Fast Cluster Exploration of BCI data using Riemannian Geometry

Gregersens, Adam Rene Jensen, Daniel Agerskov Heidemann

June 10, 2014

Abstract

Classifying Brain Computer-Interface (BCI) data can require identification and labeling of relevant signal characteristics. To aid this process, the mode-seeking algorithm Quick Shift creates a hierarchical structure of clusters from covariance matrices of BCI data; enabling data exploration at different levels of granularity. For accurate clustering, covariance matrices are measured in their Riemannian space. To approximate and speed up calculations of Quick Shift, Metric Trees are used in a Dual Tree approach. Presenting new construction methods in the Metric Tree's Agglomeration Phase, a high performance exploration tool was developed, testing these methods on our newly recorded data set. Results showed that the presented methods greatly reduced computation time, mostly on lower channel dimensions. Quantitative clustering capabilities, using our tool, show that relevant signal patterns can be found from clustered BCI data.

1 Introduction

When analyzing BCI data, popular techniques such as the Common Spatial Pattern (CSP) [5] can be used to derive filters for better classification in multiclass problems [4]. Before a classification problem can be formally stated, it is often necessarily to explore and clean the data. As a pre-step, data exploration is the basis to which data can be labeled and classified upon; such as the case of exploring and identifying eye blinks. Labeling can however be can be expensive and potentially require a great deal of training samples. Aiding this, data exploration can be used as a sanity check to determine whether a more expensive labeling experiment is worth investigating.

With the motivation of exploring unlabeled data, this paper will be presenting a tool, *BCI Explorer*, able to cluster and visualize brain signals in a sensible way. For the exploration of data on different levels of abstraction, a hierarchical approach will be taken. We achieve a hierarchical clustering by using the Quick Shift algorithm to create a tree structure of clusters from the covariance matrices of windowed BCI data. To the best of our knowledge, no such tool exists.

Quick Shift relies on distance measurements between covariance matrices, for which this study will use Riemannian geometry. This choice was inspired by the work of [4], presenting superior classification accuracy of BCI covariance matrices using Riemannian geometry, compared to Euclidean geometry. In turn, Riemannian geometry will yield more accurate clustering. However, the switch to Riemannian geometry comes with a cost. Compared to distance computations in an Euclidean space, measuring distances in a Riemannian space is computationally expensive because it induces eigenvalue decompositions. This is made worse by the running time of Quick Shift requiring N^2 measurements, making it impossible to explore large datasets within a reasonable time frame. To alleviate this problem, Metric Trees will be used in a Dual Tree approach to approximate distances, reducing the total number of computations required. The Metric Trees will be constructed using the *Anchors Hierarchy* method [14] which has been well described for the Euclidean space. Since working in a Riemannian space, this paper will propose new construction methods with respect to the limitations of a non-vector space.

Lastly, the paper will be presenting a new dataset recorded from 70 subjects using the Emotiv Epoc headset. The proposed tool and its algorithms will be tested against this dataset to verify the performance and illustrate its usage.

2 Background

This section introduces terms and theories in a relevant context as a basis for the rest of the paper. It will be covering Covariance Matrices from BCI Data, Quick Shift, and Riemannian Geometry.

2.1 Covariance Matrices from BCI Data

BCI data contains the recordings of brain activity through a number of channels over time. Using a Electroencephalography (EEG) device to record data, a channel can be defined as the electrical potential between a recording electrode and a reference electrode, on the scalp. Described by [15], EEG electrodes can be placed according to the 10-10 system shown in Figure 1. The red circles marks

the electrodes used by the Emotiv Epoc headset [1], used during the dataset recordings described by Section 5.

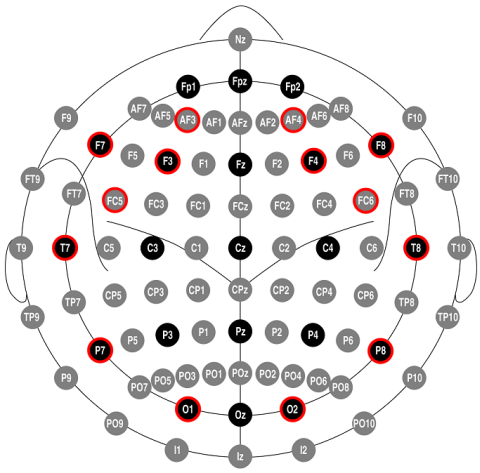

Figure 1: 10-10 System
Spatial channel layout of the 10-10 electrode system as shown in [15]. Red circles correspond to the electrodes used by Emotiv Epoc.

The analysis of BCI data, is often segmented into temporal windows defined as *epochs*. An epoch of length T will be defined in the following matrix form $X_e = [x(t), x(t+1), ..., x(t+T)]$, where e is the epoch index and $x(t)$ a column vector containing one sample from each channel at time point t.

A spatial covariance matrix can be used to describe the relationship between channels in an epoch. For a given epoch, a spatial covariance matrix is defined as

$$\Sigma = E\{(x(t) - E\{x(t)\})(x(t) - E\{x(t)\})^\top\} \quad (1)$$

where E denotes expected value over time and $^\top$ denotes transpose.

By assuming that the samples of each channel have been time-centered around zero, the expected value over time $E\{x(t)\}$, can be eliminated from Equation 1. Hence, one can define the Sample Covariance Matrix (SCM) of an epoch as

$$P_e = \frac{1}{T-1} X_e X_e^\top \quad (2)$$

The SCM is an unbiased estimate of the covariance matrix assuming that the sample size, T, is much larger than the number of channels [17]. A SCM reflects the current state-of-mind interpreted from a set of channels, within a time-window. SCMs will therefore be used as the basis feature values for distance calculations in Section 2.3.

2.2 Quick Shift

Quick Shift [18] is a mode seeking algorithm which can be used for the purpose of hierarchical clustering. The algorithm uses the kernel density estimates of each data point to create a density landscape. As illustrated in Figure 2, Quick Shift uses this landscape to join all points in a space into a single tree structure by connecting each point to its nearest neighbor at higher density. As a result, the root of the tree will always have the highest density. Clusters can afterwards be retrieved by breaking off branches having greater distance than a threshold τ. Adjusting τ will therefore increase/decrease the number of clusters, hence making the clustering hierarchical. A decrease in τ yields more clusters until each cluster only contains one point. Likewise, an increase in τ will yield less clusters until eventually one cluster contains all points.

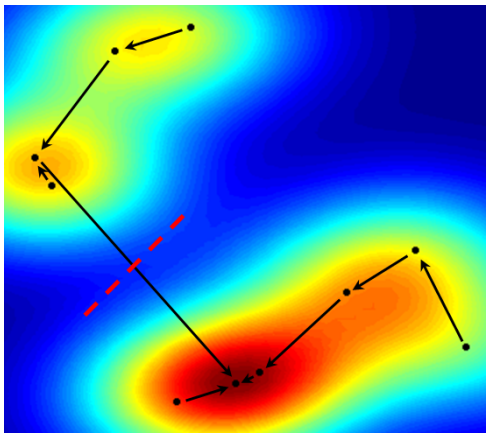

Figure 2: Quick Shift
Tree structure created by connecting each point (black dot) to its nearest neighbor at higher density. Clusters can be retrieved by breaking of large connections, exemplified by the red line [18].

The kernel density estimate, p, for a given point,

i, is calculated as the weighted sum of kernels, assuming uniform weighting of the kernels, defined as

$$p(i) = \frac{1}{N} \sum_{j=1}^{N} K(D(i,j)) \tag{3}$$

with N being number of points and K the kernel function depending on the distance between points i and j. We will in the following use a normal distribution as our kernel, but the theory presented in this paper does not depend on this choice.

Compared to other mode seeking algorithms such as Medoid Shift [16] and Mean Shift [6, 9, 7], Quick Shift [18] is simpler to implement and faster to compute. Arising from the density estimate in Equation 3, computed for each data point, Quick Shift has a complexity of $O(dN^2)$ with d being the dimensionality of the data such as the number of channels. The constant d relates to the complexity of the distance calculation $D(i,j)$, in the evaluation of a kernel. In a Euclidian space with few dimensions, this constant might be computationally insignificant. However, since each data point is a SCM for a potentially large amount of channels, this computation becomes expensive, because the accurate distance between two SCMs must be computed using Riemannian geometry (Section 2.3). Making matters worse, multiplying a significant d with N^2 density estimates, yields an impossible problem when exploring dataset consisting of many epochs. A solution yielding a better complexity will be described later in Section 3, with the use of Metric Trees.

