LU-TP 22-31
June 2022

Accelerating MCR-ALS decomposition of hyperspectral images using k-means clustering

Eric Kull

Department of Astronomy and Theoretical Physics, Lund University

Bachelor thesis supervised by Carl Troein

LUND
UNIVERSITY

Abstract

A hyperspectral image may be decomposed into component spectra and their distribution in the image to simplify analysis by revealing underlying patterns and reducing the dimensionality of the image; this may be achieved by the algorithm MCR-ALS. However, the algorithm is time consuming, but could be accelerated by a data reduction. Data reduction can be done by using a clustering method. In this project, the aim is to determine how clustering, k-means in particular, can be incorporated with MCR-ALS to achieve an accelerated decomposition.

We measured how different losses and time consumption were influenced by different parameter choices, e.g, initialization of the k-means. Clustering can result in a reduction in the time-consumption independently of the choice of parameters, but the choices altered the decomposition substantially.

From the results, we concluded that k-means can be incorporated into MCR-ALS, and that the method for selection of centroids is the most crucial step. Accordingly, an optimal set of parameters could be determined.

Snabbare separation av ljus i sökande av livets hemligheter

Att analysera livets biokemi, kemin bakom biologiska processer, är viktigt för att få en förståelse av livets mekanismer, speciellt för små längdskalor. Biokemin hos olika typer av celler skiljer sig åt, vilket kan användas för att t.ex. skilja mellan elakartade och icke-elakartade tumörer. Vilka ämnen som finns i en cell kan bli identifierade via deras respektive absorptionspektrum. Ett absorptionsspektrum berättar hur stor andel av en specifik våglängd av ljus som har absorberats och är unik för varje ämne. Så kallade hyperspektrala bilder, bilder där pixlar representerar var sitt absorptionsspektrum, är tagna för att spåra ämnena. Dock finns det ett problem: ämnena i bilden är blandade och därför är spektrumen från ämnena också blandade. Det bildas en 'soppa' av kemikaliernas spektrum som gör det svårt att särskilja olika ämnen.

Denna 'soppa' av spektrum kan 'oblandas' och uppdelas i 'rena' spektrum och bilder som representerar hur de spektrumen är 'blandade' med diverse olika metoder. MCR-ALS är en metod som gör just detta. Ett problem med MCR-ALS är att när en bild blir större med högre upplösning hos dess spektra, tar det längre tid för algoritmen att 'oblanda' bilden. Det förekommer att större uppsättningar bilder ska analyseras samtidigt, vilket ökar tidsåtgången ännu mer. En tidsreduktion kan ske genom att utnyttja det faktum att många spektrum är likadana och kan därmed representeras med ett spektrum, vilket minskar mängden data som behövs arbetas med. En sådan gruppering kan göras med hjälp av en klustrings-algoritm.

Klustring är ett begrepp som omfattar alla typer av data-reduktion som automatiskt sorterar data i grupper. Dessa grupper kan sedan representeras av så kallade centroider. Dock, ett problem med klustring är att det inte bevarar data perfekt och kräver tid utöver steget då spektran 'oblandas', som beror på det valda antalet centroider. Konsekvenserna kan liknas med en sammanfattning av en film. Har man med för många detaljer hade det tagit kortare tid att se filmen, men om man har för lite detaljer försvinner filmens andemening. I vårt arbete har vi utvecklat en metod när man arbetar med klustrad data i MCR-ALS, namngiven CAM-A, och studerar vilka effekter uppstår av olika val av parametrar.

Contents

1 Background

A hyperspectral image is an image where each pixel in the image contains a spectrum represented by a number of features, representing the intensity of a certain wavelength or frequency of light at that pixel. Therefore, the image has a 3-dimensional structure where its 'depth' is governed by the number of features, see figure 1. There are many methods to record such data, such as Raman and IR spectroscopy [1].

The hyperspectral image can be decomposed into constituent spectra and their respective contributions at each pixel in the image, thus achieving a dimensional reduction of the data, simplifying the analysis of the contents of the hyperspectral image, and possibly unraveling faint physical patterns [1, 2]. The chemical composition of an object in a hyperspectral image can then be analyzed to e.g., distinguish non-malignant from malignant tumors [3]. Two examples of methods by which such a decomposition may be done are principal component analysis (PCA) and Multivariate Curve Resolution–Alternating Least Squares (MCR-ALS).

PCA can extract a set number of components and their respective contributions which explain most of the variance within a dataset [4]. As such, it can efficiently describe the spectra in spectral data using a few pure spectra and their concentration profiles. However, PCA has two major shortcomings. The first is that PCA allows the contribution and features of the constituent spectra to be negative. Whilst it is possible for an absorption spectra to be negative in some cases, it is not possible have a negative amount of a chemical compound. The second is that the spectra are orthogonal to each other, which there is no physical precedent for. Consequently, the results are difficult to interpret in a physical manner [5].

MCR-ALS also decomposes the data into spectra and contributions, but uses a non-negative bilinear model. The decomposition is done by alternating between determining the set of spectra or their contribution with only positive elements that minimizes the square-error, based on respectively the contribution or spectra respectively found previously, and it is initialized by postulating either an initial set of constituent spectra or the contributions [1, 5, 6]. Thus, the result is easier to interpret in a physical manner [5]. Despite the simpler interpretation of results, there may be different sets of spectra and their contributions such that an image may be described nearly equally well by different solutions which are rotations of each other [2, 7, 8], and a way to combat this is to allow the spectra to have negative values. Furthermore, MCR-ALS treats the image as a matrix where the rows and columns respectively correspond to features and pixel number, and several images may be combined into one spectral data set and be analyzed in tandem [9].

A major problem with MCR-ALS is that it becomes slow and memory-intensive when working with large amounts of data. In conversations with C. Troein (2022), Anderson acceleration was brought up as a way to speed up MCR-ALS; the algorithm reduces the number of times a time consuming step is taken by making a polynomial prediction, based on previous steps, to predict subsequent steps. Another possibility is to utilize the fact

that there are many spectra which are similar to other spectra in the spectral data. We call groups of similar spectra 'clusters'. Thus, clusters may be merged into a single representative spectrum, reducing the amount of data in the data set. A way to systematically find representative spectra is to use a clustering algorithm which tries find data points, centroids, that minimizes a certain criterion such as distance or variance [6, 10]. K-means is such an algorithm which also can be used for image segmentation [6, 11], where MCR-ALS is run separately on each segment.

Despite the promise of less memory usage and quicker process, problems arise when clustering on the data; there are several aspects we must consider apart from time and memory requirement. These aspects are how the result can be translated to its original shape and what the impact is compared to the original result. Regarding the aspect of time and memory-usage, there is an important question: how well is the data preserved given an original decomposition.

Consequently, we have developed a method of compression and reconstruction of spectral data. The performance in several aspects for different choices of parameters has been evaluated by using and modifying code from the OCTAVVS project [1] and writing complementary code. To have a reliable baseline, the goal of the method is to recreate the result from MCR-ALS without clustering as closely as possible. Figure 1 depicts the essence of the method we call clustering-accelerated MCR-ALS (CAM-A). In section 2, CAM-A is presented along with descriptions and definitions for its different parts along with corresponding parameters. The found effects on performance and time consumption are presented and discussed in section 3. Lastly, we conclude the effects of different choices of methods and present an optimal set of choices; further investigations on how to potentially improve CAM-A are also discussed.

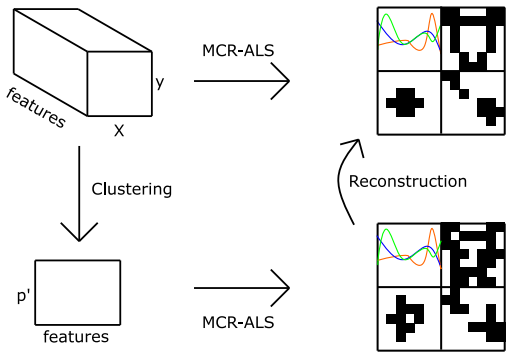

Figure 1: A schematic portraying the essence of the CAM-A method. p' represents the number of centroids used for reducing the dimensionality of the original hyperspectral image.

2 Theory and method

2.1 MCR-ALS

Multivariate-Curve-Resolution (MCR) is a framework where spectral data, such as hyperspectral images, is attempted to be expressed as a distribution of underlying pure spectra without any premises on the information in the images, and a variety of different MCR methods are commonly used in chemistry [1, 2]. Mathematically, the hyperspectral data can be represented by a matrix, D, with dimensions (pixels $\times$ features), the pure spectra by a spectral matrix, S, with dimensions (features $\times$ components), and the pure spectra-contributions by a contribution matrix, C, with dimensions (pixel $\times$ components). Using these matrices, the bilinear MCR model can be described by the following equation:

$$D = CS^T + E, \tag{1}$$

where E is a correction matrix which makes up for what the product CS^T misses [2]. Because we want to express the data as effectively as possible, we want to minimize the loss, L, given by the square Frobenius norm of E:

$$L = \sum_{ij} E_{ij}^2 = ||E||^2 = ||D - CS^T||^2. \tag{2}$$

There are several ways to achieve such a decomposition, and we use non-negative Multivariate Curve Resolution-Alternating Least Squares (MCR-ALS). The non-negativity condition is enforced on C and S, but may only need to be enforced on the spectral matrix. "Alternating Least Squares" in the name comes from the fact MCR-ALS is alternating between updating the C and S matrix using a least squares method, expressed mathematically as

$$C = \underset{C}{\operatorname{argmin}} \, ||D - CS^T|| \quad \text{or} \quad S = \underset{S}{\operatorname{argmin}} \, ||D - CS^T||, \tag{3}$$

to minimize (2) using the S or C matrix respectively. Because the minimization needs one of the matrices, an initial postulation of either a C_{init} or an S_{init} is required. Consequently, the loss is improved, or equal, after each update in MCR-ALS. The non-negativity condition is fulfilled by setting the following non-negativity conditions on C and S when updating them:

$$S_{ij} \geq 0 \quad \text{and} \quad C_{ij} \geq 0. \tag{4}$$

The conditions given in equation (4) necessitates the use of a non-negative least squares (NNLS) algorithm, and we use the *scipy.optimize.nnls* implementation. NNLS is iterative and must be run for each pixel in the image separately [12]. Compared to standard least

squares (LS), NNLS takes a substantially longer time. A dimensional reduction of the data, fewer spectra, in a data set consequently reduces the time required for a decomposition.

MCR-ALS is an iterative method and can breifly be described as follows:

1. Postulate an initial C or S.
2. Find a new S or C using D and the current C or S by minimizing equation (2) as dictated by equations (3) and (4).
3. Find a new C or S using D and the newly found S or C by the same process as for step 2.
4. Repeat step 2-3 using the C or S found in step 3 until a stopping criterion has been reached.
5. Return S and C from step 4.

Furthermore, a weakness of MCR methods overall, including MCR-ALS, is that new solutions for C and S can be obtained by introducing an invertible transformation-matrix, A. By defining $C' = CA$ and $S' = S(A^{-1})^T$, the product between C' and S'^T becomes:

$$C'S'^T = CAA^{-1}S^T = CIS^T = CS^T, \tag{5}$$

where the loss is consequently equal [2]. These kinds of solutions are, mathematically speaking, rotations of each other in high-dimensional spaces [2, 7, 8], and are therefore called rotations.

Because of the possibility of rotations in the solutions, C and S vary depending on S_{init} and C_{init}, but, due to the fact that the loss cannot increase each update, C and S are also susceptible to getting caught in a local minimum. Thus, the loss may vary between different initializations.

In this project, MCR-ALS is always initialized using an S_{init} generated from the SIMPLISMA algorithm. SIMPLISMA is a widely used and deterministic algorithm for generating initial spectra [13]. Thus, the research carries relevance to actual usage and the effects caused by rotations originating from the choice of initial S_{init} are eliminated.

The implementations of both the SIMPLISMA and MCR-ALS algorithms used are taken from the OCTAVVS project [1]. However, the code for the MCR-ALS algorithm is altered to take a weight-vector as an argument to enable MCR-ALS to decompose a compressed image correctly.

2.2 Overarching description of CAM-A

CAM-A can be split into four different steps: the pre-processing, clustering, decomposition, and reconstruction step, and their relation is depicted in figure 2. In the pre-processing step, the image is represented as a matrix D where each row-vector corresponds to a pixel in the image. The spectrum in each row may be L2-normalized, creating the normalized matrix $\bar{D}$. L2-norm is chosen to remove the importance of the intensity of the spectra when clustering, and to make their shapes the critical feature. An initial spectral matrix S_{init} is generated from D using SIMPLISMA.

A matrix, Q, whose rows are equal to centroids obtained by clustering on the rows of D (pixels) is constructed in the clustering step. A weight vector, w, whose elements are the weights for each centroid is determined concurrently. How the weights are determined is found in section 2.3. Q is then used to construct a new compressed image, D', where the rows in D' are rows in Q weighted using the weights in $\boldsymbol{w}$ as described in section 2.4.

In the decomposition step, MCR-ALS is run on D' to determine a contribution matrix for the centroids, C', and a matrix of the component spectra, S'. To recreate a C from an original decomposition as accurately as possible, a $\hat{C}$ is constructed using C' with a reconstruction method of choice presented in section 2.5; this is the reconstruction step. A reconstruction method may use both the centroid matrix Q and image D to create a reconstruction matrix V for reconstruction, if V is necessitated by the method. $\hat{C}$ and S' are multiplied to recreate a new image $\hat{D}$ used to calculate the loss.

