Multiple Static Segmentation of Videos using a Convolution of Mixtures of Gaussian processes

Master Project, 2015

Jacob Jon Jensen, Christoffer Samuel Nielsen, Niels Nørgaard Samuelsen
Spring 2015, Aalborg University

June 8, 2015

Resume

This project builds on our pre-thesis on multiple static segmentation of videos, that further builds on an earlier masters thesis from AAU on segmentation of images and image-stacks. Multiple static segmentation of videos is the creation of multiple complementary segmentations of the pixels in a video. These can be visualised as images that shows the underlying structures of a video. This is not a common task, and to our knowledge this work is novel in its approach, both in providing providing multiple complementary segmentations and providing static segmentations in general.

In this project we change the underlying probabilistic model from being non-temporal to being based on Gaussian processes which can be used to represent continuous Gaussian distributions through time. We start by explaining the original method before going into the changes, which have some implications on the algorithm that we use. We show how one part of the algorithm can be changed to work with the new model, so we still optimise our objective function and the rest of the algorithm and implementation can be reused. We implement the method in MATLAB and use this implementation to run our experiments. We experiment with different mean functions for the Gaussian process, both simple polynomial functions and piecewise linear functions, and show how these can be used to obtain good segmentations. We experiment on four different videos, two of which we have also used in our previous work. We find that the method is able find multiple interesting structures in videos, and also segmentations that we were unable to find with our previous method, it does however not give better results in all cases, and especially when looking at the movement in videos as features the results are often less intuitive than those of our previous work. The program runs iteratively with random restarts and therefore it can take a few hours to get the best segmentation. We experiment on how we can improve this running time by performing dimensionality reduction on the video using Principal Component analysis, and observe that we can obtain good results with a far shorter running time. The method is still novel and therefore there is still work to be done in adapting the method for real world applications.

AALBORG UNIVERSITY
STUDENT REPORT

The Faculty of Engineering and Science
Department of Computer Science
Address: Selma Lagerlöfs Vej 300
9220 Aalborg Øst
Phone no.: 99 40 99 40
Fax no.: 99 40 97 98
Homepage: http://www.cs.aau.dk

Project title:
Multiple Static Segmentation of Videos using a Convolution of Mixtures of Gaussian processes

Project period:
Spring 2015

Group name:
mi104f15

Supervisor:
Manfred Jaeger

Group members:
Jacob Jon Jensen
Christoffer Samuel Nielsen
Niels Nørgaard Samuelsen

Copies: 2

Pages: 72

Finished: June 8, 2015

Abstract:

In this project we work with multiple static segmentation of videos, which is the creation of multiple complementary segmentations of the pixels in a video. The work is based on a recently proposed method for multiple segmentation of image stacks and our own work on improving this method. We change the underlying probabilistic model from being non-temporal to being based on Gaussian processes. We experiment with different mean functions for the Gaussian process, and show how we can use a piecewise linear function as a mean function to obtain good segmentations. We find that the method is able find multiple interesting structures in videos, and also segmentations that we were unable to find with our previous method, it does however not give better results in all cases. We also do dimensionality reduction on the data using PCA, and show that we can obtain good results with a far shorter running time.

Contents

CHAPTER 1

Introduction

1.1 Introduction

In this masters thesis we improve on an approach for video segmentation that is based on the image segmentation model from [1] and further improved by us in [2]. Parts of this introduction was also part of our earlier report [2].

Video segmentation is a well-known and researched area in the domain of video processing. The concept encapsulates a number of separate methods that focus on different areas of segmenting a video such as temporal segmentation that typically focuses on splitting a video into scenes [3, 4], but also methods that focus on separating the physical objects that appear in a video into distinct clusters, such as tracking the athletes in a video of a sports event [5].

Our focus is in the new domain of doing clustering on videos to extract static structures that may or may not be easy to discover otherwise. This is to our knowledge an unexplored research area. This means that we provide spatial segmentations of the pixels in the whole video, and unlike when tracking objects, not a segmentation per frame. Videos with a fixed perspective generally reduces the complexity of the solutions, and assuming the input video is of this form, enables us to specialise our method. We define videos with a fixed perspective as videos for which the orientation and position of the camera and landscape as a whole remains fixed. Assuming that videos are filmed with a fixed perspective in our video segmentation enables us to extract structure from the entire video instead of doing it per-frame.

One state of the art method from image segmentation [1], focuses on multiple segmentation of images, i.e. to find several valid segmentations of one or more images. This is an interesting proposition as it is clear that there can be multiple valid interpretations of images or videos in regards to clustering. We have used this method as the basis for our work as the method have shown promising results. As example of the our version of the methods ability to do multiple segmentation on videos, consider Figure F1-1 showing frames from a video of a 4-way intersection. By basing the segmentations on the motion in the video, we could imagine a result that gives us one of any number of paths through the intersection that the cars are using. In Figure F1-2 we can see the result from using the algorithm to give three complementary segmentations that shows the paths the cars are taking in the intersection.

Figure F1-1: Two frames from a 4-way intersection video.

Figure F1-2: The resulting segmentations of the intersection video.

Another possible use case for our method could be segmentation of motion flows in more complicated contexts such as crowds of people moving through each other in various directions. A simulated example of such a video can be seen on Figure F1-3, and a segmentation that captures two crossing directions can be seen on figure F1-4.

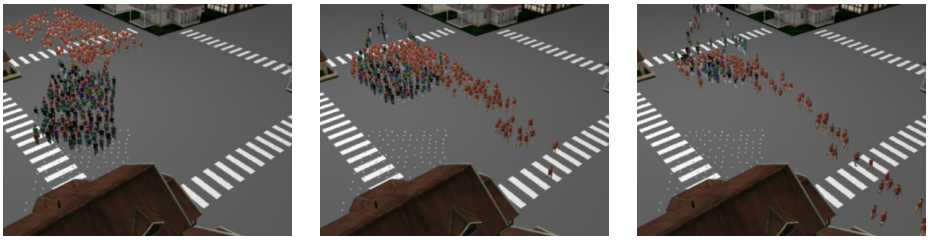


Figure F1-3: Three frames from a simulated crowd video.

Figure F1-4: A resulting segmentations of the crowd video.

1.1.1 Previous results

In our previous work [2] we managed to retrieve video segmentations based directly on the model for image segmentation [1] in which it treats the video as an unordered stack of images. By constructing new features for each pixel that are dependant on the order of the images, we were able to incorporate some temporal information. The result of this feature construction can however be seen as aggregating a video or a stack of images into a single image

and will therefore pose some limitations on how temporal information is utilized for the segmentation. A video with different scenes is not represented very well by this method due to the feature aggregation. For example if we do a simple average of the values we lose the information that some parts of the video might have entirely different values. This previous method can work when the features are constructed carefully and tailored to the video, so aggregation is done more intelligently which may in some cases allow each segmentation to correctly represent a different parts of the video, examples of which can be found in [2].

The frames in most videos are somewhat related to each other and there is a form of progressive change. This information is discarded with the previous method, and we expect a model to be more consistently able to find good solutions if we can take advantage of temporal information, and possibly find new solutions that we were previously unable to find.

Improving on these limitations and better representing the progression of values in a video is also part of the proposed future work in [2] and we proposed to extend the model with temporal information directly incorporated into the model. This is as opposed to simply being incorporated into feature construction. Creating a temporal clustering model is therefore the main contribution of this work.

The chapters is organized so the following chapter introduces the original model and then afterwards we discuss the changes and extensions we have made. After that we provide some implementation details before we show the results of our experiments and conclude on the work.

CHAPTER 2

Convolutional Clustering Algorithm

We will in this chapter first describe the convolutional clustering model for video segmentation. Then we will describe the clustering algorithm that seeks to optimize our objective function. These first two sections are largely based similar sections, that appeared in our previous report [2], which in turn was based on [1].

2.1 The Convolutional Clustering Model

In the this section we use notation where bold symbols are used to denote tuples of variables, e.g. $\boldsymbol{a} = a_1, a_2, \ldots$. Random variables in upper case is the variable itself, and lower case are concrete values of the variables.

2.1.1 Latent Variable Model

The convolutional clustering model is a probabilistic approach based on a latent variable model. In a latent variable model we assume that a data point x is sampled from a joint distribution $P(X, L \mid \theta)$ where X is the observed data variable, $L \in \{1, \ldots, k\}$ is the latent variable that specifies the cluster label, and θ are the parameters of the model. The clustering problem in this model is to learn the parameters θ and finding the label assignment l for point x such that $P(X = x, L = l \mid \theta)$ is maximal.

When the latent variable model is generalized to multiple clustering we have multiple latent variables $\boldsymbol{L} = L_1, \ldots, L_m$. We then have to find multiple labels $L_1 = l_1, \ldots, L_m = l_m$ (abbreviated $\boldsymbol{L} = \boldsymbol{l}$) that maximises $P(X = x, \boldsymbol{L} = \boldsymbol{l} \mid \theta)$.

2.1.2 Model for Multiple Segmentation of Videos

In our model we say that we want to find m segmentations, each of which consist of several clusters, in a video with I pixels. We do not count a pixel multiple times across the frames of the video, meaning that I is the number of pixels we find in a single frame. This also means that we should see the observed data X_i for pixel i as a feature vector with values constructed from all frames in the video. The perhaps simplest feature vector is the three RGB values of a pixel in a frame concatenated over all frames, so with F frames the length is $|X_i| = 3F$. Our focus in our previous work was to examine different methods for constructing X_i.

Our model is structurally identical to a multi-layer hidden Markov random field, as illustrated in Figure F2-1, where each variable X_i represents a pixel. The hidden variables in each layer is organised in a grid and influences the observed variables also organised in a grid. There are m latent variables associated to each pixel $\boldsymbol{L}_{i,\bullet} = L_{i,1}, \ldots, L_{i,m}$. The variables in one hidden layer defines one segmentation $\boldsymbol{L}_{\bullet,k} = L_{1,k}, \ldots, L_{|I|,k}$ and takes values in the set $\{1, \ldots, n_k\}$. Theoretically the model supports a different number of clusters in each segmentation in that the kth segmentation will have n_k clusters, but in our implementation the number of clusters is the same for all segmentations.

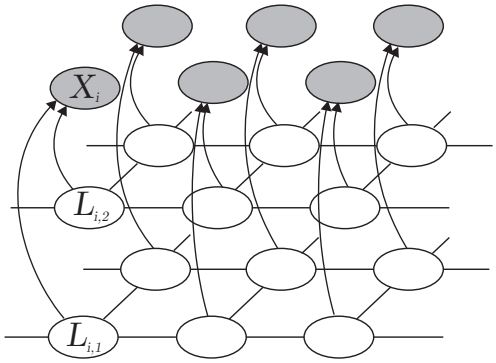


Figure F2-1: Illustration of a multi-layer hidden Markov random field

The joint distribution for all pixels and all labels $P(\boldsymbol{X}, \boldsymbol{L} \mid \theta)$ will serve as our objective function. It is found by obtaining the the marginal distribution $P(\boldsymbol{L} \mid \theta)$ and the conditional probability distribution $P(\boldsymbol{X} \mid \boldsymbol{L}, \theta)$. The marginal distribution is defined using the Potts model, and the conditional distribution is defined by a convolution of Gaussian mixtures. Each of these are defined in the following sections.

2.1.3 The Potts Model

We define the distribution $P(\boldsymbol{L} = \boldsymbol{l} \mid \theta)$ using the Potts model, which is a model inspired from particle physics in which the state of one point is influenced by the state it's neighbouring points. For our application it is used to represent a smoothing effect, where pixels are more likely to be in the same clusters as their neighbours as defined in the square grid structure in the Markov Random Field. It does not necessarily have to be a square grid structure. In other works on segmentation it is also typical to see 8-pixel neighbourhoods.

The smoothing effect is calculated using the latent variables and the parameter $\beta = 1/T$ which is the inverse temperature from the model in physics. Low temperatures mean that particles have a large influence on their neighbours, and thus a high value of β means that pixels are more likely to be placed in the same cluster as their neighbours which creates spatially connected clusters. The model is defined as:

$$P(\boldsymbol{L} = \boldsymbol{l} \mid \beta) = \frac{1}{Z} \prod_{k=1}^{m} e^{V(\boldsymbol{L}_{\bullet,k} = \mathbf{l}_{\bullet,k})\beta}$$

where Z is a normalisation constant and the function V is called the neighbours potential function and is equal to the number of neighbouring pixels that are in different clusters in the current layer:

$$V(\mathbf{L}_{\bullet,k} = \mathbf{l}_{\bullet,k}) = \sum_{i,j:i\sim j} \mathbb{I}(l_{i,k} \neq l_{j,k})$$

with $\mathbb{I}$ being a binary indicator function, that returns 0 if $l_{i,k} = l_{j,k}$ and 1 otherwise. We can rewrite it to a log-likelihood to get a more intuitive negative sum:

$$log(P(\boldsymbol{L} = \boldsymbol{l} \mid \beta)) \approx -\beta \sum_{k=1}^{m} \sum_{i,j:i\sim j} \mathbb{I}(l_{i,k} \neq l_{j,k})$$

which gives us a a lower likelihood for solutions for which pixels share less cluster indices with their neighbours. Notice that we have removed normalization constant, hence the $\approx$. It can be freely removed, because a constant has no influence in determining what is the most likely solution to the objective function. We call this part of the objective function the *neighbourhood cost*.

2.1.4 Convolution of Mixtures

To complete the model we then need the conditional probability distribution $P(\boldsymbol{X} \mid \boldsymbol{L}, \boldsymbol{\theta})$. In this model the pixel features are seen as sampled from a convolution of mixtures of Gaussians. The feature values in the layer is sampled from a mixture distribution consisting of a Gaussian for each cluster/label, with a mean μ and a constant covariance, that is simply the unit covariance matrix.

There is one mixture for each layer each corresponding to a clustering, to get to the model, we take the convolution of those. The convolution of two Gaussians results in a new Gaussian, where the mean and variance is the sum of those of the original distributions. As such this representation allows us to represent X_i as a sum of the underlying layers that can be diverse clusterings. We define the Gaussian means for the mixture components as $\boldsymbol{\mu} \in \{\mu_{1,1}, ..., \mu_{m,1}, ..., \mu_{m,n_k}\}$ where $\mu_{k,j} \in \mathbb{R}^{|\boldsymbol{X}|}$.

The mixture distribution $Z_{i,k}$ in layer k is then defined as a mixture of all the Gaussians of layer k for pixel i:

$$P(Z_{i,k} \mid L_{i,k}, \boldsymbol{\mu}_k) = \sum_{j=1}^{n_k} \mathcal{N}(\mu_{k,j}, \mathbb{1})\mathbb{I}(L_{i,k} = j)$$

where $\mathbb{1}$ is the unit covariance matrix. The indicator function $\mathbb{I}$ ensures that in practice the model for a single variable $Z_{i,k}$, is just the Gaussian corresponding to the label $L_{i,k}$ that is assigned to the variable.

We can then define the model for X_i as the convolution of each mixture, i.e., the probability distribution of the sum of each variable: $P(X_i) = P(Z_{i,1} + ... + Z_{i,m})$. We denote the convolution of two probability distributions $P(A)$ and $P(B)$ as $P(A) * P(B)$. We then have the complete model for X_i:

$$P(X_i \mid \boldsymbol{L}_{i,\bullet}, \boldsymbol{\mu}_1, ..., \boldsymbol{\mu}_m) = P(Z_{i,1} \mid L_{i,1}, \boldsymbol{\mu}_1) * ... * P(Z_{i,m} \mid L_{i,m}, \boldsymbol{\mu}_m)$$

Converting this expression to a log-likelihood we obtain:

$$log(P(X_i \mid \boldsymbol{L}_{i,\bullet}, \boldsymbol{\mu}_1, ..., \boldsymbol{\mu}_m)) = -\sum_{i \in I} ||x_i - \sum_{k=1}^{m} \mu_{k,l_{i,k}}||^2$$

We are calling this second part of the objective function the *data cost*.