2.3 Riemannian Geometry

In Riemannian geometry, differentiable geometry is used to study Riemannian manifolds. The minimum length curve, the Riemannian distance, between two points on such a manifold is defined as the Riemannian *geodesic*. The motivation for measuring Riemannian distances as opposed to Euclidean is that we study SCMs of epoched channel data. Since a SCM has the property that it is symmetric positive-definite (SPD), it lies on a Riemannian cone-shaped manifold [13]. This manifold lies in a Euclidian subspace [8], containing the set of all SPD matrices $\mathcal{P}(n)$, with n denoting the number of channels.

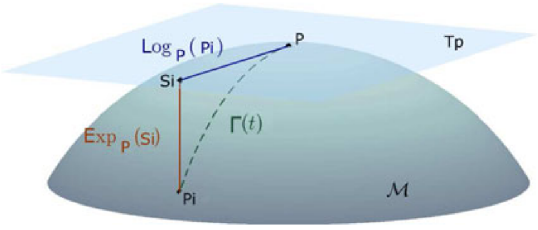

Figure 3: Riemannian manifold *Riemannian SPD manifold [4], $\mathcal{M}$, with $\boldsymbol{S}_i$ as tangent vector to $\boldsymbol{P}$. $\boldsymbol{\Gamma}(t)$ denotes the Riemannian geodesic between $\boldsymbol{P}$ and $\boldsymbol{P}_i$.*

Figure 3 illustrates a SPD manifold, $\mathcal{M}$, and a Euclidean tangent space, $T_{\mathbf{P}}$, to the point $\boldsymbol{P} \in \mathcal{P}(n)$. This tangent space is also a vector space consisting of all $n \times n$ symmetric matrices. The Riemannian geodesic, $\boldsymbol{\Gamma}(t)$, between $\boldsymbol{P}$ and $\boldsymbol{P}_i$ follows the curvature of the manifold, $\mathcal{M}$. Translating points between the space of the manifold and the tangent space, can be done using the *exponential* and *logarithmic mapping*.

Using the *exponential mapping*, a tangent vector $\boldsymbol{S}_i$ at $\boldsymbol{P}$ can be mapped to a point on the manifold, $\boldsymbol{P}_i$, reached at time 1 for the geodesic $\boldsymbol{\Gamma}(t)$ [4, 10]. The exponential map it given by

$$\mathrm{Exp}_{\boldsymbol{P}}(\boldsymbol{S}_i) = \boldsymbol{P}_i = \boldsymbol{P}^{\frac{1}{2}} \exp(\boldsymbol{P}^{-\frac{1}{2}} \boldsymbol{S}_i \boldsymbol{P}^{-\frac{1}{2}}) \boldsymbol{P}^{\frac{1}{2}} \tag{4}$$

The inverse of the exponential mapping, the *logarithmic mapping*, is given by

$$\mathrm{Log}_{\boldsymbol{P}}(\boldsymbol{P}_i) = \boldsymbol{S}_i = \boldsymbol{P}^{\frac{1}{2}} \exp(\boldsymbol{P}^{-\frac{1}{2}} \boldsymbol{P}_i \boldsymbol{P}^{-\frac{1}{2}}) \boldsymbol{P}^{\frac{1}{2}} \tag{5}$$

producing a tangent vector $\boldsymbol{S}_i$, with respect to $\boldsymbol{P}$. This vector is equal to the tangent distance between $\boldsymbol{P}_i$ and $\boldsymbol{P}$.

Compared to the Riemannian geodesic, the Euclidian geodesic between points on an SPD manifold would be shorter and incorrect, as it travels through space outside the manifold. Described by [13], the Riemannian geodesic between two SPD matrices, $\boldsymbol{P}_1$ and $\boldsymbol{P}_2$, can be calculated by the following

$$\delta_R(\boldsymbol{P}_1, \boldsymbol{P}_2) = \| \, log(\boldsymbol{P}_1^{-1} \boldsymbol{P}_2) \, \|_F = \left[\sum_{i=1}^{n} log^2 \lambda_i \right]^{1/2} \tag{6}$$

where $\lambda_i, i = 1, ..., n$, are real and positive eigenvalues of $\boldsymbol{P}_1^{-1}\boldsymbol{P}_2$.

The Riemannian geodesic satisfies the usual axioms for being a valid metric in a metric space [12].

1) $\delta_R(\boldsymbol{P}_1, \boldsymbol{P}_2) \geq 0$

2) $\delta_R(\boldsymbol{P}_1, \boldsymbol{P}_2) > 0, \textit{iff } \boldsymbol{P}_1 \neq \boldsymbol{P}_2$

3) $\delta_R(\boldsymbol{P}_1, \boldsymbol{P}_2) = \delta_R(\boldsymbol{P}_2, \boldsymbol{P}_1)$

4) $\delta_R(\boldsymbol{P}_1, \boldsymbol{P}_3) \leq \delta_R(\boldsymbol{P}_1, \boldsymbol{P}_2) + \delta_R(\boldsymbol{P}_2, \boldsymbol{P}_3)$

These axioms are important to establish. Computing the Riemannian geodesic between two SPD matrices, requires computationally expensive eigenvalue decompositions, logarithm and square root operations. By ensuring these properties, Metric Trees (Section 3) can be constructed allowing distances to be approximated. This approximation reduces the number of measured points, N, in Quick Shifts $O(dN^2)$ complexity (Section 2.2); thus lowering the significance of d and providing faster computation times.

3 Metric Trees and Anchors Hierarchy

A Metric Tree is a hierarchical representation of points bounded by a range in the metric space. In this study, Metric Trees will be used as basis for distance approximation between points to avoid the worst case alternative, of measuring N points against each other; the *Naive* method, which has a complexity of $O(N^2)$. As described by [14] and illustrated with an intuitive focus on the Euclidian space, Metric Trees can efficiently be constructed using the *Anchors Hierarchy* method. A Metric Tree constructed using the Anchors Hierarchy method, contains a tree structure of anchors and will in what follows simply be referred to as an *Anchor Tree*. In [3], this method is further expanded upon for general metric spaces, yielding a construction complexity of $O(N^{1.5} \log N)$ on balanced data. This complexity accounts for the number of distance measurements done during the tree construction. The Anchors Hierarchy method is useful when no notion of a coordinate system are available, as is the case for the space of SPD matrices. The method relies purely on the relative distances between points. It is however required of the distance metric to satisfy all metric axioms, as is the case for the Riemannian geodesic on SCMs.

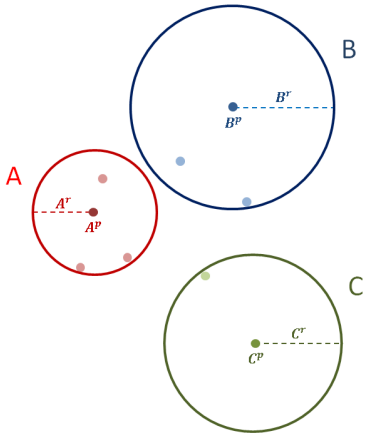

Figure 4: Growing Phase
Initial $\sqrt{N}$ anchors A, B and C created from points closest to their pivot.

Following the general tree construction described in [14], Anchor Trees are constructed in two phases. In the first phase, the *Growing Phase*, $\sqrt{N}$ anchors are constructed from the total set of points, N, illustrated by Figure 4. An anchor, A, has a center point, denoted A^p and referred to as *pivot*. During the growing phase, points are added to the anchor with the closest pivot. These anchor points will be denoted as A^{points} for future reference. In an Euclidean space, an anchor pivot can be computed as half the distance between the two anchor points furthest away from each other. However, as discussed later, other approaches are needed for finding a pivot in the Riemannian space. Lastly, an anchor has a radius, denoted A^r, which is set to cover all its anchor points.

In the second phase, *Agglomeration Phase*, anchor pairs obtained from the Growing Phase are recursive merged into parent anchors, forming a hierarchy. As seen in Figure 5, a parent anchor contains references to a pair of child anchors, which yields the smallest radius. All agglomerated anchors in the tree have the same properties as the original anchors; that is a pivot, a radius and the points that the anchor contains. At the lowest tree level, an anchor will cover a single point.