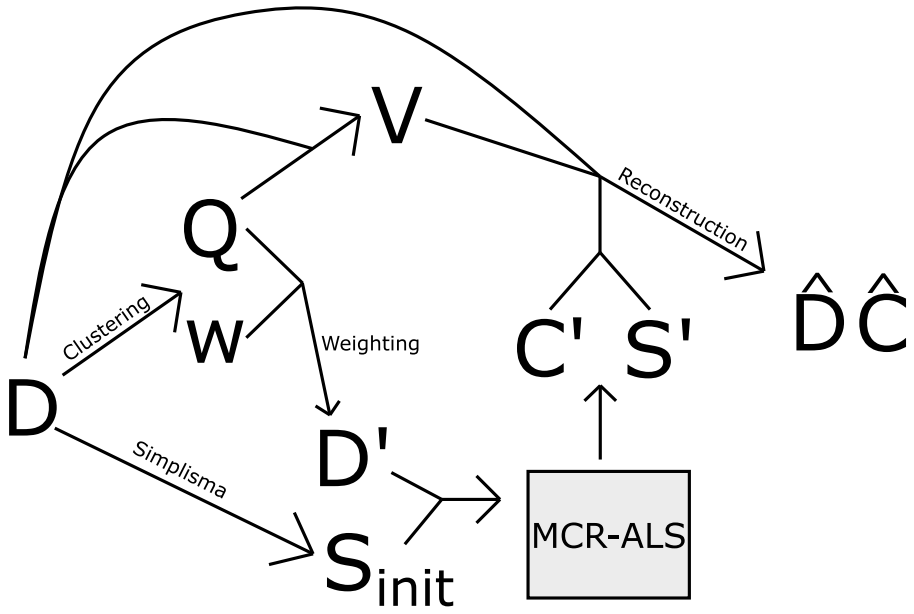

Figure 2: A flowchart of the CAM-A method illustrating the description of the steps in section 2.2

2.3 K-means clustering

In section 2.1, it was stated that we may be able to reduce the time required for decomposition by reducing the dimensionality on a spectral data-set. The clustering algorithm k-means is used for dimensional reduction as it can find so called centroids to represent spectra in a spectral data-set efficiently using fewer spectra. K-means is chosen in particular due to its simplicity compared to other clustering algorithms [14]. The implementation of k-means and its initializations are from the module *sklearn.cluster.KMeans* from the *scikit-learn* library [15].

To explain k-means, define the set of all centroids, $\boldsymbol{q}_k$, and the set of all spectra, $\boldsymbol{d}_i$, respectively, in a spectral data set by:

$$\mathcal{Q} = \{\boldsymbol{q}_k\}, \mathcal{D} = \{\boldsymbol{d}_i\}, \tag{6}$$

where $\boldsymbol{q}_k$ and $\boldsymbol{d}_i$ are the row-vectors of Q and D, respectively. The objective function, ϕ, for k-means is given by:

$$\phi(C, D) = \sum_{\boldsymbol{d}_i \in \mathcal{D}} \min\{(\boldsymbol{d}_i - \boldsymbol{q}_k)^2, \boldsymbol{q}_k \in \mathcal{Q}\}, \tag{7}$$

which measures how 'poorly' a set of centroids correspond to the data. By iterative updates, the k-means algorithm tries to minimize the value of ϕ for the given $\mathcal{Q}$ [10, 14]. Consequently, the update is as in equation (9), where a set, $\mathcal{A}$, for each $\boldsymbol{q}_k \in \mathcal{Q}$ is by

$$\mathcal{A}_{\boldsymbol{q}_k} = \{\boldsymbol{d}_i : (\boldsymbol{d}_i - \boldsymbol{q}_k)^2 = \min\{||\boldsymbol{d}_i - \boldsymbol{q}_k||^2, \forall \boldsymbol{q}_k \in \mathcal{Q}\}. \tag{8}$$

$\boldsymbol{q}_k$ is updated each iteration as follows:

$$\frac{1}{|\mathcal{A}_{\boldsymbol{q}_k}|} \sum_{\boldsymbol{d}_i \in \mathcal{A}_{\boldsymbol{q}_k}} \boldsymbol{d}_i \to \boldsymbol{q}_k, \tag{9}$$

and new $\mathcal{A}_{\boldsymbol{q}_k}$ are generated for the new $\boldsymbol{q}_k$ as given in equation (8), and the updates are repeated until a stopping criterion is reached. When the stopping criterion has been reached, a weight-vector, $\boldsymbol{w}$, is created. The weight-vector's elements are defined as

$$\boldsymbol{w}_k = ||\mathcal{A}_{\boldsymbol{q}_k}||. \tag{10}$$

There are several ways to choose an initial set of centroids, $\mathcal{Q}$. The 'Points' initialization works by sampling d_i from $\mathcal{D}$ with equal probability for each d_i to be chosen once at most. '++' is found to improve the accuracy of K-means; the initialization starts by sampling one $\boldsymbol{q}_1$ like as for 'Points' and then choosing a new $\boldsymbol{q}_k$ from X with a probability proportional to the square of the minimal distance from $\boldsymbol{d}_i$ to any $\boldsymbol{q}_k \in \mathcal{Q}$ until a specified amount

of centroids has been chosen [10]. Because the initializations we chose are stochastic, the compressed image, D', is not equal each time it is compressed, resulting in different solutions.

2.4 Weighting of centroids

When the centroid matrix, Q, has been found, we want an MCR-ALS decomposition on D' to be equivalent to a decomposition of $\tilde{D}$ where the spectrum in each pixel is replaced by the centroid the pixel belongs to. To achieve the equivalence, the following weighting, version of D', and loss are used:

$$D'_{k,j} = \sqrt{w_k} Q_{k,j} \quad \text{and} \tag{11}$$

$$L = ||E||^2 = \sum_{k,j} w_k (Q_{k,j} - \boldsymbol{C'}_k \bullet \boldsymbol{S'}_j)^2 = \sum_{k,j} (D'_{k,j} - \sqrt{w_k} \boldsymbol{C'}_k \bullet \boldsymbol{S'}_j)^2, \tag{12}$$

where k and j respectively denote row and column number, and $\boldsymbol{C}_k$ and $\boldsymbol{S}_j$ are row and column matrices respectively. In appendix A.1, the proof for equivalent loss between D' and $\bar{D}$ is shown. It is used for convenience of implementation, but comes with the requirements that we must take the weighting into account when determining C', and consequently we must initialize MCR-ALS with S_{init}.

2.5 Reconstruction methods

When we have found C' and S', the contribution matrix for the centroids and pure spectra matrix, respectively, we used them with a reconstruction method to create a reconstruction of C, $\hat{C}$. The general formula for the construction of $\hat{C}$ is:

$$\hat{C} = VC', \tag{13}$$

and V is called a reconstruction matrix. An important property which V must abide by is that $\hat{C}_{i,j} \geq 0$. Using V, the reconstruction loss is defined as:

$$L_{\text{recon}} = ||E_{\text{recon}}||^2 = ||D - VQ||^2, \tag{14}$$

which is a measure of how well the centroids represent the image given a certain reconstruction method.

For the simplest reconstruction method, 'Simple', the elements in V are defined as following:

$$V_{i,k} = \begin{cases} 1 & \text{if} \quad \boldsymbol{D}_i \in A_{q_k} \\ 0 & \text{else} \end{cases} \quad \text{or} \quad V_{i,k} = \begin{cases} ||\boldsymbol{D}_i|| & \text{if} \quad \bar{\boldsymbol{D}}_i \in A_{q_k} \\ 0 & \text{else} \end{cases}, \tag{15}$$

where $\boldsymbol{D}_i$ and $\bar{\boldsymbol{D}}_i$ are the i:th row vectors in respective matrices.

We assume that the pixels in the image consist of positive linear combinations of different component spectra already when using MCR-ALS. By that assumption, the component distributions describing the pixels should be possible to describe as linear combinations of the distributions describing the centroids in the same way as the pixels can be described by the centroids. Ergo, a reconstruction method 'NNLS' is created, and it finds a V_{NNLS}, given by minimizing the value:

$$V = \underset{\mathcal{V}}{\operatorname{argmin}} \, ||D - \mathcal{V}Q|| : \mathcal{V}_{i,j} \geq 0, \tag{16}$$

to ensure a positive contribution matrix. Another reconstruction method tested is what we call 'LSW', short for "least squares weighting". It follows the same argument, but allows the contribution from a pixel to be negative. Moreover, the negative contributions are limited by the condition that $\hat{C}$ must be non-negative. How 'LSW' finds V_{LSW} is now described. V_{LS} is be the $\mathcal{V}$ minimizing the square norm $||D - \mathcal{V}Q||^2$ and V_{NNLS} the matrix minimizing the expression in equation (16). V_{LSW} is accordingly given by

$$\boldsymbol{V}_{\text{LSW},i} = (1 - \lambda_i)\boldsymbol{V}_{\text{NNLS},i} + \lambda_i \boldsymbol{V}_{\text{LS},i} \ , \tag{17}$$

where λ_i is the maximal inclusion of $V_{\text{LS},i}$ which conserves the non-negativity condition on $\hat{C}$, and $\boldsymbol{V}_{\text{LSW},i}$, $\boldsymbol{V}_{\text{NNLS},i}$, and $\boldsymbol{V}_{\text{LS},i}$ are the i:th row vectors in the respective matrices. λ_i is defined as follows:

$$\lambda_{i,j} = \begin{cases} -\frac{\boldsymbol{V}_{\text{NNLS},i} \cdot \boldsymbol{C'}_j}{(\boldsymbol{V}_{\text{NNLS},i} - \boldsymbol{V}_{LS,i}) \cdot \boldsymbol{C'}_j} & \text{if} \quad (\boldsymbol{V}_{\text{NNLS},i} - \boldsymbol{V}_{\text{LS},i}) \cdot \boldsymbol{C'}_j < 0 \\ 1 & \text{else} \end{cases}, \tag{18}$$

$$\lambda_i = \min(\{\lambda_{i,j} : \ 1 \leq j \leq J\} \cup \{1\}), \tag{19}$$

where J is the number of components and λ_{ij} is the maximal contribution of $\boldsymbol{V}_{\text{LS},i}$ in $\boldsymbol{V}_{\text{LSW},i}$ allowed by component j. The derivation is found in appendix A, where it also is shown that $\hat{C}$ always is positive by construction.

Another reconstruction method, called 'Strong', is to find $\hat{C}$ is by taking a 'half-step' of MCR-ALS:

$$\hat{C} = \underset{\Gamma}{\operatorname{argmin}} ||D - \Gamma(S')^T||, \ \Gamma_{ij} \geq 0. \tag{20}$$

The disadvantage of using the 'Strong' reconstruction method is that no reconstruction matrix V is created, and consequently a reconstruction loss cannot be determined, making it difficult to measure how efficiently it preserves information in an image.

2.6 Comparison loss

An error measure we call 'comparison loss' is designed to measure how similar and rotated the row vectors in $\hat{C}$ from CAM-A are compared to the row-vectors in C_{orig} from the original MCR-ALS process. A measure which 'greedily' determines the best comparison of rows of C_{orig} and $\hat{C}$ was implemented due to the number of possible permutations of the rows being proportional to $n!$, where n is the number of components. The procedure is as follows:

1. Divide all elements of the rows in $\hat{C}$ or C_{orig} with the largest feature in the corresponding row of S' or S_{orig}, respectively. S_{orig} is the spectra from MCR-ALS without clustering.

2. Choose a random sequence, $\mathcal{I}$, of the rows in C_{orig}, with equal probability for each sequence.

3. Pair the first row of the sequence that has not yet been paired with the row in $\hat{C}$ with a row in $\hat{C}$ which minimizes $||\boldsymbol{C}_{\text{orig},i} - \hat{\boldsymbol{C}}_k||$, where i and k are row numbers, and has not been paired with a row in C yet.

4. Repeat step 3 until all rows have been paired. Construct a new matrix $\hat{C}_{\text{opt}}$ where $\hat{\boldsymbol{C}}_{\text{opt},i}$ is the $\hat{\boldsymbol{C}}_k$ paired with $\boldsymbol{C}_{\text{orig},i}$

5. The partial comparison loss is given by $L_{c,p} = \frac{1}{|\mathcal{I}|} \sum_{i=1}^{|\mathcal{I}|} \frac{||\boldsymbol{C}_{\text{orig},i} - \hat{\boldsymbol{C}}_{\text{opt},i}||^2}{||\boldsymbol{C}_{\text{orig},i}||^2}$.

6. Repeat step 2-5, and calculate the the comparison loss as the mean value of all $L_{c,p}$ from each repetition, $L_c = \frac{1}{P} \sum_{p=1}^{P} L_{c,p}$, where P is the number of repetitions.

The number of repetitions in step 4 is a parameter that must be set; the more repetitions, the more precise the value becomes. Another thing to note is that the value of L_c is punished by rotations. The reconstruction may produce a solution close to the loss from an original MCR-ALS decomposition, $||E_{\text{orig}}||^2$, but a rotation between any components may give rise to a large $L_{c,p}$ despite that, increasing the value of L_c. Another aspect is

that due to the stochastic nature of L_c, the measure fluctuates more between runs when L_c is large, both caused by a greater discrepancy between C_{orig} and $\hat{C}$ allowing for a wider variety of pairings. Thus, it is a more qualitative measure when L_c is large, but can used quantitatively when L_c is small due to steps 2 and 3 suppressing the stochastic nature when $\hat{C}_k$ is similar to C.

2.7 Data and measurement

We chose 3 hyperspectral images containing infrared absorption spectra from measurements using different spectroscopic methods on different biological samples as sources. A variety of sources and methods allows for a greater generalizability of the results.

The loss, reconstruction loss, and comparison loss for a combination of choices is not equal each run due to the stochastic nature of the initializations of k-means. Similarly, fluctuations in time consumption could occur since the environment was not entirely isolated from other processes. Hence, the average losses and time consumptions were averaged out over 40 runs with the same parameter settings.

Furthermore, there were termination-conditions dictating when both k-means and MCR-ALS should terminate for the measurements of the performance parameters presented in section 3. k-means was set to terminate when either $||Q_{i-1} - Q_i|| \leq 10^{-5}$ or reaching Q_{200}, where Q_i is the ith iteration. MCR-ALS was also run for 200 iterations. Additionally, 100 different $L_{\text{c,p}}$ were used to calculate L_c each run of CAM-A.

2.8 Hardware and software

CAM-A was implemented in the programming language *Python*, version 3.6, and the code for the implementation is found in appendix B.2. The modified MCR-ALS code is found in appendix B.1. Libraries and their respective version used in the implementation are shown in table 1. The CPU used is an AMD Ryzen Threadripper 3990X 64-Core processor.

