrary cell, a list of points that lie in c is stored in a separate list for later use.

Therefore, to find a potential match for a point p_i for a given transformation T, the algorithm finds a cell c of the grid that contains $T(p_i)$. Then, 9 cells of the grid closest to c (including c itself) are used to find a potential match, see Figure 2.1. By doing so, the query time for a given transformation and a given position requires $O(1)$ time to search for a potential match.

In the project, *Grid* class, see C.1, is provided to make a grid over the target point set to facilitate the searching process for a possible match.

2.2.3 Find a Global Match

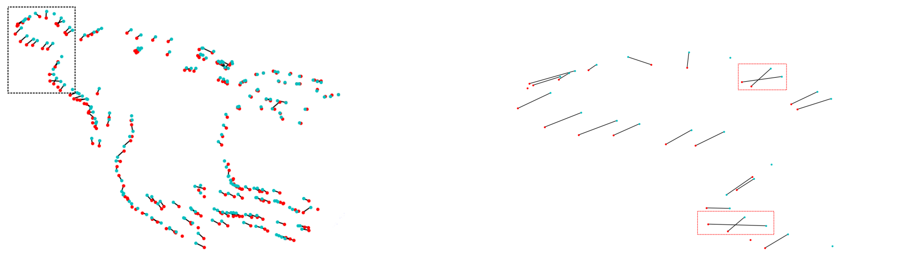
After finding a local match, that is, a transformation T that can transform some neighbors of a randomly selected point p in P to neighbors of a point q in Q, then, the process to check if T can be a global transformation is easy. The only thing is to apply T over the whole point set P, and for each point p in P test if it is possible to find a match in Q which is not further than a constant distance (i.e. within a threshold t). If the number of selected matches is more than $\rho.N$, then the transformation is considered as a global match (ρ is a matching probability). In the following, the pseudo code of the process is shown.

Algorithm 2 Global Match Test Algorithm

```
1: procedure IsGlobalMatch
2: input:
3:     $P$, $Q$, $T$, $\rho$, and $t$
4: main computations:
5:     $L_1 = \{\}, L_2 = \{\}$
6:     for $i = 1$ to $n$ do
7:         $tp_i = T(p_i)$
8:         $list = \mathbb{G}(tp_i)$
9:         if $list$ has a member ($q_j$) within distance $t$, and $q_j$ is not marked then
10:            append $p_i$ to $L_1$, and mark $q_j$ and add it to $L_2$
11:    if $|L_1| \geq \rho.n$ then
12:        return $T$
13:    return No global match found.
```

2.3 Finding Correspondences Strategies

Suppose FPPMA computed a transformation T from P to Q. Next, the question is after transforming point set P and having $T(P)$, how are we going to choose a match for a point t_i of $T(P)$? Therefore, we need a strategy to pick matches for each transformed point. It is worth to mention that one of the fundamental shape analysis tasks is to find a relevant correspondence among two or more points or shapes [42]. The original method that is used in [43] is to find the nearest neighbor of each point (the *Nearest Neighbor* match method) and consider it as a match. However, by having this strategy the algorithm makes some mismatches, see Figure 2.2. In this example, some points are mismatched or unmatched. To address this issue, we also study and implement two other strategies (based on unweighted and weighted Euclidean graph of the points) in our project to have better matches (i.e. to reduce the number of mismatches). In the following subsections, we describe these strategies, and in Chapter 3 we give an evaluation among these three different strategies.


(a) A computed transformation using FPPMA-kNN. The box shows a selected area.

(b) The figure shows some mismatched and unmatched points in the selected area. Matched points are connected together using an edge. Red rectangles illustrate some samples of mismatches. An isolated point shows an unmatched point.

Figure 2.2: Mismatches occurs due to the presence of the noise.

2.3.1 Unweighted Bipartite Matching Strategy

Let $G = (V, E)$ be an unweighted undirected graph with vertex set V, edge set E. Here, V is a union of points of $T(P)$ and Q where T is a computed transformation between point sets P and Q. Then, for each pair (u, v) of points where $u \in T(P)$ and $v \in Q$, $(u, v) \in E$ if u and v are within distance t of each other. Note that, by definition of the edges in G in which $u \in P$ and $v \in Q$, this graph is a bipartite graph.

By having the graph G, in *Unweighted Bipartite Matching* strategy, FPPMA computes a maximum bipartite matching [9] of G to make matches between points in $T(P)$ and Q.

2.3.2 Weighted Bipartite Matching Strategy

Let $G = (V, E)$ be an weighted undirected graph with vertex set V, edge set E, and a weight function w. Here, V is a union of points of $T(P)$ and Q where T is a computed transformation between point set P and Q. Then, for each pair (u, v) of points where $u \in T(P)$ and $v \in Q$, $(u, v) \in E$ while $w(u, v)$ denotes the weight of the edge which equals to the Euclidean distance between u and v.

By having the graph G, in *Weighted Bipartite Matching* strategy, FPPMA computes a maximum bipartite matching of G with the minimum cost [9] to make matches between points in $T(P)$ and Q. Note that, the cost of the matching is the sum of the weights of the edges of the matching. Therefore, the goal is to find a matching with maximum number of edges with the minimum cost.

Chapter 3

Pairwise Experiments

The pairwise point pattern matching algorithm only considers two point sets and looks for only one transformation. In this chapter, we present experimental results on the implementation of point pattern matching algorithm proposed in [43] using different kinds of point sets for pairwise pattern matching.

All of our experiments were performed on a PC with 16 GB of RAM and an Intel(R) Core(TM) i7-3770 CPU (3.40GHz).

3.1 Data and Point Set Generation

The algorithm is tested using many different 2D point-sets, including complete matching and partial matching options. For partial matching we select a different part of a point set in portion (i.e. a subset of the points that lie on a connected subset of the 2D-space), or randomly removed some data points, see Figure 3.1.

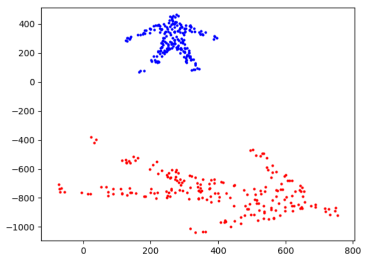
To do that, a point set generation code is implemented. Here, by doing some random transformation on a point set P, the code generates a point set Q. This is done by applying some random translation (a random move between 0 to 15 along the x and y axis), rotation (a random rotation between 0 to 360 degrees), and scaling (a random scaling between 0.1 and 5) over the point set P. Also, it applies some random noise for each point independently.

3.2 Evaluation

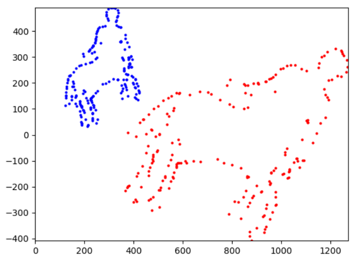
In this section, we are going to talk about the experiments that run over FPPMA-kNN, FPPMA-dNN, and FPPMA-gNN algorithms. The performances of these representative point set registration algorithms are validated using different kinds of data sets.

To evaluate the performance of the rivals, we use the Mean Squared Error Distance (MSED) as the cost function of the optimization problem. It is worth to mention that all the algorithms were implemented in C++.