In the following Section 3.1, the Growing Phase will be explained as described by [14]. Next, in Sec-

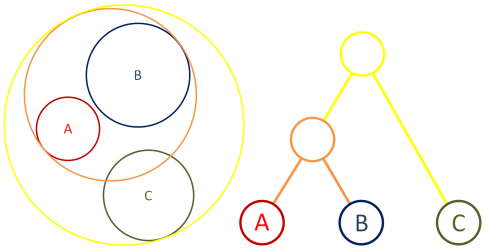

Figure 5: Agglomeration Phase
Hierarchy of anchors constructed bottom-up from the nodes at leaf level.

tion 3.2 the Agglomeration Phase will be outlined and expanded upon. This section introduces specific algorithms for efficiently merging anchors into a tree structure, which has been left unspecified by the authors of [14].

3.1 Growing Phase

As described by Algorithm 1, the first step in the *Growing Phase* assigns all points to an initial seed anchor with a random point as pivot. Points in an anchor are always sorted by distance to pivot in descending order. Hence, the anchor radius covering all children is set as the distance to the first contained point.

In Algorithm 2, a tree structure is recursively built from sub-trees. At each tree level, $\sqrt{N}$ anchors are created from N number of points in the seed anchor. This creation continues recursively subdividing each new anchor until reaching leaf level; when a seed anchor contains only one point. If points in the seed are evenly distributed, each new anchor will contain approximately $\sqrt{N}$ points.

Algorithm 3 describes the step of creating a new anchor. From the set of $\sqrt{N}$ anchors, the point furthest away in the anchor with the largest radius, is selected as pivot for a new anchor, A. For all points in all anchors, those closest to A's pivot are assigned to A and removed from the containing anchor. Measuring distances between points is computationally expensive (see Section 2.3). As an efficiency gain to avoid unnecessary measurements, the Anchor Tree construction method introduces the following threshold

$$t = \frac{Dist(anchor^p, A^p)}{2} \tag{7}$$

which defines t as half the distance between the pivot of an existing anchor, $anchor^p$, and the new anchor A. With points sorted by distance on an existing anchor, reaching a point than t, subsequently means that it and all remaining points are closest to their containing anchor; hence the iteration can be stopped. To maintain efficiency, the $Dist(.)$ function caches all computations ensuring that the distances between points cannot be computed more than once.

Lastly, the recursively constructed anchors are merged into one tree structure by the Agglomeration Phase (Section 3.2).

The tree construction complexity arises from the recursive anchor sorting, described in the appendix of [2]. The seed anchor is sorted in $O(N \log N)$. With the recursive calling of $\sqrt{N}$ created anchors, the complexity becomes $\sqrt{N} \cdot O(N \log N)$, and hence, the final complexity becomes $O(N^{1.5} \log N)$ on balanced data.

Algorithm 1 Growing Phase - First Anchor

```
1: procedure BuildTree(Points)
2:     Let SeedAnchor be a new anchor
3:
4:     SeedAnchor.Add(Points)
5:     SeedAnchor^p ← random p ∈ Points
6:     SeedAnchor.SortPointsDescending()
7:     SeedAnchor^r ←Dist(SeedAnchor^p, SeedAnchor^points[0])
8:     SeedAnchor ← BuildAnchors(SeedAnchor)
9: end procedure
```

Algorithm 2 Growing Phase - Recursive Build

```
1: procedure BuildAnchors(SeedAnchor)
2:     if IsLeaf(SeedAnchor) then
3:         return SeedAnchor
4:     end if
5:
6:     Let Anchors be a new empty set
7:     Let N be a counter
8:     Anchors.Add(SeedAnchor)
9:     N ←Len(SeedAnchor^Points)
10:
11:    while Len(Anchors) < √N do
12:        Anchors.Add(NextAnchor(Anchors))
13:    end while
14:
15:    for each anchor in Anchors do
16:        anchor^r ←Dist(anchor^p, anchor^points[0])
17:        anchor ← BuildAnchors(anchor)
18:    end for
19:
20:    return Agglomerate(Anchors)
21: end procedure
```

Algorithm 3 Growing Phase - Get Next Anchor

```
1: procedure NextAnchor(Anchors)
2:     Let A be a new anchor
3:     A^p ←MaxRadius(Anchors).PopPoint()
4:
5:     for each anchor in Anchors do
6:         t ← Dist(anchor^p, A^p) / 2
7:
8:         for each p in anchor^points do
9:             distToCur ←Dist(p, anchor^p)
10:            distToNew ←Dist(p, A^p)
11:
12:            if distToNew < distToCur then
13:                A.Steal(p)
14:            else if distToCur ≤ t then
15:                Break to next anchor
16:            end if
17:        end for
18:    end for
19:
20:    A.SortPointsDescending()
21:    return A
22: end procedure
```

3.2 Agglomeration Phase

Described by Algorithm 4, the *Agglomeration Phase*, creates a tree structure bottom-up from a set of anchors by merging the pairs which results in the smallest radius. The pair (A, B) is assigned to a new parent C and removed from the set. C is assigned a pivot and a radius encapsulating all points in $A \cup B$. The merging continues until only one anchor is left in the set.

Algorithm 4 Agglomeration Phase

```
1: procedure Agglomerate(Anchors)
2:     while Len(Anchors) > 1 do
3:         Let A and B be anchors
4:         Let C be a new anchor
5:
6:         A, B ← SmallestRadius(Anchors)
7:         C.AddChild(A)
8:         C.AddChild(B)
9:         C^p ← GetPivot(A, B)
10:        C^r ← GetRadius(C^p, A^points, B^points)
11:
12:        Anchors.Remove(A)
13:        Anchors.Remove(B)
14:        Anchors.Add(C)
15:    end while
16:
17:    return Anchors[0]
18: end procedure
```

In general, the Agglomeration phase can be described in three main steps.

Pair Selection: An initial best anchor pair is selected among all $\sqrt{N}$ anchors, based on a rough pivot and a rough radius estimation.

Pivot Selection: Given the pair selection, an improved pivot is found among their containing points, also yielding a better radius.

Radius Selection: From the pair and the improved pivot, an even better radius can be computed from the points in the pair.

For illustration purposes, the figures in the following sections build on the intuition of a Euclidean space, but should however be imagined in the cone-shaped Riemannian space.

3.2.1 Pair Selection

For performance reasons, we resort to an approximated way of selecting the pair of anchors with the smallest radius, described as

$$\underset{A,B \in Anchors}{argmin} \; max\{A^r + Dist(A^p, B^p), \; B^r + Dist(A^p, B^p)\} \tag{8}$$

The intuition behind this equation is further elaborated in Figure 6.

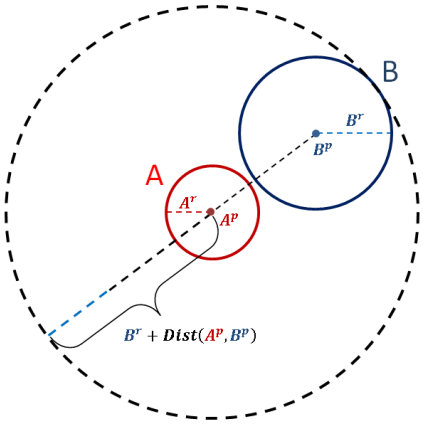

Figure 6: Anchor Pair
Pair of anchors from an approximated radius.

The approximated radius encapsulating an anchor pair is taken from the distance between their

pivots, as well as the maximum radius of the two anchors. This encapsulating radius is an upper bound to the smallest possible radius. As clearly seen in Figure 6, this radius can become very large compared to the best possible encapsulating radius. With N as the number of all points in a sub-tree branch, the Growing Phase yields $R = \sqrt{N}$ anchors, with each anchor containing approximately R points. Computing an exact radius to determine the smallest anchor pair would require checking all anchor points against each other, giving a $O(R^4) = O(N^2)$, for finding the best anchors to merge. At this step, it would not only be too costly but also defeat the purpose of constructing Metric Trees to alleviate Quick Shifts $O(dN^2)$ complexity (Section 2.2).

For the proposed approximate best pair selection, the set of R anchors contains one less anchor after each merge. Equation 8 therefor amounts to $R^2 + (R-1) + (R-2) + \cdots + 2 + 1$ uncached computations. This yields a complexity of $O(R^2) = O(N)$ for the merging decision during the construction of the full sub-tree.

3.2.2 Pivot Selection

Working in an Euclidian space, the mean of all points in $A \cup B$ could serve as pivot for C. However, computing an Euclidian mean in a Riemannian space would likely yield an invalid point lying outside the space of the SPD manifold. Two methods are presented for selecting valid pivots for C.