Library	Version
statsmodels	0.12.2
numpy	1.17.2
scipy	1.5.4
scikit-learn	0.24.2

Table 1: Python libraries used and their respective versions.

3 Results and discussion

CAM-A was run using different combinations of choices of pre-processing, number of components, reconstruction methods, and hyperspectral images to investigate the interaction between the choices and to isolate respective effects on rotations, loss, time consumption, etc. The combinations of choices were tested for 3 images we named "Ex_90", "Tumor", and "PAI".

3.1 Reconstruction methods

We examined the effects on the loss and reconstruction loss using different reconstruction methods and initializations of k-means to find which of respective choice gives the least respective errors and why. The results of the examination when using 4 components and the image "Ex_90" is presented in figure 3. "Ex_90" was selected because it exemplifies the general behavior, and 4 components was chosen to more clearly display the difference between the 'LSW' and 'NNLS' reconstruction methods. In section 3.4, how the number of components and image influence 'LSW' is presented.

Among the reconstruction methods, 'Strong' reconstruction consistently yields the lowest loss for a given number of centroids when chosen, meaning it reconstructed the original image better than the other methods. The 'LSW' reconstruction method yields a loss like the 'Strong' reconstruction for 4 components in figures 3**C** and 3**D**. In contrast, the 'NNLS' and 'Simple' reconstruction methods gave larger losses, of which 'Simple' yields the largest.

'Strong' reconstruction yields the lowest loss because of its construction. The reconstructed contribution matrix, $\hat{C}$, in equation (20); the equation yields the $\hat{C}$ minimizing the difference $||D - \hat{C}(S')^T||^2$ within the boundary conditions, for a given spectra matrix after decomposition, S', and an image D. The other reconstruction methods do not necessarily give the same $\hat{C}$. Consequently, the loss may only be equal or larger for the other reconstruction methods.

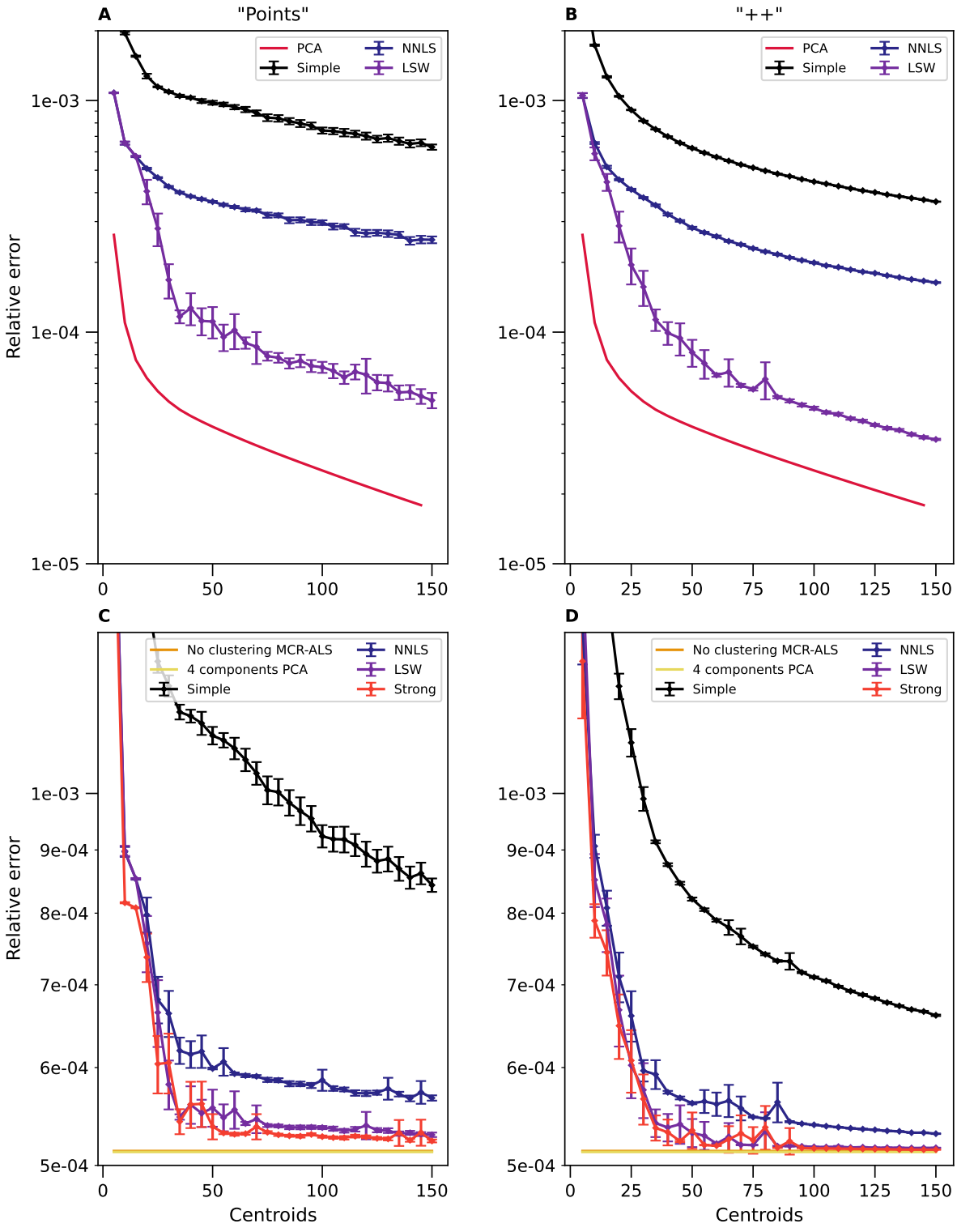

Figure 3: Loss and reconstruction loss of different reconstruction methods and intializations of k-means, with 95% confidence intervals. The relative error is given by $||E_{\text{recon}}||^2/||D||^2$ in (**A**) and (**B**) and $||E||^2/||D||^2$ in (**C**) and (**D**). "PCA" in (**A**) and (**B**) is the unexplained variance against number of components.

'Simple' reconstruction performs poorly, compared to the other methods, in terms of both the loss and reconstruction loss because k-means group spectra of varying intensities together. When $\hat{C}$ is constructed using 'Simple', it is presumed that all pixels belonging to a centroid are built up by the constituent spectra the exact same way as the centroid is in the compressed contribution matrix, C', including the amplitudes of the contributions of component spectra. Consequently, many pixels in the reconstructed image, $\hat{D}$, have an incorrect intensity. Since the losses defined in equations (12) and (14) are dependent on the absolute values of the features, the incorrect intensities cause the errors in representation.

The 'NNLS' reconstruction method generates a substantially larger loss and reconstruction loss than 'LSW' and does so because of a more limited subspace of linear combinations of centroids. Only positive linear combinations of centroids are allowed to be used for reconstruction of the pixels, and the possible reconstructions are limited to a subspace whose 'borders' are defined by the centroids, see figure 4. The reconstructions of pixels lying outside the 'borders' of the subspace are projections of the pixels onto the subspace.

Moreover, because the centroids defining the borders are the mean-values of the pixels belonging to them, some pixels will always be outside the 'borders' independently of how many centroids are used. The same effect also causes an increment in the number of centroids to give diminishing returns, as the 'push' the centroids exert on each other decrease due to a screening-effect caused by the pixels belonging uniquely to their closest centroid.

Reconstruction using the 'LSW' method allows linear combinations of centroids representing pixels to contain negative contributions from centroids, as long as the contributions in $\hat{C}$ are positive. How the subspace is expanded is depicted in figure 4. Thus, more of the pixels lying close to the borders are captured, allowing for a more accurate representation of the pixels; consequently decreasing the loss and reconstruction loss.

Figure 4 is an illustration of subspace created by positivity condition of the 'NNLS' reconstruction method when only two features are present. Black lines indicating the span of the subspace have their direction determined by the centroids, which the lines pass through. A red line in the figure illustrates how 'LSW' can expand the subspace, and the green line the new 'border' replacing the 'border' which the red line emerges from.

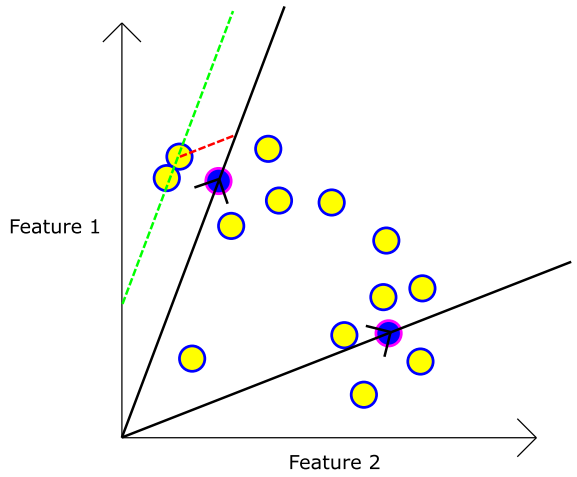

Figure 4: An illustration of the limitation of 'NNLS'-reconstruction (black lines) and how 'LSW'-reconstruction decrease the limitation (green dashed line). Centroids are represented by blue dots with purple rings and spectra by yellow dots with blue rings.

3.2 Initialization of k-means

The '++' initialization of k-means yields a smaller loss and reconstruction loss than 'Points' independently of number of components, reconstruction method, normalization, and image. More of an image is captured by a centroids when using '++', exemplified by the steeper slope of reconstruction loss as a function of centroids in figure 3**B** compared to 3**A** for all reconstruction methods. The loss also follows the same pattern, exemplified by a comparison between figure 3**D** and 3**C**.

Compared to 'Points' initialization, '++' initialization causes k-means to generally produce better centroids in terms of minimizing the square-Euclidian distance [10]. Thus, the spectra represented by the centroids are more similar each other, and can represent more of the image, when initializing with '++' rather than 'Points'. MCR-ALS, consequently, has more variation between the spectra to work with when '++' is chosen, despite the number of centroids not changing. Thus, $\hat{C}$ and S' from the decomposition represent the image better since more nuances in the original image are caught.

Additionally, because the probability of choosing a pixel as an initial centroid is proportional to the square distance to the closest centroid, the centroids are, on average, spread out further from each other in spectra-space when using '++' compared to 'Points'. Consequently, the greater spread causes the subspaces restricting the 'NNLS' and 'LSW' reconstruction methods to span a greater subspace in comparison. The centroids are also more distributed according to their respective intensity, causing the pixels belonging to a centroid to be more uniform in intensity, further decreasing the loss when using the 'Simple' reconstruction method.

3.3 Normalization of spectral data

We investigated the difference in reconstruction loss and loss when normalizing the image before clustering for all reconstruction methods to determine how normalization interacts with the different steps of CAM-A. From the investigation, the result of CAM-A decompositions of image "Ex_90", both normalized and unnormalized, using 4 components and initializing k-means with '++' is presented in figure 5. "Ex_90" was chosen to portray the general case, '++' because it is the most error-efficient initialization of k-means and does not interact with normalization, and 4 components to differentiate 'LSW' from 'NNLS' more clearly, as further discussed in section 3.4. A peculiar difference between figure 5**A** and 5**B** is that the reconstruction loss for the 'LSW' and 'NNLS' is smaller when the image is normalized, but their loss is larger; the loss also is larger for the 'Strong reconstruction method. In contrast, 'Simple' reconstruction has a smaller loss and reconstruction loss when the image is normalized.

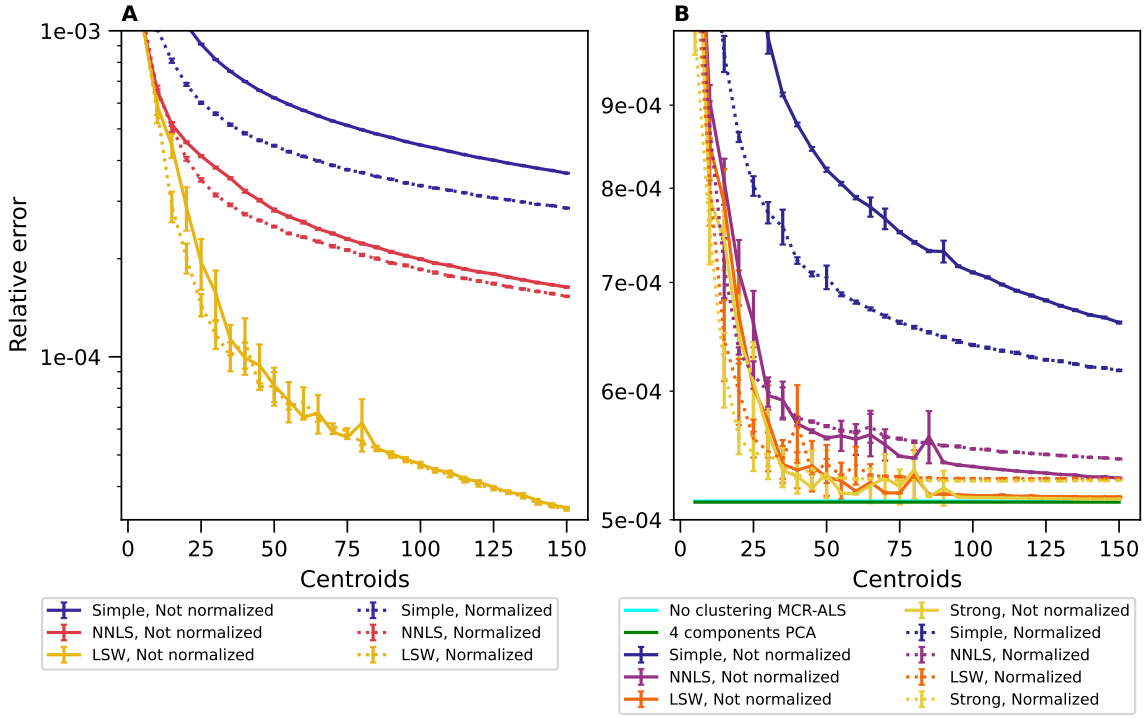

Figure 5: Loss and reconstruction loss when normalizing not normalizing D before clustering, with 95% confidence intervals. The relative error on the y-axis is given by $||E_{\text{recon}}||^2/||D||^2$ in (**A**) and $||E||^2/||D||^2$ in (**B**). Each color represent a reconstruction method, and dashed lines represent losses when normalizing the image.