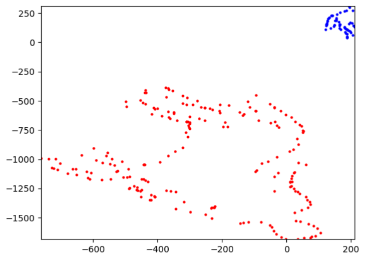
In the following, a discussion and result data (including average number of selected pairs, average running time, average error distances) for each type of data including *complete*



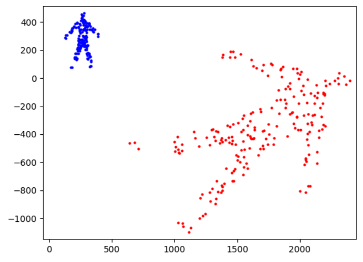
(a) Sample data for complete matching with the same size point sets.



(b) Sample data for complete matching (only border) with the same size point sets.



(c) Sample data for partial matching with different size point sets.



(d) Sample data for partial matching (random removal - 20% of the blue points are removed randomly).

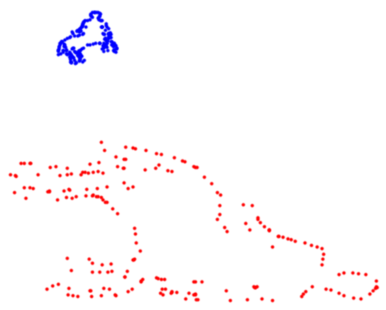
Figure 3.1: This figure shows some examples of transformations data. Blue and red points indicate model and scene, respectively.

matching and partial matching is presented. Note that, as pairwise matching case, here the algorithm looks only for one transformation to transform P into Q. We summarize the results of the evaluation in tables in the following. Note that, the results are the average results over 20 trials. In all the tables, for each data set the size of each set of points is about 200. ρ is the matching probability, λ is the matching factor, see Section 1.1. k is the number of neighbors, $k2$ and $k3$ are the number of neighbors of a point in P and Q to consider to find a global match, respectively. *#pairs* is the average number of pairs of points that are checked for a local match, $\#pairs(min)$ is the minimum number of checked pairs among samples, $\#pairs(max)$ is the maximum number of checked pairs among samples, *time* is the average time in milliseconds, *ave dist* is the average distance (i.e. Euclidean distance) of the transformed points to the nearest corresponding matching points, and *ave max dist* is the average distance of the farthest points in the results.

3.2.1 Complete Matching (CM-DATA)

The first data type that is used to test our implementation is *complete matching*. Here, we know that the cardinalities of the point sets are equal, for example see Figures 3.1a and 3.1b.

In the following tables, the summarized information of the evaluation for FPPMA-kNN, FPPMA-dNN, and FPPMA-gNN with CM-DATA are presented, respectively. Also, in Figuer 3.2a, a sample input of CM-DATA is shown, and Figure 3.2b shows the transformed points by FPPMA-kNN.


(a) An example input of CM-DATA type where point set P and Q are shown using blue and red points, respectively.

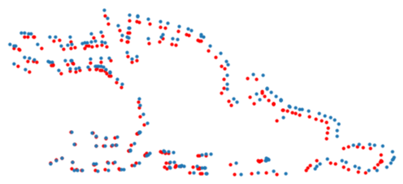

(b) In the figure, green points indicate transformed points using FPPMA-kNN.

Figure 3.2: This figure shows an example result of FPPMA-kNN with CM-DATA.

Table 3.1: For each argument, the best parameters and results for FPPMA-kNN runs with complete matching data (CM-DATA) are provided.

	arguments		parameters			results						
	ρ	λ	k	k2	k3	#soln	#pairs	#pairs (min)	#pairs (max)	ave time	ave dist	ave max dist
1	0.9	0.2	5	1	2	20 / 20	3534.25	169	8033	452.5	7.27	17.07
2	0.9	0.2	5	2	3	20 / 20	2360.4	25	8743	1293.24	6.3	15.23
3	0.9	0.2	5	3	3	20 / 20	1729.55	57	10346	1490.16	7.09	16.41
4	0.9	0.4	5	3	3	20 / 20	704.1	47	2007	636.94	9.95	24.57
5	0.9	0.2	5	3	4	20 / 20	1180.5	5	4425	1370.78	6.97	16.28
6	0.9	0.4	5	3	4	20 / 20	534.9	4	3171	662.41	12.31	30.57
7	0.9	0.2	6	1	2	20 / 20	2881.15	47	7862	399.47	6.22	14.67
8	0.9	0.2	6	2	3	20 / 20	1717.2	29	7067	1030.27	7.14	16.63
9	0.9	0.2	6	3	3	20 / 20	1265.25	39	4689	1205	7.12	16.65
10	0.9	0.4	6	3	3	20 / 20	694	10	3477	692.9	10.98	26.95
11	0.9	0.2	6	3	4	20 / 20	1158.05	166	4175	1503.33	7.01	16.34
12	0.9	0.4	6	3	4	20 / 20	801.9	48	3313	1109.88	12.72	30.91
13	0.75	0.4	6	3	4	20 / 20	171.95	10	463	236.01	13.54	34.81
14	0.75	0.4	7	2	3	20 / 20	229.55	44	602	166.27	15.63	40.29
15	0.75	0.4	7	3	3	20 / 20	178.45	2	777	202.51	13.6	36.04
16	0.75	0.4	7	4	4	20 / 20	213.9	12	606	447.51	15.9	43.96
17	0.75	0.4	9	3	4	20 / 20	236.3	9	1020	454.2	14.29	36.06
						average:	1152.44	42.53	4163.29	785.49	10.24	25.5

Table 3.2: For each argument, the best parameters and results for FPPMA-dNN runs with complete matching data (CM-DATA) are provided.

	arguments		parameters			results						
	ρ	λ	k	k2	k3	#soln	#pairs	#pairs (min)	#pairs (max)	ave time	ave dist	ave max dist
1	0.9	0.2	5	1	2	20 / 20	1881.1	87	9675	189.91	6.24	15.19
2	0.9	0.2	5	2	3	20 / 20	896.3	20	4048	461.19	7.08	16.57
3	0.9	0.2	5	3	3	20 / 20	942.25	10	2627	776.52	6.7	15.42
4	0.9	0.4	5	3	3	20 / 20	681.7	39	2833	601.23	10.63	26.09
5	0.9	0.2	5	3	4	20 / 20	1062.25	30	3593	1339.51	7.26	16.51
6	0.9	0.4	5	3	4	20 / 20	876.65	18	2856	1171.96	11.98	29.91
7	0.9	0.2	6	1	2	20 / 20	2505.5	62	11925	171.09	6.59	15.35
8	0.9	0.2	6	2	3	20 / 20	1454.65	42	6427	629.16	7.18	16.31
9	0.9	0.2	6	3	3	20 / 20	874.7	18	4244	670.53	6.34	15.2
10	0.9	0.4	6	3	3	20 / 20	729.35	76	3121	536.01	10.13	23.64
11	0.9	0.2	6	3	4	20 / 20	1947.15	124	7313	2355.15	6.88	15.31
12	0.9	0.4	6	3	4	20 / 20	777.1	2	3210	1060.12	10.53	25.66
13	0.75	0.4	6	3	4	20 / 20	500.65	12	1590	623.6	14.17	37.15
14	0.75	0.4	7	2	3	20 / 20	968.2	48	3208	203.49	10.35	25.74
15	0.75	0.4	7	3	3	20 / 20	747.2	1	2682	402.9	12.3	31.66
16	0.75	0.4	7	4	4	20 / 20	734.7	14	3033	1168.54	12.63	33.81
17	0.75	0.4	9	3	4	20 / 20	4209.1	47	23903	1454.71	8.75	21.24
						average:	1281.68	38.24	5664	812.68	9.16	22.4