The first method finds the point closest to A and B's shared center, an approximate *centroid*, as given by

$$C^p = \underset{x \in \{A,B\}}{argmin}\ max\{A^r + Dist(x, A^p), \\ B^r + Dist(x, B^p)\} \tag{9}$$

Equation 9 is valid as the triangle inequality holds for the Riemannian SPD space (Section 2.3, Axiom 4). Illustrated by Figure 7, the centroid is found by iterating through each point, x, in $A \cup B$. By taking the maximum distance between x and the two anchor pivots plus radius, the centroid is determined as the x with the minimum of these max distances.

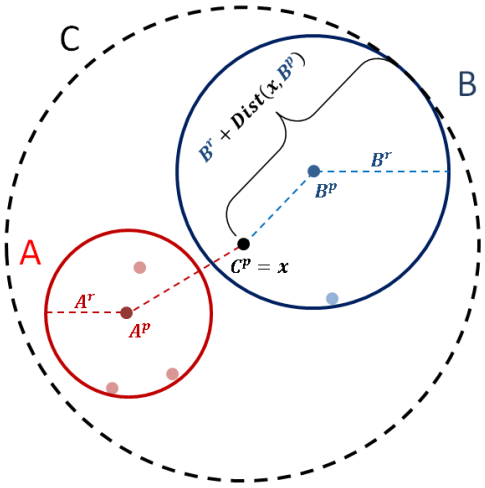

Figure 7: Pivot Selection
Anchor C with C^p set from centroid; the point, x, in $A \cup B$ with the minimum max distance to either pivots, plus radius.

At each level in the tree, R anchors are used to find $\frac{1}{2}R$ parent anchor pivots. This amounts to a complexity of $O(R^2 \log R)$, translating into $O(N \log \sqrt{N})$ for the full tree construction.

The second method constructs a valid pivot by linearly interpolating between points within the Riemannian space [10]. Using the *logarithmic mapping* (Equation 5), the distance between two SPD matrices in the tangent space is defined as

$$\overrightarrow{\boldsymbol{P}_1\boldsymbol{P}_2} = \mathrm{Log}_{\boldsymbol{P}_1}(\boldsymbol{P}_2) \tag{10}$$

A function for Riemannian linear interpolation can then be defined as

$$Int(\boldsymbol{P}_1, \boldsymbol{P}_2, \omega) = \mathrm{Exp}_{\boldsymbol{P}_1}(\omega \overrightarrow{\boldsymbol{P}_1\boldsymbol{P}_2}) \tag{11}$$

with the interpolation amount as $\omega : [0, 1]$. The expression $\omega\overrightarrow{\boldsymbol{P}_1\boldsymbol{P}_2}$, interpolates between $\boldsymbol{P}_1$ and $\boldsymbol{P}_2$ in the tangent space. Using the *exponential mapping* (Equation 4), the interpolated tangent distance can then be projected back onto the manifold, yielding a valid point to use as pivot.

When computing a center pivot for a pair of anchors, the anchor sizes must be taken into account.

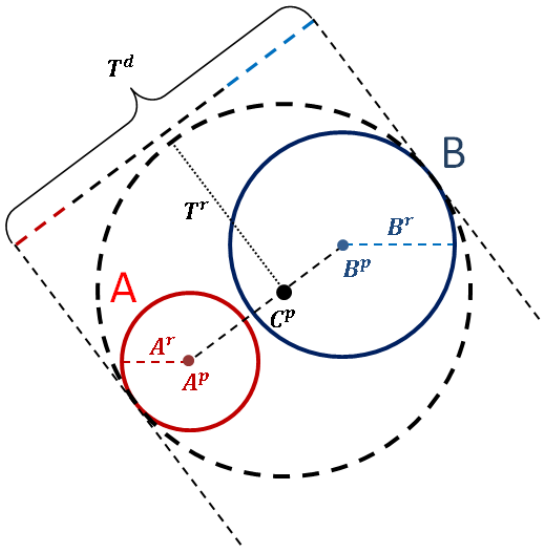

Figure 8: Interpolated Center
From the anchor pairs total diameter, T^d, an interpolated center can be determined using the total radius, T^r.

Illustrated in Figure 8, let T^d be the total diameter covering the entire metric span of A and B, defined as

$$T^d = A^r + Dist(A^p, B^p) + B^r \tag{12}$$

with the exception that one anchor completely covers the other; to which $T^d = max(A^r, B^r)$. The total radius, T^r is thus defined as

$$T^r = \frac{T^d}{2} \tag{13}$$

The following equation describes how a midpoint ratio, $r : [0, 1]$, is found by

$$r = \frac{B^r - A^r + Dist(A^p, B^p)}{2Dist(A^p, B^p)} \tag{14}$$

which leads to the correct interpolated center between two anchors in a Riemannian space as

$$C^p = Int(A^p, B^p, r) \tag{15}$$

For finding the ratio r in Equation 14, a single distance measure between pivots is required. As this computation has already been cached during anchor creation in the Growing Phase (Section 3.1), the interpolation method has a low complexity of $O(1)$. However, creating the interpolated point on the manifold (Equation 15), does in our implementation, require three (worst case) eigenvalue decompositions. This yields a high constant with respect to computation time.

A perfect pivot for all points in $A \cup B$ can be taken as the interpolated center between the pair of points having the largest distance, as seen by Figure 9. This would however bring the tree construction complexity back to $O(N^2)$.

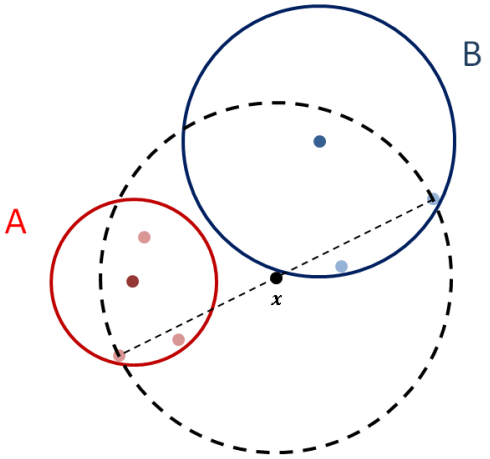

Figure 9: Perfect Pair Pivot
A perfect pivot, $\boldsymbol{x}$, as the interpolated center between the pair of points with largest distance in $A \cup B$.

3.2.3 Radius Selection

Using either the *centroid* or *interpolation* pivot methods, a *measured radius* for C can be found from the point furthest away. This is illustrated in Figure 10 and given by

$$C^r = \max_{x \in \{A,B\}} \{Dist(x, C^p)\}, \tag{16}$$

using the max distance (worst case) between C^p and all points in $A \cup B$ as radius. In the case where one anchor is completely contained within the other, the radius is set as the maximum of the two radiuses. The complexity of a measured radius is the same as the one for the centroid method, described by Equation 9; at $O(N \log \sqrt{N})$ for the full tree construction.

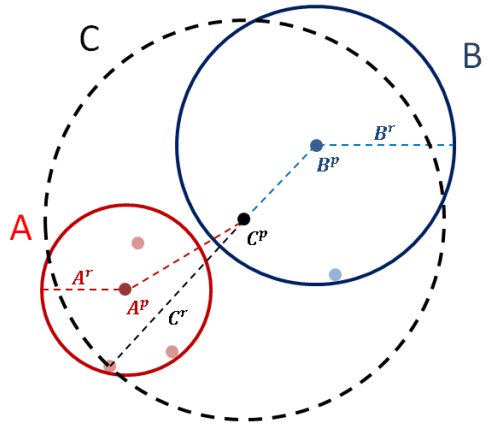

Figure 10: Measured Radius
Anchor C with C^p set from centroid and C^r from the point furthest away.

With respect to a measured radius, the *interpolation* method is not guaranteed to produce a better pivot than the *centroid* method. From Figure 11, good and bad cases of both pivot methods are shown when their radius is set from the point furthest away.

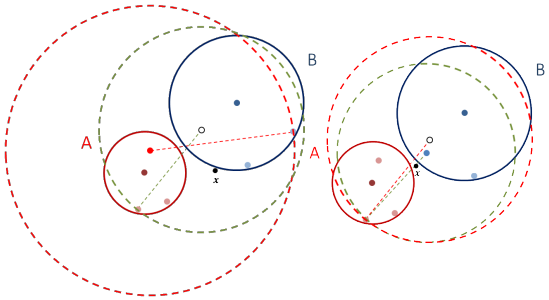

Figure 11: Pivot Radius Compared
Left: *Interpolation produces the best pivot (black circle point) by being closest to the perfect center $\boldsymbol{x}$. This yields a narrow radius (green line), compared to the centroid (red dot).* ***Right:*** *Here the centroid (blue dot) yields the smallest radius, as it is now closest to the perfect center.*

As seen from the anchor pair to the left in Figure 11, no point in $A \cup B$ is close to their perfect center, $\boldsymbol{x}$. The centroid pivot (red dot) therefore fails to produce the best radius compared to the interpolated pivot (black circle point). In this case, the interpolated pivot is closer to the perfect center, hence yielding a superior minimal radius. From the anchor pair to the right, the centroid (blue dot) measures a better radius than the interpolated pivot, because it is slightly closer to the perfect center of all points in $A \cup B$.