2.1.5 The Objective Function

We can then combine our equations for $P(\boldsymbol{L} = \boldsymbol{l} \mid \beta)$ and $P(\boldsymbol{X} \mid \boldsymbol{L}, \boldsymbol{\mu})$ and obtain our objective function that we want to maximise:

$$log(P(\boldsymbol{L} = \boldsymbol{l}, \boldsymbol{X} = \boldsymbol{x} \mid \boldsymbol{\mu}, \beta)) \approx$$

$$-\beta \sum_{k=1}^{m} \sum_{i,j:i \sim j} \mathbb{I}(l_{i,k} \neq l_{j,k}) - \sum_{i \in I} ||x_i - \sum_{k=1}^{m} \mu_{k,l_{i,k}}||^2$$

By a change of sign we get an energy function that instead needs to be minimised. It is this energy function that in practice will be the target of optimisation in the clustering algorithm.

2.2 Clustering Algorithm

The clustering algorithm is a two-step iterative process similar to the EM algorithm. We will start with an overview of the algorithm, and the describe the individual steps in more detail afterwards.

The pseudo code can be seen in Algorithm 1. We saw in the previous section that the parameters of the model are the means $\boldsymbol{\mu}$ and β. The smoothing parameter β will in the algorithm be taken as a hyper parameter, and the parameters that will have be learned is then the means. The output is the segmentations specified by the labellings $\boldsymbol{L}$. The first step in the algorithm is the feature creation, where the feature vectors for each pixel is created. In a later section we give some examples of video features. The second step is a random initialisation of the means. Then follows the two-step iterative process that continues until convergence to a local minimum of the energy function. The first step, called the MAP-step, calculates an approximation to the most likely label assignment $\boldsymbol{L} = \boldsymbol{l}$ using an algorithm called α-expansion given the current setting of $\boldsymbol{\mu}$. The following step, called the M-step (maximisation of likelihood), updates the means by coordinate descent on the energy function.

From here we refer to X_i as the pixels, defined as consisting of their associated feature values.

2.2.1 Initialization

The means are initialised using a method devised in our previous work [2]. Assume that we want m segmentations with c clusters each. We then randomly pick c pixels $x_1, \ldots, x_c$. For each picked pixel x_j we use it to initialise

Algorithm 1: Overview of the clustering algorithm.

input : A video and the smoothing parameter β
output: The labellings $\boldsymbol{L}$

Construct x_i for each original pixel i;
Initialize $\boldsymbol{\mu}$;
while *energy has not converged* **do**
    Find label assignment $\boldsymbol{L} = \boldsymbol{l}$ with the α-expansion algorithm;
    Update means $\boldsymbol{\mu}$ by coordinate descent;
end

the means belonging to cluster index j in all m layers, such that a convex combination of the mean vectors is equal to x_j. Formally, the means are initialised as $\mu_{k,j} = x_j * w_{k,j}$ where we have random weights $w_{1,1}, \ldots, w_{m,c}$ such that $\sum_{k=0}^{m} \mu_{k,j} = x_j$. This method ensures that in the first MAP-step the three pixels are heavily biased to be placed separate clusters. This helps guarding against results with one or more empty clusters, which was a problem we faced previously. The weights are random instead of fixed, e.g. always $\frac{1}{3}$ for three layers, for having as much variance in the means as possible for multiple runs of the algorithm.

2.2.2 The MAP-step with α-expansion

The α-expansion algorithm [6] can find a joint labelling s of all pixels that is a local minimum of an energy function $E(s)$ using a graph cut approach. Since we are doing multiple segmentation and every pixel is assigned a cluster label for each segmentation, the labelling of a pixel i can be seen as a tuple $(l_{i,1}, \ldots, l_{i,m})$. In order to use the α-expansion algorithm, we have to interpret every tuple labelling as a single label $s(i) = (l_{i,1}, \ldots, l_{i,m})$. This means that we are creating $\prod_{k=1}^{m} n_k$ labels for the α-expansion algorithm, one for each possible combination of cluster assignments across the segmentations. This also means that the we are gaining a complexity that is exponential in the number of segmentations.

For the α-expansion algorithm the energy function must be on the following form

$$E(s) = \sum D_i(s(i)) + \sum_{i,j:i\sim j} V_{i,j}(s(i), s(j))$$

Here $D_i(s(i))$ is the independent data cost of the labelling $s(i)$ given the observed data for pixel i. The term $V_{i,j}(s(i), s(j))$ is the smooth cost that gives a preference for spatial smoothness. This requirement is the reason we change sign on the log-likelihood function. In our case the data cost will be

$$D_i(s(i)) = \| \boldsymbol{x}_i - \sum_{k=1}^{m} \mu_{k,l_{i,k}} \|^2$$

and the smooth cost will be

$$V_{i,j}(s(i), s(j)) = \beta \sum_{k=1}^{m} \mathbb{I}(l_{i,k} \neq l_{j,k})$$

The algorithm guarantees that the returned solution is a local minimum that is within a factor of the global minimum. For our algorithm this factor is $2m$ [2], two times the number of layers.

2.2.2.1 Algorithm Complexity

Though we gain a complexity that is exponential in the number of layers, the α-expansion is linear in the number of pixels, and the computation of the data cost makes it linear in the size of the feature vector $|X_i|$. The full complexity is $O(I \times |X_i| \times k^m)$, where we use k to denote the number of clusters in a layer.

2.2.3 The M-step with Coordinate Descent

During the M-step the means are updated by Coordinate descent on the energy function and thus increasing the likelihood of the current setting. In this step new optimal values are computed for every μ, and then every mean takes simultaneously a step towards their new optimal values. The approach is similar to [7], except that we are not using a regularisation term.

We will in this section describing the general procedure for updating a single mean. The specific mean we are updating is denoted by $\tilde{\mu}$, and its computed new optimal value will be denoted by $\hat{\mu}$. The optimal values can be found by taking the partial derivative of our energy function w.r.t. $\tilde{\mu}$. Since we assume that everything but $\tilde{\mu}$ is constant, we can then just take the partial derivative to the following simplified energy function, where we have removed the terms that does not involve $\tilde{\mu}$

$$\sum_{i \in \tilde{I}} ||x_i - \sum_{k=1}^{m} \mu_{k,l_{i,k}}||^2$$

In the above expression $\tilde{I}$ is the set of pixels which is assigned to the cluster of $\tilde{\mu}$. The equation can be simplified a little further by subtracting all means except $\tilde{\mu}$ from each x_i to get a value y_i

$$y_i = x_i - \sum_{k=1}^{m} \mu_{k,l_{i,k}} + \tilde{\mu} \tag{2.1}$$

We can then use y_i instead in the function

$$\sum_{i \in \tilde{I}} ||y_i - \tilde{\mu}||^2 \tag{2.2}$$

When we take the partial derivative of the above function we get

$$\frac{\partial}{\partial \tilde{\mu}} \sum_{i \in \tilde{I}} ||y_i - \tilde{\mu}||^2 = -2 \sum_{i \in \tilde{I}} (y_i - \tilde{\mu})$$

We set the gradient equal to the zero vector to get the optimal mean value, and remove the -2 by multiplying each side by $-\frac{1}{2}$

$$\mathbf{0} = \sum_{i \in \tilde{I}} (y_i - \hat{\mu})$$

Now we can isolate the mean on the left-hand side of the equation, and that will be the optimal mean value, which is actually just the mean value of all y_i

$$\hat{\mu} = \frac{1}{|\tilde{I}|} \sum_{i \in \tilde{I}} y_i \tag{2.3}$$

CHAPTER 3

A Temporal Model

This chapter will introduce our proposal for a new model for video segmentation that incorporates the temporal aspect of videos. The first sections in the chapter will explain the more formal parts, and the last few sections will explain what changes the new model brought to the algorithm.

3.1 A Temporal View of the Model

Let us first review why the old model is incapable of capturing the the temporal aspects of a video. The old model uses multivariate Gaussian distributions. If the constructed feature vectors for each pixel had for example length 100, then it would be 100 dimensional Gaussian distributions. The mean of each dimension in the Gaussian distribution is optimised independently from the others. This is why the model is non-temporal. In a video we can make the reasonable assumption that the mean values of a cluster of pixels at frame f is close to the mean of the same cluster at frame $f-1$ and $f+1$. Imagine for example a part of a video that gets brighter throughout the video. The mean of the pixels in that area will then gradually increase, perhaps in a linear fashion. A temporal view we could bring into the model is then to say that we have a (multivariate) Gaussian distributions per frame, and that these distributions changes during the video. This is what is known as a Gaussian process.

3.2 Gaussian Processes

A Gaussian process is a stochastic process, that describes a continuous Gaussian distribution which means we can evaluate the probability distribution in infinitely many locations [8, 9]. However it is often sufficient to say that we can evaluate it at any location on the real line. Often it is evaluated in time, but it can be a multidimensional process for example a spatial process where we can evaluate it at any point in two dimensions. In our case we only need one dimension which corresponds to the temporal nature of videos.

A simple example of a Gaussian Process is the temperature in the world. Each year we could sample the temperature different places around the globe and they might fit a Gaussian distribution. This distribution is similar the following year, but probably not identical, the temperature have been rising for the last couple of decades. This behaviour could be modelled by a Gaussian process where the parameters of the process determines that the temperature rises in the long term.

A Gaussian process is however much more flexible than simply describing an increase over time. If, for example, the temperature changes periodically

because we measure the temperature on a monthly basis instead of a yearly, the long term tendency is still an increase, but it does not increase every month because of the seasons. To model this, we need a periodic trend besides the long term trend, and this can also be modelled by a Gaussian process.

A multivariate Gaussian distribution is defined by two parameters, the covariance matrix and the mean vector of the distribution. When we are considering a Gaussian process the parameters for the associated Gaussian distribution of each point, changes depending on the location. The mean value is easy to describe as a function which is often constant. But since we cannot write a covariance matrix that handles an infinite number of locations we also need a function in this case and we use a *covariance kernel function* $k(t_i, t_j)$. It gives the covariance between two locations t_i and t_j. We can then define the covariance matrix for a set of locations $\boldsymbol{t}$ as

$$\boldsymbol{K}(\boldsymbol{t}, \boldsymbol{t}) = \begin{bmatrix} k(t_1, t_1) & k(t_1, t_2) & \cdots & k(t_1, t_n) \\ k(t_2, t_1) & k(t_2, t_2) & \cdots & k(t_2, t_n) \\ \vdots & \vdots & \vdots & \vdots \\ k(t_n, x_1) & k(t_1, t_2) & \cdots & k(t_n, t_n) \end{bmatrix}$$

So to describe a Gaussian process we need a mean function and a covariance kernel function. Formally we can write the evaluation of a Gaussian process at locations $\boldsymbol{t}$ as

$$p(\boldsymbol{t}) = \mathcal{N}(\boldsymbol{m}(t), \boldsymbol{K}(\boldsymbol{t}, \boldsymbol{t}))$$

where $\boldsymbol{m}$ is the mean function.

The covariance function k can be in many different forms but the constraint is that the covariance matrix is generates must be *positive semi definite* which is the case if and only if the eigenvalues of the matrix are non-negative. This still allows for a lot of different functions, so one has to choose one based on the use case. In many cases we assume that the data is dependent on past data and how long ago the past data was observed. To incorporate this information in a covariance function a *stationary* covariance function is used which means that the function is dependant on $|t_i - t_j|$. The most widely used function of this class is the *squared exponential function* which is defined as

$$k(t_i, t_j) = h^2 exp[-(\frac{t_i - t_j}{\lambda})^2]$$

The hyper parameter h decides the output scale of the function and λ the time scale i.e. how fast the correlation between locations decreases. Often a certain amount of noise in observations is assumed, this can also be modelled in the covariance function by adding a noise term. We expect this noise to be uncorrelated between different locations and therefore we multiply it by the identity matrix $\boldsymbol{I}$ and add it to the covariance function. A squared exponential function with noise would then be

$$k(t_i, t_j) = h^2 exp[-(\frac{t_i - t_j}{\lambda})^2] + \sigma^2 \boldsymbol{I}$$

where σ^2 is the noise variance.

The mean function can be any continuous function but often the average values of the sample functions drawn from a Gaussian process is assumed to

be 0, and thus the mean function is assumed to be constant. This means that the functions drawn from a Gaussian process is purely based on the covariance function. However in our model we work with the opposite case, we have knowledge of the mean values from the videos, but assume the covariance to be zero, there for we simply use the identity matrix as our covariance matrix. This makes for a simpler case in the maximisation step, where we are optimising the parameters, because we only need to optimise the mean functions of the Gaussian processes. If we also had to learn covariance function parameters it would make the maximisation step harder. So we assume it is enough to model the temporal relationship only through mean functions.

3.3 The Temporal Convolutional Clustering Model

First and foremost the temporal model requires a specific view of the pixel data. In the old model the feature vector x_i that you create for a pixel i depended entirely on the feature creation method. For example you could aggregate the motion flow information of any number frames into just 8 features per pixel (an example is our *SIFT histogram* features from [1]). Because we now need to model a Gaussian processes it is a requirement that x_i contains data from every frame. Therefore we now introduce the notation $x_{i,f}$ which is the pixel value(s) at frame f for pixel i. In reality we would most likely have a vector of pixel values per frame, for example when we use RGB values we would have both a red, green and blue component, but for simplicity the model is easier explained if we assume that $x_{i,f}$ is just a single value.

The temporal model definition will in some parts be identical to the old model's definition. An structural overview of the model is illustrated in Figure F3-1. The illustration is similar to Figure F2-1 except that each pixel variable X_i is split into three variables $X_{i,1}$, $X_{i,2}$ and $X_{i,3}$; one for each of three frames. We may still refer to X_i, in which case we mean $X_i = X_{i,1}, \ldots, X_{i,F}$ for F frames. The notation to index the individual frames will only be used when we talk about specific the specific values for these frames.

The joint distribution for all pixels and all labels is $P(\boldsymbol{X}, \boldsymbol{L} \mid \boldsymbol{\theta}, \beta)$. In the old model θ was used as the general symbol for model parameters, but now $\boldsymbol{\theta}$ is used specifically as the mean function parameters.

The joint distribution will still be found by combining two distributions: the marginal distribution $P(\boldsymbol{L} \mid \beta)$ and the conditional distribution $P(\boldsymbol{X} \mid \boldsymbol{L}, \boldsymbol{\theta})$. The model's latent variable structure remains unchanged so there is still one latent variable per layer associated to each pixel. Therefore the Potts model specification for use in the marginal distribution in Section 2.1.3 also remains unchanged. The definition of the conditional distribution has changed, because it has now become a convolution of mixtures of Gaussian processes.

3.3.1 Convolution of Mixtures of Gaussian Processes

The model now defines a Gaussian process, instead of a Gaussian distribution, for each cluster. The parameters $\boldsymbol{\theta}$ is the set of mean function parameters for each Gaussian process in each layer $\boldsymbol{\theta} = \theta_{1,1}, \ldots, \theta_{m,1}, \ldots, \theta_{m,n_k}$.