The reason the reconstruction loss is smaller when normalizing is that the centroids have a greater variety of shapes after k-means on the original image, D. k-means is very sensitive to the intensity of the spectra due to it minimizing the square-Euclidian distance each iteration. Therefore, it groups spectra after both shape and intensity. When clustering on a normalized image, $\bar{D}$, the intensities of all the spectra are the same, and they are only grouped after their shapes. Therefore, a wider range of shapes of centroids are obtained when clustering on $\bar{D}$.

For the 'Simple' reconstruction, the shapes of the centroids are more like the shapes of the pixels belonging to the centroid due the shapes being be less mixed. The centroids are brought to an intensity of the same magnitude as respective original spectrum they represent in the reconstruction by the reconstruction method. Consequently, the reconstruction loss is lower because the greater variety of shapes of centroids allows the reconstruction methods to create spectra more similar to the original pixels in D' and captures a greater portion of the lower intensity spectra.

The normalization removes the importance of the image's higher intensity spectra in MCR-ALS. Thus, the MCR-ALS decomposition distributes the importance more evenly between the spectra during decomposition. Therefore, C' and S' will be more adapted to explain the lower intensity spectra more precise, but the higher intensity less precise. The loss, L, is more susceptible for changes in high-intensity spectra and therefore is larger, demonstrated by the graphs in figure 5**B** for 'NNLS', 'LSW', and 'Strong'.

3.4 Components

An investigation on how whether using 8 or 4 components the loss for a CAM-A decomposition compares to an original decomposition was conducted. Our aim was to uncover how information in the image is distorted and to find interactions between the choice of reconstruction method and number of components. The findings from the investigation are presented in figure 6, for the case of '++' k-means initalization. The 'points' initalization gave similar but less clear results.

More components yielded a lower loss invariably of the other choices. However, the difference in the relative error between the CAM-A loss and the original decomposition is larger. Additionally, the loss using 'LSW' reconstruction is increasing with the number of centroids in figure 6**B**; the increase is not present for 8 components. For all images, the 'LSW' reconstruction method yields a loss close to the 'NNLS' reconstruction method when decomposing the image into 8 consituent spectra.

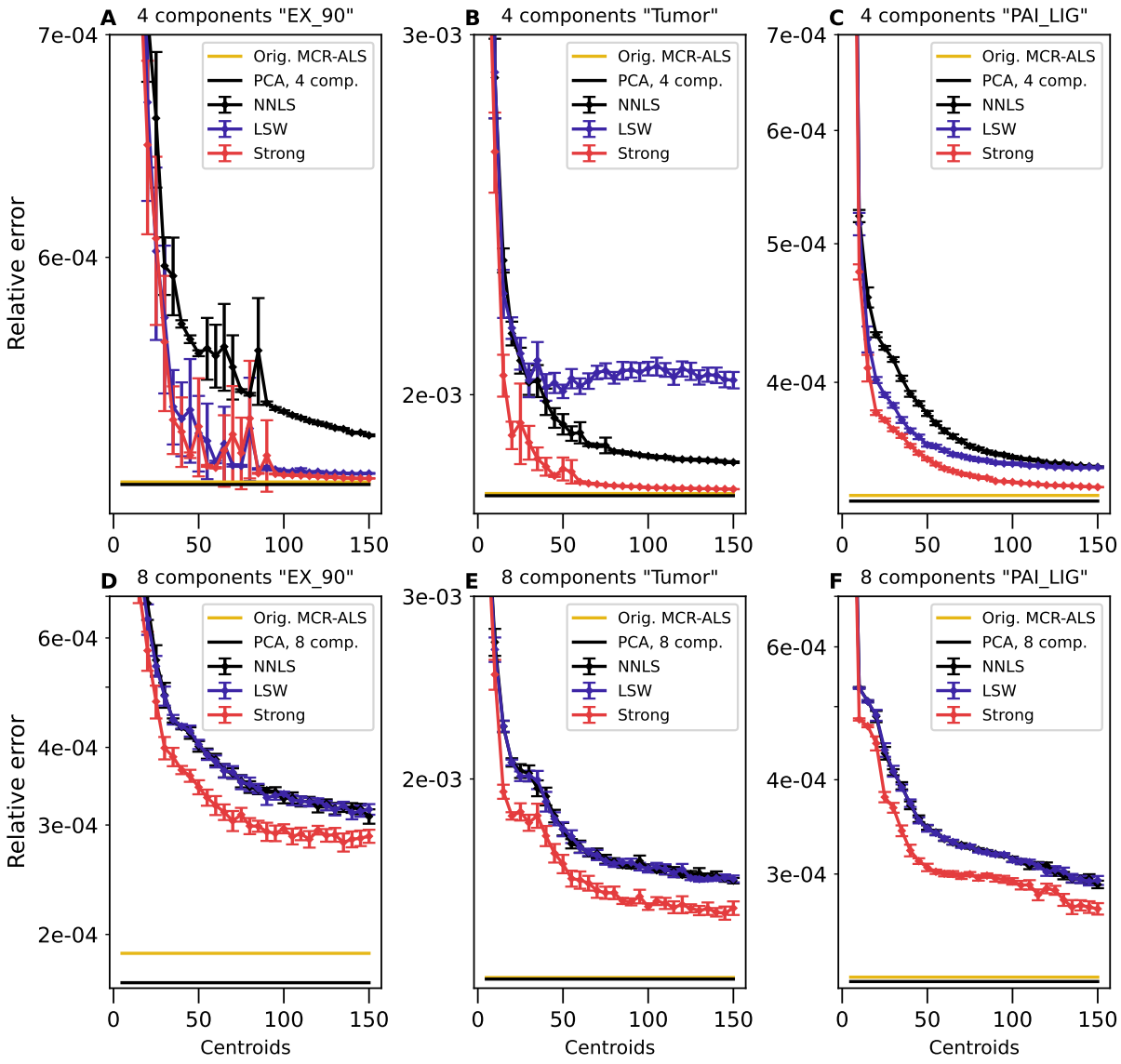

Figure 6: The relative error, $||E||^2/||D||^2$, for different reconstruction methods, number of components, and images. The confidence intervals are 95%.

The larger difference in relative error of the original MCR-ALS and reconstruction methods with more components is a consequence of that the original MCR-ALS can account for components constituting a smaller portion of the image. Thus, the decomposition can account for more of the image when more components are used, reducing the loss. However, when clustering, the subtler components are not necessarily conserved in D', due to being overshadowed in the clustering-step by the more prominent components. Therefore, MCR-ALS cannot use the subtler components to describe the image, and the loss is larger.

The loss, L, using the 'LSW' method is more like the 'NNLS' method when using 8 components compared to 4 due to an more constraints limiting the reconstruction. 'LSW'

reconstruction has an equal number of components and $\lambda_{i,j}$. The value λ_i is the smallest $\lambda_{i,j}$, and λ_i is then likely be smaller with more components. Thus, the incorporation of the V_{LS} matrix into the V_{LSW} matrix is also likely to be smaller with more components, making the V_{LSW} matrix increasingly alike the V_{NNLS} matrix. Consequently, the 'LSW' reconstruction becomes more like the 'NNLS' reconstruction when increasing the number of components.

3.5 Rotations

We examined whether rotations could occur when using CAM-A relative the original decomposition, and which choices may cause rotations. A baseline is given when 4 components are used, the image is not normalized, and the '++' clustering initialization. This baseline is chosen to study the effect on comparison loss, defined in section 2.6, when changing either the number of components to 8, clustering on a normalized image, $\bar{D}$, or k-means initialized with 'Points'. The choice of parameters in the baseline gave the smallest reconstruction error. Any choice of reconstruction could not be determined to influence the comparison more than the margin of error, and 'Strong' reconstruction was therefore chosen as a representative reconstruction method in the baseline. The results for 2 different images is presented in figure 7 as the consequences for the comparison error of changing different parameters may be different depending on the image.

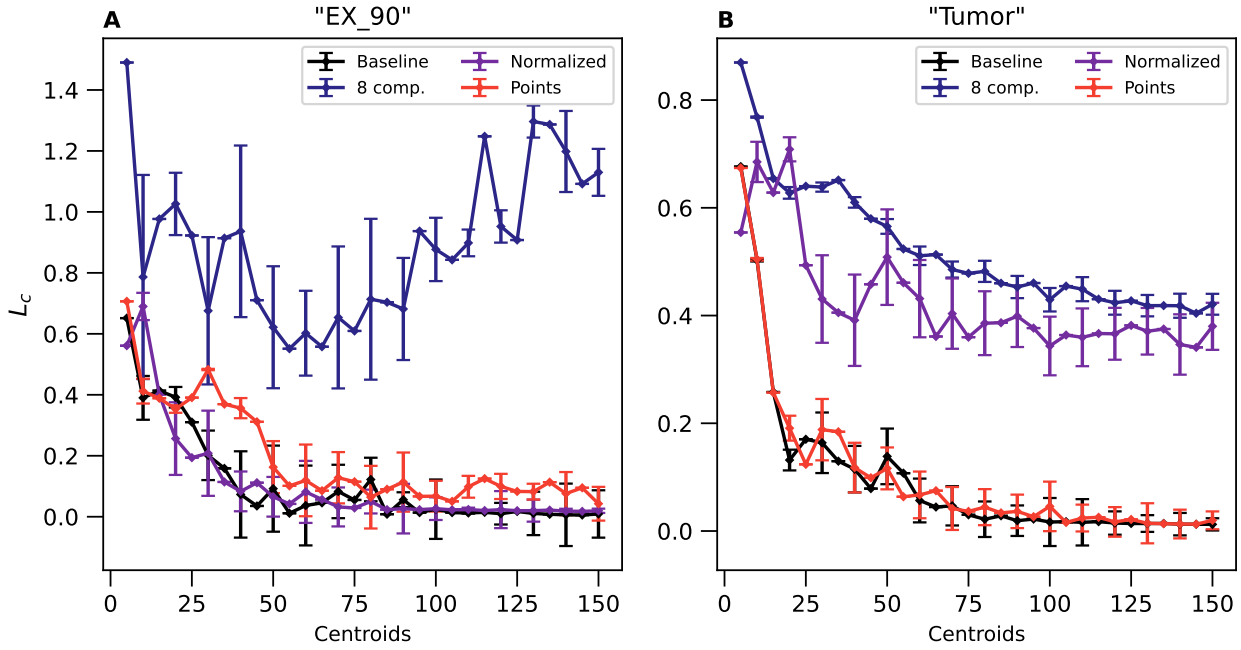

Figure 7: The comparison loss, measuring rotation relative original MCR-ALS decomposition, by choice of parameters and image, with 95% confidence intervals. The baseline is a set of parameters chosen because they give a low comparison loss in all images.

Normalization causes a larger comparison loss compared to the baseline for the image "Tumor" but not for the image "Ex_90". Likewise, the comparison loss is larger when initializing using 'Points' for image "Ex_90" but not for image "Tumor", see figure 7. A decomposition by CAM-A is more rotated with more components, exemplified by the larger comparison loss decomposing an image into 8 components compared to the baseline in figures 7**A** and 7**B**.

Normalization of D may cause a rotation in the data as the loss dictating how MCR-ALS decomposes an image is sensitive to intensities of the spectra. Hence, changing the intensities will alter the decomposition. However, normalization does not cause a rotation for "Ex_90", indicating that the effect is image dependent.

More components increase the probability of rotations in the MCR-ALS due to a greater subspace of transformations. The compressed image, D', consist of centroids and do not represent the original image, D, perfectly. Thus, C' and S' will be rotated, and consequently $\hat{C}$ to also be rotated. Thusly, the comparison error is larger with more components.

Also, the centroids from k-means clustering removes information about components explaining less of the image making it more difficult for MCR-ALS to properly distinguish between them. Hence, the weak components are prone to be 'mixed' into other components; this is equivalent to a rotation. More weak components are present when more components are used, which increases this effect. Thus, more components results in a larger comparison loss.

A larger number of centroids allows D' to better represent D. Hence, the rotations and comparison loss decrease with centroids, and do so invariably of choices for the steps of CAM-A. The decrease in comparison loss with centroids is present in the data presented in figure 7.

3.6 Time consumption

CAM-A reduces the total time-requirement for decomposition of the images invariably of the number of components used, etc, within the range of centroids used. The different parts of CAM-A are also scaling differently with the number of centroids, see figure 8, where the time-requirement for the different parts of CAM-A is represented using image "Ex_90". To compare the total time-requirement of CAM-A and its individual segments, the quotient between respective time-consumptions and the time-consumption for the original MCR-ALS decomposition, T_0, is presented in figure 8.