Because the *interpolation* pivot selection method creates an entirely new point, finding a radius for this new pivot, using Equation 16, will result in $2\sqrt{N}$ new (uncached) distance calculations for each anchor pair. Optionally, an *approximated radius* can be computed avoiding new measurements to interpolated pivots by taking the distance of T^r, defined by Equation 13 and seen in Figure 8. As computing T^r only requires one cached distance, the complexity becomes $O(1)$.

Selecting a minimal radius is important as it leads to better performance and cluster approximation, which is further explained in Section 4.

The *centroid* and *interpolation* pivot methods as well as the *measured* and *approximated* radius methods, will be referred to by these names in the BCI application interface described in Section 6.

4 Dual Trees

In a Dual Tree traversal algorithm [11], explored for fast kernel density estimates in [2], two partition trees referred to as *data* and *kernel* partition trees, are constructed. In this study, the data partition tree is the constructed Metric Tree (Section 3) with each leaf node being a data point on a Riemannian SPD manifold. The kernel partition tree is a copy of the data tree and together they are used to construct a block partition. If the kernel evaluations for a set of kernel and data points do not vary in any significant way, they can be represented by a single value in a block partition.

As detailed in [2], a Marked Partition Tree (MPT) contains references to data and kernel partition nodes yielding a block partition. Figure 13 illustrates the structure of a MPT which represents the data and kernel trees in Figure 12. With green leaf numbers pointing to data tree nodes and red numbers pointing to kernel tree nodes, a block partition can be derived from these references. Kernel blocks $1-2$ and $5-6$ exists in data columns 3 and 4. Data columns 3 and 4 each has separate blocks for

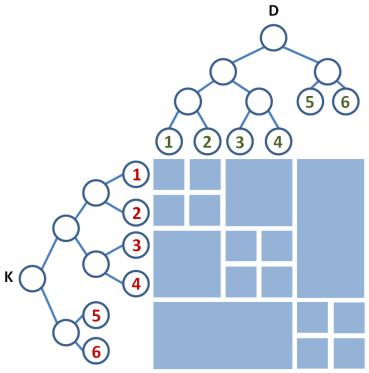

Figure 12: Dual Tree
Block partition (matrix) for data (top) and kernel (left) partition trees. For illustration purposes, the kernel tree has been transposed.

the 3, 4 kernel items.

A block partition is constructed by comparing data and kernel nodes. Non-overlapping nodes are grouped together whereas overlapping ones are partitioned into separate blocks. In the Anchor Tree, a given anchor encapsulates all its descending sub-anchors within its radius. The basic intuition behind the block construction is that if A does not overlap with B, then A and all its descendants are unlikely to significantly affect the density of B, or any of B's descendants; hence A and its descendants can be represented by a single value, with respect to B. Likewise, if A overlaps with B, then A will need further refinement as A might contain descendants which could affect the density of B significantly enough to be represented by a separate value.

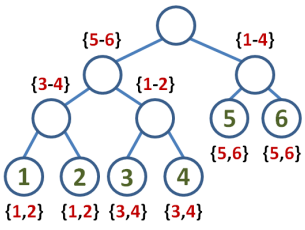

Figure 13: Marked Partition Tree
MPT representing block partitions from Figure 12. Color-coded numbers correspond to the data and kernel partition tree nodes.

The amount of blocks represents the level of point approximation. By recursively refining overlapping nodes and checking for overlap between sub-nodes, more blocks can be constructed. Fewer blocks yield a less accurate representation of points, in return for less distance measurements. Likewise, more blocks yields a more accurate representation at the expense of computation speed, as more points will be measured. A maximum of N^2 partition blocks can be acquired to which no performance gain is achieved.

By using a MPT with approximated distances, the time complexity $O(dN^2)$ for Quick Shift (Section 2.2) can be improved. The significance of d for density estimates (Equation 3) becomes less with fewer blocks. Likewise, more blocks will make d more significant giving more accurate density estimates, at the expense of computation speed.

5 BCI Recordings

A BCI experiment was conducted in cooperation with the Department of Computer Science and Department of Humanistic Informatics at Aalborg University. Here, test subjects were asked to watch an unknown series of video clips while having their brain activity recorded. The clips selected were picked to evoke a wide range of emotions, spanning from happiness to shock. The experiment took place during two days using three Emotiv Epoc headsets, a consumer grade EEG device. The Emotiv Epoc is capable of recording brain activity at 128 Hz from 14 electrodes and has a built-in Notch filter at $50 - 60$ Hz. Along with channel data, the contact quality for each electrode is also included in the recordings, which for most subjects were perfect at all times.

Subjects had little or no knowledge of EEG. Out of 70 subjects the experiment resulted in 65 complete datasets. An incomplete dataset is defined as a prematurely ended recording. The gender distribution is approximately 1/3 males and 2/3 females. In groups of one to three, subjects were isolated with one or more test-leaders quietly positioned in the back. Each subject was instructed to sit relaxed while watching a 10 minute video consisting of smaller commercial clips. A 30 second black screen marked the beginning and end of the

video. All clips within the video were separated by 10 seconds of black screen.

Since recordings happened on up to three subjects at a time, it is possible that they have affected each other. It is also possible for subjects to previously have seen some of the presented commercial clips, which might affect their reaction to seeing them again. Also, data sometimes contain noise from activities such as head movement and eye blinks. This makes for a realistic case as any algorithm analyzing BCI data in a real scenario, would have to resolve noisy recordings.

5.1 Synchronization

For convenience, the BCI recordings were initiated before the video start. Once all subjects were ready, the test leader would mark the actual video beginning in each recording; done on up to three separate PCs (one per headset). Since this marking was done by hand, the marker has a slight delay relative to the video start (up to 8 seconds). To remove this delay, each dataset has been synchronized by finding an approximate video start, based on signal peaks within each dataset. Using approximated times for three distinctive scare events, the channel peeks were analyzed within windows expected to capture the events. The largest peak across all channels within a time window is assumed to be the beginning of the event. An offset is then calculated by subtracting peak time with expected event time. The intuition is that scare scenes would cause a symmetrical spike in EEG activity, mostly due to head movement. Computed averages for all offsets from the three events according to peaks, yields an approximation of the actual video start.

Synchronized versions of the source datasets were created removing non-video EEG readings from the beginning of each dataset. Each synchronized dataset then becomes exactly 9 minutes 51 seconds long.

6 Application

As part of this study, the application *BCI Explorer* was created. This program is able to visualize a BCI dataset with the intent of exploring brain patterns, using the theory and algorithms described by this paper. An overview of the application interface is given by Figure 14.

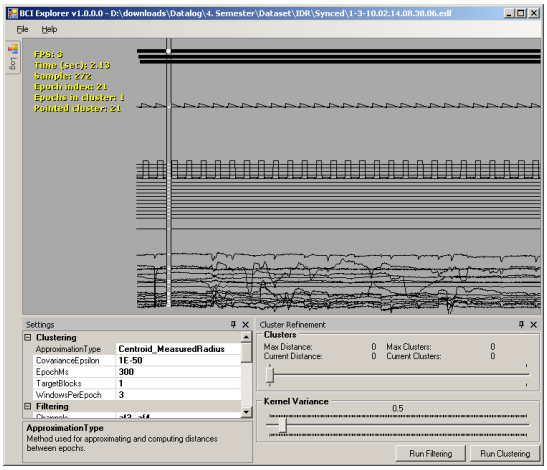

Figure 14: BCI Explorer: Interface
The middle blue window, the Signal View, contains channel data. Remaining windows have been docked and hidden.

The main blue window, the *Signal View*, visualizes data across all recorded channels, horizontally over time; referred to as the *timeline*. From the Signal View, channel data can be dragged in the main window using the left mouse button, and zoomed in/out using the scroll wheel. Hovering the cursor on the timeline will highlight a given epoch. With a flexible interface, all sub-windows can be docked, hidden, resized and rearranged at the users' convenience.