Based on the information provided in the tables 3.1, 3.2, and 3.3, the *average time/number of pairs* required to find transformation for data sets with size 200 are 785/1152, 812/1281, and 4082/29404 milliseconds/pairs for FPPMA-kNN, FPPMA-dNN, and FPPMA-gNN, respectively. Thus, we can say FPPMA-kNN and FPPMA-dNN act almost the same and they are much faster than FPPMA-gNN. On the other hand, by comparing average distances information the story can be changed. Based on the average timing, although FPPMA-gNN is slow, its accuracy can be much better than FPPMA-kNN and FPPMA-dNN. The *average distances/average max distances* are 10.24/25.5, 9.16/22.4, and 8.1/20.99 for FPPMA-kNN, FPPMA-dNN, and FPPMA-gNN, respectively. All in all, we can say for timing purposes FPPMA-kNN and FPPMA-dNN are better, and for accuracy, FPPMA-gNN is better than the others.

Table 3.3: For each argument, the best parameters and results for FPPMA-gNN runs with complete matching data (CM-DATA) are provided.

	arguments		parameters			results						
	ρ	λ	k	k2	k3	#soln	#pairs	#pairs (min)	#pairs (max)	ave time	ave dist	ave max dist
1	0.9	0.2	5	1	2	4 / 20	38026.35	10729	53361	777.61	4.11	9.82
2	0.9	0.2	5	2	3	15 / 20	20054.85	712	44100	1618.34	6.15	15.42
3	0.9	0.2	5	3	3	18 / 20	16720.7	377	42849	3181.28	6.34	14.91
4	0.9	0.4	5	3	3	18 / 20	10244.85	50	42849	1859.94	9.95	24.3
5	0.9	0.2	5	3	4	17 / 20	16805.1	177	42849	6680.59	6.21	14.68
6	0.9	0.4	5	3	4	18 / 20	11741.5	473	42849	4992.88	11.52	28.6
7	0.9	0.2	6	1	2	1 / 20	41103.8	32041	53361	692.21	7.02	17.01
8	0.9	0.2	6	2	3	3 / 20	38166	9233	53361	2133.93	4.3	10.8
9	0.9	0.2	6	3	3	6 / 20	36562.65	4791	53361	4096.25	5.42	14.27
10	0.9	0.4	6	3	3	11 / 20	26303.6	1876	50176	2802.22	9.95	25.23
11	0.9	0.2	6	3	4	7 / 20	34464.7	5333	53361	8603.99	5.59	14.44
12	0.9	0.4	6	3	4	11 / 20	28399.3	1149	53361	6970.09	9.25	23.69
13	0.75	0.4	6	3	4	17 / 20	23333.7	9042	50176	5558	13.45	36.58
14	0.75	0.4	7	2	3	1 / 20	41103.8	32041	53361	1994.67	7.22	18.8
15	0.75	0.4	7	3	3	1 / 20	40444.65	21042	53361	3201.17	9.16	29.53
16	0.75	0.4	7	4	4	6 / 20	35294.15	2289	53361	10322.98	14.12	43.78
17	0.75	0.4	9	3	4	1 / 20	41103.8	32041	53361	3923.53	8.01	15
						average:	29404.32	9611.53	49968.12	4082.92	8.1	20.99

3.2.2 Partial Matching (PM-DATA)

Partial matching means the cardinalities of the point sets are not equal, that is, a smaller set is matched to a subset of equal size of the bigger set, see Figures 3.1c and 3.1d. In this project, different kinds of partial matching data are considered.

Partial Matching - Portion (PM-P-DATA):

In this case, for the data set mentioned in 3.1, the point set P is extracted in portion, the neighborhood of the points are intact. For example, take a look at Figure 3.1c, we run FPPMA-kNN, FPPMA-dNN, and FPPMA-gNN with different PM-P-DATA data, and summarized information can be found in the following tables.

Table 3.4: Summary of results of FPPMA-kNN with partial matching data (PM-P-DATA) is provided.

	results						
	#soln	#pairs	#pairs (min)	#pairs (max)	time	ave dist	ave max dist
average:	60 / 60	621.96	13.67	2731.19	309.42	9.82	24.33

Table 3.5: Summary of results of FPPMA-dNN with partial matching data (PM-P-DATA) is provided.

	results						
	#soln	#pairs	#pairs (min)	#pairs (max)	time	ave dist	ave max dist
average:	60 / 60	1116.57	15.83	2084.11	456.57	9.11	22.83

Table 3.6: Summary of results of FPPMA-gNN with partial matching data (PM-P-DATA) is provided.

	results						
	#soln	#pairs	#pairs (min)	#pairs (max)	time	ave dist	ave max dist
average:	17 / 60	11718	1793.05	2539.43	1880.76	4.35	13.65

Partial Matching - Randomly Removed (PM-R-DATA):

Here, for the data set mentioned in 3.1, the point set P is extracted by random removal of points. Therefore, it is worth to mention that unlike PM-P-DATA, for PM-R-DATA the neighborhoods of the points are changed. For example, take a look at Figure 3.1d. We run FPPMA-kNN, FPPMA-dNN, and FPPMA-gNN with different PM-R-DATA data, and summarized information can be found in the following tables where for each data set only 20 percent of the points are removed randomly.

Table 3.7: Summary of results of FPPMA-kNN with partial matching data (PM-R-DATA) is provided.

	results						
	#soln	#pairs	#pairs (min)	#pairs (max)	time	ave dist	ave max dist
average:	20 / 20	1957.46	54.24	5460.89	768.62	10.04	24.59

Table 3.8: Summary of results of FPPMA-dNN with partial matching data (PM-R-DATA) is provided.

	results						
	#soln	#pairs	#pairs (min)	#pairs (max)	time	ave dist	ave max dist
average:	20 / 20	1319.33	89.38	3830.57	649.21	8.77	20.89

Table 3.9: Summary of results of FPPMA-gNN with partial matching data (PM-R-DATA) is provided.

	results						
	#soln	#pairs	#pairs (min)	#pairs (max)	time	ave dist	ave max dist
average:	7.9 / 20	25858.35	8448.65	1622.52	3186.01	9.64	23.71

3.3 Experimental Running Time of FPPMA-kNN

In table 3.10, FPPMA-kNN is run over sample data points with different sizes. In the table, #pairs shows the average number of pairs of points that are checked for a local match and time is the running time in milliseconds. Each row shows the average number of pairs and time of running FPPMA-kNN over 10 trials on each specific sample data. For each row, in each trial, the sample data remains intact. However, consider that in each trail the order of picking points from P and Q is by random order. All in all, based on the table 3.10, in the worst case, the time is basically the number of pairs times $O(n)$.