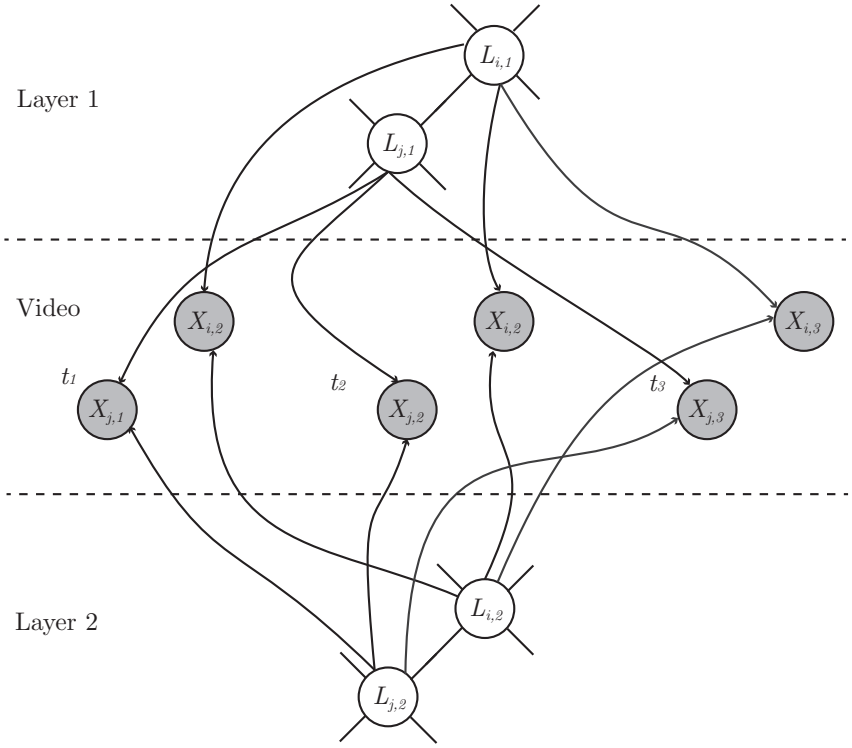


Figure F3-1: Illustration of the temporal model for three frames and two pixels

The feature values in a layer is now sampled from a mixture of Gaussian processes, whose mean function is defined by some parameters θ and the unit covariance matrix. Our decision to keep zero covariance between the frames for a pixel matches the illustration in Figure F3-1, because we can here see that $X_{i,2}$ is conditionally independent from $X_{i,1}$ and $X_{i,3}$ given $L_{i,1}$ and $L_{i,2}$.

The change from Gaussian distributions to Gaussian processes is perhaps easiest explained by thinking of the mean function as a vector, in which there is an entry for each frame f with the mean function value $M\theta, f)$ as specified by θ. In fact, there is also some mathematical convenience in this. We can keep the mean vectors $\boldsymbol{\mu}$ in our model, but redefine them so the mean vector in layer k and cluster j is $\mu_{k,j} = [M\theta_{k_j}, 1), \ldots, M\theta_{k_j}, F)]^\top$. So instead of the mean vectors being parameters, they can now be thought of as being generated by the mean function. This is not only to simplify the model definition, but also because it matches the algorithm implementation, since the algorithm can then support both the non-temporal model and the temporal model. Therefore our way of defining the model also helps explaining how the clustering algorithm is backwards compatible.

With $\boldsymbol{\mu}$ defined by $\boldsymbol{\theta}$ we are allowed to have a similar definition of the mixture distribution as we used in Section 2.1.4. We define $Z_{i,k}$ in layer k as a mixture of all the Gaussians processes of layer k for pixel i:

$$P(Z_{i,k} \mid L_{i,k}, \boldsymbol{\theta}_k) = \sum_{j=1}^{n_k} \mathcal{N}(\mu_{k,j}, \mathbb{1})\mathbb{I}(L_{i,k} = j)$$

The mean of the convolution of two Gaussian distributions was the sum of the two Gaussian distributions' means. Similarly, the function that describes the means of a convolution of Gaussian processes is the sum of the mean functions of each Gaussian processes. Note that this is only the case for Gaussian distributions and thus differs from the definition of a convolution of two arbitrary functions f and g which is not the simply sum. The easiest way of thinking of the convolution of the processes is to add them on top of each other, as illustrated in Figure F3-2 which shows the convolution of two example mean functions. In our context the horizontal axis represents a videos frame number. We can therefore again reuse the definitions from the

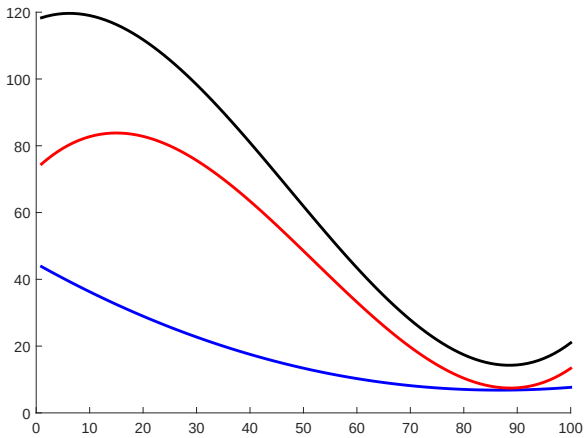


Figure F3-2: The convolution (black line) of two mean functions.

old model and define the conditional distribution for X_i using the sum of the mean vectors because they were defined by the mean function parameters:

$$P(X_i \mid \boldsymbol{L}_{i,\bullet}, \boldsymbol{\theta}_1, ..., \boldsymbol{\theta}_m) = P(Z_{i,1} \mid L_{i,1}, \boldsymbol{\theta}_1) * ... * P(Z_{i,m} \mid L_{i,m}, \boldsymbol{\theta}_m)$$

As a result, when we combine the marginal distributions and conditional distribution we get a similar log-likelihood

$$log(P(\boldsymbol{L} = \boldsymbol{l}, \boldsymbol{X} = \boldsymbol{x} \mid \boldsymbol{\theta}, \beta)) \approx$$

$$-\beta \sum_{k=1}^{m} \sum_{i,j:i\sim j} \mathbb{I}(l_{i,k} \neq l_{j,k}) - \sum_{i \in I} ||x_i - \sum_{k=1}^{m} \mu_{k,l_{i,k}}||^2$$

3.4 Mean Functions for Gaussian Processes

The mean function in a Gaussian Process can have any shape, but we are restricting ourselves to only focus on a few types of simple mean functions. For example we could restrict the mean functions to only be linear. The reason for this restriction is both because of scope, since we need to evaluate the usefulness of the mean functions somehow, but also because too much time spent on choosing appropriate mean functions is over-engineering or overfitting the mean function to the video. Instead it would be more practical to have a few functions that can be used for any video. The multiclustering tool has from the start been seen as a general purpose video-exploration tool, so in that context it is not really possible to reason much about which complex

mean function that would be the most appropriate. We have chosen to focus on the polynomial family of functions and the piecewise linear function, which is a special kind of polynomial function.

3.4.1 Polynomial Mean Function

Polynomial functions as mean functions can effectively be used to represent videos with smooth feature value progression throughout the videos frames. The complexity of the polynomial function is determined the by the degree d of the function. The polynomial mean function with $d = 1$ is the linear mean function. We will treat the degree as a hyper parameter that is chosen before the algorithm is run. This is both because of the above mentioned issue of overfitting, but also because it would be a much more complex problem if this parameter also had to be learned. The two mean functions in Figure F3-2 are examples of two polynomial mean functions of degree $d = 3$.

Polynomial functions can be least squares fitted to a pixel trajectory with a computational complexity of $\mathcal{O}(n^2)$, with n as the number of frames of the video, using Vandermonde matrices [10]. Polynomial function fitting is available in MATLAB by the function *polyfit*.

3.4.2 Piecewise Mean Functions

The polynomial functions seems an appropriate choice when we can assume that the mean of pixels' trajectories is somewhat smooth. You could though image that the mean values change dramatically one or multiple times throughout a video, for example if the video contains scene changes. In that case it would be appropriate to have multiple mean functions that correspond to different parts of the video. So what we need is in fact piecewise mean functions, since a piecewise function is the concatenation of multiple functions, also called pieces.

There is two basic methods to fitting a piecewise function. The first is where you control the number of knots k, which is the number of endpoints in the piecewise. A piecewise function with two pieces have three knots; one in either end, and one where they two pieces meet. An algorithm for fitting such a piecewise function will then determine where to place the knots to reduce squared error. The second method is where an algorithm is given a bound on the error the piecewise function may at maximum give. Such an algorithm would create as few functions as possible needed to be within the error bound. This method is not a least squares fitting method, because its first priority is to minimise the number of knots.

We decided to only focus on piecewise linear functions. We do not believe there is a need to consider piecewise polynomial functions of higher degree, since they already get flexibility by the freedom to determine where the knots should be. We choose two fitting algorithms that are based on the two methods explained above. The algorithm that uses a fixed number of knots fits a continuous piecewise function, while the algorithm that uses an error bound fits a discontinuous piecewise function.

Continuous Piecewise Linear Mean Function with Fixed Number of Knots

The algorithm we use is from the Shape Language Modeling toolbox for MATLAB[11]. The toolbox provides a function that fits a piecewise function on an input vector. The parameters of the function is the number of knots k and the degree of individual functions, which we set to 1, because we want piecewise linear functions. It utilises the MATLAB optimization toolbox to find a least square fit of the function with the chosen number of knots. The complexity of the fitting is not provided, but empirical testing shows that the running time is linear in the number of input points which is in our case the number of frames, and linear in the number of knots but with a higher constant. The testing showed that running with up to 9 knots is reasonable. An example of two continuous piecewise linear mean functions is shown in Figure F3-3 along with their convolution.

Discontinuous Piecewise Linear Mean Function with Error Bound

The algorithm we use comes from [12], and implemented by ourselves in MATLAB. It is an method that tries to fits as few linear functions as possible to the input data that all lie with the error bound. The resulting piecewise function is discontinuous because it does not attempt to match the endpoints of the linear function it is made up of. It is an approximate method in that the fitted discontinuous piecewise function only approximately minimises squared error w.r.t. the number of knots the algorithm found. Because it is an approximate method it achieves an efficient running time that is linear in the input size.

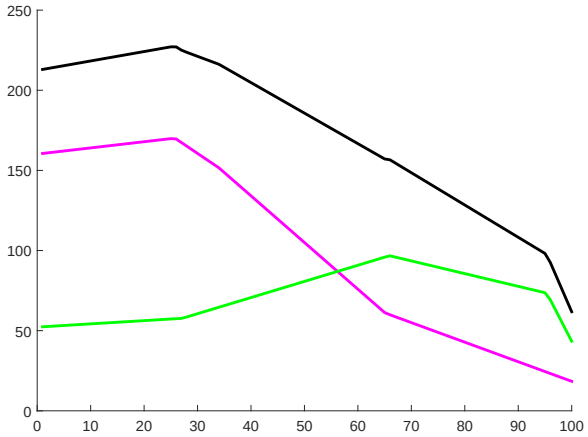


Figure F3-3: The convolution (black line) of two continous piecewise linear mean functions with $k = 5$.

3.5 Clustering Algorithm

In this section we will introduce the main parts of the clustering algorithm based on the model presented. Focus is on the contributions of this work which lies primarily with the M-Step using coordinate descent.

3.5.1 Initialization of Mean Functions

As explained in Section 2.2.1 we initialize the means by choosing a number of pixels x and base the initial means on their values. However when we are using mean functions we cannot use the feature values for the pixel directly, but need to fit the function to the pixel values from each frame. The method is very similar, but instead multiplying the pixel features x_j by the weights w, we weight the function parameters θ. Formally this means that for m segmentations with c clusters, we fit the desired function type to x_j to obtain function parameters $\breve{\theta}$. Then we weight them to obtain the final parameters used for initialisation $\theta_{k,j} = \breve{\theta}_j * w_{k,j}$ where we use random weights $w_{1,1}, \ldots, w_{m,c}$ such that $\sum_{k=0}^{m} \theta_{k,j} = \breve{\theta}_j$. We can do this because our functions are polynomials or piecewise polynomials, and the evaluation of a polynomial, where the parameters is the sum of the parameters for a set of other polynomial functions of the same degree, is equal to the sum of the evaluations of the polynomials in the set. This results in the convex combination of the mean functions being equal to the function with parameters $\breve{\theta}$.

3.5.2 The MAP-step with α-*expansion*

The MAP-step remains unchanged, because it just finds a joint labelling of all pixels using the α-*expansion* algorithm.

3.5.3 Coordinate Descent for Mean Functions

The algorithm for the non-temporal model uses coordinate descent in the M-step where the mean vectors are updated one by one. Now since the mean vectors are created from mean functions, we will have to find the parameters of the mean functions instead of updating the mean vectors directly. If $\tilde{\theta}$ is the current setting of parameters of a piecewise function we wish to update, $\hat{\theta}$ are the new calculated optimal parameters. In this section we will show how to obtain $\hat{\theta}$ by starting with a mathematical explanation and then follow with the pseudo code in Section 3.5.3.2 to get an overview of the method.

In the old model we first computed $\hat{\mu}$ to

$$\hat{\mu} = \frac{1}{|\tilde{I}|} \sum_{i \in \tilde{I}} y_i$$

and then updated the means. Coordinate descent for mean functions will work in a similar way. When iterating over the mean functions we apply the following procedure: From a current setting of parameters $\tilde{\theta}$ we have $\tilde{\mu}$ from which we can find $\hat{\mu}$. We then do a least squares refit of the mean function to $\hat{\mu}$ giving $\hat{\theta}$, which then becomes the new parameters for that mean function.

As an example lets consider the case where we are in the midst of updating one from the set of linear mean functions. The parameters for the mean function we are updating consists of two coefficients $\tilde{\theta} = (\tilde{a}, \tilde{b})$. Let $linear\_mean$ be a function that we use to create the corresponding mean vector from the parameters

$$\tilde{\mu} = linear\_mean(\tilde{a}, \tilde{b})$$

From $\tilde{\mu}$ we calculate $\hat{\mu}$ and do a least squares fit a new linear function to $\hat{\mu}$ giving us a new set of parameters.

$$(\hat{a}, \hat{b}) = linear\_fit(\hat{\mu})$$

This process is the same for other mean functions except that we use other fitting functions giving other sets of parameters, and that the mean vectors are created differently of course. It might not be immediately obvious that least squares fitting of mean functions to $\hat{\mu}$ does in fact minimise our objective function, so we will dedicate the following section to showing it.

3.5.3.1 Correctness of Coordinate Descent Method

The argument we will make in this section holds for the class of mean functions that can be least squares fitted to a vector. This does *not* include the discontinuous piecewise linear mean function, since it is constrained to be least squares fitted within the error bound.

First we will introduce some notation for use in this section. We use the notation $y_{i,f}$ to denote the value of the fth element of the vector y_i. We use the function $m(\tilde{\theta}, f)$ to give us the value at frame f of a mean function specified by parameters $\tilde{\theta}$. Using this notation we can write Equation 2.2 from Section 2.2.3 in a "constrained" form where we replace $\tilde{\mu}$ with the mean function

$$\sum_{f=1}^{F} \sum_{i \in \tilde{I}} (y_{i,f} - m(\tilde{\theta}, f))^2 \tag{3.1}$$

When we fit a mean function that minimizes the squared error, to $\hat{\mu}$ we are minimising

$$\sum_{f=1}^{F} (\frac{1}{|\tilde{I}|} \sum_{i \in \tilde{I}} y_{i,f} - m(\tilde{\theta}, f))^2 \tag{3.2}$$

The equation is similar, but instead of minimising the sum of the individual squared errors we are minimising the square of the average error.