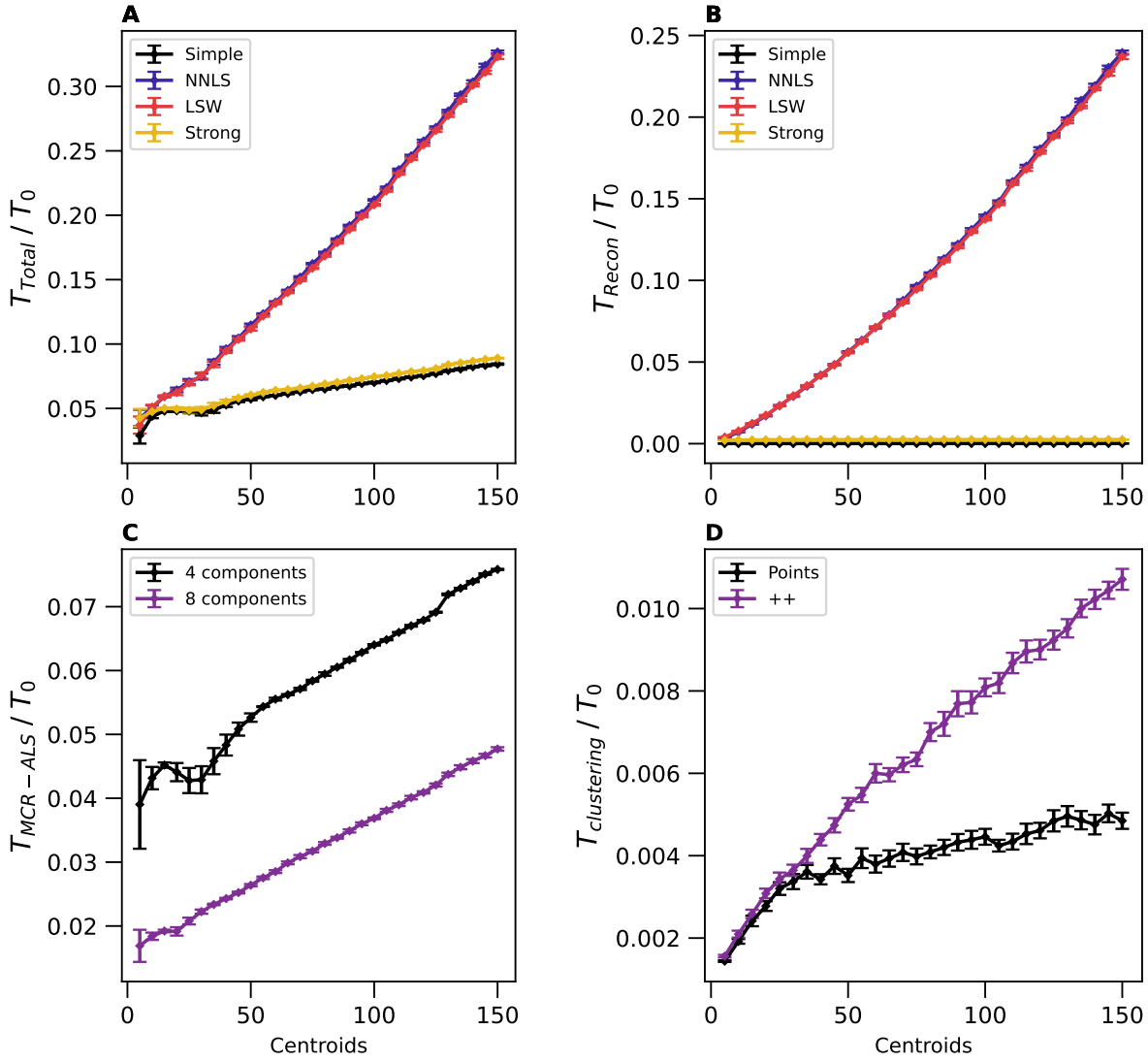

Figure 8: Time-requirements for the whole CAM-A process and the different steps for some choices of parameters. The confidence intervals are 95%. (**A**,**B**,**D**): 4 components. (**A-C**): Clustering is initialized using '++'. (**D**):The scaling of the time consumption with centroids is linear for '++' and logarithmic for 'Points'.

The time for MCR-ALS decomposition of the compressed image D' scales linearly with the number of centroids, exemplified in figure 8**C**, which has to do with the implementation of NNLS. NNLS must be run on each pixel in D' separately, and due to MCR-ALS being run for 200 iterations, the average time required per centroid is about the same. Furthermore, the reason the time for decomposition does not start at zero is an initial setup-time in the implementation of MCR-ALS.

The time-requirement of the clustering step scales linearly for '++' with centroids but logarithmically for 'Points'. The reason is that the probability of choosing a centroid must be re-calculated each time. Their exact scaling is dependent on the implementation.

There is no practical difference in time consumption between the 'LSW' and 'NNLS' methods, and their time consumptions scales like $O(n^{1+\epsilon})$, $\epsilon < 1$, with centroids. In contrast, the time-requirements of the Strong' and 'Simple' reconstruction methods do not scale with the number of centroids. Moreover, the 'Strong' and 'Simple' reconstruction methods also require considerably less time than the 'LSW' and 'NNLS' methods; of them, 'Simple' requires the least time.

The total time-consumption is dominated by the time taken to construct the reconstructed contribution matrix, $\hat{C}$, when using either the 'LSW' or 'NNLS' as the reconstruction method. Aside from using those reconstruction methods, when using the 'Simple' or 'Strong' reconstruction method, the total time consumption is at most a tenth of the original time, within the range of centroids used; the largest portion of time was used for decomposition.

We also examined how efficient the methods are in terms of minimizing the loss against relative time elapsed, because we want to accelerate the process as much as possible with the least error possible. Examples on how the loss using CAM-A evolved as a function of the time quotient between time required for all parts of CAM-A and the time required for non-clustered MCR-ALS, T_0, depends on the number of components and reconstruction method are presented in figure 9 for the images "Tumor" and "Ex_90"; the initialization of k-means is '++', to minimize the loss per centroid. From the examination, we found that the 'Strong' reconstruction method is the most efficient reconstruction method.

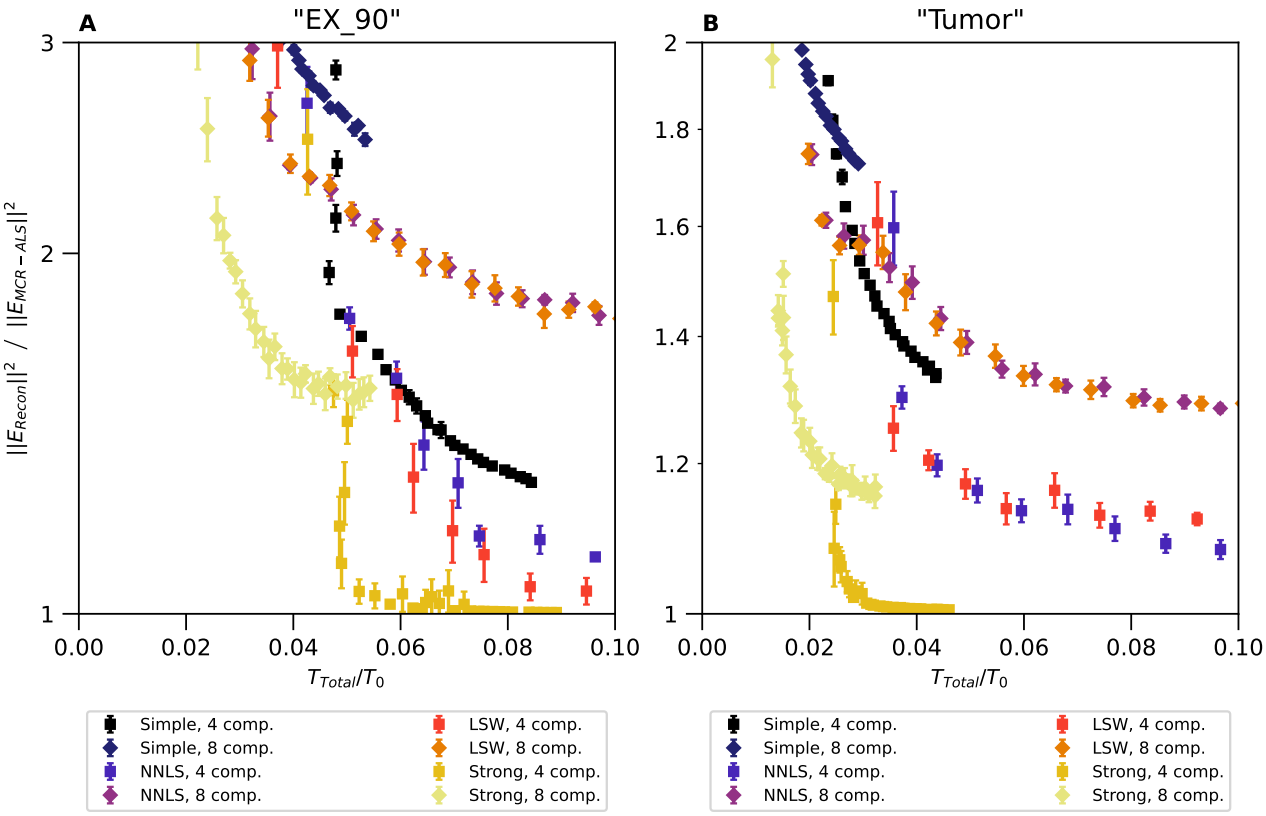

Figure 9: Loss against time elapsed for all reconstruction methods of CAM-A, with 95% confidence intervals.

4 Conclusion

We have developed a method for accelerating MCR-ALS, CAM-A, which can reduce the time requirement, but at a cost in accuracy compared to the original decomposition. However, the resulting decomposition is profoundly dependent on the choices made for the different steps in CAM-A.

Of the four reconstruction methods presented in this project, the 'Strong' method is the most efficient method in terms of loss as a function of time elapsed. The 'Strong' method also yields the least loss per centroid, and, consequently, is the most memory-efficient reconstruction method since fewer centroids are needed compared to the other methods. It is also not found to exhibit any unfavorable behaviors, unlike the 'LSW'-method which can introduce an error increasing with the number of centroids.

The preferable method to initialize k-means is '++'. The extra time-requirement for using '++' instead of the 'Points' initialization is outweighed by its benefits. '++' initialization is also less prone to rotations and produces a smaller loss due to a better representation of spectra. Thus, fewer centroids are needed to reach a better representation of an image, reducing the time and memory required.

Spectral data should not be normalized. The intensity in a hyperspectral image is critical to determine the importance of spectra. Without the information about the intensity, new rotations may emerge because the lower-intensity spectra may be heavily influenced by random linear fluctuations in the original hyperspectral image.

Overall, how well centroids from clustering can represent the original hyperspectral image is crucial to MCR-ALS. The choices which allow the centroids to represent of the original hyperspectral more accurately consistently yields a smaller loss. Thus, usage of other clustering methods than k-means may be investigated, or other initializations, to see if the representation can be improved comparatively.

The optimal set of parameters for CAM-A presented in this project, is initializing k-means with '++', using 'Strong' as reconstruction method, and not normalizing the image. CAM-A subsequently produces a reliable decomposition and reduces both the time and memory usage needed for decomposition. Nevertheless, there are further applications and improvements.

The decomposition is the step which takes the most time, even when using the optimal set of parameters for clustering. The time required by decomposition may be reduced by utilizing Anderson acceleration [16]. It is a method to accelerate MCR-ALS which does not care about the distribution of the spectra in the image and can straightforwardly be implemented with clustering. Thus, MCR-ALS will be accelerated further. However, there may be unintended consequences combining CAM-A with Anderson acceleration, and further research is required.

CAM-A may also be useful when working with data sets consisting of several hyperspectral images. CAM-A can be run on images or a subset of the images separately. The centroids found from clustering are subsequently properly weighted and merged into one matrix. The matrix is then decomposed into a spectral and contribution matrix. From the decomposition, each image can be reconstructed using any of the reconstruction methods proposed here.

References

[1] Troein C, Siregar S, Op De Beeck M, Peterson C, Tunlid A, Persson P. OCTAVVS: A graphical toolbox for high-throughput preprocessing and analysis of vibrational spectroscopy imaging data. Methods Protoc. 2020;3(2):34. Available from: http://dx.doi.org/10.3390/mps3020034

[2] de Juan A, Tauler R. Multivariate curve resolution (MCR) from 2000: Progress in concepts and applications. Crit Rev Anal Chem. 2006;36(3–4):163–76. Available from: http://dx.doi.org/10.1080/10408340600970005

[3] Ruiz JJ, Marro M, Galván I, Bernabeu-Wittel J, Conejo-Mir J, Zulueta-Dorado T, et al. Novel non-invasive quantification and imaging of eumelanin and DHICA subunit in skin lesions by Raman spectroscopy and MCR algorithm: Improving dysplastic nevi diagnosis. Cancers (Basel). 2022;14(4):1056. Available from: http://dx.doi.org/10.3390/cancers14041056

[4] Jolliffe IT. Principal Component Analysis. 2nd ed. New York, NY: Springer; 2002.

[5] Chatterjee S, Singh B, Diwan A, Lee ZR, Engelhard MH, Terry J, et al. A perspective on two chemometrics tools: PCA and MCR, and introduction of a new one: Pattern recognition entropy (PRE), as applied to XPS and ToF-SIMS depth profiles of organic and inorganic materials. Appl Surf Sci. 2018;433:994–1017. Available from: http://dx.doi.org/10.1016/j.apsusc.2017.09.210

[6] Piqueras S, Duponchel L, Tauler R, de Juan A. Resolution and segmentation of hyperspectral biomedical images by multivariate curve resolution-alternating least squares. Anal Chim Acta. 2011;705(1–2):182–92. Available from: http://dx.doi.org/10.1016/j.aca.2011.05.020

[7] Ghaffari M, Hugelier S, Duponchel L, Abdollahi H, Ruckebusch C. Effect of image processing constraints on the extent of rotational ambiguity in MCR-ALS of hyperspectral images. Anal Chim Acta. 2019;1052:27–36. Available from: http://dx.doi.org/10.1016/j.aca.2018.11.054

[8] Ahmadi G, Abdollahi H. A systematic study on the accuracy of chemical quantitative analysis using soft modeling methods. Chemometr Intell Lab Syst. 2013;120:59–70. Available from: http://dx.doi.org/10.1016/j.chemolab.2012.11.007

[9] Tauler R. Multivariate curve resolution applied to second order data. Chemometr Intell Lab Syst. 1995;30(1):133–46. Available from: http://dx.doi.org/10.1016/0169-7439(95)00047-x

[10] Arthur D, Vassilvitskii S. k-means++: the advantages of careful seeding. In: Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms;

2007 Jan 7-9; New Orleans, United States. United States: Society for Industrial and Applied Mathematics; 2007 [cited 2022 Apr 10]; p.1027-1035. Available from: http://ilpubs.stanford.edu:8090/778/1/2006-13.pdf

[11] Piqueras S, Krafft C, Beleites C, Egodage K, von Eggeling F, Guntinas-Lichius O, et al. Combining multiset resolution and segmentation for hyperspectral image analysis of biological tissues. Anal Chim Acta. 2015;881:24–36. Available from: http://dx.doi.org/10.1016/j.aca.2015.04.053

[12] Lawson CL, Hanson RJ. Solving Least Squares Problems. Harlow, England: Longman Higher Education; 1974

[13] Windig W. Spectral data files for self-modeling curve resolution with examples using the Simplisma approach. Chemometr Intell Lab Syst. 1997;36(1):3–16. Available from: http://dx.doi.org/10.1016/s0169-7439(96)00061-5

[14] Miyamoto S, Ichihashi H, Honda K. Algorithms for fuzzy clustering: Methods in c-means clustering with applications. Berlin, Germany: Springer; 2010.