Table 3.10: This table shows running time information of FPPMA-kNN over sample data with different sizes.

sample size	#pairs	time
200	694	406
500	871	1177
1000	641	1581
1500	2882	10892
2000	14561	70971
3000	13826	113625
5000	44391	771616

3.4 Correspondence Strategies Evaluation

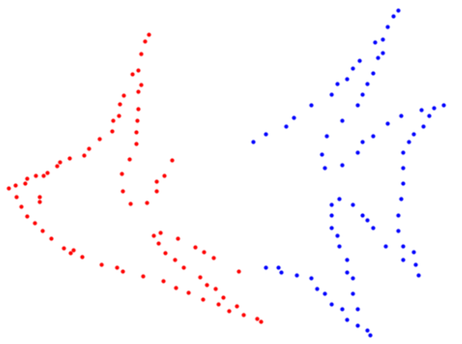
As we mentioned in Section 2.3, three different matching strategies are studied and implemented in the current project. We use different data points to evaluate the performance of each of the strategies. For evaluation, we count the number of matched pairs that are not correct, i.e. mismatches. In table 3.11, summarized information of evaluations of strategies with different data points and configurations are given. In all evaluations, the size of the input is about 200. Based on the table 3.11, the average number of mismatches are 49, 40, and 9 for the *Nearest Neighbor*, *Unweighted*, and *Weighted* matching strategies, respectively. Therefore, the original method is the worst strategy with the highest number of mismatches while it is fast. On the other hand, *Weighted Matching* is the best strategy with the lowest number of mismatches while it is slow. All in all, to have an acceptable correspondence among among matched points, *Weighted Matching* strategy is the better choice.

Table 3.11: This table shows average number of mismatches among points for evaluation of FPPMA-kNN with different matching strategies and configurations over different data points. The size of the input is about 200 for each data set.

	rho	lambda	k	k2	k3	ave time	number of mismatches
Nearest Neighbor						883.13	50
Unweighted Matching	0.9	0.2	5	2	3	964.86	40
Weighted Matching						1480.91	8
Nearest Neighbor						1041.65	48
Unweighted Matching	0.9	0.2	5	3	3	1121.34	40
Weighted Matching						2093.65	10

3.5 Method to Compare - Gaussian Mixture Models

In this section, *Robust Point Set Registration Using Gaussian Mixture Models* (GMM-Reg) [30] is chosen as a method of point set registration to compare against our implementation. In this registration method, the key idea is to represent the input point sets using *Gaussian Mixture Models* (GMMs). Therefore, aligning the two GMMs can be seen as a point set registration. Minimizing the Euclidean distance of the two GMMs estimates the transformation of two point sets, as an example of matching see Figure 3.3.


(a) A sample input of two incomplete fishes.

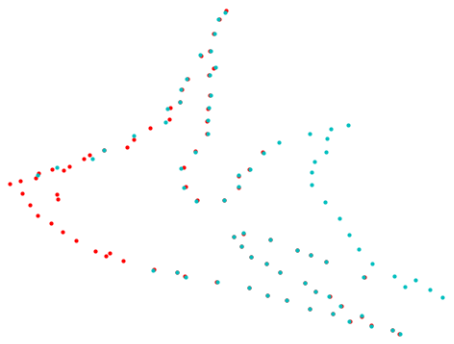

(b) Green points are the transformed points using GMMReg.

Figure 3.3: An experiment on a pair of incomplete fish shapes by GMMReg method.

It is worth to mention that, in [13], to improve the robustness to outliers, noises, and occlusions, Campbell et. al. developed a *support vector-parametrized Gaussian mixture* (SVGM) method, which is an adaptive data representation method of point sets. Here, the point set is shown by a one-class support vector model (SVM) and the output function is approximated by a GMM.

To evaluate and compare the performance of the GMMs, as before we use the C++ implementation of GMMReg [30], and the Mean Squared Error Distance (MSED) as the cost function of the optimization problem. The transformed function in the GMMReg is chosen using a nonrigid transformation, and the parameters in this representative algorithm are set as in the original paper. GMMReg is carried out until it is converged or runs at least 50 iterations. The evaluation information is presented in the following table 3.12.

Now, we have enough information to compare FPPMA and GMMReg with different types of data including *complete matching* and *partial matching*.

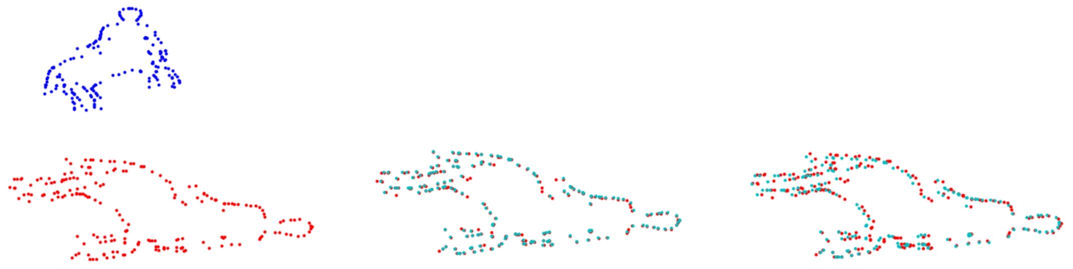
First, for the *complete matching*, based on the provided information in tables 3.1, 3.2, and 3.3, by setting proper parameters using FPPMA the average errors are near 7 and 15 for average distance and average maximum distance, respectively. However, based on the table 3.12, for the same set of data, using GMMReg we can achieve 3.48 and 9.14 for average distance and average maximum distance errors, respectively. Clearly, GMMReg beats FPPMA for the *complete matching* data.

Table 3.12: Experiment using GMM with different types of data.

	results		
	time	ave dist	ave max dist
Complete Matching	5442.82	3.48	9.14
PM-P-DATA	3623.56	11.98	30.54
PM-R-DATA (20%)	4946.22	5.62	15.25

On the other hand, for the *partial matching* including PM-P-DATA and PM-R-DATA the story is different. Based on the tables 3.4, 3.5, 3.6, and 3.12, the average distances/average max distances are 3.1/9.5 and 11.98/30.54 for for FPPMA and GMMReg with PM-P-DATA data, respectively. Also, for PM-R-DATA, the average distances/average max distances are 3.2/8.8 and 5.62/15.25 for FPPMA and GMMReg, respectively. In both cases, the accuracy of FPPMA beats GMMReg.

All in all, we can say GMMReg and FPPMA are better for *complete matching* and *partial matching*, respectively. In Figure 3.4, an example of the result for partial matching is shown.


(a) An example input of PM-R-DATA where point set P and Q are shown using blue (20% of the points are removed) and red points, respectively.

(b) Green points indicate transformed points using FPPMA-kNN. The average distances and max distance are 2.6 and 5.4, respectively.

(c) Green points indicate transformed points using GMMReg. The average distances and max distance are 7.01 and 16.14, respectively.

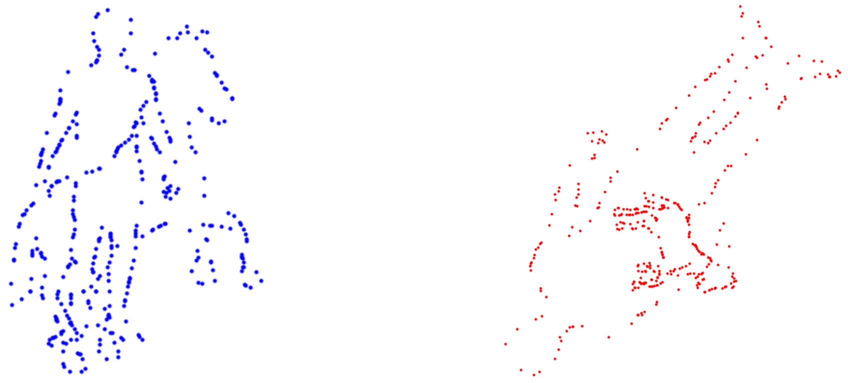
Figure 3.4: A comparison result between FPPMA and GMMReg with PM-R-DATA.