The correctness of our coordinate descent method comes from the fact that minimising Equation 3.1 is equivalent to minimizing Equation 3.2. It can be shown in the following way: We take the partial derivative of Equation 3.1 w.r.t. a parameter $\theta \in \tilde{\theta}$ and set it to zero and with some rewritings arrive at the same equation as when we similarly take the partial derivative of Equation 3.2. We will start with Equation 3.1

$$\frac{\partial}{\partial \tilde{\theta}} \sum_{f=1}^{F} \sum_{i \in \tilde{I}} (y_{i,f} - m(\tilde{\theta}, f))^2 = -2 \sum_{f=1}^{F} \sum_{i \in \tilde{I}} (y_{i,f} - m(\tilde{\theta}, f)) \frac{\partial m(\tilde{\theta}, f)}{\partial \tilde{\theta}}$$

We set the partial derivative equal to zero and remove the -2 by multiplying both sides by $-\frac{1}{2}$.

$$0 = \sum_{f=1}^{F} \sum_{i \in \tilde{I}} (y_{i,f} - m(\tilde{\theta}, f)) \frac{\partial m(\tilde{\theta}, f)}{\partial \tilde{\theta}}$$

By doing a series of rewrites we can make the mean of $y_{i,f}$ over all i appear in the equation. First $\frac{\partial m(\tilde{\theta}, f)}{\partial \tilde{\theta}}$ is moved outside the second summation

$$0 = \sum_{f=1}^{F} \frac{\partial m(\tilde{\theta}, f)}{\partial \tilde{\theta}} \sum_{i \in \tilde{I}} (y_{i,f} - m(\tilde{\theta}, f))$$

Then instead of subtracting $m(\tilde{\theta}, f)$ inside the summation term, we subtract $|\tilde{I}|$ times $m(\tilde{\theta}, f)$ afterwards

$$0 = \sum_{f=1}^{F} \frac{\partial m(\tilde{\theta}, f)}{\partial \tilde{\theta}} (\sum_{i \in \tilde{I}} y_{i,f} - |\tilde{I}| m(\tilde{\theta}, f))$$

And lastly the multiplication of $|\tilde{I}|$ is moved outside the second summation term

$$0 = \sum_{f=1}^{F} \frac{\partial m(\tilde{\theta}, f)}{\partial \tilde{\theta}} |\tilde{I}| (\frac{1}{|\tilde{I}|} \sum_{i \in \tilde{I}} y_{i,f} - m(\tilde{\theta}, f))$$

Now for Equation 3.2 we similarly take the partial derivative

$$\frac{\partial}{\partial \tilde{\theta}} \sum_{f=1}^{F} (\frac{1}{|\tilde{I}|} \sum_{i \in \tilde{I}} y_{i,f} - m(\tilde{\theta}, f))^2 = -2 \sum_{f=1}^{F} \frac{\partial m(\tilde{\theta}, f)}{\partial \tilde{\theta}} (\frac{1}{|\tilde{I}|} \sum_{i \in \tilde{I}} y_{i,f} - m(\tilde{\theta}, f))$$

Again the partial derivative is set equal to zero and -2 is removed

$$0 = \sum_{f=1}^{F} \frac{\partial m(\tilde{\theta}, f)}{\partial \tilde{\theta}} (\frac{1}{|\tilde{I}|} \sum_{i \in \tilde{I}} y_{i,f} - m(\tilde{\theta}, f))$$

We can multiply both sides by $|\tilde{I}|$ to arrive at the same equation as above

$$0 = \sum_{f=1}^{F} \frac{\partial m(\tilde{\theta}, f)}{\partial \tilde{\theta}} |\tilde{I}| (\frac{1}{|\tilde{I}|} \sum_{i \in \tilde{I}} y_{i,f} - m(\tilde{\theta}, f))$$

which shows that the parameters of any mean function that minimises squared error w.r.t. $\hat{\mu}$ does in fact minimize our energy function.

3.5.3.2 The pseudo code

The pseudo code can be found in Algorithm 2.

Algorithm 2: Overview of the coordinate descent algorithm. To bring down the complexity a little we assume only one feature being used. It is easy to generalize simply by using matrices of feature values instead of vectors, and fitting a function for each feature.

input : A video $\boldsymbol{I}$ with labels $\boldsymbol{L}$, a type of mean function M and the current mean parameters $\tilde{\theta}$
output: Updated mean function parameters $\hat{\theta}$

```
/* Calculate energy using Equation 3.1                              */
```
OldEnergy = CalculateEnergy($\boldsymbol{I}$,$\boldsymbol{L}$,M, $\tilde{\theta}$);
$\hat{\theta} = \tilde{\theta}$;
Layers = Random order of layers;
for *Layer in Layers* **do**
  Clusters = Random order of clusters in Layer;
  for *Cluster in Clusters* **do**
    correspondingClusters = All clusters in all *other* layers to which *Pixel* belongs;
```
    /* We now calculate the sum of the corresponding
       means as in Equation 2.1                                     */
```
    μ_{sum} = Zero-vector of length $|frames|$;
    for *Cluster* l, c *in correspondingClusters* **do**
      $\mu_{sum} = \mu_{sum} + M$ evaluated using $\tilde{\theta}_{l,c}$ at each frame;
    end
    $y_{sum} = 0$;
    for *Pixel* X *in* $\boldsymbol{I}$ *that belongs to Cluster* **do**
```
      /* X and DataError is a vector of values for
         every frame                                                */
```
      $y_i = X$ - μ_{sum};
      $y_{sum} = y_{sum} + y_i$;
    end
```
    /* Normalize to get μ̂ according to Equation 2.3               */
```
    $\hat{\mu} = y_{sum}/|\tilde{I}|$;
```
    /* Fit returns parameters for M fitted to the data,
       depending on the type of M, see Section 3.4                */
```
    $\hat{\theta}_{Layer,Cluster}$ = Fit(M, $\hat{\mu}$);
  end
end
NewEnergy = CalculateEnergy($\boldsymbol{I}$,$\boldsymbol{L}$,M, $\hat{\theta}$);
if *OldEnergy*>=*NewEnergy* **then**
  return $\hat{\theta}$;
else
  TerminateRun;
end

The idea behind the code is to update each of the mean functions in turns, and then use the updated mean function when updating the next mean function. Since we have one mean function for each cluster in each layer we loop through the layers and the clusters, and perform the update inside this loop. To ensure that a specific mean does not dominate the results

because it is chosen first in every iteration of the overall algorithm we pick the order of the layers and clusters at random.

Inside the loop we take the feature values of each pixel X and subtract the mean values from the clusters in the other layers, that the pixel belongs to. Unless the means fit perfectly to every pixel which is never happening in practice, we then end up with a deviation for every pixel y_i on the remaining amount in every time step. The total error is divided by the number of pixels to get the average error for a pixel. We have then obtained a vector of the optimal mean values for every time step $\hat{\mu}$. Finally we fit our desired mean function to these values to obtain the new parameters $\tilde{\theta}$. After this we continue with the same procedure for the remaining means in the other layers and clusters until every mean function have been updated.

If the total error i.e. the energy of the solution have not improved by this procedure we assume that we have arrived at a local optimum and terminate the current run of the overall algorithm, otherwise we continue to the MAP step to retrieve a new labelling.

The complexity of the algorithm is better than that of the MAP step mentioned in Section 2.2.2.1, however the constant time it takes to fit a mean function means that this part of the algorithm still makes for most of the time spent running our method. The complexity is linear in most of the parameters, but the influence of the number of frames depends on which mean function is being used. The complexity of fitting a polynomial function is $O(Frames^2)$, which means we end up with a total complexity of $O(Layers \times Clusters \times Pixels \times Features \times Frames^2)$.

3.6 Features

In our previous work [2] we constructed a number of feature types with the purpose of covering different style and videos and also to respect the temporal aspect of videos. Our RGB features were represented as mean the mean RGB values aggregated over all frames or as as a long feature vector containing all RGB values of all frames. Our trajectory features did incorporate temporal information but only by creating feature values that covered 10 predetermined discrete parts of the video, with each piece aggregated over its frames. In this work we have incorporated the temporal aspects of videos into the model itself and we don't need to rely on features to represent the temporal structure. This means that the number of relevant feature types have been reduced to RGB and SIFT flow. These can be used by themselves or combined. Furthermore the framework is still easily extensible with new feature types.

3.6.1 RGB-features

RGB-features for a given pixel are given by the 0 to 255 Red, Green, Blue values. RGB is the general purpose feature that covers use cases in the widest range of videos used to extract basic visual structures.

3.6.2 Motion Features based on SIFT Flow

Scale Invariant Feature Transform (SIFT) [13] is a standard method for describing local features in an image and has been shown to have a wide spectrum of applications in computer vision. Given an image it will create a

large feature vector for a number of *points of interest* that are detected. This feature vector is called a SIFT descriptor and captures the local gradient information around the point. It has the advantage that it can reliable detect features under changes to orientation, scale, camera perspective and lightning.

The SIFT flow algorithm [14] creates SIFT descriptors for every pixel in an image and attempts to match these descriptors between two images, and in doing so calculates the displacement of every SIFT descriptor. The algorithm can be used for motion field prediction if the two images are consecutive frames from a video, because the calculated displacement can be interpreted as a motion vector. The motion vector calculated between frames i and $i+1$ is denoted $mv_i = (dx_i, dy_i)$. The simplest approach is then simply to use these two coordinate values as they are in the feature vector. We convert the motion vector into three values, which are the sine, cosine and length of the vector.

CHAPTER 4

Implementation

The implementation is written in MATLAB and is available on the attached DVD along with the data and configurations we have used for the experiments.

4.1 Architecture Overview

An illustration of the overall architecture of the implementation is shown in Figure F4-1. The flow of the program working on a video follows the illustration from top to bottom. To run the program on a video a configuration file needs to be specified for the application that specifies which video files that should be used and the values of various algorithm parameters that are listed later in this chapter.

Before videos can be used by the main application they need to be converted into its constituent frame images in either .png or .tif file format. This is not done by the our implementation and needs to be done by a third party tool. Once the frames have been obtained, the application starts constructing feature vectors for each pixel from the frames specified by the active configuration file. This utilizes a number of distinct feature conversion functions depending on which features are used.

After feature vectors for the video have been obtained the application enters the main part of the algorithm which begins with an initialisation step that provides randomized initial values for the models parameters. The next step is a derivation of the Hard EM-algorithm. The mapping step consists of matching each individual pixel to Gaussian cluster means, which is done using an external library for the α-expansion algorithm [6, 15]. The maximization step involves fitting new mean functions to the newly assigned pixels in each cluster using our coordinate descent method.

4.2 Configuration Files

The configuration files specify the parameters that the application should use for a particular video. This includes some basic information such as the path of the video frames and which part of the video, specified as frame indices, that should be used and whether to skip frames to reduce the amount of features and speed up the run. The configuration files also specifies a number of parameters relevant to how the algorithm should run:

- *NumberOfLabelsPerLayer*
- *NumberOfLayers*

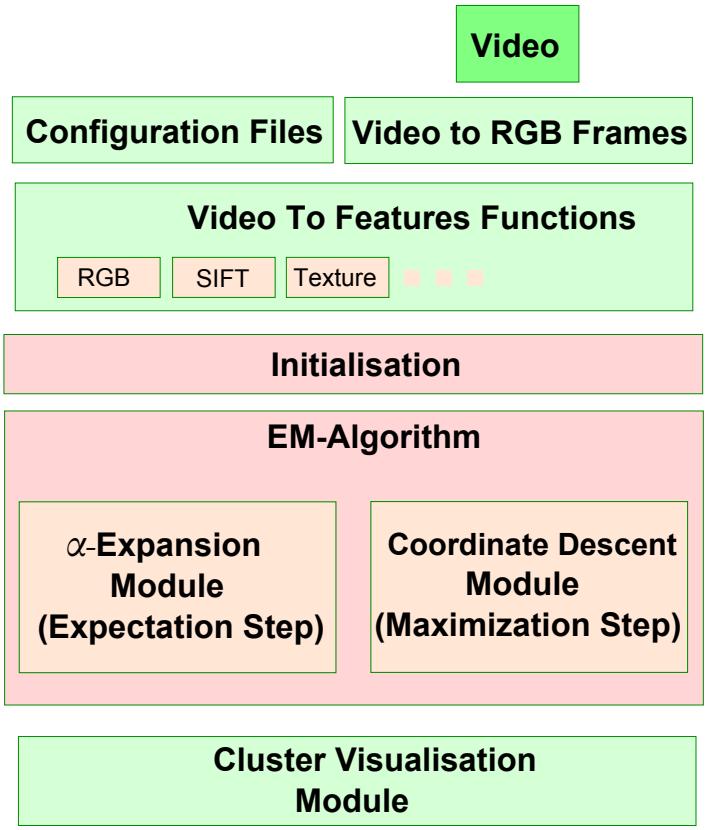


Figure F4-1: The architecture of the implementation.

- *NeighbourWeight* (β)
- *NumberOfRuns*

  The number of runs specifies the number of times the application should restart the initialisation and EM part of the application. This is useful due to the nature of the random initialisation. The EM-algorithm that will not always converge on the globally optimal solution, and as such multiple runs is recommended.

- *MaxEMIterations*

  While the EM-algorithm is guaranteed to converge in a finite number of steps under normal circumstances, it can still be useful to limit this number to prevent slowing down the application with an unnecessary number of small adjustments.

- *TerminationThreshold*

  The termination threshold determines a minimum change in energy for the application to continue doing EM iterations.

- *UsePCAonFeatures and pctVarianceCovered*

  Specifies whether to use the features directly or perform dimensionality reduction, as explained in 5.6, and what percentage of the original variance the reduced set of components should cover.

- *MeanType and parameters* Tells the algorithm which mean function family should use and the hyper parameters of the function depending

on the function type. For polynomial that is the polynomial degree. For the continuous piecewise linear it is the number of knots and for the discontinuous piecewise linear it is the ϵ-value.

Additionally a few less significant parameters for the output format are specified.

4.3 Video to Features

The application utilizes a number of conversion functions that take the video frames as input and constructs a feature vector per pixel per frame. These feature vectors will automatically be concatenated when multiple feature types are specified in the configuration files. The implementation currently has three conversion functions:

- *RBG-features*

  RGB features consist of Red, Blue and Green values.

- *SIFT-features*

  SIFT features are calculated using the SIFT flow algorithm [14] calculated using an external library. The motion is represented as Sine Cosine and length of the pixel motions from one frame to the next.

- *SFTA Texture features*

  The program also supports texture features that calculated using the SFTA algorithm [16] using an external library for the core computations. The texture of the local area around each pixel is then converted into 9 values which are used in the feature vector. This feature type has however not been used in our experiments, as feature types is not the main contribution of this project.

All feature values are normalized to lie between 0 and 255, this value is not important but rather chosen to match the natural normalisation of RGB-values in images.

4.4 Initialisation

The initialisation module is responsible for random initialisation of the cluster means.

4.5 EM-Algorithm

The core part of the algorithm is based on the hard EM algorithm which in generally works in two alternating steps: mapping and maximisation. The interfacing between the two steps consists of passing the cluster means and the label assignments. In the interfacing, the means are represented as a vector of values for each feature value per frame, to avoid constantly re-evaluating the mean function.

4.5.1 MAP Step

The map step consists of taking the current cluster means and reassigning all pixels to best fitting cluster using a squared distance function. Cluster assignments are computed using the α-expansion algorithm [6]. The α-expansion algorithm require that the distance from each pixel to each cluster mean is pre calculated in what is called the data cost, the α-expansion also supports parameters for a neighbourhood structure as well as a weighting factor. The data cost is implemented as specified in Section 2.1.4.

4.5.2 Maximisation Step

The maximisation step computes the new cluster means provided the cluster assignments in the expectation step. This is computed using the coordinate-descent method presented in Section 3.5.3.

4.6 Cluster Visualisation

When a run terminates due to TerminationThreshold or MaxEMIterations being exceeded a visualisation function is called to display the computed segmentations. The segmentations are shown as grey scale images with the clusters in the different layers displayed above each other. Each output file is named with the resulting energy of the solution.

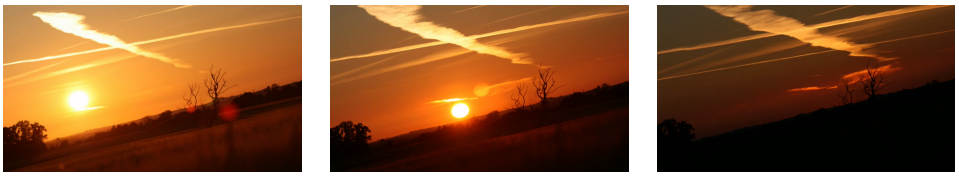
CHAPTER 5

Experiments

In the following experiments we will use the shorthand names cpl, dpl and poly for the mean function types continuous piecewise linear, discontinuous piecewise linear and polynomial respectively. When we present the results of a specific configuration of the algorithm we will write the mean function along with the mean function's hyperparameters, e.g. poly $d = 1$ for when the algorithm used polynomial mean functions of degree 1. In this section when comparing experimental results with our previous method [2], we refer to it as the old model.