[15] Pedregosa F, Varoquaux G, Gramfort A, Michel V, Thirion B, Grisel O, et al. Scikit-learn: Machine Learning in Python. arXiv [cs.LG]. 2012. Available from: http://arxiv.org/abs/1201.0490

[16] Walker HF, Ni P. Anderson acceleration for fixed-point iterations. SIAM J Numer Anal. 2011;49(4):1715–35. Available from: http://dx.doi.org/10.1137/10078356x

A Derivations

A.1 Derivation of weighting

To start with, define a matrix G as follows:

$$G = CS^T, \tag{App.1}$$

where

$$S = \underset{\mathcal{S}}{\operatorname{argmin}} ||D - C\mathcal{S}^T|| \quad \text{or} \quad C = \underset{\mathcal{C}}{\operatorname{argmin}} ||D - \mathcal{C}S^T||. \tag{App.2}$$

$G' = C'(S')^T$ for the compressed image, D', by extrapolation. Also, define a set consisting of the pixels corresponding to a centroid by

$$I_k = \{i : \boldsymbol{d}_i \in \mathcal{A}_{\boldsymbol{q}_k}\}. \tag{App.3}$$

The partial loss from a row,i, in a matrix $\tilde{D}$, where the each pixels are simply replaced with the centroid representing it, $\tilde{D_{ij}} = Q_{kj} : i \in \mathcal{I}_k$ has the form:

$$||E_i||^2 = \sum_j (\tilde{D}_{ij} - \tilde{G}_{ij})^2 = \sum_j (Q_{kj} - \tilde{G}_{ij})^2. \tag{App.4}$$

Consequently, we can observe that $\tilde{G_{ij}} = G_{pj} := G_{kj} : i, p \in \mathcal{I}_k$ due to the fact that they should minimize the same total loss using the same centroid and S or C -matrix. Ergo, the total loss becomes:

$$\begin{aligned} ||E_{\text{tot}}|| &= \sum_{ij} (Q_{ij} - \tilde{G}_{ij})^2 = \sum_{ij} w_k (Q_{kj} - G_{kj})^2 = \sum_{kj} w_k (Q_{kj} - G'_{kj})^2 = \\ &= \sum_{kj} (\sqrt{w_k} Q_{kj} - \sqrt{w_k} G'_{kj})^2 = \sum_{kj} (\sqrt{w_k} Q_{kj} - \sqrt{w_k} \boldsymbol{C'}_k \boldsymbol{\cdot} \boldsymbol{S'}_j)^2, \end{aligned} \tag{App.5}$$

and we find that the same effective decomposition is conserved with proper weighting of the loss or D' as given in equation (12).

If MCR-ALS is initialized using S, the decomposition returns a C' whose components are effectively weighted after importance due to the interaction between the construction of G' in equation (App.1) and (App.5). Similarly, the interaction between the weighted C' and equations (App.1) and (App.5) results in a non-weighted S', and the cycle repeats each iteration. In contrast, initializing using an improperly weighted C_{init} will cause a major initial rotation of S', influencing the subsequent iterations.

A.2 Derivation of expression of λ_i for LSW.

Define $\hat{C}$ from equation (13). Let V_{LSW} be the reconstruction matrix, and its rows defined as in equation (17). We know that $V_{\text{LS},i}$ produces the lowest reconstruction loss of the two matrices and $V_{\text{NNLS},i}$ produces the lowest loss given positivity boundary-conditions. As such, we want $0 \leq \lambda_i \leq 1$ but as close to 1 as possible. Another condition on V_{LSW} is that

$$\hat{C}_{ij} \geq 0. \tag{App.6}$$

By construction, $V_{\text{NNLS},i} \bullet C'_j \geq 0$. Emerging from equations (13), (17) and (App.6), we find the following limitation on λ_{ij}:

$$((1-\lambda_{ij})V_{\text{NNLS},i} + \lambda_{ij}V_{\text{LS},i}) \bullet C'_j \geq 0 \Rightarrow$$

$$\Rightarrow (\lambda_{ij}(V_{\text{LS},i} - V_{\text{NNLS},i}) + V_{\text{NNLS},i}) \bullet C'_j \geq 0 \Rightarrow$$

$$\Rightarrow \begin{cases} \lambda_{ij} \leq -\frac{C'_j \bullet V_{\text{NNLS},i}}{(V_{\text{LS},i} - V_{\text{NNLS},i}) \bullet C'_j} \geq 0 & \text{if } (V_{\text{LS},i} - V_{\text{NNLS},i}) \bullet C'_j < 0 \\ \lambda_{ij} \geq -\frac{C'_j \bullet V_{\text{NNLS},i}}{(V_{\text{LS},i} - V_{\text{NNLS},i}) \bullet C'_j} \leq 0 & \text{if } (V_{\text{LS},i} - V_{\text{NNLS},i}) \bullet C'_j > 0 \\ \lambda_{ij} \in \mathbb{R} & \text{if } (V_{\text{LS},i} - V_{\text{NNLS},i}) \bullet C'_j = 0. \end{cases} \tag{App.7}$$

Thus, we find that we can let λ_{ij} to be arbitrarily large when $(V_{\text{LS},i} - V_{\text{NNLS},i}) \bullet C'_j \geq 0$, but as λ_i is confined to the region $[0, 1]$, we may set $\lambda_{ij} = 1$ for simplicity in implementation. Equation (18) arises as a result. Subsequently, because the elements of $\hat{C}$ are to be strictly non-negative and we we are confined in the region $[0, 1]$, we find the expression for λ_i given in equation (19).

B Code

B.1 Modified MCR-ALS code from the OCTAVVS project

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Tue May 18 14:45:53 2021

@author: carl, eric (small modification)
"""

import numpy as np
import scipy
import math
import time
from threadpoolctl import threadpool_limits

def simplisma(d, nr, f):
    """
    The SIMPLISMA algorithm for finding a set of 'pure' spectra to serve
    as starting point for MCR-ALS etc.
    Reference Matlab Code:
        J. Jaumot, R. Gargallo, A. de Juan, R. Tauler,
        Chemometrics and Intelligent Laboratoty Systems, 76 (2005)
    101-110

    Parameters
    ----------
    d : array(nspectra, nwavenums)
        input spectra.
    nr : int
        number of output components.
    f : float
        noise threshold.

    Returns
    -------
    spout: array(nr, nspectra)
        concentration profiles of 'purest' spectra.
    imp : array(nr, dtype=int)
        indexes of the 'purest' spectra.
    """

    nrow = d.shape[0]
    ncol = d.shape[1]
    s = d.std(axis=0)
    m = d.mean(axis=0)
    mf = m + m.max() * f
    p = s / mf
```

```
46
47      # First Pure Spectral/Concentration profile
48      imp = np.empty(nr, dtype=np.int)
49      imp[0] = p.argmax()
50
51      #Calculation of correlation matrix
52      l2 = s**2 + mf**2
53      dl = d / np.sqrt(l2)
54      c = (dl.T @ dl) / nrow
55
56      #calculation of the first weight
57      w = (s**2 + m**2) / l2
58      p *= w
59      #calculation of following weights
60      dm = np.zeros((nr+1, nr+1))
61      for i in range(1, nr):
62          dm[1:i+1, 1:i+1] = c[imp[:i], :][:, imp[:i]]
63          for j in range(ncol):
64              dm[0, 0] = c[j, j]
65              dm[0, 1:i+1] = c[j, imp[:i]]
66              dm[1:i+1, 0] = c[imp[:i], j]
67              w[j] = np.linalg.det(dm[0:i+1, 0:i+1])
68          imp[i] = (p * w).argmax()
69
70      ss = d[:,imp]
71      spout = ss / np.sqrt(np.sum(ss**2, axis=0))
72      return spout.T, imp
73
74  def clustersubtract(data, components, skewness=300, power=2):
75      """
76      Create initial spectra for MCR-ALS based on successively removing
77      what appears to be the strongest remaining component.
78
79      Parameters
80      ----------
81      data : array (nspectra, nfeatures)
82          Spectral data.
83      components : int
84          Number of components to return.
85      skewness : float, optional
86          Asymmetry between positive and negative residuals when computing
87          how much of the previous component to remove from the data.
88          The default is 100.
89      power : float, optional
90          The sum of residuals is raised to this power before summation to
91          determine the leading remaining component.
92
93      Returns
94      -------
95      initial_spectra : array (components, nfeatures)
96      """
```

```
    def typical_cluster(data, first):
        # draw sqrt(n) random, r
        # find closest in r for each s
        # for r with most s, return mean of s (or iterate?)
        r = np.random.choice(len(data), math.floor(math.sqrt(len(data))))
        rd = data[r]
        #Cosine norm used to ignore difference in magnitude
        nearest = scipy.spatial.distance.cdist(
            rd, data, 'cosine').argmin(axis=0)
        # Mean of those who are nearest the biggest cluster
        if first:
            selected = np.bincount(nearest).argmax()
        else:
            sums = data.sum(1)**power
            selected = np.bincount(nearest, weights=sums).argmax()
        return data[nearest == selected].mean(0)

    comps = []
    for c in range(components):
        tc = typical_cluster(data, c == 0)
        tc = np.maximum(tc, 0)
        tc = tc / (tc * tc).sum() ** .5

        comps.append(tc)
        sgn = np.ones_like(data, dtype=bool)
        for i in range(10):
            ww = 1 * sgn + skewness * ~sgn
            a = (data * ww * tc).sum(1) / (ww * tc * tc).sum(1)
            oldsgn = sgn
            sgn = data > a[:, None] @ tc[None, :]
            if np.array_equal(sgn, oldsgn):
                break
        data = data - a[:, None] @ tc[None, :]
    return np.array(comps)

def numpy_scipy_threading_fix_(func):
    """
    This decorator for mcr_als prevents threading in BLAS if scipy's NNLS
    is used, because for some reason NNLS won't be parallelized if called
    shortly after lstsq or @. This makes a *massive* difference to the
    time needed for Anderson acceleration, where the BLAS calls
   themselves
    take negligible time. For mixed NNLS/lstsq solving (of MCR-ALS on
    derivatives) it's less obvious whether NNSL or lstsq should be
   allowed
    to be parallelized.
    Note: This issue is seen on
    """
    def check(*args, **kwargs):
        if np.any(kwargs['nonnegative']):
            with threadpool_limits(1, 'blas'):
```

```
                return func(*args, **kwargs)
        else:
            return func(*args, **kwargs)
    return check

@numpy_scipy_threading_fix_
def mcr_als(sp, initial_A, *, maxiters, nonnegative=(True, True),
            tol_abs_error=0, tol_rel_improv=None, tol_iters_after_best=None,
            maxtime=None, callback=None, acceleration=None, normalize=None,
            contrast_weight=None, return_time=False, weight_vector = None, **kwargs):
    """
    Perform MCR-ALS nonnegative matrix decomposition on the matrix sp

    Parameters
    ----------
    sp : array(nsamples, nfeatures)
        Spectra to be decomposed.
    initial_A : array(ncomponents, nfeatures)
        Initial spectra or concentrations.
    maxiters : int
        Maximum number of iterations.
    nonnegative : pair of bool, default (True, True)
        True if (initial, other) components must be non-negative
    tol_abs_error : float, optional
        Error target (mean square error).
    tol_rel_improv : float, optional
        Stop when relative improvement is less than this over 10 iterations.
    tol_iters_after_best : int, optional
        Stop after this many iteratinos since last best error.
    maxtime : float, optional
        Stop after this many seconds of process time have elapsed
    callback : func(it : int, err : float, A : array, B : array)
        Callback for every iteration.
    acceleration : str, optional
        None or 'Anderson'.
        Anderson acceleration operates on whole iterations (A or B updates),
        mixing earlier directions to step towards the fixed point. This
        implementation restarts from basic updates when those would be
        better.
    normalize : str, optional
        Which matrix to l2 normalize: None, 'A' or 'B'
    contrast_weight : (str, float), optional
        Increase contrast in one matrix by mixing the other, named matrix
        ('A' or 'B') with the mean of its vectors. If A is spectra,
        try contrast_weight=('B', 0.05) to increase spectral contrast.
        See Windig and Keenan, Applied Spectroscopy 65: 349 (2011).
```

```
    return_time : bool, default False
        Measure and return process_time at each iteration.
    weight_vector: array(nfeatures,1), default None
        Vector containing the weigths corresponding to cluster. initial_A must
        be the initial spectra.

    Anderson acceleration parameters in kwargs
    -------
    m : int, >1, default 2
        The maximum number of earlier steps to consider.
    alternate : bool, default True
        Alternate between accelerating A and B, switching when restarting.
    beta : float, default 1.
        Scaling factor for accelerated step length.
    betascale : float, default 1.
        Reduction factor for beta after each restart.
    bmode : bool, default False
        Start with accelerating B instead of A.