Chapter 4

Groupwise Point Set Registration

In this section, we consider another variant of the point pattern matching problem, that is, Group-wise point set registration.

Let $P' = \{p'_1, p'_2, ..., p'_{n'}\}$ and $P'' = \{p''_1, p''_2, ..., p''_{n''}\}$ be two point sets. Let $P = P' \cup P''$ denote the union of the two point sets. Note that, we don't know about point sets P' and P'', and we only have point set P as a union of the two point sets. However, we know that it is exactly two different transformations that can transform point set P into the target point set. For example, imagine a situation where P is a point set that contains two different objects (each point set P' and P'' represents a different object). Therefore, we need to find two different transformations to have a good match, as an example see Figure 4.1.



(a) Point set P contains data points for two different objects.

(b) Point set Q contains data points with different transformed objects of point set P.

Figure 4.1: This figure shows a point set P, and its transformation by two different sets of parameters.

4.1 Groupwise method using FPPMA

In this project, the FPPMA algorithm is used sequentially to overcome the challenge of multiple point sets registration problem. Here, the idea is to find a transformation for one of the subsets of P, and then go for the rest of the points. However, the issue is that we don't know about P' and P''. Also, due to the presence of the two objects, the neighborhoods of the points change, and this makes the process more difficult. Clearly, if the number of points in one of the subsets (say P') is known then we can find a transformation (say T') for P'. To do that, FPPMA is used by setting ρ close to $\frac{|P'|}{|P|}$. Then, we remove the selected points in P and their corresponding matches in Q. Next, FPPMA is run once more over the remaining points. Note that, if T' is a good transformation, FPPMA has to find a good transformation for the rest of the points. Therefore, for the second round of the algorithm, we should set ρ with a high percentage (say 0.8). Therefore, the idea is to try different values for ρ to get the result. For example, we can do binary search between $\frac{1}{100}$ and $\frac{5}{10}$, and stop the process whenever FPPMA can find a good transformation with a high ρ as a parameter in the second phase.

4.2 Groupwise Experiments

The implementation is tested using some 2D point-sets. To evaluate *Groupwise* method, FPPMA-kNN is used because it is faster. Again, to evaluate the performance of the experiment, we use the Mean Squared Error Distance (MSED) as the cost function of the optimization problem.

Here, like before, all of our experiments were performed on a PC with 16 GB of RAM and an Intel(R)Core(TM) i7-3770 CPU (3.40GHz). Also, C++ is used for implementation.

Summarized information and sample illustration can be found in the following tables and figures.

Table 4.1: Runtime information of Pairwise implementation of FPPMA for different values of ρ (time is represented in milliseconds).

rho	#pairs	time	ave dist	ave max dist
0.5	22627	105918.54	49.4	315.34
0.48	9664	64926.31	47.94	293.9
0.46	4014	54822.51	50.29	312.21
0.44	7308	226825.63	43.84	294.34
0.42	5641	235061.92	50.99	314.89
0.4	23021	1821238.16	45.35	300.23
0.38	25370	2633175.98	45.35	300.23
0.36	17145	2534273.23	45.35	300.23

Based on the Figure 4.2, the required time to do Pairwise FPPMA is increased by moving away from $\rho = 0.5$ for the set of points that are almost in the same size, that is, ρ should be close to 0.5.

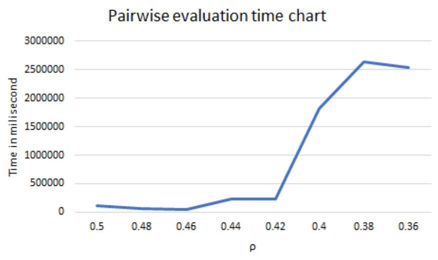
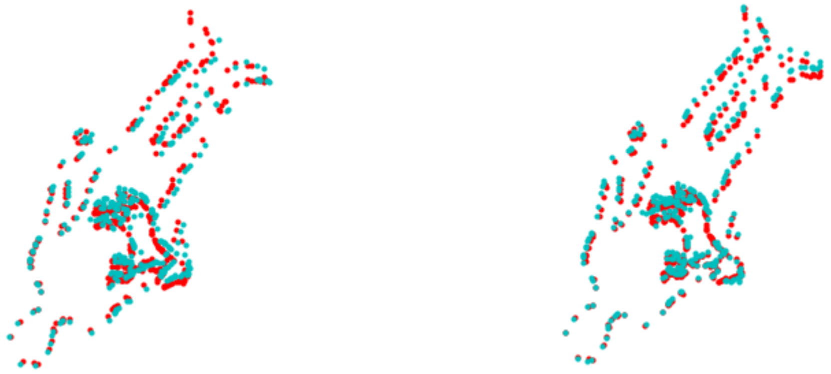


Figure 4.2: Time chart based on the table 4.1. This figure shows the required time for Pairwise FPPMA in milliseconds for different ρs. Time is increased by moving away from $\rho = 0.5$.



(a) Output example for Pairwise FPPMA with $\rho = 0.42$.

(b) Output example for Pairwise FPPMA with $\rho = 0.5$.

Figure 4.3: This figure shows output examples of Pairwise FPPMA for $\rho = 0.42$ and $\rho = 0.50$ with input data of Figure 4.1.

Chapter 5

Conclusion

In this project, we studied, implemented, tested, and evaluated FPPMA which is one of the fastest algorithmic approaches to the point pattern matching problem [43]. The elements of the set are points in R^2, and the pattern set can undertake translation, rotation, and scaling. We consider Euclidean distance as similarity and dissimilarity measures.

In Chapter 2, we give a description of FPPMA [43] and the details of its implementation. We implemented FPPMA with three different neighborhood metrics, that is, FPPMA-kNN, FPPMA-dNN, and FPPMA-gNN for k nearest, Delaunay, and Gabriel neighbors, respectively. Moreover, we extended FPPMA by adding two different strategies of picking matched points, i.e. *Unweighted* and *Weighted* matching strategies.

In Chapter 3, we evaluated different implementations of FPPMA including FPPMA-kNN, FPPMA-dNN, and FPPMA-gNN with three different matching strategies. We then showed the experimental results (including the average number of selected pairs, average running time, average error distances) of these evaluations. All in all, we can say for timing purposes FPPMA-kNN and FPPMA-dNN are better, and for accuracy, FPPMA-gNN is better than the others. For matching strategies, *Weighted Matching* strategy is the better choice while it is a little bit slow. In the end, we had a comparison between FPPMA-kNN and the well-known GMMReg method. We can say GMMReg and FPPMA are better for *complete matching* and *partial matching* (including PM-P-DATA and PM-R-DATA), respectively.

In Chapter 4, we used and evaluated FPPMA-kNN to solve groupwise version of the registration problem. It is observed that FPPMA can be generalized to solve this version of the problem as well.

Many different tasks have been left for the future due to lack of time. For example, all the ideas explored in this project can be easily extended to higher dimensions. As another example, the problem of efficiently localizing the search for a matched point under general transformation is still unresolved. This problem is solved here by a randomized exhaustive search method.

