Annex: Fast Cluster Exploration of BCI data using Riemannian Geometry

Gregersens, Adam Rene
Jensen, Daniel Agerskov Heidemann

June 9, 2014
Chapter 1

Quick Shift: Density Estimation

As described in the main paper, Quick Shift computes kernel density estimates for each point to produce a density landscape. As observed during our testing, two or more points can end up having the same density estimate. This is illustrated in Figure 1.1, where multiple symmetrically distributed points have equal densities.

![Figure 1.1: Quick Shift Density Issue](image)

*Figure 1.1: Quick Shift Density Issue*

Lower density points \( P_1, P_3, P_4, P_6 \) connects properly to higher density sub-root points \( P_2, P_5 \). However, both sub-roots have equal densities leaving Quick Shift indeterminate on how to connect these. As a result one sub-root branch is excluded from the tree.

The mode seeking of Quick Shift works properly on points with equal densities \( P_1, P_3, P_4, P_6 \), as long as the full set of points contains some other point at higher density. However, a corner case arises if two or more points at the highest density, a sub-root, have equal density values \( P_2, P_5 \). While the scenario
of sub-roots having equal density values is possible, it is highly unlikely, as it
would require points to be distributed in a very unique way in the Riemannian
space. Nevertheless, if no sub-root has the bigger density, Quick Shift cannot
connect either to a higher density node. As a result, potentially large sub-trees
will not be joined into the final tree. Intuitively, if a point cannot be connected
to any other point, it has the highest density; hence it becomes the root. Our
solution to sub-roots with equal densities is simply to add one root as a child
of the other root. The final tree will then connect all points. The problem of
points being unconnected can only occur at the highest density level, among
sub-roots. We suspect that adding one root as a child of the other root will
have no negative or incorrect implications for the clustering of data patterns.
Chapter 2

KD-trees

This section will explain KD-Trees, as defined in [3], and explain why KD-Trees cannot be used in a Riemannian space. KD-Trees can work for k-dimensions, but the concept is explained on the intuition of two dimensions.

KD-Trees only work in a vector space, with each point having a coordinate. Figure 2.1 shows the tree structure of a KD-Tree as well as its subdividing planes. Intermediate nodes have two children, a splitting value and depth of the node in the tree. The depth is used to determine whether a node is split on the x or y-coordinate. A node is split on the x-coordinate if the depth is even. Likewise, an uneven depth will split a node on the y-coordinate. However, other ways of splitting also exist. The left child node contains all points having a smaller than or equal coordinate to the splitting value. The right child node contains all points at larger coordinates.

A KD-Tree can be constructed in $O(N \log N)$ time and store $O(N)$ elements. KD-Trees are therefore an efficient data structure to store elements. The Riemannian space is a non-vector space, as point coordinates are unknown. We therefore cannot use KD-Trees which requires a vector-space; hence, the reason for resorting to Metric Trees which does not rely on having a coordinate system.
Chapter 3

Closest Pair of Points

During the Agglomeration Phase of the Anchor Tree construction, an additional option based on the Closest Pair of Points algorithm, was considered for finding anchor pairs with the smallest combined radius. As time constraints did not allow us to implement and test this method, it will only be described as a theoretical addition to the study.

As the pair selection method described in the main paper is already fast, this method will not reduce the tree complexity. It can however improve the running time of the pair selection step. That being said, the motivation to use the Closest Pair of Points method is to avoid the need to check all anchors against all other anchors. It is important to note that the method must account for the anchor radius when computing distances. This is done by computing the distance as the value of $T^d$, described in the main paper as the total metric space of two anchors; that is $T^d = A^r + \text{Dist}(A^p, B^p) + B^r$. To ease the understanding, the distance between two anchors will from now on implicitly be equal to $T^d$.

Closest Pair of Points finds the minimum distance between two points in $O(N \log N)$. The algorithm is presented in [4, 2] for the Euclidean space, to which we will be extending it to the Riemannian space. The closest pair problem for multiple dimensions in the Euclidean space was solved in [1]. To the best of our knowledge, an algorithm for the Closest Pair of Points problem has not been developed for the Riemannian space. In this section, anchors will be referred to as points, easing the transition from the Euclidean to the Riemannian space.
3.1 Closest Pair: Euclidean Space

Based on [4, 2], let $P$ be a set of points sorted according to any coordinate axes in $O(N \log N)$; e.g., on $x$-coordinates.

![Figure 3.1: Closest Pair Partitions](image1)

Points $P$ are split into partitions. The orange line marks the first splitting. The red lines split the two partitions into sub-partitions, each containing two or three points.

Using a divide and conquer approach, points are split into left and right partitions by the median until the base case is satisfied. As seen in Figure 3.1, the base case is satisfied when a partition contains three points or less.