6.1 Settings

From the *Settings* window (Figure 15) filtering and clustering options can be changed.

To remove noise or isolate certain frequencies, BCI Explorer has built-in customizable *Low-Pass*, *High-Pass*, *Band-Pass* and *Band-Stop* Butterworth filters.

The temporal window size (*epoch*), used to compute SCMs (Section 2.1), is defined in milliseconds by *EpochMs*. To our experience, important signal patterns are often not aligned with fixed window segmentations; such as an eye-blink artifact being split in half between epochs. To avoid missing important patterns due to bad epoch partitioning, the option of running multiple sliding-windows was

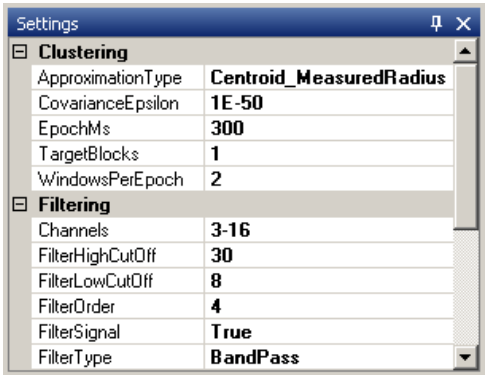

Figure 15: BCI Explorer: Settings
All filter and cluster settings can be changed from this window.

introduced, set from *WindowsPerEpoc* in the Settings window. According to the number of sliding-windows, epochs will be started at an offset. Setting this to 2, will start a new epoch halfway into the previous; 3, will start a new epoch one third into the previous, etc. More epochs will of course result in a longer computation time, as these become points on the Riemannian manifold to which distances are measured.

Described by Equation 6, measuring the distance between two SCM matrices requires their eigenvalues. In BCI Explorer, the eigenvalues of two SCMs are computed from a generalized eigenvalue problem. However, due to floating point rounding errors, this problem becomes computationally unstable on matrices with little variance. One option to solve this would simply be to exclude SCMs with little variance. However, since a SCM with little variance might still represent usable information on some channels, this issue was solved by constructing computationally stable versions of the original SCMs. By adding a positive offset to each computed eigenvalue from a single SCM, a computationally stable SCM can be constructed almost identical to the original. The *CovarianceEpsilon* option controls this positive eigenvalue offset. Ideally, this should be set as close to zero as possible, without yielding unstable SCMs.

The *ApproximationType* selects the approximation algorithm to use during Anchor Tree construction, as described by Section 3. To perform comparison experiments, the *Naive* method is also included as an option.

As described in Section 4, the level of point approximation can be refined with the number of block partitions. The target block partition size is set from the *TargetBlocks* setting. This does not guarantee a certain partition size, e.g. a partition size of 1 is not possible, but will yield the minimum possible number of partitions. This option does not apply when using the *Naive* method, as this method does not rely on the estimations of an Anchor Tree.

6.2 Hierarchical clustering

Once the dataset has been filtered and clustered, the hierarchical representation of data can be explored from the Signal View window.

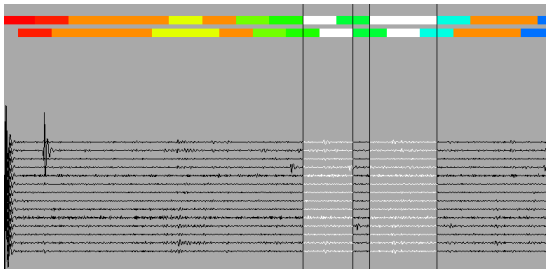

Figure 16: BCI Explorer: Clusters
Signal View after clustering. The color-coded cluster bars in the top shows which epochs belongs to which clusters.

As shown in Figure 16, channel data is aligned underneath each other in the bottom part of the window. Color-coded *cluster bars* are shown in the top of the window. In this case, channel data has been partitioned using two sliding windows yielding two cluster bars. Each color on the bars represents a unique cluster of epochs. The cluster locations correspond to channel data on the timeline, visualizing what data belongs to which cluster. Hovering a cluster on the timeline will highlight and outline the epochs which are part this cluster.

Clustering statistics, such as method, time, epochs, etc. are output to the *Log* window shown in Figure 17

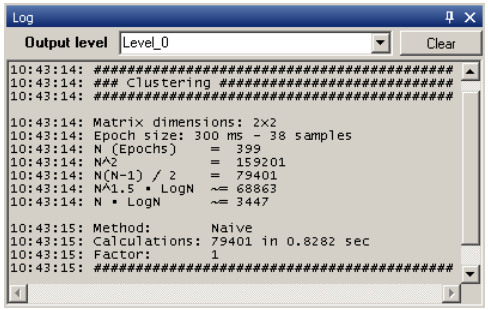

Figure 17: BCI Explorer: Log *Log window outputting clustering statistics.*

The tree structure created by Quick Shift (Section 2.2), can hierarchically be explored by adjusting the sliders in the *Cluster Refinement* window, shown in Figure 18.

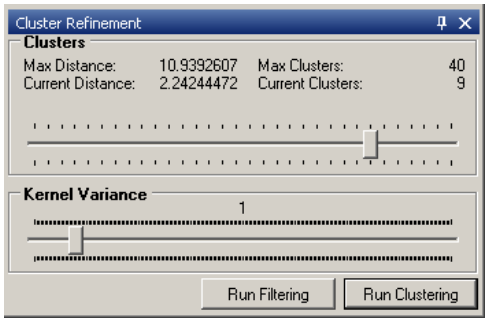

Figure 18: BCI Explorer: Cluster Refinement *Quick Shifts kernel variance and the threshold for breaking off tree branches, can be refined from this window.*

Using the cluster slider, tree branches longer than the specified distance are broken off, yielding clusters. Likewise, the kernel variance slider controls the variance on the normal distribution used by Quick Shift, to estimate kernel densities (Equation 3). Changing this slider will imply a recalculation of the estimated kernel densities. This is however fast as all distances are cached at this stage. The approximation methods tries to estimate a reasonable variance, based on the average distance between k random points and their k'th nearest neighbor, where $k = \sqrt{N}$. The kernel variance slider is therefore only enabled for the Naive method, for the option of manually controlling variance.

Adjustments made with these sliders will, in real time, change the colors of the cluster bars in the Signal View; making data exploration fast and responsive.

7 Results

This section presents the evaluations for the described methods in Section 3 using BCI Explorer. Section 7.1 will present the quantitative evaluations of each method according to computation factor, number of partition blocks and time. The following Section 7.2 visually presents a qualitative demonstration of the hierarchical clustering capabilities using Quick Shift.

7.1 Quantitative Evaluation

The quantitative performed tests were run on a 2.1 Gz AMD Athlon (tm) II P320 Dual-Core Processor machine with 6 GBs of RAM. Throughout this section, we evaluate each of the three approximation methods to the Naive method; described in Section 3. As all methods have been implemented to take advantage of a multi-threaded environment, their performance can vary greatly depending on processor. The approximation methods will for simplicity be referred to by the abbreviations: Centroid Measure Radius (CMR), Interpolated Measured Radius (IMR) and Interpolated Approximated Radius (IAR).

For each method test, the results are presented in terms of *computation factor*, *computation time* and *partition blocks*. Computation factor, referred to as *c-factor*, denotes the number of computed distances compared to the naive method. Given N points, the naive method computes $\frac{N \cdot (N-1)}{2}$ distances, equal to a c-factor of 1. A c-factor of 2 is equivalent to half the naive computations. Time is highly correlated with c-factor, however time may also reflect performance overhead and additional time consuming operations from each method. Explained in Section 4, fewer partition blocks means less anchor overlap. With the goal of producing the least amount of blocks, the first set of tests were run against an interval of 100, 500, 1000 and 3000 epoch, on 2 and 14 channels. Few epochs and channels will emphasize potential overhead from