Bibliography

[1] ANN: A library for approximate nearest neighbor searching. `https://www.cs.umd.edu/~mount/ANN/`. Accessed: 2019-09-15.

[2] Oswin Aichholzer, Helmut Alt, and Günter Rote. Matching shapes with a reference point. *International Journal of Computational Geometry & Applications*, 7(04):349–363, 1997.

[3] Dror Aiger and Klara Kedem. Geometric pattern matching for point sets in the plane under similarity transformations. *Information Processing Letters*, 109(16):935–940, 2009.

[4] Helmut Alt, Bernd Behrends, and Johannes Blömer. Approximate matching of polygonal shapes. In *Proceedings of the Seventh Annual Symposium on Computational Geometry*, pages 186–193, 1991.

[5] Helmut Alt and Leonidas J Guibas. Discrete geometric shapes: Matching, interpolation, and approximation. *Handbook of Computational Geometry*, 1:121–153, 1999.

[6] Helmut Alt, Kurt Mehlhorn, Hubert Wagener, and Emo Welzl. Congruence, similarity, and symmetries of geometric objects. *Discrete & Computational Geometry*, 3(1):237–256, 1988.

[7] Marc Benkert, Joachim Gudmundsson, Damian Merrick, and Thomas Wolle. Approximate one-to-one point pattern matching. *Journal of Discrete Algorithms*, 15:1–15, 2012.

[8] Arijit Bishnu, Sandip Das, Subhas C Nandy, and Bhargab B Bhattacharya. Simple algorithms for partial point set pattern matching under rigid motion. *Pattern Recognition*, 39(9):1662–1671, 2006.

[9] John Adrian Bondy, Uppaluri Siva Ramachandra Murty, et al. *Graph theory with applications*, volume 290. Macmillan London, 1976.

[10] Lisa Gottesfeld Brown. A survey of image registration techniques. *ACM Computing Surveys*, 24(4):325–376, 1992.

[11] Sergio Cabello, Panos Giannopoulos, Christian Knauer, and Günter Rote. Matching point sets with respect to the earth mover's distance. In *European Symposium on Algorithms*, pages 520–531, 2005.

[12] Tiberio S Caetano, Terry Caelli, Dale Schuurmans, and Dante Augusto Couto Barone. Graphical models and point pattern matching. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 28(10):1646–1663, 2006.

[13] Dylan Campbell and Lars Petersson. An adaptive data representation for robust point-set registration and merging. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 4292–4300, 2015.

[14] L.Paul Chew, Michael T. Goodrich, Daniel P. Huttenlocher, Klara Kedem, Jon M. Kleinberg, and Dina Kravets. Geometric pattern matching under euclidean motion. *Computational Geometry*, 7(1):113 – 124, 1997.

[15] S. Cohen and L. Guibasm. The earth mover's distance under transformation sets. In *Proceedings of the Seventh IEEE International Conference on Computer Vision*, volume 2, pages 1076–1083, 1999.

[16] Mark De Berg, Marc Van Kreveld, Mark Overmars, and Otfried Schwarzkopf. Computational geometry. In *Computational geometry*, pages 1–17. Springer, 1997.

[17] Hu Ding and Jinhui Xu. Fptas for minimizing the earth mover's distance under rigid transformations and related problems. *Algorithmica*, pages 1–30, 2016.

[18] Boris A Dubrovin, Anatolij Timofeevic Fomenko, and Sergei Petrovich Novikov. *Modern geometry methods and applications: Part II: The geometry and topology of manifolds*, volume 104. Springer Science & Business Media, 2012.

[19] Alon Efrat, Alon Itai, and Matthew J Katz. Geometry helps in bottleneck matching and related problems. *Algorithmica*, 31(1):1–28, 2001.

[20] Paul Finn, Dan Halperin, Lydia Kavraki, Jean-Claude Latombe, Rajeev Motwani, Christian Shelton, and Suresh Venkatasubramanian. Geometric manipulation of flexible ligands. *Applied Computational Geometry Towards Geometric Engineering*, pages 67–78, 1996.

[21] PW Finn, LE Kavraki, JC Latombe, R Motwani, C Shelton, S Venkatasubramanian, and A Yao. Rapid: Randomized pharmacophore identification for drug design. In *Proceedings of the Thirteenth Annual Symposium on Computational Geometry*, 1997.

[22] David Forsyth, Joseph L Mundy, Andrew Zisserman, Chris Coelho, Aaron Heller, and Charlie Rothwell. Invariant descriptors for 3 d object recognition and pose. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 13(10):971–991, 1991.

[23] K Ruben Gabriel and Robert R Sokal. A new statistical approach to geographic variation analysis. *Systematic zoology*, 18(3):259–278, 1969.

[24] David Gans. *Transformations and geometries.* Appleton-Century-Crofts New York, 1969.

[25] Warren F Gardner and Daryl T Lawton. Interactive model-based vehicle tracking. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 18(11):1115–1121, 1996.

[26] Michael T Goodrich, Joseph SB Mitchell, and Mark W Orletsky. Approximate geometric pattern matching under rigid motions. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 21(4):371–379, 1999.

[27] WEL Grimson, RJFA Kikinis, Ferenc A Jolesz, and PM Black. Image-guided surgery. *Scientific American*, 280(6):54–61, 1999.

[28] Liisa Holm and Chris Sander. Mapping the protein universe. *Science*, 273(5275):595, 1996.

[29] Daniel P Huttenlocher, Klara Kedem, and Micha Sharir. The upper envelope of voronoi surfaces and its applications. *Discrete & Computational Geometry*, 9(3):267–291, 1993.

[30] Bing Jian and Baba C Vemuri. Robust point set registration using gaussian mixture models. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 33(8):1633–1645, 2010.

[31] Oliver Klein and Remco C Veltkamp. Approximation algorithms for the earth mover's distance under transformations using reference points. 2005.

[32] Marc Levoy, Kari Pulli, Brian Curless, Szymon Rusinkiewicz, David Koller, Lucas Pereira, Matt Ginzton, Sean Anderson, James Davis, Jeremy Ginsberg, et al. The digital michelangelo project: 3d scanning of large statues. In *Proceedings of the Twenty Seventh Annual Conference on Computer Graphics and Interactive Techniques*, pages 131–144, 2000.

[33] Julian J McAuley, Tibério S Caetano, and Marconi S Barbosa. Graph rigidity, cyclic belief propagation, and point pattern matching. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 30(11):2047–2054, 2008.

[34] David M Mount. Cmsc 754 computational geometry. *Lecture Notes for Spring*, 2007.

[35] David M Mount, Nathan S Netanyahu, and Jacqueline Le Moigne. Efficient algorithms for robust feature matching. *Pattern recognition*, 32(1):17–38, 1999.

[36] David Mumford. Mathematical theories of shape: Do they model perception? In *proceeding of the International Society for Optics and Photonics*, pages 2–10, 1991.

[37] F Murtagh. A new approach to point pattern matching. *Publications of the Astronomical Society of the Pacific*, 104(674):301, 1992.

[38] Raquel Norel, Daniel Fischer, Haim J Wolfson, and Ruth Nussinov. Molecular surface recognition by a computer vision-based technique. *Protein Engineering*, 7(1):39–46, 1994.

[39] James B Orlin. A faster strongly polynomial minimum cost flow algorithm. *Operations Research*, 41(2):338–350, 1993.