5.1 Experiments on Sunset Video

The first experiment will be on a time-lapse video of a sunset containing 800 frames. Figure F5-1 shows three frames from the start, middle and end of the video. The video is chosen because the RGB trajectories of the pixels in video are very smooth, and it should therefore be an example where it is easy to find convolutions of mean functions that match the pixel trajectories.



(a) Video at 0:01 (b) Video at 0:15 (c) Video at 0:29

Figure F5-1: Three frames from the start, middle and end of the sunset video.

For the experiments we are only including every eighth frame of the video to speed up the algorithm. This does not impact the shape of the trajectories, because the RGB values for each pixel changes very slowly during the video. For this video we are creating a $(2, 3)$-segmentation, which is two segmentations with 3 clusters each. The algorithm is run 200 times for each type of mean function with a different settings of the functions hyperparameters. We set $\beta = 500$ for all runs of the algorithm.

Table T5-1 shows energy statistics for the result of each run of the algorithm. The energy values are the final energy value of the segmentations after the algorithm have converged. The minimum values is especially interesting because they tell us which type of mean functions resulted in the best fit to the pixel data. The other values helps us to get a picture of how often the algorithm gives good results when using each mean function. This is of

	min	mean	max	median	std. dev
cpl $k = 5$	6697	7675	10086	7582	605
poly $d = 4$	6734	7721	9694	7659	561
old model	6736	7707	9509	7637	541
cpl $k = 4$	6741	7743	9971	7740	581
poly $d = 5$	6754	7727	9734	7664	494
cpl $k = 6$	6754	7776	10145	7813	578
poly $d = 3$	6770	7789	9685	7746	551
cpl $k = 3$	6851	7826	9678	7757	563
dpl $\epsilon = 3$	6945	8125	10108	7987	719
poly $d = 2$	7036	7927	9880	7845	553
dpl $\epsilon = 5$	7058	8536	10916	8341	884
poly $d = 1$	7436	8432	10421	8429	475
dpl $\epsilon = 10$	7503	9254	12710	9007	954
dpl $\epsilon = 20$	8044	9921	13280	9695	925

Table T5-1: Statistics for energy of results for mean functions with varying parameter values. The mean functions are sorted by minimum energy values. Values are in millions.

course also tied to the choice of hyper parameters. We immediately notice that the dpl functions generally performs worst. Part of the explanation can be that, since the dpl fitting method is heuristic and does not fit an optimal function inside its error bound, then we will see some approximation error and a higher error bound will lead to a higher approximation error. This effect is especially significant with the sunset video because the temporal relationship between pixel values is so smooth that a function can easily be fitted to it. The cpl and polynomial functions perform on par with respect to minimum energy results. Also we see that higher values of k and d gives lower energy results. This is expected since as the complexity of the mean functions increase they can better fit the pixel trajectories. Cpl and polynomial functions also compete with the original method that fitted mean vectors in terms of energy. When looking at the mean, maximum, median and standard deviation values we see that polynomial functions seem to have a slight advantage over cpl functions.

The lowest energy segmentation using cpl $k = 5$ is shown in Figure F5-2. The right segmentation managed to separate the pixels into foreground, background and clouds. This segmentation can be explained by the fact that there is a general decline in RGB values for all pixels, but the starting point and rate of decline is different for these three parts of the video. The left segmentation appears to have largely separated the pixels based the the distance in the video to the setting sun which influenced the red colour. In Figure F5-3 the lowest energy segmentation using the old method is shown, and we can see that it is very similar to the lowest energy segmentation using cpl $k = 5$. This is not surprising since the pixel trajectories are very smooth, so the mean vectors that fit well will tend to look like they were generated by smooth mean functions.

The lowest energy result using cpl $k = 5$ was the lowest energy result for all the different configurations, so the convolution of the mean functions must fit the pixel data quite well. Each pixel has in this video been assigned to two clusters, which mean there are a total of 9 different combinations of cluster

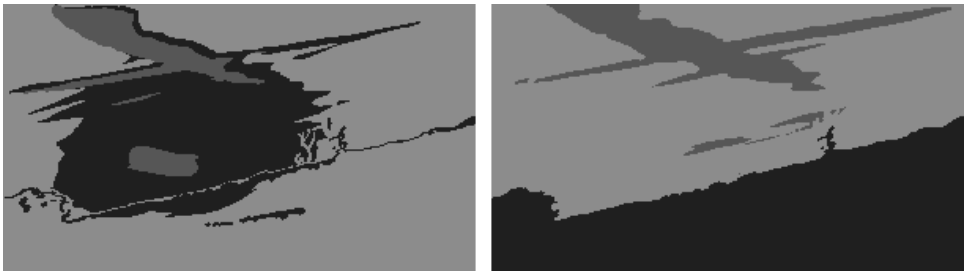


Figure F5-2: Lowest energy $(2, 3)$-segmentation by algorithm using cpl with $k = 5$.

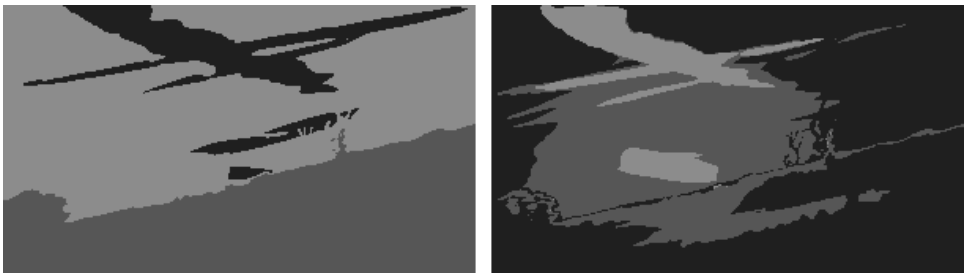
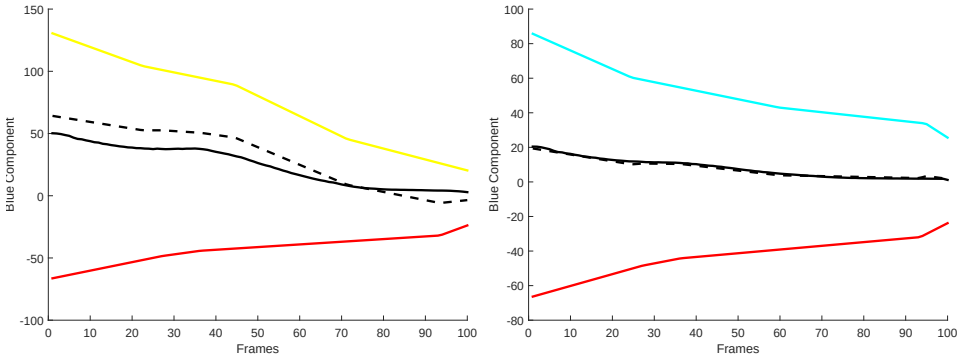


Figure F5-3: Lowest energy $(2, 3)$-segmentation by algorithm using the old model.

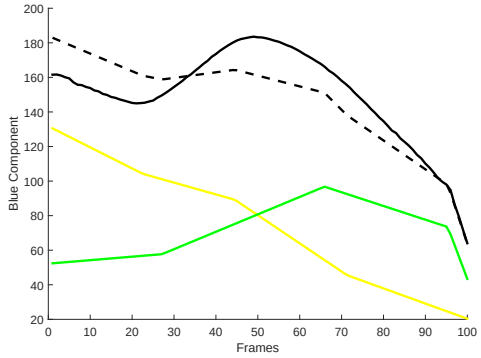
assignments (3 cluster and 2 layers give 3^2 combinations). In Figure F5-4 we have plotted the two mean functions for each cluster combination, their convolution, and the mean of the pixels for the blue RGB component. There are only 8 plots in the figure because there was a single cluster combination which had no pixels. The mean functions have been drawn with the same colour in all the plots, i.e. the red mean function in Figure F5-4a is the same as the red mean function in Figure F5-4b. The figure is helpful in showing how each mean function is a part of multiple pixel means.

As an example let us first focus on the combination in Figure F5-4d. This combination contains the majority of the pixels located in the path of the sun. We can see in the left segmentation in Figure F5-2 that the dark grey cluster have captured the path of the sun. It is also evident from the pixels' mean in Figure F5-4d that these are in fact the pixels in the path of the sun, since the pixels' mean contain an upward bumb in the first part of the video which corresponds to the time the sun passes over the pixels. The purple mean function helps explain this early upward bumb in the pixels' mean. The purple mean function is also in the cluster combination in Figure F5-4g where it also helps explaining an early upward bumb. The green mean function also has an upward bumb, but it appears later in the video. In Figure F5-4c and Figure F5-4e, we can see it helps explaining two upwards bums that appear in the pixels' mean.

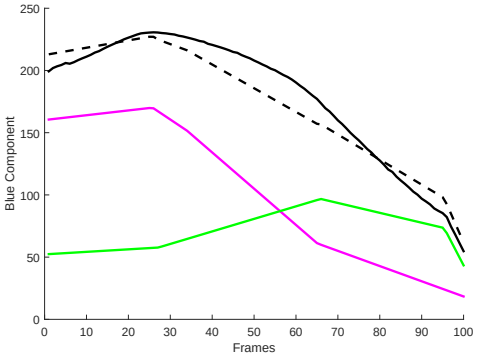


(a) Cluster combination 1

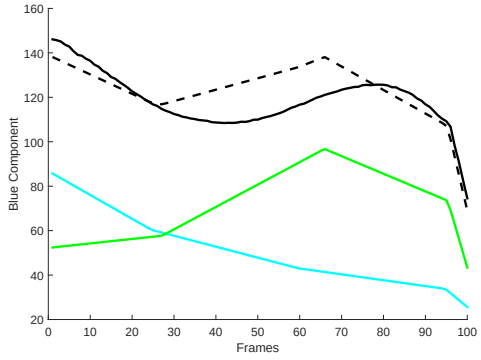
(b) Cluster combination 2



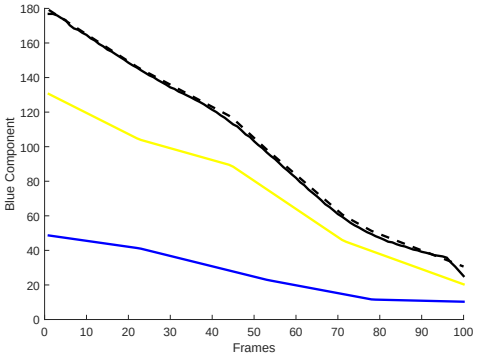
(c) Cluster combination 3



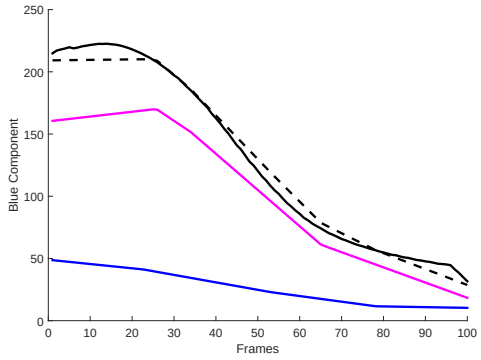
(d) Cluster combination 4



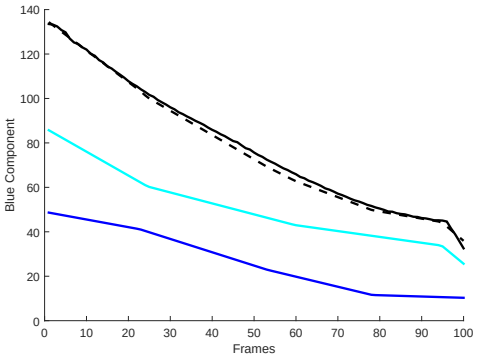
(e) Cluster combination 5



(f) Cluster combination 6



(g) Cluster combination 7



(h) Cluster combination 8

Figure F5-4: The 8 cluster combinations from the lowest energy result with cpl $k = 5$. Every figur shows the mean of the blue RGB component for pixels with that cluster combination (black line) and the two means for the two clusters (non-black lines) and their convolution (dashed black line).

A second statistic we produced in this experiment was the average num-

cpl $k = 3$	cpl $k = 4$	cpl $k = 5$	cpl $k = 6$
12.6	13.4	13.4	13.1

dpl $\epsilon = 3$	dpl $\epsilon = 5$	dpl $\epsilon = 10$	dpl $\epsilon = 20$
11.1	9.0	6.3	4.8

poly $d = 1$	poly $d = 2$	poly $d = 3$	poly $d = 4$	poly $d = 5$
12.3	12.6	13.2	12.8	13.4

Table T5-2: Average number of iterations before convergence of algorithm for different configurations of the algorithm.

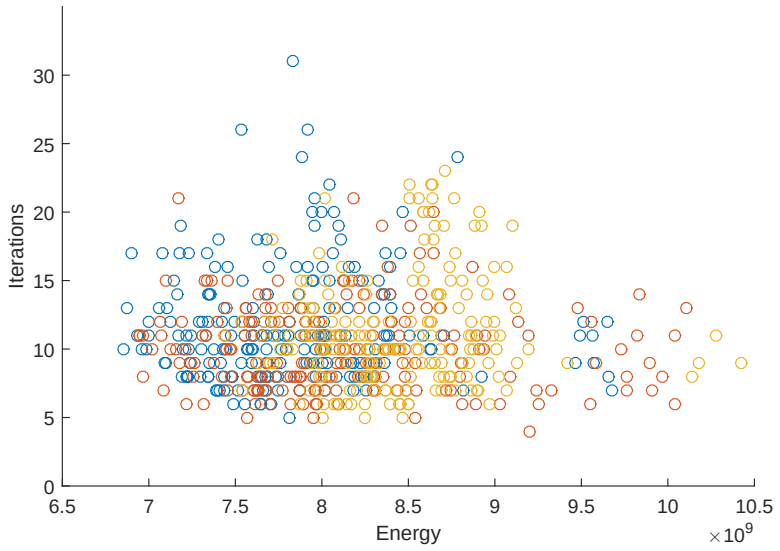
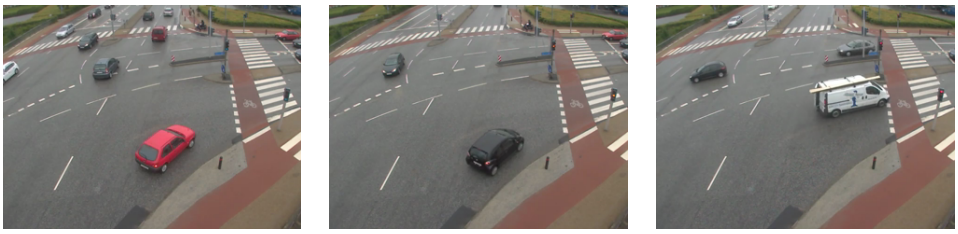


Figure F5-5: Scatter plot over energies of segmentations and the number of iterations before convergence in 200 runs of the algorithm for three different mean functions. Blue: cpl $k = 3$. Red: dpl $\epsilon = 3$. Yellow: Poly $d = 1$.

ber of iterations used by each configurations of the algorithm. The results can be seen in Table T5-2, the original model used 12.6 iterations on average. The cpl and poly mean functions both have similar converge speed regardless of the complexity of the functions (k and d values). On the other hand we see that when using dpl mean functions the converge speed improves the larger the error bound gets, but as we saw in the Table T5-1 a larger error bound was also tied to worse results. This could be an indication that there is a negative correlation between the energy of a segmentation and the number of iterations, so in Figure F5-5 we have drawn a scatter plot with the energy and number of iterations for each of the 200 runs of the algorithm for three different mean functions. We can see that there is no trend in the plot that supports this hypothesis. Therefore we believe that the dpl mean functions with a high error bound does not converge fast, because it finds high energy segmentations, but rather because the high error bound enables it to quickly find a satisfying solution. The high energy is simply another by-product of the approximation error that can be the result of a high energy bound.