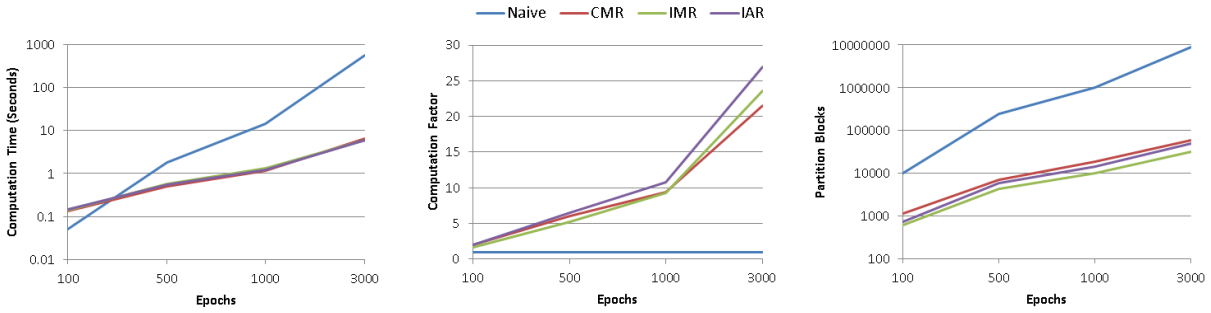

Figure 19: Performance at low dimensionality
Performance results at 100*,* 200*,* 1000 *and* 3000 *epochs, using* 2 *channels.*

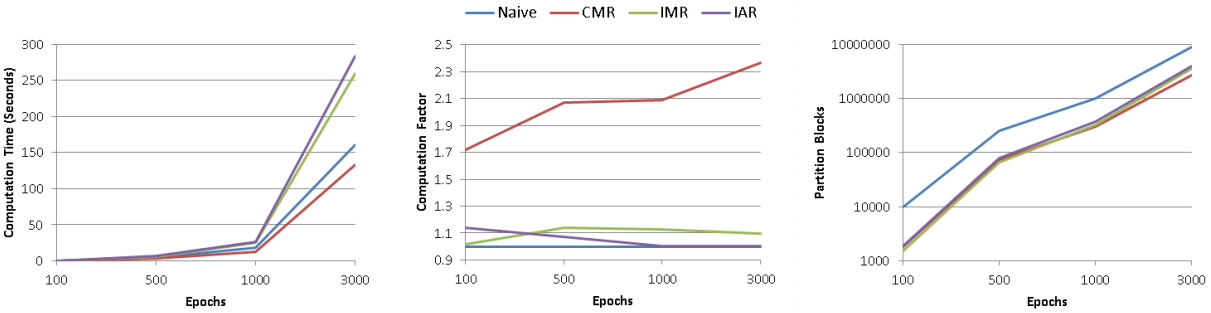

Figure 20: Performance at high dimensionality
Performance results at 100*,* 200*,* 1000 *and* 3000 *epochs, using* 14 *channels.*

the method implementations. Higher numbers will highlight the general tendencies of each method. These results are seen in Figure 19 and 20. Note that some scales in these figures are logarithmic.

Looking at time and c-factor, the three approximation methods are far superior to the naive method on 2 channels. Although only doing half the computations of the naive method, the approximation methods are slower than the naive at 100 epochs. This indicates a minor performance overhead, most likely originating from the Anchor Tree construction process. Compared to CMR, the interpolated pivots of IMR and IAR yields fewer blocks implying smaller anchor radiuses with less overlap. The measured radius of IMR further reduces anchor overlap resulting in the least amount of blocks.

On 14 channels, CMR performs best on all scales. Opposite the results using 2 channels, CMR is now producing fewest blocks. The additional uncached measurements introduced by IMR and IARs interpolated pivots, results in no significant c-factor improvement compared to other methods. Combined with a low c-factor, the expensive computational constant related to creating new points on the Riemannian manifold, makes the computation time of these methods worse than the naive.

With parameters ranging from 1 to 14 channels at a fixed epoch count of 1500, the next performance tests was conducted with the intent of highlighting method trends, at increased dimensionality. Again, the approximation methods must produce the least amount of partition blocks. Seen in Figure 21, at few channels, all approximation methods have far superior c-factors compared to the naive. This however declines rapidly with the increased dimensionality. At 4 channels, the declining c-factor slowly breaks off for the CMR method, keeping a steady c-factor of around 2.5, up to 14 channels. IMR and IAR keep declining until they are just as bad, or even worse than the naive. Re-

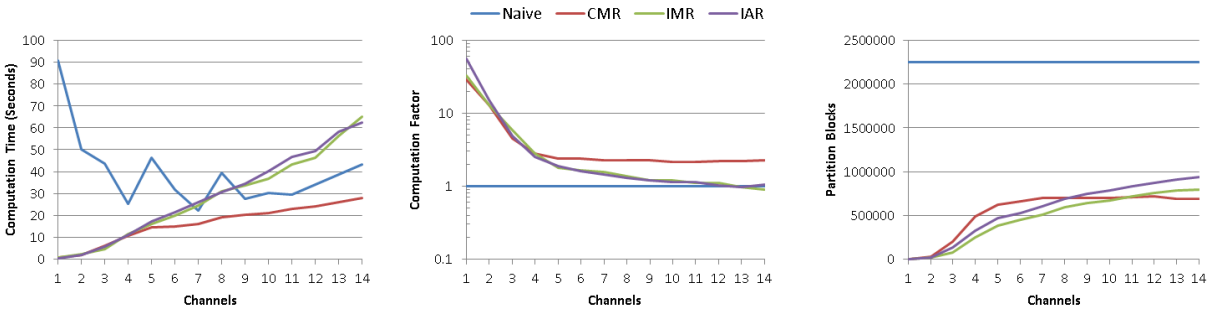

Figure 21: Performance vs dimensionality
Performance results at 1500 *epochs, using* 1 *to* 14 *channels.*

sults show that the increased dimensionality greatly impacts both time and c-factor for all approximation methods. Because more dimensions increase the probability of points having a greater spread, we suspect that the threshold distance, introduced in the Anchor Tree's Growing Phase by Equation 7, fails to uphold efficiency. As a result, more points are measured during Anchor Tree construction thus reducing the c-factor. Furthermore, the increased overlap will also impact the Dual Tree step. If two anchor nodes do not overlap, any further measurements to child nodes can be omitted. As more anchors overlap, more child nodes will also be measured.

Interestingly, on lower dimensions the naive method has a big overhead, to which it performs worse than on higher dimensions. At 4 channels, the naive times begin to increase, although still performing better at 14 channels compared to 1. This overhead has been observed to originate from the programming library used to compute eigenvalue decompositions[1]. At few channels, IMR and IAR produce fewer blocks than CMR. Interestingly, at higher channels CMR starts yielding the least amount of blocks implying overall smaller anchor radiuses.

7.1.1 Quantitative Conclusion

Compared to other methods, test results show that CMR performs substantially better at higher dimensions, across all parameters. At varying epoch intervals CMR is just as good, or better, than IMR and IAR. Despite CMR performing marginally worse at lower dimension, mostly in terms of blocks, it is still a good all-around choice which in most cases yields the highest performance. On lower dimensions, IMR and IAR produces fewer blocks making them a considerable choice if higher levels of approximation are desired.

[1] Microsoft Research: Sho 2.1
http://research.microsoft.com/en-us/projects/sho/

7.2 Qualitative Evaluation

Using the naive method in BCI Explorer, the qualitative performance of the Quick Shift based hierarchical clustering has been evaluated. Clustering unknown data leaves little predetermined answers about cluster formations. Clusters must therefore be visually explored based on a given set of parameters. A simple case of clustering of eye-blinks is evaluated.

7.2.1 Eye-blink Evaluation

From the recorded BCI datasets (Section 5), all subjects are expected to produce eye-blink artifacts during the almost 10 min of video watching. Identifying these eye-blinks thus makes for a simple case of testing clustering capabilities. Figure 22 shows a chunk of data over a period of approximately 35 *sec*, containing eye-blink spikes from subject 1. Muscle activity is expected to appear on lower frequencies [19]. Data from the frontal channels ($AF3$, $AF4$) close to the eyes, has therefore been Low-Pass filtered at 8 Hz to isolate eye-blink activity. As eye-blinks are expected to happen over a short period of time, the epoch size was set at 300 ms. Furthermore, the data has been partitioned using three sliding windows to maximize the chances of capturing all eye-blinks, at the right moments.

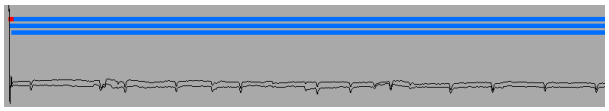

Figure 22: Hierarchical clustering
First emerging clusters at a high threshold distance.

As described in Section 2.2, clusters are retrieved from the tree structure created using Quick Shift, by breaking of branches greater than a specified threshold distance. When gradually reducing this threshold, the first emerging clusters separates the data (blue cluster), from the Butterworth filter calibration at the beginning (red cluster).

In Figure 23, the next cluster to appear further segments the blue cluster by isolating most eye-blink activity (orange cluster).