[40] Stefano Rebay. Efficient unstructured mesh generation by means of delaunay triangulation and bowyer-watson algorithm. *Journal of Computational Physics*, 106(1):125–138, 1993.

[41] Azriel Rosenfeld. Image analysis and computer vision. *Computer Vision and Image Understanding*, 74(1):36–95, 1999.

[42] Oliver Van Kaick, Hao Zhang, Ghassan Hamarneh, and Daniel Cohen-Or. A survey on shape correspondence. In *Computer Graphics Forum*, volume 30, pages 1681–1707. Wiley Online Library, 2011.

[43] Paul B Van Wamelen, Z Li, and SS Iyengar. A fast expected time algorithm for the 2-d point pattern matching problem. *Pattern Recognition*, 37(8):1699–1711, 2004.

[44] Remco C. Veltkamp and Michiel Hagedoorn. *State of the art in shape matching*, pages 87–119. Springer London, 2001.

[45] Xiaoyun Wang and Xianquan Zhang. Point pattern matching algorithm for planar point sets under euclidean transform. *Journal of Applied Mathematics*, 2012, 2012.

[46] Gerald Weber, Lars Knipping, and Helmut Alt. An application of point pattern matching in astronautics. *Journal of Symbolic Computation*, 17(4):321–340, 1994.

[47] Thomas D Wu, Scott C Schmidler, Trevor Hastie, and Douglas L Brutlag. Modeling and superposition of multiple protein structures using affine transformations: Analysis of the globins. *Patient Assessment Of Constipation Symptoms on Bio*, pages 507–518, 1998.

Appendix A

Point Pattern Matching Implementation

Implementation A.1: Transformation Finder Class

```

class TransformationFinder {
protected:
    // parameters
    int k = 5;
    int k2 = 1, k3 = 2;

    float t = 0.05;
    float lambda = 0.2;

    float rho = 0.90;

    // 0 = the default i.e. no matching,
    // 1 = simple matching, 2 = weighted matching
    int _matching_mode = 0;


    // data
    ANNpointArray p_data;
    int p_size;
    vector<int> p_indices;

    ANNpointArray q_data;
    int q_size;
    vector<int> q_indices;


    // tools
    BaseFinder *pFinder;
    BaseFinder *qFinder;

    Grid *_grid;

    vector<int> G_L1;
    vector<int> G_L2;

    bool is_found = false;
    double _tx;
    double _ty;
    double _rotation;
    double _scale;
    ANNpointArray transformed_data;

    clock_t pStart, pEnd;
    clock_t qStart, qEnd;
    clock_t tStart, tEnd;

    int p_considered_num = 0;

    // set data
    void setData(ANNpointArray &p_set, int p_size, vector<int>
    &p_indices, ANNpointArray &q_set, int q_size, vector<int>
    &q_indices);

    // set parameters
    void setParameters(int k, int k2, int k3, float lambda, float rho);

    // get average distance from min circle (i.e. r / sqrt(size))
    void getAverageDist(ANNpointArray &data, int size, double &res);

    bool find_a_local_match(int idx_p, int idx_q);

    bool is_a_global_match(Transform2 &the_transformation);

    bool is_a_global_match_graph(Transform2 &the_transformation);

    bool is_a_global_match_graph_weighted(Transform2 &the_transformation);

public:
    // constructor definition
    TransformationFinder(ANNpointArray &p_set, int p_size, vector<int>
    &p_indices, ANNpointArray &q_set, int q_size, vector<int>
    &q_indices);

    TransformationFinder(ANNpointArray &p_set, int p_size, vector<int>
    &p_indices, ANNpointArray &q_set, int q_size, vector<int>
    &q_indices, int k, int k2, int k3, float lambda,
                         float rho);

    ~TransformationFinder();

    // initialization
    void init(int _mode);

    void setMatchingMode(int _mode);

    bool to_find_transformation();

    double get_time(char _t);

    void save_transformed_data(string path);

    void print_transformation();

    void display();

    void save_matched_points(string path);

    // to get matched points data to a file
    void save_matched_points_data(string path);

    int get_number_matched();

    void get_unmatched_points();

    void get_unmatched_points(ANNpointArray &e_p_set, int &e_p_size,
    vector<int> &e_p_indices, ANNpointArray &e_q_set, int &e_q_size,
    vector<int> &e_q_indices);

    void save_unmatched_points(string path);

    int get_number_unmatched();

    void get_mismatched_points(vector<int> &shuffle_vector);

    void save_mismatched_points(vector<int> &shuffle_vector, string path);

    int get_number_mismatches(vector<int> &shuffle_vector);

    void save_info(string path, vector<int> &shuffle_vector);

    static double get_hausdorff_dist(ANNpointArray &t_data, int t_size,
    ANNpointArray &q_data, int q_size);

    double get_hausdorff_distance();

    static void make_ini(string path, string base_path);

    int get_p_considered_num() {
        return p_considered_num;
    }
};

//—————————————————————— Implementation Part ——————————————————
```

Appendix B

Proximity Related Code

Implementation B.1: Base Proximity Finder Class

```

// ======================= BNFinder ============================
class BaseFinder {
protected:

    double eps = 0;

    int nPts;                       // actual number of data points
    ANNpointArray dataPts;                   // data points

    vector <vector<int>> nn_list;

public:
    BaseFinder(ANNpointArray &points, int p_size, double e = 0) {
        this->eps = e;

        // dataPts = annAllocPts(maxPts, dim);
        // allocate data points
        this->dataPts = points;
        this->nPts = p_size;
    }

    ~BaseFinder() {}

    virtual void makeNNList(int k) = 0;

    ANNpointArray getDataPts() {
        return dataPts;
    }

    vector <vector<int>> get_nn_list() {
        return nn_list;
    }
};
```

Implementation B.2: Nearest Neighbor Finder Class

```

// ======================== NNFinder ==============================
class NNFinder: public BaseFinder {

    ANNkd_tree *kdTree;                     // search structure

    ANNidxArray nnIdx;                         // allocate near neigh indices
    ANNdistArray dists;                         // allocate near neighbor dists
public:
    NNFinder(ANNpointArray &points, int p_size, double e = 0)
    :BaseFinder(points, p_size, e) {
        // build search structure
        this->kdTree = new ANNkd_tree(
                dataPts,                      // the data points
                nPts,                          // number of points
                2);                          // dimension of space
    }

    ~NNFinder() {
        delete kdTree;
    }

    void makeNNList(int k) {
        if(k >= nPts)
            k = nPts - 1;

        // allocate near neigh indices
        nnIdx = new ANNidx[k + 1];
        // allocate near neighbor dists
        dists = new ANNdist[k + 1];

        for (int i = 0; i < nPts; i++) {
            kdTree->annkSearch(     // search
                    dataPts[i],     // query point
                    k + 1,          // number of near neighbors
                    nnIdx,          // nearest neighbors (returned)
                    dists,          // distance (returned)
                    this->eps);     // error bound`

            vector<int> temp_list;
            // start from 1 because index 0 is the query point itself
            for (int j = 1; j <= k; j++)
                temp_list.push_back(nnIdx[j]);
            nn_list.push_back(temp_list);
        }

        // clean things up
        delete[] nnIdx;
        delete[] dists;
    }

    vector<int> getNN(ANNpoint _query, int k) {
        if(k >= nPts)
            k = nPts - 1;

        nnIdx = new ANNidx[k + 1];       // allocate near neigh indices
        dists = new ANNdist[k + 1];      // allocate near neighbor dists

        kdTree->annkSearch(      // search
                _query,          // query point
                k + 1,           // number of near neighbors
                nnIdx,           // nearest neighbors (returned)
                dists,           // distance (returned)
                this->eps);      // error bound`

        vector<int> temp_list;
        // start from 1 because index 0 is the query point itself
        for (int j = 1; j <= k; j++)
            temp_list.push_back(nnIdx[j]);

        delete[] nnIdx;        // clean things up
        delete[] dists;

        return temp_list;
    }

    ANNpointArray getDataPts() {
        return dataPts;
    }

    vector <vector<int>> get_nn_list() {
        return nn_list;
    }
};
```