![Figure 3.2: Closest Pair Between Partitions](image2)

Vector distance $d_{left}$ and $d_{right}$ marks the minimum distance in both partitions. The smallest of these minimum partition distances, in this case $d_{left}$, is used to limit investigated points while searching for smaller distance between partitions. The points with an $x$-coordinate less than $d_{left}$ away from the median are considered. In this case, two of the considered points form the closest pair, with a distance of $d_{min}$.

6
Figure 3.2 illustrates the computed closest vector distances in the left partition as \( d_{\text{left}} \) and the right partition as \( d_{\text{right}} \). From the smallest distance in each partition, the minimum of these distances is used to limit investigated points while searching for a smaller distance between partitions. In Figure 3.2, \( d_{\text{left}} \) is the minimum vector distance in the two partitions. When checking distance across partitions, only points closer than \( d_{\text{left}} \) from the median on the x-axis, has to be considered, as no other points can form a closest pair. This can potentially yield an even smaller distance, \( d_{\text{min}} \) in Figure 3.2. Let a candidate be defined as a point which may form a closest pair with another candidate. The performance for finding the closest pair in two partitions can be improved further. As explained in the following, each candidate only needs to consider the next five points ahead.

![Diagram showing closest pair and improved performance](image)

**Figure 3.3: Closest Pair: Improved performance**

The four points on the orange line all share the same x-coordinate. The red point needs three measurements to the right partition and two measurements to the left partition. The cross nodes can be omitted since their distance will always be larger than \( d \), yielding a total of five measurements.

Figure 3.3 shows the maximum number of relevant points to measure distances against. By sorting candidates on the y-coordinate, candidates above the current point (red point) do not need to be considered. The red point measures against all points in the right partition, except for the point with the red cross; summing to three calculations. For the left partition, the red point only needs two measurements, because the distance to the point with the red cross will always be further than \( d \) away. Using the Pythagorean Theorem, it becomes clear why the cross points can be omitted. As the length of the legs is \( d \), the hypotenuse is given as

\[
\text{hypotenuse} = \sqrt{d^2 + d^2} > d
\]

thereby showing that the distance from the red point, to any of the crossed points, is always greater than \( d \).
3.1.1 Euclidian Algorithm

Based on the described theory, Algorithm 1 recursively finds the minimum distance from a list of points, by dividing them into sub-partitions. The minimum distance found is used to determine a possible closest pair of candidates.

Algorithm 1

Closest Pair in a Euclidean Space

1: \textbf{procedure} ClosestPair\((P)\)
2: \quad \text{//Determine sub-partitions}
3: \quad \text{if} \ \text{Len}(P) \leq 3 \ \text{then}
4: \quad \quad d_{\text{min}} \leftarrow \text{BruteForceMinDist}(P)
5: \quad \text{else}
6: \quad \quad \text{median} \leftarrow \lceil P \rceil_2
7: \quad \quad d_{\text{left}} \leftarrow \text{ClosestPair}(P[0, \ldots, \text{median}])
8: \quad \quad d_{\text{right}} \leftarrow \text{ClosestPair}(P[\text{median}, \ldots, n])
9: \quad \quad d_{\text{min}} \leftarrow \text{Min}(d_{\text{left}}, d_{\text{right}})
10: \quad \text{end if}
11: \quad \text{//Let C be a new list}
12: \quad C \leftarrow \emptyset
13: \quad \text{//Find candidates}
14: \quad \text{for each} \ p \ \text{in} \ P \ \text{do}
15: \quad \quad \text{if} \ p_x \geq P[\text{median}]_x - d_{\text{min}} \ \text{and} \ p_x \leq P[\text{median}]_x + d_{\text{min}} \ \text{then}
16: \quad \quad \quad C.\text{Add}(p)
17: \quad \quad \text{end if}
18: \quad \text{end for}
19: \quad \text{//Sort candidates by y-coordinate}
20: \quad C.\text{SortDescendingY()}
21: \quad \text{//Compute possible closest pair}
22: \quad \text{for} \ i \leftarrow 0 \ \text{to} \ \text{Len}(C) \ \text{do}
23: \quad \quad \text{for} \ j \leftarrow i+1 \ \text{to} \ \text{Len}(C) \ \text{do}
24: \quad \quad \quad \text{if} \ j - i > 5 \ \text{then}
25: \quad \quad \quad \quad \text{Break}
26: \quad \quad \quad \text{end if}
27: \quad \quad \quad \text{if} \ \text{Dist}(C[i], C[j]) < d_{\text{min}} \ \text{then}
28: \quad \quad \quad \quad d_{\text{min}} \leftarrow \text{Dist}(C[i], C[j])
29: \quad \quad \quad \text{end if}
30: \quad \quad \text{end for}
31: \quad \text{end for}
32: \quad \return \ d_{\text{min}}
33: \text{\end procedure}

From [2], the complexity of Algorithm 1 is given by the recursion

\[
T(N) = \begin{cases} 
O(1) & \text{if } N \leq 3 \\
2T\left(\frac{N}{2}\right) + O(N) & \text{if } N > 3
\end{cases}
\]

with \( N \) denoting number of points and \( T(N) \) the complexity of each recursive step. As the algorithm requires pre-sorted points, performed in \( O(N \log N) \), the total complexity becomes

\[
T'(N) = T(N) + O(N \log N) = O(N \log N) + O(N \log N) = O(N \log N)
\]
3.2 Closest Pair: Metric Space

As the Riemannian space is a non-vector space, positions are unknown. The situation can be seen in Figure 3.4, where the distance between two points represent possible locations on a circle.