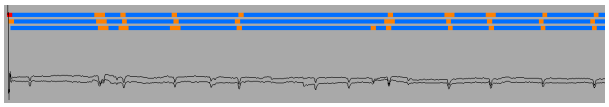

Figure 23: Hierarchical clustering
Emerging cluster of eye-blinks segmented from the blue cluster.

It makes sense for eye-blinks to appear very early into the hierarchical structure, as these are the epochs (points) with the biggest covariance and hence has the greatest distance to more subtle epochs with less covariance.

Further down into the hierarchy, the last eye-blinks are capture by the green cluster, seen in Figure 24.

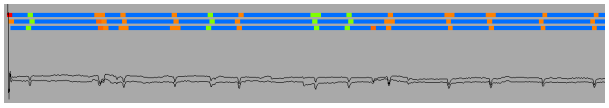

Figure 24: Hierarchical clustering
The last slightly different eye-blinks are capture by the green cluster.

Noticeably, all eye-blink are now split between two clusters. A possible explanation for them being split is that the eye-blinks in the green cluster appear to exhibit slightly different patterns than those captured by the orange cluster. This variance could place the green eye-blinks closer to the epochs of the blue cluster, on the Riemannian manifold. The distance threshold therefore has to be low to separate the green and blue clusters; substantially lower than the threshold for separating orange and blue clusters.

In Figure 25, a very low threshold distance retrieves the clusters being far down into the hierarchical structure. The most subtle epochs (highlighted in white) are now separated from eye-blinks and other epochs with high covariance.

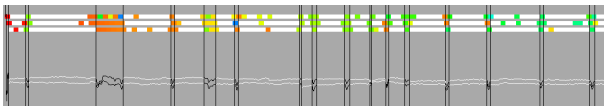

Figure 25: Hierarchical clustering
At a small threshold distance, epochs with low covariance are clustered separately from epochs with high covariance.

The hierarchical approach has in this case shown effective at clustering eye-blinks and other sources of noise. On the Riemannian manifold, points with high covariance have relatively large distances to points with low covariance, thus resulting in an early separation when reducing the threshold distance.

8 Conclusion

This paper has presented the high performance tool, BCI Explorer, enabling visual exploration of clustered BCI data. The mode-seeking algorithm Quick Shift, has been used to create a tree structure of BCI data points, which is used to hierarchically retrieve clusters. We have shown that the slow $O(dN^2)$ complexity of Quick Shift can be alleviated through an approximative solution. With d as the computational constant of each distance measure between N points, reducing N through approximation will thus produce fewer measurements and decrease Quick Shifts tree construction time.

A Dual Tree approach has been used to achieve this approximation, with the intuition that distant points can be represented by the same value. As each of our data points belong to the non-vector space of a Riemannian manifold, Metric Trees has been constructed using Anchors Hierarchy which does not require data coordinates. These Metric

Trees was then used as a set of Dual Trees. For the creation of Metric Trees, we have presented three methods of constructing anchors in the Agglomeration Phase: Centroid Measured Radius (CMR), Interpolated Measured Radius (IMR) and Interpolated Approximated Radius (IAR). With Metric Trees constructed from these methods, we have evaluated the quantitative performance of the Dual Tree.

Compared to the naive method, the presented methods have been shown to greatly improve computation times on lower channel dimensions. At the greatest level of approximation, the CMR method performs overall best resulting in 2.5 to 25 times fewer distance measurements than the naive method. At lower dimensions, the IMR and IAR methods yields fewer blocks than CMR, at the cost of computation speed. CMR is however best at producing fewer blocks on higher dimensions. Increased dimensionality has a poor effect on these methods as this increase the likelihood of bigger anchor radiuses, which increases computation time. Tested on our own dataset, visual demonstrations have shown the hierarchical clustering capability of a Quick Shift based approach. In a simple case, we have seen how clusters of eye-blinks and noise were separated from clusters of brain activity. This exploration of BCI clusters could be used as a potential pre-step to a classification problem.

In future works we would like to further experiment with distance approximations in the Riemannian space. By constructing a KD-Tree from points projected to the Euclidian tangent space, an approximative solution might be possible without the use of Metric and Dual Trees. Many enhancements could also be made to BCI Explorer. Currently the tool can only explore a single dataset. Allowing the exploration of multiple datasets at once, might reveal interesting data clusters across subjects. Another interesting option would be to cluster data based on variable epoch sizes. Imagined as a hierarchy, epochs might be largest at the root while gradually decreasing in size until reaching leaf nodes. At varying sizes, bigger epochs would highlight general tendencies over long periods, while being segmented by the smaller epochs representing short local tendencies.

References

[1] *Emotiv EPOC Specifications.* `http://emotiv.com/upload/manual/EPOCSpecifications.pdf`. Visited at May the 7th 2014.

[2] Saeed Amizadeh, Bo Thiesson, and Milos Hauskrecht. Variational Dual-Tree Framework for Large-Scale Transition Matrix Approximation. In *The 28th International Conference on Uncertainty in Artificial Intelligence*, UAI '12, Catalina Island, CA, August 2012.

[3] Saeed Amizadeh, Bo Thiesson, and Milos Hauskrecht. The Bregman Variational Dual-Tree Framework. In *The 29th International Conference on Uncertainty in Artificial Intelligence*, UAI '13, Seattle, WA, July 2013.

[4] Alexandre Barachant, Stéphane Bonnet, Marco Congedo, and Christian Jutten. Multiclass Brain-Computer Interface Classification by Riemannian Geometry. *IEEE Trans. Biomed. Engineering*, 59(4):920–928, 2012.

[5] Benjamin Blankertz, Ryota Tomioka, Steven Lemm, Motoaki Kawanabe, and K-R Muller. Optimizing Spatial Filters for Robust EEG Single-Trial Analysis. *Signal Processing Magazine, IEEE*, 25(1):41–56, 2008.

[6] Yizong Cheng. Mean Shift, Mode Seeking, and Clustering. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 17(8):790–799, 1995.

[7] Dorin Comaniciu and Peter Meer. Mean Shift: A Robust Approach Toward Feature Space Analysis. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 24(5):603–619, 2002.

[8] Thomas P. Fletcher and Sarang Joshi. Riemannian Geometry for the statistical analysis of diffusion tensor data. *Signal Processing*, 87(2):250–262, 2007.

[9] Keinosuke Fukunaga and Larry Hostetler. The Estimation of the Gradient of a Density Function, with Applications in Pattern Recognition. *Information Theory, IEEE Transactions on*, 21(1):32–40, 1975.

[10] Alvina Goh and René Vidal. Unsupervised Riemannian Clustering of Probability Density Functions. In Walter Daelemans, Bart Goethals, and Katharina Morik, editors, *ECML/PKDD (1)*, volume 5211 of *Lecture Notes in Computer Science*, pages 377–392. Springer, 2008.

[11] Alexander G Gray and Andrew W Moore. 'N-Body' Problems in Statistical Learning. In *NIPS*, volume 4, pages 521–527, 2000.

[12] Jürgen Jost and J Jost. *Riemannian Geometry and Geometric analysis*, volume 4. Springer, 2005.

[13] Maher Moakher. A differential geometric approach to the geometric mean of symmetric positive-definite matrices. *SIAM Journal on Matrix Analysis and Applications*, 26(3):735–747, 2005.

[14] Andrew W. Moore. The Anchors Hierarchy: Using the Triangle Inequality to Survive High Dimensional Data. In *Proceedings of the 16th Conference on Uncertainty in Artificial Intelligence*, UAI '00, pages 397–405, San Francisco, CA, USA, 2000. Morgan Kaufmann Publishers Inc.

[15] Robert Oostenveld and Peter Praamstra. The five percent electrode system for high-resolution EEG and ERP measurements. *Clinical Neurophysiology*, 112(4):713–719, 2001.

[16] Yaser Ajmal Sheikh, E.Khan, and Takeo Kanade. Mode-seeking by Medoidshifts. In *Eleventh IEEE International Conference on Computer Vision (ICCV 2007)*, number 141, October 2007.

[17] Steven Thomas Smith. Covariance, Subspace, and Intrinsic Crameŕ-Rao Bounds. *Signal Processing, IEEE Transactions on*, 53(5):1610–1630, 2005.

[18] Andrea Vedaldi and Stefano Soatto. Quick Shift and Kernel Methods for Mode Seeking. In *Computer Vision–ECCV 2008*, pages 705–718. Springer, 2008.

[19] Jonathan Wolpaw and Elizabeth Winter Wolpaw. *Brain-Computer Interfaces: Principles and Practice*. Oxford, 2012.