5.2 Experiments on Intersection Video

Our second experiment will be with the intersection video that was given as an example in the introduction. Three frames form the video can be seen in Figure F5-6. In our previous project we managed to get the result shown in Figure F1-2 by engineering a set of features, which we called a *SIFT histogram*. Basically we divided all motion vectors into 8 bins which represented 8 directions, e.g. left, down-right etc. and incremented a pixel's bin counter based on the direction of the motion in a frame.



(a) Video at 0:01 (b) Video at 0:05 (c) Video at 0:10

Figure F5-6: Three frames from the start, middle and end of the intersection video.

The purpose of this experiment is to see if the mean functions allow us to replicate this result without any specific feature engineering. The three SIFT flow features are computed per frame, in the same way that we have three RGB features per frame.

For these experiments the algorithm have also been run 200 times for each configuration and with $\beta = 75.000$. It is a high value for β compared to the other videos, but we also find it important in this video that the clusters are connected, since we are hoping to find the traffic lanes in the results. As the first experiment we will see what happens, when we fit mean vectors using the old method instead of mean functions. In Figure F5-7 we see the lowest energy segmentation, and notice that the clusters does not correspond to the traffic lanes. Since this experiment is different in that we are hoping for a specific result, we should look at the 20 best results. This is because the results in the top results will not necessarily look alike, so one of them might have managed to find the traffic lanes, but was not shown to be the case in this experiment.

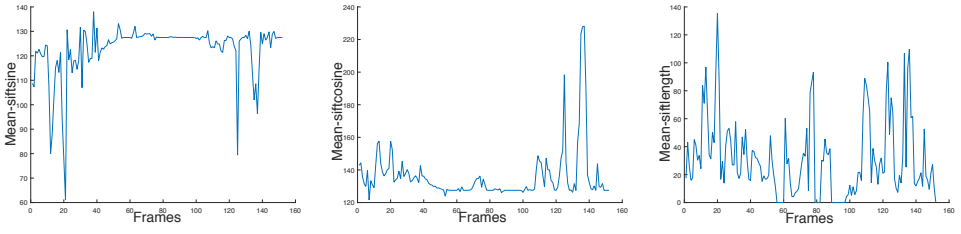
Figure F5-7: Lowest energy result from intersection video when fitting the old model.

For the experiments with mean functions we use the same configurations as presented in the sunset experiment section. In Appendix 1.3 we have shown one result each algorithm configuration, the one that best captured the traffic lanes for each algorithm configuration. In Figure F5-8 we have

presented what we deem to be the best segmentation w.r.t. capturing the traffic lanes. It was the second lowest energy segmentation for poly $d = 2$. Many of the results using mean functions have two segmentations that are identical. Typically they only found the up-going lane and the right-going lane, and then one of these segmentations were duplicated. The result in Figure F5-8 stood out because it found three different lanes.

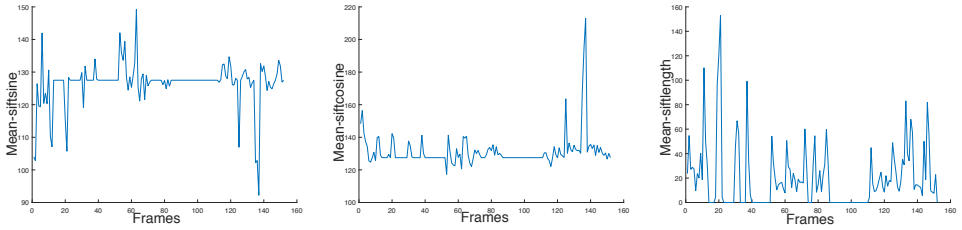
Figure F5-8: Lowest energy result from intersection video when using poly $d = 2$. Left segmentation: Right-going lane. Middle segmentation: Down-going lane. Right segmentation: Up-going lane.

If we look more closely at the mean of the feature values in these results we can attempt to understand the structure of the segmentations. One interesting observation can be made of the results from our previous work in Figure F1-2 is that in the first cluster we have the upwards going lane and the curved bottom-to-right going lane merged into a single cluster. If we look at our new models result, we have the upwards going lane by itself in the rightmost cluster of Figure F5-8 . If we look at the mean feature values of the pixels that belong to the upwards going cluster and compare them to the mean feature values of the pixels in the curved lane, we can try to understand why these two are not clustered together using the new model. Figure F5-9 shows us the mean feature values of the rightmost cluster. Note that all feature values are normalized to lie between 0 and 255, and that neutral sine and cosine values lie at 127.5. Comparing this with the values of the curved bottom-to-right going lane in Figure F5-10 we see significant differences in the fluctuations of the curves and it becomes clear that the upwards going lane should not be in the same segment as the curved bottom-to-right going lane based on these features.



(a) Mean sine values (b) Mean cosine values (c) Mean length values

Figure F5-9: Mean feature values of rightmost segmentation in Figure F5-8.

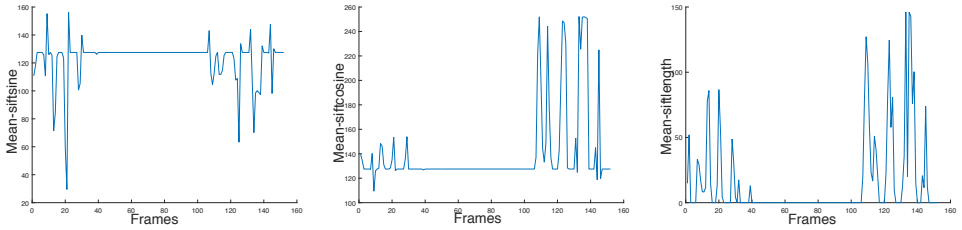


(a) Mean sine values (b) Mean cosine values (c) Mean length values

Figure F5-10: Mean feature values of curved bottom-to-right going lane.

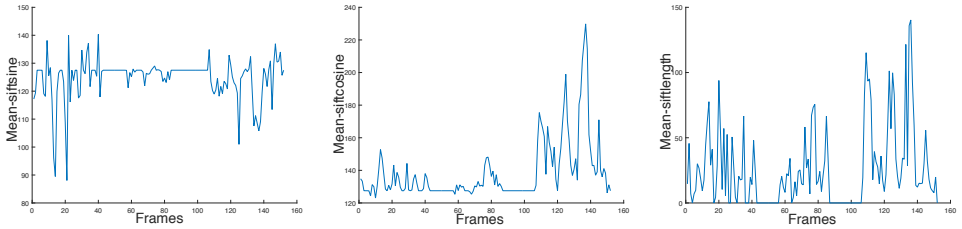
.

We can similarly look at the other results from the old model. Looking at the middle segmentation in Figure F5-7 which appears to capture a small part of the right going lane, it would be interesting to see why the rest of the lane is not segmented. To figure out why, we can look at the difference of the feature values of the pixels inside the stumped cluster seen in Figure F5-11 and the feature values belonging to the full right going lane seen in Figure F5-12. Looking at these trends we see that there is a rather significant difference between the mean feature values in the stumped cluster and the entire lane. Especially during frames 40 to 100 we have very little change in the stump, but if we look at the entire lane we do have activity there. This difference can be explained by the other lanes crossing the right going lane and causing activity in parts of the video where we otherwise do not have cars moving right.



(a) Mean sine values (b) Mean cosine values (c) Mean length values

Figure F5-11: Mean feature values of middle segmentation in Figure F5-7.



(a) Mean sine values (b) Mean cosine values (c) Mean length values

Figure F5-12: Mean feature values of right going lane.

The results from the intersection video shows that the new model may not always provide more intuitive results than the old method. It might be the case that the various parameters can be changed to find the same segmentations, but it is not something one would discover without specifically looking for these segmentations. In which case there is not much point in

finding the segmentation, other than for experimental purposes. It is probable that we need other features to find that kind of segmentation, and that it is not possibly with the generic SIFT flow features which we have used here.

5.3 Experiments on Scene Change Video

An often seen task when doing video segmentation in other contexts, is a temporal segmentation of the different scenes in a video. While our method is not meant to do this kind of segmentation, we can make a different segmentation for each scene due to the multiple layers.

To test the viability of this we have used our method on a simple video showing a person playing drums, but from different angles.

The video is 12 seconds long and changes back and forth between two angles 4 times. Three frames from the video can be seen in Figure F5-13.

(a) Video at 0:01 (b) Video at 0:05 (c) Video at 0:10

Figure F5-13: Three frames from the start, middle and end of the drumming video.

For this video we used time standard RGB-features and created a 3-2 segmentation. In theory only two layers should be required to make a segmentation based on the two angles, but it is interesting to see what additional segmentation is found when looking at the third layer. The algorithm is run 100 times for each configuration and a number of different configurations are used.

The statistics for each method can be seen in T5-3. The best result from *dpl* $\epsilon = 3$ can bee seen in Figure F5-14, and a collection with the best results from each method can be found in Appendix A.

Figure F5-14: Lowest energy result using dpl with $\epsilon = 3$.

It is clear that the piecewise linear fits provides the best results energy-wise. The polynomial mean function is slightly better than using the old model that treats the video as an image stack, and importantly polynomial mean function provides much more stable results. The resulting images from the polynomial mean results shows the two different camera angles in two of the segmentations, but they are not very clear. The two different angles of the video can not be seen in the results from the old model.

	min	mean	max	median	std. dev
dpl $\epsilon = 3$	12703	13807	18757	13649	999
dpl $\epsilon = 5$	12753	13829	19253	13660	994
dpl $\epsilon = 10$	12820	14574	22132	14050	1736
dpl $\epsilon = 20$	13576	18266	21427	19510	2360
cpl $k = 9$	13988	15051	19155	14961	894
cpl $k = 7$	14164	15052	17027	14896	758
cpl $k = 4$	15760	17725	21122	17642	1326
poly $d = 1$	19014	19764	21625	19593	586
poly $d = 5$	19691	20255	21726	20201	348
Image stack	20334	32886	210612	24681	24576

Table T5-3: Statistics for energy of results for mean functions with varying parameter values. The mean functions are sorted by minimum energy values. Values are in millions.

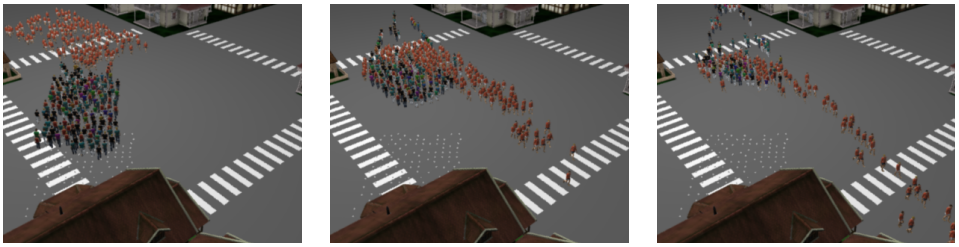
Looking at the results from the piecewise fit we can see that the discontinuous method provides the results with the lowest energy, but with a slightly higher standard deviation. If we look at the resulting images it is clear that the different angles are seen in different segmentation in both results, and actually it is not possible to see a difference in the segmentations between the best results from the discontinuous mean function and those from the continuous mean function. If we look at results with the discontinuous mean function that has the same energy as the best results from the continuous case the latter are clearly more useful segmentations even though the energy is the same. Thus it is hard to conclude a clear better method, the energy is better for the discontinuous function, but with a less stable result compared to the results with the continuous function.

5.4 Experiments with Simulated Crowd Movement

One interesting domain where our method could be useful is segmenting motion flow in videos. An example of a type of video in which motion flow could be interesting, would be surveillance of crowded areas to for example analyse where adverts should be placed to be seen. In order to evaluate our method on a completely static and interesting video we have used a crowd simulation plugin for the 3d-animation program maya [17] to generate videos with crowds moving through a street. The video shows two groups of people moving in opposing directions, cluttering while trying to move through each other and then finally getting through.

	min	mean	max	median	std. dev
cpl $k = 2$	46057	48286	76847	47386	4664
cpl $k = 3$	46270	47630	50463	47333	936
cpl $k = 4$	45915	47920	77081	47081	3785
cpl $k = 5$	46465	48922	79651	48274	3785
dpl $\epsilon = 3$	45717	47452	50402	47202	1030
dpl $\epsilon = 5$	47491	49101	77637	48549	2569
dpl $\epsilon = 7$	49039	51096	55988	50162	2083
dpl $\epsilon = 10$	47397	50824	79642	49889	3658
dpl $\epsilon = 20$	45635	47450	76903	46790	3697
poly $d = 1$	47534	48284	51040	48018	876
poly $d = 2$	46814	48300	50990	47880	1128
poly $d = 4$	46274	48145	76965	47272	3829
poly $d = 5$	46085	47523	50430	47136	989

Table T5-4: Statistics for energy of results of crowd simulation video for mean functions with varying parameter values. The mean functions are sorted by minimum energy values. Values are in millions.



(a) Video at 0:01 (b) Video at 0:04 (c) Video at 0:15

Figure F5-15: Three frames from the start, middle and near the end of the simulated crowd video.

For this video we have used SIFT flow features and created 2-2 segmentations and the $\beta = 75.000$. The algorithm have been run 200 times for each configuration and a number of different configurations are used.

Similar to the previous experiments the statistics for each configuration can be seen in T5-4

Energy wise all the methods provide results that are very similar. Looking at the quality of the segmentations it seems that many of the methods provide near-identical segmentations in both layers which is the case both with the lowest energy solutions shown in Appendix A and most of the slightly higher energy solutions. It seems that in about one half of the cases we have the segmentation of the upwards moving crowd seen in Figure F5-16 and the other half the method finds the group moving downwards on the video seen in Figure F5-17. One exception to this is the discontinuous piecewise linear method with $\epsilon = 7$ where both directions are captured in very well defined in clear clusters seen in Figure F5-18. It is probably somewhat a coincidence that we find both directions in that single case, and it is just unlikely to arrive at that solution so even more runs might find it with the other configurations. It is however evident that the method only identifies both directions in rare cases.

Figure F5-16: Lowest energy result using cpl with =¸3.

Figure F5-17: Lowest energy result using cpl with =¸4.

5.5 Experimental Observations

In the intersection and crowd experiments we have experienced a high number of results where several layers of the results provide very similar or even identical clusters, which is in contrast with the goal of the method that is finding distinct but complimentary clusters. One common point for these two experiments is that they use SIFT flow features, and it is possible that the behaviour of the SIFT features could explain the phenomenon. One possible explanation could be that it is difficult to match the three SIFT flow feature values for distinct pixels which results in the algorithm only finding very few good solutions.

A possible solution to this could be to reintroduce the regularization term from [1] which penalizes similar clusters explicitly in the objective function but at that time did not make a big difference, this is however beyond the scope of this work. One could also look more into the SIFT features and try to represent the motion vectors differently.

5.6 Experiments with Dimensionality Reduction

Before we extended the model to be temporal and use a mean function, we discovered that we could not work with videos of long durations due to computer resources, so we looked into methods of reducing memory consumption and running time of the algorithm. For a video with pixel dimensions 300x200 we began to meet the memory ceiling on a 8GB laptop when the video length approached a thousand frames, which is a duration of less than a minute for a typical 24 frames per second video. There are possibly parts of the source code that could benefit from some memory optimisations, but the memory usage reduction would only be a small constant factor, and not enough to actually enable us to run the algorithm on longer videos. There is of course not a corresponding hard limit on the running time, but since we

Figure F5-18: Lowest energy result using dpl with $\epsilon = 7$.

are restarting the algorithm multiple times, any reduction in running time would also be beneficial. For the video described above a single iteration of the algorithm takes 10 seconds on a quad-core 2.3 GHz laptop with the old method, and with an average of 12 iterations per run, 200 runs takes relatively long time to complete.

5.6.1 Dimensionality reduction in the old model

Due to the above reasons we looked into using principal component analysis (PCA) [18]. It is used in statistics and relating fields in many different contexts. We can use it to create a linear transformation that maps the feature vector of every pixel to a another space, where the dimensions can be sorted according to importance. Several good tutorials that explains how to use PCA in practice can be found, e.g. [19] and [20].