    Returns
    -------
    A : array(ncomponents, nfeatures)
        Spectra (at lowest error)f
    B : array(ncomponents, nsamples)
        Concentrations at lowest error
    error : list(float)
        Mean square error at every iteration
    process_time : list(float)
        Time relative start at each iteration, only if return_time is True.
    """
    if normalize not in [None, 'A', 'B']:
        raise ValueError('Normalization must be None, A or B')
    unknown_args = kwargs.keys() - {
        'm', 'alternate', 'beta', 'betascale', 'bmode'}
    if unknown_args:
        raise TypeError('Unknown arguments: {}'.format(unknown_args))

    nrow, ncol = sp.shape
    nr = initial_A.shape[0]
    if normalize == 'A':
      #L2-norm
        norm = np.linalg.norm(initial_A, axis=1)
        A = np.divide(initial_A.T, norm, where=norm!=0,
                      out=np.zeros(initial_A.shape[::-1]))
    else:
        #Transpose od A-matrix already done
        A = initial_A.T.copy()
    B = np.empty((nr, nrow))
```

```
    errors = []
    errorbest = None # Avoid spurious warning
    # prevA, prevB = (None, None)
    newA = newB = None
    error = preverror = None

    if weight_vector is not None:
      pass
        #print('Warning: You must initalize using spectra.')

    cw = 0
    if contrast_weight is not None:
        if contrast_weight[0] == 'A':
            cw = contrast_weight[1]
        elif contrast_weight[0] == 'B':
            cw = -contrast_weight[1]
        else:
            raise ValueError("contrast_weight must be ('A'|'B', [0-1])")


    if acceleration == 'Anderson':
        ason_Bmode = kwargs.get('bmode', False)
        ason_alternate = kwargs.get('alternate', True)
        ason_m = kwargs.get('m', 2)
        ason_beta = kwargs.get('beta', 1.)
        ason_betascale = kwargs.get('betascale', 1.)
        ason_g = None
        ason_G = []
        ason_X = []
    elif acceleration:
        raise ValueError("acceleration must be None or 'Anderson'")

    starttime = time.process_time()
    if return_time:
        times = []
    tol_rel_iters = 10

    for it in range(maxiters):
        ba = 0
        retry = False
        while ba < 2:
            if not retry:
                preverror = error
            if ba == 0:
                if newA is None:
                    newA = A
                prevA = newA
                if cw > 0:
                    newA = (1 - cw) * newA + cw * newA.mean(1)[:,None]
                if nonnegative[1]:
                    error = 0
```

```
                    if not retry:
                        B = np.empty_like(B)
                    for i in range(nrow):
                        B[:, i], res = scipy.optimize.nnls(newA, sp[i,
:])
                        error += res * res
                else:
                    B, res, _, _ = np.linalg.lstsq(newA, sp.T, rcond=-1)
                    error = res.sum()
                if normalize == 'B':
                    norm = np.linalg.norm(B, axis=1)
                    B = np.divide(B.T, norm, where=norm!=0, out=B.T).T
                newA = None
            else:
                if newB is None:
                    newB = B
                prevB = newB
                if cw < 0:
                    newB = (1 + cw) * newB - cw * newB.mean(1)[:,None]
                if nonnegative[0]:
                    error = 0
                    if not retry:
                        A = np.empty_like(A)
                    for i in range(ncol):
                        A[i, :], res = scipy.optimize.nnls(newB.T, sp[:,
i])
                        error += res * res
                else:
                    A, res, _, _ = np.linalg.lstsq(newB.T, sp, rcond=-1)
                    A = A.T
                    error = res.sum()
                if normalize == 'A':
                    norm = np.linalg.norm(A, axis=0)
                    np.divide(A, norm, where=norm!=0, out=A)
                newB = None

            if acceleration is None:
                pass
            elif ba == ason_Bmode:
                if retry:
                    retry = False
                    if ason_alternate:
                        ason_Bmode = not ason_Bmode
                    ason_beta = ason_beta * ason_betascale
                elif len(ason_X) > 1 and error > preverror:
                    ason_X = []
                    ason_G = []
                    retry = True
                    ba = ba - 1
                else:
                    pass
```

```
            elif ason_Bmode == 1 and it == 0:
                pass
            else:
                prevg = ason_g
                ason_g = ((A - prevA) if ba else (B - prevB)).flatten()
                if len(ason_X) < 1:
                    ason_X.append(ason_g)
                else:
                    ason_G.append(ason_g - prevg)
                    while(len(ason_G) > ason_m):
                        ason_G.pop(0)
                        ason_X.pop(0)
                    Garr = np.asarray(ason_G)
                    try:
                        gamma = scipy.linalg.lstsq(Garr.T, ason_g)[0]
                    except scipy.linalg.LinAlgError:
                        print('lstsq failed to converge; '
                              'restart at iter %d' % it)
                        # print('nans', np.isnan(Garr).sum(),
                        #       np.isnan(ason_g).sum())
                        ason_X = []
                        ason_G = []
                    else:
                        gamma = ason_beta * gamma
                        dx = ason_g - gamma @ (np.asarray(ason_X) + Garr)
                        ason_X.append(dx)
                        if ba:
                            newA = prevA + dx.reshape(A.shape)
                            if nonnegative[0]:
                                np.maximum(0, newA, out=newA)
                        else:
                            newB = prevB + dx.reshape(B.shape)
                            if nonnegative[1]:
                                np.maximum(0, newB, out=newB)
            ba = ba + 1
        # error = error / weight_vector.sum()

        curtime = time.process_time() - starttime
        if return_time:
            times.append(curtime)
        errors.append(error)
        if not it or error < errorbest:
            errorbest = error
            Abest = A
            Bbest = B
            iterbest = it
        if it:
            if error < tol_abs_error:
                break
            if tol_rel_improv and it > tol_rel_iters:
                emax = max(errors[-tol_rel_iters-1:-2])
```

```
                if (emax - errors[-1]) * tol_rel_iters <= \
                    tol_rel_improv * emax:
                        break
            if tol_iters_after_best is not None:
                if iterbest + tol_iters_after_best < it:
                    break
        if it and maxtime and curtime >= maxtime:
            break
        if callback is not None:
            callback(it, errors, A.T, B)
    if weight_vector is not None:
      Bbest = (((weight_vector)**(-1/2)) * Bbest.T).T
    if return_time:
        return Abest.T, Bbest, errors, times
    return Abest.T, Bbest, errors
```

B.2 Code used for implementations of k-means and reconstruction methods

```

import numpy
import scipy
import numpy
import random
import time
from scipy.cluster import hierarchy
import matplotlib.pyplot as plt
import sklearn.cluster

def normalize(A,labels, init_centroids, trgt_nrm, message,
    centroids_from_labels = False):
  '''
  centroids_from_labels: bool
    Added to solve problem of possible inconsistency between labels and
    centroids
  '''

  nrow = labels.shape[0]

  centroids_mapping = [l for l in set(labels)]
  centroid_pixel_arrs = [A[labels == l] for l in set(labels)]
  centroid_weights = numpy.array([item.shape[0] for item in
    centroid_pixel_arrs]).reshape(len(centroids_mapping),1)

  square_sums = numpy.array([(item**2).sum(axis = 0) for item in
    centroid_pixel_arrs])
  means = numpy.array([item.mean(axis = 0) for item in
    centroid_pixel_arrs])


```

```
  if trgt_nrm == 'mean':
    sums = numpy.array([item.sum(axis = 0) for item in centroid_pixel_arrs])
    normalization = (centroid_weights**(-1/2)) * sums
    errors_tot = (square_sums - means*sums).sum()
    if centroids_from_labels:
      return centroids_mapping, centroid_weights, normalization, errors_tot, means
    else:
      return centroids_mapping, centroid_weights, normalization, errors_tot

  elif trgt_nrm ==  'square':
    normalization = square_sums**(1/2)
    square_means = square_sums/centroid_weights
    sqrt_square_means = square_means ** (-1/2)
    error_coeffs = numpy.true_divide(square_means-(means**2),(sqrt_square_means+means),where = square_means != 0, out = numpy.zeros_like(means))
    if centroids_from_labels:
      return centroids_mapping, centroid_weights, normalization, error_coeffs, means
    else:
      return centroids_mapping, centroid_weights, normalization, error_coeffs

  elif trgt_nrm == 'nnls':
    membership_coeffs = numpy.empty((nrow,len(centroids_mapping)))
    for i in range(nrow):
      membership_coeffs[i:] = scipy.optimize.nnls(init_centroids.T,A[i,:].T)[0].reshape((1,len(centroids_mapping)))
    centroid_weights = membership_coeffs.sum(axis = 0).reshape(len(centroids_mapping),1)
    normalization = (centroid_weights**(1/2)) * init_centroids
    if centroids_from_labels:
      return None,centroid_weights, normalization, None, means
    else:
      return None,centroid_weights, normalization, None

  else:
    raise Exception('trgt_nrm must either be mean, square or nnls')

def kMeans_old(A, maxit, thresh, nclus = 'sqrt', minit = 'points', check_finite = False, trgt_nrm = 'mean', message = False):
  '''
  Currently, thresh does not have any functionality
  '''
  time1 = time.time()
  npixels = A.shape[0]
  if nclus == 'sqrt':
    nclus = int(numpy.sqrt(npixels))
```

```
68
69    init_centroids, labels = scipy.cluster.vq.kmeans2(A,nclus, maxit,
       thresh = thresh, minit = minit, check_finite = check_finite)
70
71    if message:
72      print('Clustering done!')
73
74    centroids_mapping, centroid_weights, new_target, error = normalize(A,
       labels, init_centroids, trgt_nrm, message)
75
76    return new_target, centroid_weights, init_centroids, centroids_mapping,
       labels, error, time.time()-time1
77
78  def kMeans(A, maxit, thresh, nclus = None, minit = 'points',check_finite
       = False, trgt_nrm = 'mean', message = False):
79
80    time1 = time.time()
81    if minit == 'points':
82      minit = 'random'
83    elif minit == '++':
84      minit = 'k-means++'
85
86    if nclus is None:
87      raise Exception('You must designate a number of centroids')
88
89
90    cluster_data = sklearn.cluster.KMeans(nclus,init = minit,max_iter =
       maxit,tol = thresh,n_init = 1).fit(A)
91
92    labels = cluster_data.labels_
93    init_centroids = cluster_data.cluster_centers_
94    centroids_mapping, centroid_weights, new_target, error, init_centroids
       = normalize(A,labels, init_centroids, trgt_nrm, message,
       centroids_from_labels = True)
95
96    return new_target, centroid_weights, init_centroids, centroids_mapping,
       labels, error, time.time()-time1
97
98  def fuzzy(data,nclusters,maxiter,m = 2, tol = None , seed = None,
       initalization = 'simple' ,trgt_nrm = 'fuzzy',init_clusters = None,
       e_rel_tol=None):
99    '''
100   Implmentation proposed by (Qian Liu, et.al, https://doi.org/10.1016/j.
       tcs.2021.06.035)
101
102   Inputs:
103     data: ndarray
104
105     nclusters: int
106
107     seed:
```

```
108
109   '''
110   if m <= 1:
111     return ValueError('m must be larger than 1')
112
113   fuzzy_constant = 1/(m-1)
114
115   time1 = time.time()
116   ##### Definitions of useful functions #####
117   def memberships(data,clusters):
118
119     '''
120     It was hell to code this section so that it would not be too resource
      intensive
121     '''
122     distances_to_cluster = []
123     for i in range(clusters.shape[0]):
124       # This loop is expensive, but easier to implement than using a 3-
     tensor and cheaper than iterating over the objects
125       cluster = clusters[i,:].reshape(1,clusters.shape[1])
126
127       distance = (((data-cluster)**2).sum(axis=1))
128       distances_to_cluster.append(distance)
129
130     distances_to_cluster = numpy.array(distances_to_cluster)
131     distances_untouched = distances_to_cluster.copy()
132     #Distances defined in square euclidian measure
133     #Matrix used for handling of  points on top of clusters
134     locations = (distances_to_cluster == 0).any(axis = 0)
135     distances_to_cluster  = numpy.power(distances_to_cluster,-(
      fuzzy_constant),where = distances_to_cluster != 0,out = numpy.
      zeros_like(distances_to_cluster))
136     distances_to_cluster[:,locations] = (distances_to_cluster[:,locations
      ] == 0)*1.0
137
138     total_membership = distances_to_cluster.sum(axis = 0).reshape(data.
      shape[0],1).T
139     membership_degrees = numpy.true_divide(distances_to_cluster,
      total_membership)
140
141     potential = ((membership_degrees**m) * distances_untouched).sum()
142
143     return membership_degrees,potential
144
145
146   def phi(data,clusters,membership_degrees=None):
147     '''
148     Outdated function, used for verification
149     '''
150     distances_to_cluster = []
151     for i in range(clusters.shape[0]):
```

```
      # This loop is expensive, but easier to implement than using a 3-tensor and cheaper than interating over the objects
      cluster = clusters[i,:].reshape(1,clusters.shape[1])
      # Distances is defined in the square-euqlidian norm for convenience
      distance = (((data-cluster)**2).sum(axis=1))
      distances_to_cluster.append(distance)
    distances_to_cluster = numpy.array(distances_to_cluster)

    if membership_degrees != None:
      potential = ( distances_to_cluster * membership_degrees**m).sum()
    else:
      locations = (distances_to_cluster == 0).any(axis = 0)
      distances_to_cluster[:,locations] = 0.0
      distances_to_cluster  = numpy.power(distances_to_cluster,-1,where = distances_to_cluster != 0,out = numpy.zeros_like(distances_to_cluster))
      total_membership = distances_to_cluster.sum(axis = 0).reshape(data.shape[0],1).T
      total_membership = numpy.power(total_membership,-1,where = total_membership != 0,out = numpy.zeros_like(distances_to_cluster))
      potential = total_membership.sum()

    return potential

#### Initialization step ####
if nclusters > data.shape[0]:
  raise ValueError(f'number of objects must be less or equal to clusters. {nclusters} > {data.shape[0]}')


rng = numpy.random.default_rng(seed)

if initalization == 'points':
  indicies = rng.choice(data.shape[0],size =nclusters, replace = False)
  clusters = data[indicies,:]

elif initalization == 'choice':
  clusters = init_clusters

elif initalization == '++':
  unpicked_data = data.copy()
  ncolumns = unpicked_data.shape[1]
  index = rng.integers(0,unpicked_data.shape[0])
  clusters = unpicked_data[index,:].reshape((1,ncolumns))
  unpicked_data = numpy.delete(unpicked_data,index,axis = 0)

  #new_data_pool = data[numpy.arange(data.shape[0]) != index,:]
  count = 0
  for _ in range(nclusters-1):
    full_weight = memberships(unpicked_data,clusters)[-1]
    while True:
```