Implementation B.3: Delaunay Finder Class

```

// ======================= DNFinder ============================
class DNFinder: public BaseFinder
{
protected:
    vector<DPoint<float>> d_points;
    Delaunay<float> _triangulation;
public:
    DNFinder(ANNpointArray &points, int p_size, double e = 0)
    : BaseFinder(points, p_size, e) {
        for(int i = 0; i < p_size; i++)
            this->d_points.push_back({i, points[i][0], points[i][1]});
    }

    ~DNFinder() {}

    void makeNNList(int k) {
        if(k >= this->nPts)
            k = this->nPts - 1;

        this->_triangulation = triangulate(this->d_points);

        std::vector<std::vector<int>> vec(this->nPts, std::vector<int >(0));

        for (auto const& e : this->_triangulation.edges) {
            if(!(find(vec[e.p0.idx].begin(), vec[e.p0.idx].end()
            , e.p1.idx) != vec[e.p0.idx].end()))
```

```
                vec[e.p0.idx].push_back(e.p1.idx);

            if(!(find(vec[e.p1.idx].begin(), vec[e.p1.idx].end()
            , e.p0.idx) != vec[e.p1.idx].end()))
                vec[e.p1.idx].push_back(e.p0.idx);
        }

        for(int i = 0; i < this->nPts; i++) {
            nn_list.push_back(vec[i]);
        }
    }
};
```

Implementation B.4: Gabriel Finder Class

```

// ======================= GNFinder ============================
class GNFinder: public DNFinder
{
public:
    GNFinder(ANNpointArray &points, int p_size, double e = 0)
    : DNFinder(points, p_size, e) {}

    void makeNNList(int k) {
        if(k >= this->nPts)
            k = this->nPts - 1;

        this->_triangulation = triangulate(this->d_points);

        std::vector<std::vector<int>> vec(this->nPts, std::vector<int>(0));

        NNFinder *aFinder = new NNFinder(this->dataPts, this->nPts, 0);

        for (auto const& e : this->_triangulation.edges) {
            // query point to find the nearest neighbor in Q within 2t
            ANNpoint pt = annAllocPt(2);
            pt[0] = e.mid.x;
            pt[1] = e.mid.y;
            vector<int> t_list = aFinder->getNN(pt, 1);
            int id = t_list[0];
            ANNdist dist = annDist(2, pt, this->dataPts[id]);
            double _real_dist = ANN_ROOT(dist);
            if(!(_real_dist > e.rad)) continue;

            if(!(find(vec[e.p0.idx].begin(), vec[e.p0.idx].end()
            , e.p1.idx) != vec[e.p0.idx].end()))
                vec[e.p0.idx].push_back(e.p1.idx);

            if(!(find(vec[e.p1.idx].begin(), vec[e.p1.idx].end()
            , e.p0.idx) != vec[e.p1.idx].end()))
                vec[e.p1.idx].push_back(e.p0.idx);
        }

        for(int i = 0; i < this->nPts; i++) {
            nn_list.push_back(vec[i]);
        }
    }
};
```

Appendix C

Tools

Implementation C.1: Grid Class

```

class Grid {
    double side_len = 0;
    double min_x = 0, min_y =0, max_x = 0, max_y = 0;
    double l_x = 0, l_y = 0;

    int n_rows = 0, n_cols = 0;

    ANNpointArray points;
    int nPts;   // actual number of data points
    int size = 0;
    vector<int>* _grid;
private:
    void init(ANNpointArray pts, int size, double side_len) {
        this->points = pts;
        this->nPts = size;
        this->side_len = side_len;
        double min_x = points[0][0], min_y = points[0][1]
        , max_x = points[0][0], max_y = points[0][1];

        for(int i = 0; i < size; i++) {
            if(points[i][0] < min_x)
                min_x = points[i][0];

            if(points[i][0] > max_x)
                max_x = points[i][0];

            if(points[i][1] < min_y)
                min_y = points[i][1];

            if(points[i][1] > max_y)
                max_y = points[i][1];
        }

        double l_x = fabs(max_x - min_x);
        double l_y = fabs(max_y - min_y);

        this->min_x = min_x;
        this->max_x = max_x;
        this->min_y = min_y;
        this->max_y = max_y;

        this->l_x = l_x;
        this->l_y = l_y;

        this->n_rows = this->l_y / side_len + 1;
        this->n_cols = this->l_x / side_len + 1;

        _grid = new vector<int>[this->n_rows * this->n_cols];
    }

    // to find points inside each of cells of the grid
    void load_data() {
        for(int i = 0; i < this->nPts; i++) {
            int row = fabs(this->points[i][1] - this->min_y) / side_len;
            int col = fabs(this->points[i][0] - this->min_x) / side_len;

            (this->getCell(row, col))->push_back(i);
        }
    }

public:
    Grid(ANNpointArray points, int size, double side_len) {
        // initialization
        this->init(points, size, side_len);

        // load data
        this->load_data();
    }

    ~Grid() {
        delete[] this->_grid;
    }



    vector<int>* getCell(size_t row, size_t col)
    {
        //cout << "grid getCell() is called!" << endl;
        return (this->_grid + (row * this->n_cols + col));
    }

    vector<int> search_for_a_match(ANNpoint pt, double t0) {
        // cout << "grid search() is called!" << endl;
        int row = fabs(pt[1] - this->min_y) / side_len;
        int col = fabs(pt[0] - this->min_x) / side_len;

        // cout << "row: " << row << " and col: " << col << endl;

        vector<int> idxs;
        for(int i = -1; i <= 1; i++)
            for(int j = -1; j <= 1; j++) {
                int r = row + i;
                int c = col + j;
                if(r < 0 || c < 0 || r >= this->n_rows
                || c >= this->n_cols) continue;

                vector<int>* v_ptr = this->getCell(r, c);
                for(int l = 0; l < v_ptr->size(); l++){
                    ANNpoint p_l = this->points[ (*v_ptr)[l] ];
                    ANNdist dist = annDist(2, pt, p_l);
                    dist = ANN_ROOT(dist);
                    // cout << "dist: " << dist << endl;
                    // cout << "dist: " << dist << " and t0: " << t0 << endl;
                    if(dist < t0) {
                        // cout << "-- " << dist << " < " << t0 << endl;
                        idxs.push_back((*v_ptr)[l]);
                    }
                }
            }

        return idxs;
    }


    void print() {
        cout << "Grid: " << endl
            << "lx = " << this-> l_x << endl
            << "ly = " << this-> l_y << endl
            << "side len = " << this->side_len << endl
            << "nr = " << this->n_rows << endl
            << "nc = " << this->n_cols << endl;
    }
};
```