PCA allows us to discard the least important dimensions in the new space, and only work on a space of lower dimensionality (from here on called a reduced space), with minimal loss of information. We first attempted this using the original model, where we are using vectors of means instead of fitting mean functions.

PCA is performed on a dataset by computing a covariance matrix of the feature vectors where each pixel is considered a data point. You then find the eigenvalues and eigenvectors of the covariance matrix. Each eigenvector, also called a principal component, represents a dimension in the new transformed space. It can then be used transform the feature values from the original space to the values for this dimensions in the new spaces, simply by multiplying the transposed original data by the transposed eigenvectors.

$$reducedData = covEigenVectors^T \cdot originalData^T$$

$covEigenVectors$ is the eigenvectors of the covariance matrix, and $originalData$ is the pixel features where the columns are the frames and the rows the feature value.

The eigenvalues shows how much of the variance of the original data is kept in that component. You can then calculate the percentage of total variance each component explains, so that you can choose a number of components such that you preserve e.g. 99% of the variance of the original data in the transformation. When the pixels have been transformed to the reduced space the entire algorithm will only work on this space, meaning that the mean vectors will of course also be vectors in the reduced space. There are no changes to the algorithm other than the pixel data is transformed right after it is loaded.

The most important question that arise is whether the best segmentations can be found in both the original space and in the reduced space. In Figure F5-19 we see that the segmentation is almost identical to the segmentation in Figure F5-3. So it is possible to get the same segmentations when we work on the reduced space which in this case covered 99% of the variance and resulted in 6 components. Although we don't show this to be the case for all videos it does show that PCA is very beneficial in at least some cases. It is possible to form the argument that the more components you use in your transformation the more likely you are to find the same segmentations as you would in the original space because you preserve more of the information.

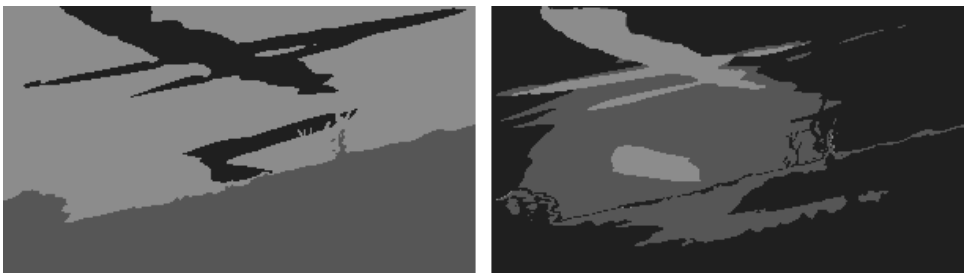


Figure F5-19: Lowest energy $(2, 3)$-segmentation using the original method in reduced space.

5.6.2 Dimensionality Reduction in the New Model

In the new model we have to consider how the algorithm uses mean functions when the data has been transformed to a reduced space. At first it might not seem logical that a mean function can be represented in a reduced space, but it is possible because we in our implementation only work on discrete evaluations of the mean function, and therefore can work work on a vector containing the mean value for every frame. Every time we fit a mean function we immediately use the newly found parameters to generate a vector of means from that function. This is done to ensure that the implementation supports both the original model and our new model. This means that we in practice can have a mean function in a the reduced space if we transform the mean values the function provides into the reduced space.

Another issue is that we have no control over which components PCA will find. This means we have no guarantee as to which degree the transformation preserves the temporal properties of the pixel trajectories when we are not using all of the components. PCA only guarantees that we preserve a certain amount of the variance. We hope however that this issue is mitigated in practice when using enough components.

Ideally we would like to fit our mean functions using coordinate descent in the original space and let the rest of the algorithm work in the reduced space. We have also tried to fit the mean functions directly to the data in the reduced space but that did not produce useful results. This is not too surprising since the assumption when we use the mean function, is that there is a correlation between the values in each frame that can be expressed by the function. PCA however guarantees to give us uncorrelated components, which means that it becomes hard to fit a function that describes the values well.

It is possible to fit the mean functions in the original space, because a useful feature of PCA is that you can transform your data into the original space. PCA can be used as a sort of lossy compression of the data, and the more components we use the more information we keep. If we use every component the transformations is lossless, since all we need to go back is the inverse of the eigenvectors, and the inverse of our eigenvectors is in this case simply equal to the transpose of the eigenvectors. This is only the case because all the elements of the eigenvectors matrix is a unit vector of our data. This means we can simply multiply the reduced data by the eigenvectors to go back to our original space [19].

$$originalData = covEigenVectors \cdot reducedData$$

Using all the components is however not useful for our application since that would not reduce the dimensionality. It is also problematic to use too few, because this of course results in an large loss of information.

Since we can transform the feature vectors of the pixels back and forth between the two spaces we can also transform the mean vectors back and forth, and this will allow us to fit our mean functions in the original space. We do not lose additional information by doing the transformation multiple times, because the eigenvalues we use for the transformation are derived from the original data and not changed.

It is quite common to use a lossy transformation such as PCA and the closely related method *Karhunen and Loéve transform(KLT)* for dimensionality reduction. It is for example used for plain compression or applications doing facial recognition [19, 21]. Despite PCAs popularity, some results however show that variants of KLT is better in the cases tested[22], which was unknown to us at the time of the experiments and could be interesting to further explore.

5.6.2.1 Experiments with PCA

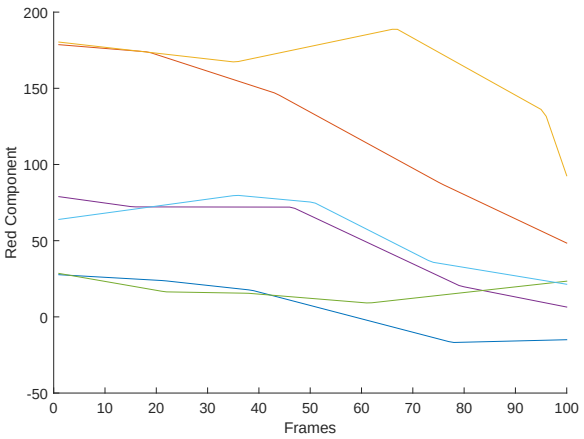
To experiment with PCA in the new model we have to make some changes to the coordinate descent step. Recall that in Section 3.5.3 we explained that the algorithm iterates through all means in a random order. When we use PCA we now go through the steps seen in Algorithm 3.

Algorithm 3: Changes to coordinate descent

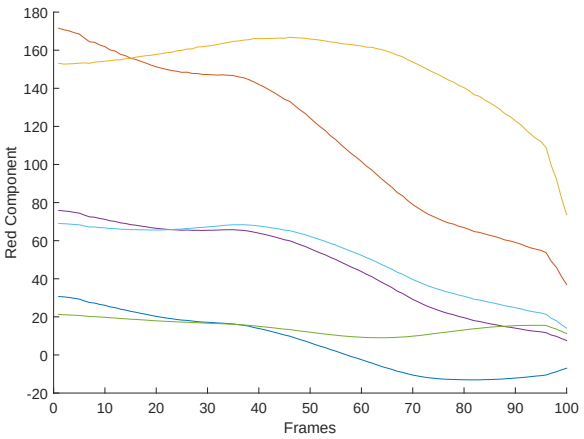
Find $\hat{\mu}_{reduced}$ in the reduced space;
Transform $\hat{\mu}_{reduced}$ to $\hat{\mu}_{original}$;
Fit a mean function $\boldsymbol{M}$ to $\hat{\mu}_{original}$ to obtain $\tilde{\theta}$;
Generate a new mean vector μ from M using $\tilde{\theta}$;
Transform μ back to the reduced space;

We reran the sunset experiment with cpl $k = 5$ using respectively 6, 22 and 74 components in PCA, which explained 99.0%, 99.9% and 99.99% of the variance. In Figure F5-20 we have shown the six mean functions (two layers with three clusters each) for the red RGB component before and after we reconstructed them. It is quite clear that when we use more components we preserve more of the functions shape in this video. In Figure F5-20f where 74 components were used we can easily recognise that it is indeed a continuous piecewise linear function. In Figure F5-21 the lowest energy segmentation for PCA with 74 components is shown.

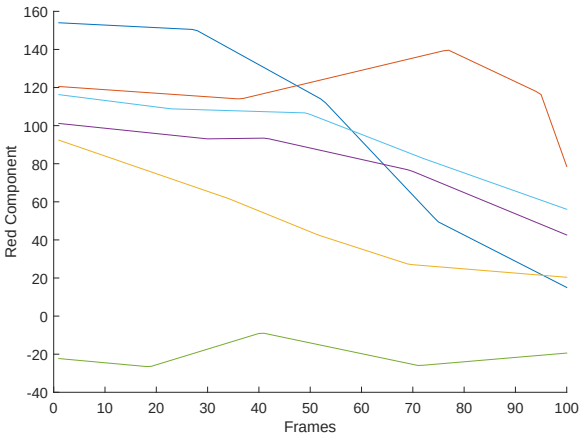
The sunset video was an example where the old model and the new model gave the same segmentations, so we also applied PCA on the intersection video, since it was only the mean functions that were able to give segmentations that represented the traffic lanes. We used poly $d = 2$ since it was the same configuration that gave the result in Figure F5-8. We set PCA to preserve 99.0 % of the variance, which lead to the use of 54 components. In Figure F5-22 we show the result within the top 10 that best captured the traffic lanes. It has two duplicate segmentations, but that was also common for the results without PCA. Another interesting thing about the segmentations in Figure F5-22 is that the right-going lane is fully connected from right to left. There is no "hole" in the lane cluster as we saw in Figure F5-8. So not only can we somewhat get the same segmentations with the PCA transformation, but the transformation resulted in it being able to create a better segmentation of the right-going lane.



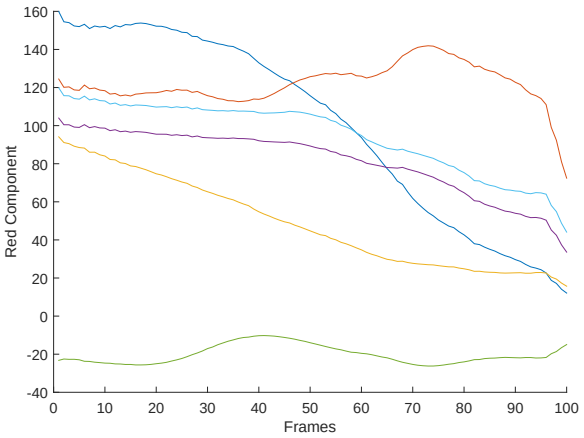
(a) Mean functions before transformation using 6 components.



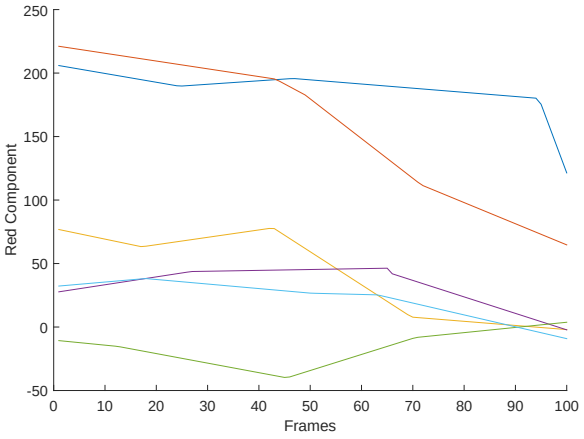
(b) Reconstruction of mean functions using 6 components.



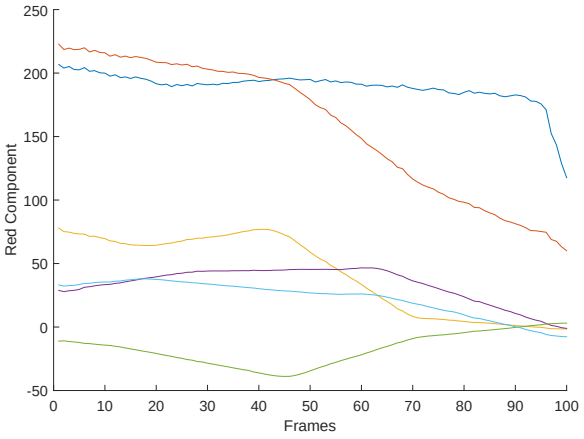
(c) Mean functions before transformation using 22 components.



(d) Reconstruction of mean functions using 22 components.



(e) Mean functions before transformation using 74 components.



(f) Reconstruction of mean functions using 74 components.

Figure F5-20: Reconstruction of means using different number of components.

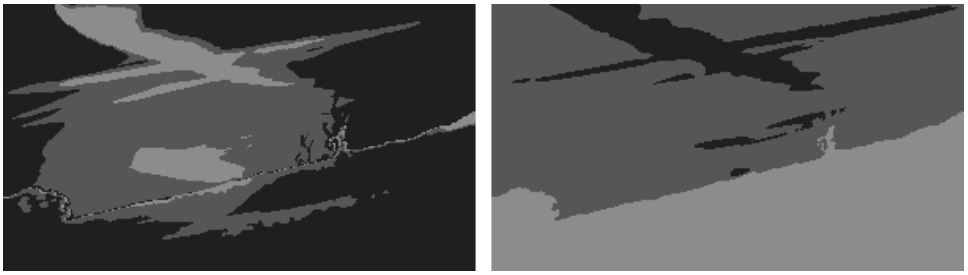


Figure F5-21: Lowest energy $(2, 3)$-segmentation by algorithm using cpl $k = 5$ as mean function and PCA with 74 components.

Figure F5-22: Best segmentation by algorithm using poly $d = 2$ as mean function and PCA with 54 components.

CHAPTER 6

Conclusion

In this project we work with and extend a novel method for image and video segmentation that provides multiple segmentations in the form of images that represents static structures in a video. We base our work on our previous project, where future work included extending the model with temporal information. Our work achieves this by modelling the temporal relationships between the pixel values throughout the frames in the input video. We do this by changing the underlying probabilistic model from being based on multivariate Gaussian distributions to Gaussian processes. The changes in the model requires a new method to fit it to the data, in which we need to fit a mean function for the Gaussian process. We present how this can be done, and show that fitting the new model still maximised the original likelihood function. The choice of mean function is not immediately clear, and we propose three different families of functions each of which can be preferable in different cases, a simple polynomial function and two different methods of fitting a piecewise linear function. In all cases the piecewise functions performs better. The whole method is implemented in MATLAB based on the implementation of the original image segmentation method. We show that our new method provides advantages to the original method, and is able to find meaningful segmentations without doing feature engineering and is able to find segmentations in videos with scene changes that the original method cannot. We did however show that in some cases our segmentation results were less intuitive than the previous method. Furthermore we experiment with dimensionality reduction on the data using Principal Component Analysis, and our results show that we improve the running time significantly, while still gaining segmentations that are as good as when using the whole feature space.

6.1 Future Work

There is still work that can be done to make the implementation more usage friendly. Our work has mainly been with the model and realising the concept without focusing much on the usability. To be used in a context where the results can be used for real world applications it would be preferable to have a more user friendly way of interacting with the program. We added configuration files that allow the user to easily rerun the algorithm with some chosen parameters, but it would also be good to be able to adjust the parameters and provide default values in a user interface. This could furthermore be complemented by an interactive visualisation tool, where for example the results can be browsed based on energy and parameters used.

Additionally a user interface could automate the explorative tasks of

finding useful parameters. The application could be set to attempt a range of possible parameter settings and then either have some strategy for changing the parameters based the energy of the results or alternatively present the results to the user who evaluate which parameter settings give the most interesting.

Furthermore it currently takes a while to get the results for a video due to the running time and the large amount of runs to get the best result. We have not focused a lot on the performance of the algorithm, but there are likely significant changes that could make it faster. It could especially use some attention if the algorithm needs to be run videos with a longer duration.