```
        index = rng.integers(0,unpicked_data.shape[0])
        choice = unpicked_data[index,:].reshape(1,ncolumns)
        small_weight = memberships(choice,clusters)[-1]
        if rng.random() < small_weight/full_weight:
          clusters = numpy.vstack((clusters,choice))
          unpicked_data = numpy.delete(unpicked_data,index,axis = 0)
          break
      #new_data_pool = data[numpy.arange(data.shape[0]) != index,:]

  else:
    raise Exception('Must initalize using either points,choice or ++')
  #### Clustering loop ####

  it = 0
  membership_matrix,loss = memberships(data,clusters)
  while maxiter > it:
    old_loss = loss
    if tol is not None:
      if tol > loss:
        break
    clusters =numpy.array([((membership_matrix[i,:].T.reshape(
     membership_matrix.shape[1],1)**m) * data).sum(axis = 0) / (
     membership_matrix[i,:]**m).sum() for i in range(nclusters)])
     membership_matrix,loss = memberships(data,clusters)

    it += 1
    if e_rel_tol is not None:
      if old_loss-loss < e_rel_tol:
        break


  normalization, norms = fuzzy_normalization(data,clusters,
     membership_matrix,m,trgt_nrm)

  return normalization,norms,clusters, membership_matrix, loss, time.time
    ()-time1

def fuzzy_normalization(data,clusters,membership_matrix,m,trgt_nrm = '
    fuzzy'):

  nrows = data.shape[0]
  nclus = clusters.shape[0]

  if trgt_nrm == 'nnls':
    membership_coeffs = numpy.empty((nrow,nclus))
    for i in range(nrow):
      membership_coeffs[i:] = scipy.optimize.nnls(init_centroids.T,
     objects[i,:].T)[0].reshape((1,nclus))
    norms = membership_coeffs.sum(axis = 0).reshape(nclus,1)
    normalization = (norms**(1/2)) * init_centroids
  else:
```

```
242     norms = numpy.array([(membership_matrix[i,:]).sum() for i in range(
        clusters.shape[0])])
243     print(norms,norms.sum())
244     new_targets = clusters * norms.reshape(norms.shape[0],1)**(1/2)
245   return new_targets, norms.reshape(norms.shape[0],1)
246
247 def fuzzy_reconstruction(A,membership_degrees,m):
248   time1 = time.time()
249   D  = (membership_degrees.T) @ A
250   #print(D.shape)
251   return D, time.time()-time1
252   #D = membership_degrees @ pixels per cluster or contributions per
        cluster
253
254 def reconstruct_image(A,B,init_centroids,centroids_mapping,labels):
255
256   D = numpy.empty_like(A)
257   B = B.T
258   for i in range(len(labels)):
259     l = labels[i]
260     D[i,:] = B[centroids_mapping.index(l),:]
261   return D
262
263 def reconstruct(A,B,centroids,centroids_mapping = None,labels = None ,
        version = 'nnls',limit = None,objects = None,norm_data = None,
        return_time = False,V = None):
264   time1 = time.time()
265   membership_coeffs = V
266   if version == 'simple':
267     if centroids_mapping is None or labels is None:
268       raise TypeError('centroids_mapping must be a one to one mapping of
        the centroids')
269     B = B.T
270     D = numpy.empty_like(A)
271     nrow, nclus = objects.shape[0], centroids.shape[0]
272     membership_coeffs = numpy.zeros((nrow,nclus))
273     for i in range(len(labels)):
274       l = labels[i]
275       membership_coeffs[i,l] = 1
276       D[i,:] = B[centroids_mapping.index(l),:]
277
278   elif version == 'nnls':
279     if type(objects) == None:
280       raise Exception('you must define the original set of objects
        clustered on \n when using NNLS')
281     nrow, nclus = objects.shape[0], centroids.shape[0]
282     if membership_coeffs is None:
283       membership_coeffs = numpy.empty((nrow,nclus))
284       for i in range(nrow):
285         membership_coeffs[i:] = scipy.optimize.nnls(centroids.T,objects[i
        ,:].T)[0].reshape((1,nclus))
```

```
    if limit == None:
      D = membership_coeffs @ B.T
    else:
      raise Exception('not yet implemented')
  elif version == 'lstsq':
    if membership_coeffs is None:
      membership_coeffs = scipy.linalg.lstsq(centroids.T,objects.T)[0].T
    if limit == None:
      D = membership_coeffs @ B.T
    else:
      raise Exception('not yet implemented')
  elif version == 'invL2sq':
    distances_to_cluster = []
    for i in range(centroids.shape[0]):
      # This loop is expensive, but easier to implement than using a 3-tensor and cheaper than iterating over the objects
      cluster = centroids[i,:].reshape(1,centroids.shape[1])

      distance = (((norm_data-cluster)**2).sum(axis=1))
      distances_to_cluster.append(distance)

    distances_to_cluster = numpy.array(distances_to_cluster)
    #Distances defined in square euclidian measure
    #Matrix used for handling of  points on top of clusters
    locations = (distances_to_cluster == 0).any(axis = 0)
    distances_to_cluster  = numpy.power(distances_to_cluster,-1,where = distances_to_cluster != 0,out = numpy.zeros_like(distances_to_cluster))
    distances_to_cluster[:,locations] = (distances_to_cluster[:,locations] == 0)*1.0

    total_membership = distances_to_cluster.sum(axis = 0).reshape(A.shape[0],1).T
    membership_coeffs = numpy.true_divide(distances_to_cluster, total_membership)
    membership_coeffs = membership_coeffs.T

    D = membership_coeffs @ B.T

  elif version == 'exponential':
    var_of_data = norm_data.var(axis = 0).sum()
    nclusters = centroids.shape[0]
    distances_to_cluster = []
    for i in range(centroids.shape[0]):
      # This loop is expensive, but easier to implement than using a 3-tensor and cheaper than iterating over the objects
      cluster = centroids[i,:].reshape(1,centroids.shape[1])

      distance = (((norm_data-cluster)**2).sum(axis=1))
      distances_to_cluster.append(distance)
    distances_to_cluster = numpy.array(distances_to_cluster)/var_of_data
```

```
330     weight_matrix = nclusters**-(distances_to_cluster)
331     partition_array = weight_matrix.sum(axis = 0)
332     partition_array = partition_array.reshape(1,A.shape[0])
333     membership_coeffs = weight_matrix / partition_array
334     D = membership_coeffs.T @ B.T
335     membership_coeffs = membership_coeffs.T
336 
337   elif version == 'LSWeighting':
338     if type(objects) == None:
339       raise Exception('you must define the original set of objects
      clustered on \n when using LSWeighting')
340     nrow, nclus = objects.shape[0], centroids.shape[0]
341     membership_coeffs_nnls = numpy.empty((nrow,nclus))
342     if V is not None:
343       membership_coeffs = V
344       D = membership_coeffs@B.T
345     else:
346       for i in range(nrow):
347         membership_coeffs_nnls[i:] = scipy.optimize.nnls(centroids.T,
      objects[i,:].T)[0].reshape((1,nclus))
348       membership_coeffs_ls = scipy.linalg.lstsq(centroids.T,objects.T)
      [0].T
349 
350       numerator = -1*membership_coeffs_nnls@B.T
351       denominator = (membership_coeffs_ls - membership_coeffs_nnls)@B.T
352       #sign_bools = numpy.where(denominator>0)
353       Coeff_matrix = numpy.true_divide(numerator,denominator,where =
      denominator < 0, out = numpy.ones_like(numerator)*numpy.Inf)
354       Coeff_matrix = numpy.where(denominator == 0,0,Coeff_matrix)
355       min_coeffs = Coeff_matrix.min(axis=1)
356       min_coeffs = numpy.where(min_coeffs<1, min_coeffs,1).reshape(
      min_coeffs.shape[0],1)
357 
358       membership_coeffs = membership_coeffs_nnls + min_coeffs*(
      membership_coeffs_ls-membership_coeffs_nnls)
359       D = membership_coeffs@B.T
360       D[numpy.logical_and(D > -1e-11, D<0) ] = 0
361       if (D<0).any():
362         print(D[ D<0 ])
363         raise Exception('Negative values not explained by truncation
      errors present in the reconstructed matrix')
364   else:
365     raise Exception('must define version')
366   if return_time:
367     return D,membership_coeffs, time.time()-time1
368   return D,membership_coeffs
369 
370 def reconstruct_from_spectra(D,spectra):
371   time1 = time.time()
372   C = numpy.empty((D.shape[0],spectra.shape[0]))
373   print(D.shape)
```

```
  print(spectra.shape)
  for i in range(D.shape[0]):
    C[i,:] = scipy.optimize.nnls(spectra.T,D[i,:].T)[0].T
  return C, time.time() - time1

##### Hierarchial clustering method ######
def hierachial(data,sigma_tol,trgt_nrm = 'mean',message = False,left_tol
    =0):
  time1 = time.time()
  class cluster:
    def __init__(self,features = None,index = None,sigma = 0):
      self.features = features
      self.sum = features
      if index != None:
        self.pixel_indicies = [index]
      else:
        self.pixel_indicies = []
      self.count = 1
      self.sigma = sigma

    def __add__(self,other):
      new_clus = cluster()
      new_clus.sum = self.sum + other.sum
      new_clus.pixel_indicies = self.pixel_indicies + other.
    pixel_indicies
      new_clus.count = self.count + other.count
      return new_clus

  cluster_list =[cluster(data[i,:],i) for i in range(data.shape[0])]
  random.shuffle(cluster_list)

  sigma_tol = sigma_tol ** 2

  total_length = len(cluster_list)
  failcount = 0
  while total_length > failcount+left_tol:
    total_length = len(cluster_list)
    print(total_length)
    failcount = 0
    fresh_clusters = []
    length = total_length
    while length > 1:
      cluster1 = cluster_list[0]
      minimal_sigma = None
      sum_sigma = 0
      found = False

      for i in range(1,length):
        cluster2 = cluster_list[i]
        count1 = cluster1.count
        count2 = cluster2.count
```

```
423         new_mean = (cluster1.sum + cluster2.sum)/(count1+ count2)
424
425         new_sigma = ((cluster1.sigma*count1 + cluster2.sigma*count2)+
    count1*(cluster1.sum/count1 - new_mean)**2 + count2*(cluster2.sum/
    count2 - new_mean)**2)/(count1+count2)
426
427         if minimal_sigma is None:
428           if (new_sigma <= sigma_tol).all() :
429             sum_sigma = new_sigma.sum()
430             minimal_sigma = new_sigma
431             min_index = i
432             found = True
433           else:
434             continue
435         if (new_sigma <= sigma_tol).all():
436           new_sum = new_sigma.sum()
437           if sum_sigma > new_sum:
438             sum_sigma = new_sum
439             minimal_sigma = new_sigma
440             min_index = i
441             found = True
442
443       if found:
444         cluster2 = cluster_list[min_index]
445         fresh_clusters.append(cluster1+cluster2)
446         fresh_clusters[-1].sigma = minimal_sigma
447         del cluster_list[min_index]
448         del cluster_list[0]
449         length -= 2
450         #print(length,total_length)
451       else:
452         fresh_clusters.append(cluster1)
453         del cluster_list[0]
454         failcount += 1
455         length -= 1
456     if len(cluster_list) == 1:
457       fresh_clusters.append(cluster_list[0])
458       cluster_list = fresh_clusters
459       failcount += 1
460
461     cluster_list = fresh_clusters
462
463
464   init_centroids = []
465   labels = numpy.empty(data.shape[0])
466   for i in range(len(fresh_clusters)):
467     centroid = fresh_clusters[i]
468     init_centroids.append(centroid.sum/centroid.count)
469     for item in centroid.pixel_indicies:
470       labels[item] = i
471
```

```
  init_centroids = numpy.array(init_centroids)
  centroids_mapping, centroid_weights, new_target, error = normalize(data
    ,labels,init_centroids,trgt_nrm, message)


  return new_target, centroid_weights, init_centroids, centroids_mapping,
     labels, error, time.time()-time1

def hierarchial2(data,thresh,criterion = 'distance',method = 'ward',
    metric = 'euclidean',trgt_nrm = 'mean',optimal_ordering = True,
    show_dendrogram = False, message = False,):


  time1 = time.time()
  Z = hierarchy.linkage(data,method=method,metric=metric,optimal_ordering
     = optimal_ordering)
  if show_dendrogram:
    hierarchy.dendrogram(Z)
    plt.show()
    plt.clf
  labels = hierarchy.fcluster(Z, t=thresh,depth = 2,criterion=criterion)
  ncolumns = data.shape[1]
  nclus = len(set(labels))
  clusters = numpy.zeros((nclus,ncolumns))
  cluster_counts = numpy.zeros((nclus,1))
  for i in range(data.shape[0]):
    cluster_index = labels[i]
    clusters[cluster_index-1,:] += data[i,:]
    cluster_counts[cluster_index-1] += 1
  init_centroids = clusters / cluster_counts

  centroids_mapping, centroid_weights, new_target, error = normalize(data
    ,labels,init_centroids,trgt_nrm, message)
  return new_target, centroid_weights, init_centroids, centroids_mapping,
     labels, error, time.time()-time1

def greedy_best_match(target_spectra,target_comp,spectra2,comp2,runs):
  nrow = target_spectra.shape[0]
  sg = numpy.random.SeedSequence()
  rg = [numpy.random.Generator(numpy.random.MT19937(s)) for s in sg.spawn
    (runs)]
  losses = numpy.zeros((nrow,runs))
  for irun in range(runs):
    norm_target = target_spectra.max(axis = 1).reshape(nrow,1).copy()
    target_c = norm_target*target_comp

    norm_spectra2 = spectra2.max(axis = 1).reshape(nrow,1).copy()
    comp2_c = comp2*norm_spectra2

    rng = rg[irun]
    order = rng.choice(target_comp.shape[0],size = target_comp.shape[0],
```

```
  replace = False)
   manipulated_matrix = comp2_c.copy()

   indices_correspondence = numpy.zeros(target_comp.shape[0])

   for index in order:
     distances = ((manipulated_matrix - target_c[index,:])**2).sum(axis
  = 1)
     temp_index = numpy.argmin(distances)
     indices_correspondence[index] = temp_index
     manipulated_matrix[temp_index,:] = numpy.Inf

   indices_correspondence = indices_correspondence.astype('int')
   norms = (target_c**2).sum(axis=1).reshape(nrow,1)
   losses[:,irun] = (((target_c - comp2_c[indices_correspondence])**2)/
  norms).sum(axis=1)
 return losses.mean(axis=1).mean(), losses.std(axis=1).mean()
```