![Figure 3.4: Possible point locations](image)

*Figure 3.4: Possible point locations*

With unknown coordinates, only distance is known. The blue point can therefore be at any position on the circle.

To overcome the challenge of not knowing positions, a reference point is used to which all other points are measured against. The points can then be sorted by distance to the reference point. As seen in Figure 3.5, the distance to the median point, c, serves as a radius (median circle) for dividing the points into partitions.

![Figure 3.5: Median Circle](image)

*Figure 3.5: Median Circle*

Points sorted by distance to reference, r. The median is shown as the orange circle dividing blue and green points into separate partitions.

Partitions not satisfying the base case (three points or less) are further divided into sub-partitions. In Figure 3.5, the blue partition satisfies the base case. The green partition however still needs splitting. As in the Euclidian space, minimum distances are computed for each partition with $d_{\text{min}}$ being set as the smallest of these.
If the closest pair consists of a point from each partition, the candidates should be within $d_{\min}$ away from the median circle. Shown in Figure 3.6, a candidate is within $d_{\min}$ from the median circle (orange), if it is between the inner and outer circles (red). With the reference point $r$ in center, let $m_{\text{len}}$ be the radius to the median circle. With respect to $r$, the inner circle has a radius of $m_{\text{len}} - d_{\min}$, and the outer circle a radius of $m_{\text{len}} + d_{\min}$. We will refer to the space between the inner and outer circles as the candidate space.

![Figure 3.6: Inner and Outer Circles](image)

The inner and outer red circles are $d_{\min}$ away from the orange median circle, defined as the candidate space. The red points are candidates because they are within the candidate space.

When checking candidates in the Euclidean space, it is possible to only check five points ahead due to the sorting on $y$-coordinates. However, in the non-vector Riemannian space we cannot sort by coordinates. It is therefore necessary to check the distances between all candidates in the candidate space. As candidates are sorted by distance to the reference point, calculations in the candidate space is expressed as an arithmetic series. In other words, with $k$ candidates the number of calculations is given by

$$
\sum_{i=1}^{k-1} i = 1 + 2 + \cdots + k = \theta((k-1)^2) \tag{3.3}
$$

In the worst case situation, all $N$ points are within the candidate space. The time complexity then becomes

$$
T(N) = \begin{cases} 
O(1) & \text{if } N \leq 3 \\
T(\text{median}(N)) + T(N - \text{median}(N)) + \theta((N-1)^2) & \text{if } N > 3 
\end{cases} \tag{3.4}
$$

By the median, the problem of $N$ points is split into the sub-problems $T(\text{median}(N))$ and $T(N - \text{median}(N))$. The combining step of going through all candidates is $\theta((N-1)^2)$. This complexity is not better than the complexity of the pair selection described in the main paper. It is however important to note that $O(N^2)$ is the absolute worst case. As the number of dimensions increase, so does the probability of points being far away from each other. As a result, this algorithm would likely become better on increased dimensions, as more points would be excluded from the candidate space.
3.2.1 Riemannian Algorithm

Finding a closest pair can be applied to multiple dimensions, but for simplicity, this section will explain it in two dimensions. Much alike the Euclidian version, Algorithm 2 computes a closest pair in the Riemannian space.

Algorithm 2 Closest Pair in Riemannian Space

```
1: procedure ClosestPairRie(P)
2: \Determine sub-partitions
3: if Len(P) \leq 3 then
4: \{d_{min}, P_1, P_2\} \gets \text{BruteForceMinDist}(P)
5: return \{d_{min}, P_1, P_2\}
6: else
7: median \gets \lceil \frac{|P|}{2} \rceil
8: \{d_l, L_1, L_2\} \gets \text{ClosestPairRie}(P[0, \ldots, median])
9: \{d_r, R_1, R_2\} \gets \text{ClosestPairRie}(P[median, \ldots, n])
10: if d_l \leq d_r then
11: d_{min} \gets d_l
12: P_1 \gets L_1
13: P_2 \gets L_2
14: else
15: d_{min} \gets d_r
16: P_1 \gets R_1
17: P_2 \gets R_2
18: end if
19: end if
20: end procedure
```

To set the bounds for the candidate space, the inner and outer circle radiuses are computed using the MakeCircle method. Points are then checked against these radiuses to determine if they are candidates for a closest pair. To make this algorithm work in multiple dimensions, one should only change the implementation of the MakeCircle method. Note that because points are pre-sorted, a candidate $C_i$ only needs to check other candidates from $i + 1$.
Bibliography