Another important direction for the method would be to test it in additional use contexts where the segmentations could provide useful insight. Such uses could include exploratory data analysis on videos or other temporal data that can be represented as images. We have done some brief experiments on spatio temporal data and think that it could be useful in this environment. Another use case could involve using the resulting images as identification for videos to compare videos for example for a search application, since they provide a representation of the different underlying structures in the video. That would require the segmentation result to be more consistent for example using a more refined, but not random initialization method.

Our work with dimensionality reduction is the result of an experiment, and if it is to be used, more investigation on the effect of this should be performed. Other dimensionality reduction techniques could also be tried that for example are tailored to this application.

APPENDIX A

Results

1.1 Lowest Energy Results from Sunset Video

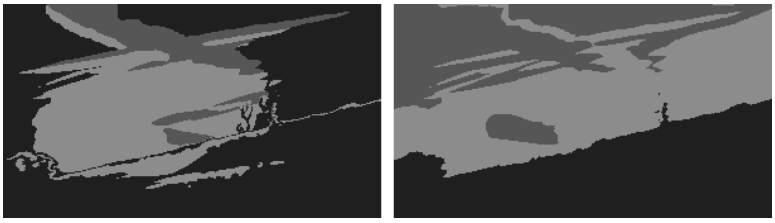


Figure F1-1: Lowest energy result using cpl with $k = 3$.

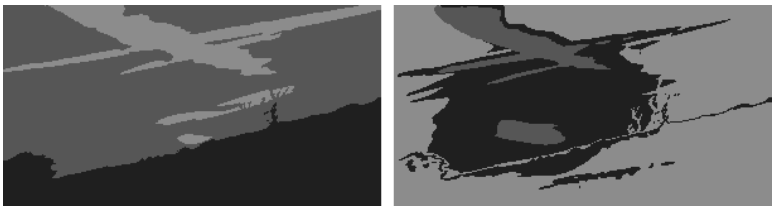


Figure F1-2: Lowest energy result using cpl with $k = 4$.

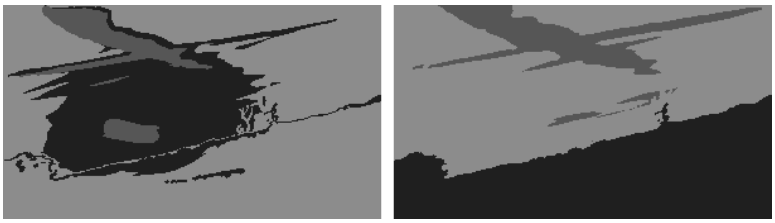


Figure F1-3: Lowest energy result using cpl with $k = 5$.

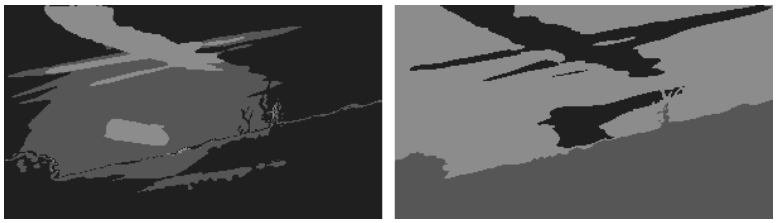


Figure F1-4: Lowest energy result using cpl with $k = 6$.

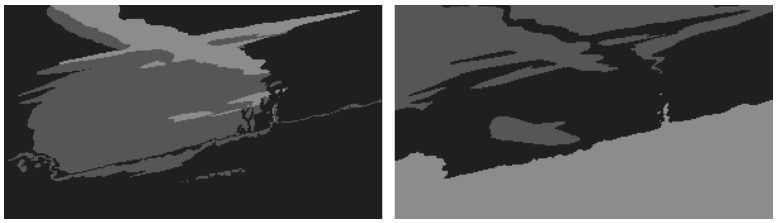


Figure F1-5: Lowest energy result using poly with $d = 1$.

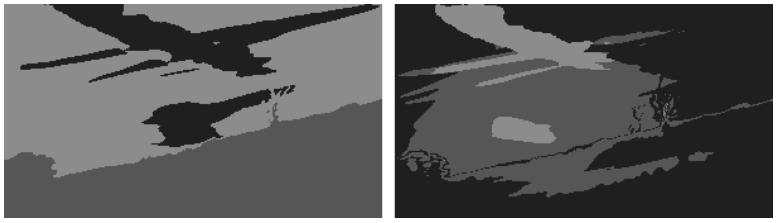


Figure F1-6: Lowest energy result using poly with $d = 2$.

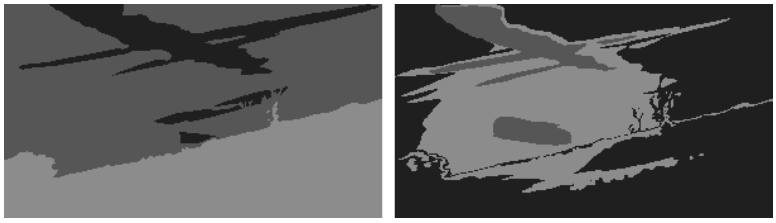


Figure F1-7: Lowest energy result using poly with $d = 3$.

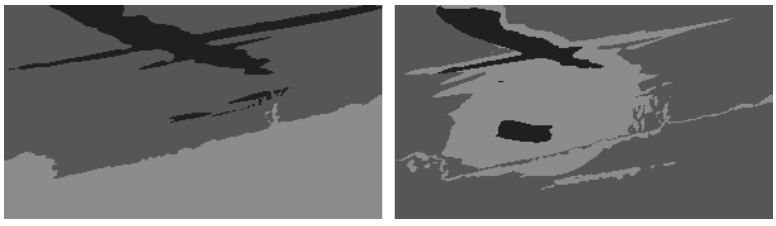


Figure F1-8: Lowest energy result using poly with $d = 4$.

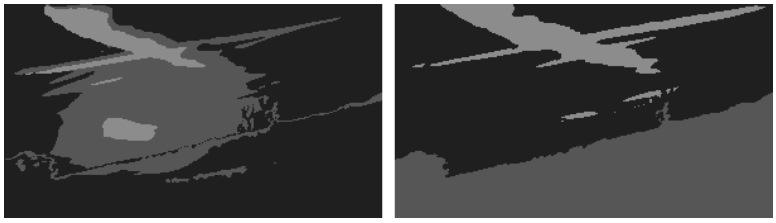


Figure F1-9: Lowest energy result using poly with $d = 5$.

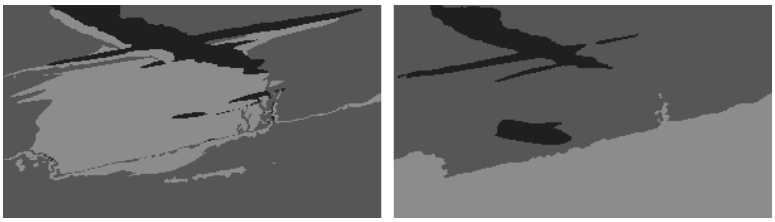


Figure F1-10: Lowest energy result using dpl with $\epsilon = 3$.

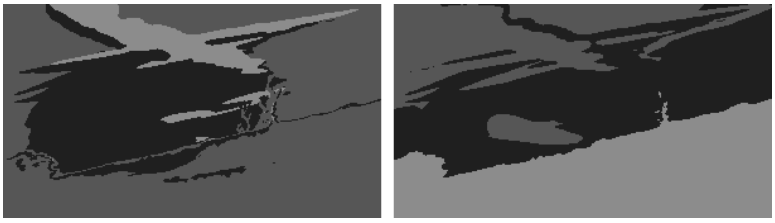


Figure F1-11: Lowest energy result using dpl with $\epsilon = 5$.

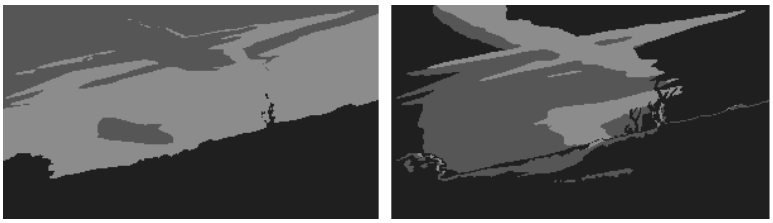


Figure F1-12: Lowest energy result using dpl with $\epsilon = 10$.

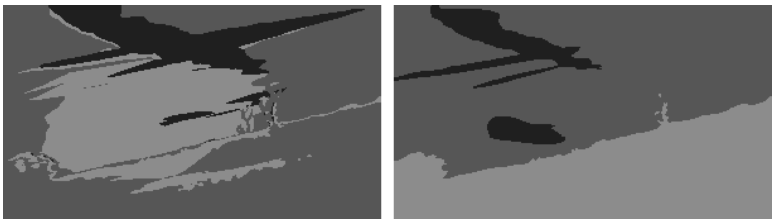


Figure F1-13: Lowest energy result using dpl with $\epsilon = 20$.

1.2 Lowest Energy Results from Drumming Video

Figure F1-14: Lowest energy result using the old model.

Figure F1-15: Lowest energy result using poly with $d = 1$.

Figure F1-16: Lowest energy result using poly with $d = 5$.

Figure F1-17: Lowest energy result using cpl with $k = 4$.

Figure F1-18: Lowest energy result using cpl with $k = 7$.

Figure F1-19: Lowest energy result using cpl with $k = 9$.

Figure F1-20: Lowest energy result using dpl with $\epsilon = 3$.

Figure F1-21: Lowest energy result using dpl with $\epsilon = 5$.

Figure F1-22: Lowest energy result using dpl with $\epsilon = 10$.

Figure F1-23: Lowest energy result using dpl with $\epsilon = 20$.

1.3 Best Segmentations of the Intersection Video

The results presented here are the segmentations which best captured the traffic lanes, and were not necessarily the ones with the lowest energy.

Figure F1-24: Best result using cpl with $k = 3$.

Figure F1-25: Best result using cpl with $k = 4$.

Figure F1-26: Best result using cpl with $k = 5$.

Figure F1-27: Best result using cpl with $k = 6$.

Figure F1-28: Best result using poly with $d = 1$.

Figure F1-29: Best result using poly with $d = 2$.

Figure F1-30: Best result using poly with $d = 3$.

Figure F1-31: Best result using poly with $d = 4$.

Figure F1-32: Lowest energy result using poly with $d = 5$.

Figure F1-33: Lowest energy result using dpl with $\epsilon = 3$.

Figure F1-34: Lowest energy result using dpl with $\epsilon = 5$.

Figure F1-35: Lowest energy result using dpl with $\epsilon = 10$.

Figure F1-36: Lowest energy result using dpl with $\epsilon = 20$.

1.4 Lowest Energy Results from Simulated Crowd Video

Figure F1-37: Lowest energy result using poly with $d = 1$.

Figure F1-38: Lowest energy result using poly with $d = 2$.

Figure F1-39: Lowest energy result using poly with $d = 4$.

Figure F1-40: Lowest energy result using poly with $d = 5$.

Figure F1-41: Lowest energy result using cpl with $k = 3$.

Figure F1-42: Lowest energy result using cpl with $k = 4$.

Figure F1-43: Lowest energy result using cpl with $k = 5$.

Figure F1-44: Lowest energy result using dpl with $\epsilon = 1$.

Figure F1-45: Lowest energy result using dpl with $\epsilon = 2$.

Figure F1-46: Lowest energy result using dpl with $\epsilon = 3$.

Figure F1-47: Lowest energy result using dpl with $\epsilon = 5$.

Figure F1-48: Lowest energy result using dpl with $\epsilon = 7$.

Bibliography

[1] Jonathan Smets and Manfred Jaeger. Multiple segmentation of image stacks. In *ICPRAM 2014 - Proceedings of the 3rd International Conference on Pattern Recognition Applications and Methods, ESEO, Angers, Loire Valley, France, 6-8 March, 2014*, pages 5–13, 2014.

[2] Jacob J. Jensen, Christoffer S. Nielsen, and Niels N. Samuelsen. Multiple static segmentation of videos, 2015. Pre-thesis Report.

[3] John R Kender and Boon-Lock Yeo. Video scene segmentation via continuous video coherence. In *Computer Vision and Pattern Recognition, 1998. Proceedings. 1998 IEEE Computer Society Conference on*, pages 367–373. IEEE, 1998.

[4] Shu-Ching Chen, Mei-Ling Shyu, Chengcui Zhang, and Rangasami L Kashyap. Video scene change detection method using unsupervised segmentation and object tracking. In *ICME*, 2001.

[5] Matthias Grundmann, Vivek Kwatra, Mei Han, and Irfan Essa. Efficient hierarchical graph-based video segmentation. In *Computer Vision and Pattern Recognition (CVPR), 2010 IEEE Conference on*, pages 2141–2148. IEEE, 2010.

[6] Yuri Boykov, Olga Veksler, and Ramin Zabih. Fast approximate energy minimization via graph cuts. *IEEE Trans. Pattern Anal. Mach. Intell.*, 23(11):1222–1239, November 2001. ISSN 0162-8828.

[7] Prateek Jain, Raghu Meka, and Inderjit S Dhillon. Simultaneous unsupervised learning of disparate clusterings. *Statistical Analysis and Data Mining: The ASA Data Science Journal*, 1(3):195–210, 2008.

[8] S Roberts, M Osborne, M Ebden, S Reece, N Gibson, and S Aigrain. Gaussian processes for time-series modelling. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 371(1984):20110550, 2013.

[9] Carl Edward Rasmussen. Gaussian processes for machine learning. 2006.

[10] A Eisinberg and G Fedele. On the inversion of the vandermonde matrix. *Applied mathematics and computation*, 174(2):1384–1397, 2006.

[11] Ingo Lütkebohle. SLM - Shape Language Modeling. `http://www.mathworks.com/matlabcentral/fileexchange/24443-slm-shape-language-modeling`, 2009. [Online; accessed 11-may-2015].

[12] Ivan Tomek. Two algorithms for piecewise-linear continuous approximation of functions of one variable. *IEEE Transactions on Computers*, C-23:445–448, 1974.

[13] David G Lowe. Distinctive image features from scale-invariant keypoints. *International journal of computer vision*, 60(2):91–110, 2004.

[14] Ce Liu, Jenny Yuen, Antonio Torralba, Josef Sivic, and William T Freeman. Sift flow: Dense correspondence across different scenes. In *Computer Vision–ECCV 2008*, pages 28–42. Springer, 2008.

[15] Andrew Delong, Anton Osokin, HossamN. Isack, and Yuri Boykov. Fast approximate energy minimization with label costs. *International Journal of Computer Vision*, 96(1):1–27, 2012. ISSN 0920-5691.

[16] Alceu Ferraz Costa, Gabriel Humpire-Mamani, and Agma Juci Machado Traina. An efficient algorithm for fractal analysis of textures. In *Graphics, Patterns and Images (SIBGRAPI), 2012 25th SIBGRAPI Conference on*, pages 39–46. IEEE, 2012.

[17] Golaem crowd. `http://golaem.com`. [Online; accessed 20-may-2015].

[18] Svante Wold, Kim Esbensen, and Paul Geladi. Principal component analysis. *Chemometrics and intelligent laboratory systems*, 2(1):37–52, 1987.

[19] Lindsay I Smith. A tutorial on principal components analysis. *Cornell University, USA*, 51:52, 2002.

[20] Jonathon Shlens. A tutorial on principal component analysis. *arXiv preprint arXiv:1404.1100*, 2014.

[21] A Levey and Michael Lindenbaum. Sequential karhunen-loeve basis extraction and its application to images. *Image Processing, IEEE Transactions on*, 9(8):1371–1374, 2000.

[22] Anil Cheriyadat and L Mann Bruce. Why principal component analysis is not an appropriate feature extraction method for hyperspectral data. In *Geoscience and Remote Sensing Symposium, 2003. IGARSS'03. Proceedings. 2003 IEEE International*, volume 6, pages 3420–3422. IEEE, 2003.