VIEW-BASED 3D OBJECTS RECOGNITION WITH EXPECTATION PROPAGATION LEARNING

Adrien BERTRAND

A Thesis

in

the Concordia Institute

for Information Systems Engineering

(CIISE)

Presented in Partial Fulfillment of the Requirements

For the Degree of Master of Applied Science

in Information Systems Security at

Concordia University

Montreal, Québec, Canada

March 2016

CONCORDIA UNIVERSITY
School of Graduate Studies

This is to certify that the thesis prepared

By: _____

Entitled: _____

and submitted in partial fulfillment of the requirements for the degree of

_____

complies with the regulations of the University and meets the accepted standards with respect to originality and quality.

Signed by the final examining committee:

Dr. J. Y. Yu _____ Chair

Dr. J. Bentahar _____ Examiner

Dr. H. Rivaz _____ Examiner

Dr. N. Bouguila _____ Supervisor

Approved by _____
Chair of Department or Graduate Program Director

_____
Dean of Faculty

Date _____

ABSTRACT

View-based 3D Objects Recognition with Expectation Propagation Learning

In this thesis, we present an improvement on the Expectation Propagation learning framework, specifically various enhancements on both speed and accuracy. We use this enhanced EP learning with the Inverted Dirichlet mixture model as well as the Dirichlet mixture model, to implement an algorithm to recognize 3D objects. Those objects are in our case from a view-based 3D models database that we have assembled. Following specific rules determined by analyzing the results of our tests, we've been able to get good recognition rates. Experimental results are presented with different object classes by comparing recognition rates and confidence level, according to different tuning parameters we're able to refine towards specific classes for better specialized accuracy.

ACKNOWLEDGMENTS

First, I would like to thank my supervisor Dr. Nizar Bouguila, for the many hours spent with me at Concordia and on Skype, answering my questions and guiding me all along the way.

Moreover, I want to express my gratitude to Dr. Taoufik Bdiri for giving me some pointers and a nice trick regarding an initialization algorithm.

I would also like to thank my friends and family, of course, for their support and continuous encouragement, and especially my father, Paul Bertrand, for his precious math help in particular.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACRONYMS

BRIEF	Binary Robust Independent Elementary Features
CSV	Comma-Separated Values *(for a spreadsheet app…)*
DMM	Dirichlet Mixture Model
Div	Divergence *(used with KL-Div)*
EM	Expectation Maximization
EP	Expectation Propagation
FAST	Features from Accelerated Segment Test
GMM	Gaussian Mixture Model
InvDMM	Inverted Dirichlet Mixture Model
KL	Kullback-Leibler
MM	Mixture Model
ORB	Oriented FAST and Rotated BRIEF
PDF	Probability Density Function
SIFT	Scale-Invariant Feature Transform
SURF	Speeded Up Robust Features

CHAPTER 1: INTRODUCTION

1.1 Background and related works

For quite some time, creating systems being able to detect and recognize (classify) objects, has been a very popular subject of research, as it goes well along with many fields such as data retrieval and analysis, signal processing, artificial intelligence applications, etc. It is even more the case nowadays thanks to a great increase in computing power, working with techniques such as neural networks, deep learning, and generally statistical-based approaches. There are several types of probabilistic classifiers often used, for example Naïve Bayes ones, which have been studied a lot for dozens of years for applications such as text/information retrieval especially since they can be fast [***1***], but also logistic regression, support vector machines (SVM)... Such classifiers are capable to predict in which probability distribution, over a set of classes, a given sample input should belong, in addition to a certainty level ("confidence" as we'll call it later). This differs from other methods that could instead predict only the most likely one. Another kind of method that sometimes can be related to a Bayes (if the variables are independent) is mixture models – in particular, have been a big focus for data clustering, and are considered efficient and are rather attractive in terms of ease and flexibility for these kinds of frameworks. A general finite mixture model has N random variables (for the observations), distributed on a mixture of K components of different parameters each having a weight and a parameter, and N corresponding "random latent variables". For instance, on Gaussian mixture models (GMM), often used for the applications mentioned earlier [***2***], [***3***], the Gaussian distributions have a mean and variance for each component. A Gaussian Mixture Model is generally characterized by a covariance matrix, and the variables are independent if this matrix is diagonal (Naïve Bayes assumption). Once a model is chosen, the next step is the "learning" of parameters from the initial data. There are several learning frameworks available, although a very common one is Expectation-Maximization (EM) [***4***]. Although it is very sensible to the initialization, it is an easily implementable way and provides accurate results. With X the data from a distribution, the goal is to maximize the likelihood $p(X|\theta)$, given the model with parameter θ. First, a hidden variable is introduced allowing for a likelihood maximization simplification. The algorithm is iterative, with the E step: where we estimate the hidden variable with the data and current parameters value; the M step consists in updating the parameters to maximize the distribution of the hidden variable and the data. The initialization is crucial, and generally used methods are K-Means, C-Means, hierarchical K-Means, Gaussian

splitting… Other learning methods include Markov-Chain Monte Carlo (MCMC) [***5***] and derivatives [***6***], moment matching, spectral method etc.

We will now focus our interest on a practical application: 3D object clustering and recognition. There exist several methods that have been researched over time – for instance, Multi-View Probabilistic model [*7*], 3D Geometric model [***8***], Discriminative mixture of templates [***9***] ... and more approaches such as the ones detailed in [***10***]. We will here focus on view-based ones. View-based 3D objects retrieval methods such as Bag of (Visual) Features [***11***], Light Field Descriptor [***12***], etc. have been researched extensively lately, and applications range from general recognition [***4***] to more precise specialized classifications [***13***], [***14***]. Two main advantages of such methods, as explained in [***15***], is that there is no need to have information about the original 3D model, allowing for real-life applications, and that 2D image processing (for working on each view) has been studied and perfected over many years already. We have seen that the GMM was largely adopted as mixture model, however, in the recent years, there has been an upsurge of research done towards more accurate models for real-life applications and learning frameworks that may be more adapted for them – these now tend to use finite Dirichlet and Inverted Dirichlet mixtures [***16***], [***17***]. Recent studies have indeed shown that they often outperform the traditional Gaussian by being more appropriate in terms of feature modeling and data clustering [***18***], [***19***].

We have also seen that the Expectation Maximization is the most often used method of learning for such models. However, recently the Expectation Propagation (EP) learning framework has been shown to be more efficient and adapted for accurate results [***20***]. We will, in this thesis, focus on the Inverted Dirichlet (and Dirichlet) based mixture models applied on Expectation Propagation learning in order to get the "best" of what has been researched lately. As far as we know, InvDMM with EP has not been put together in both a theoretical point-of-view nor algorithmically with an implementation. Thus, the implementation of this is one of our goals here, as well as some improvements on the EP learning framework itself.

1.2 Contributions

- ✓ **Improvement of the Expectation Propagation learning framework** previously proposed for the Dirichlet mixture model
- ✓ Development of an **Expectation Propagation learning framework for the Inverted Dirichlet Mixture Model**
- ✓ Integration of these two models within an **efficient view-based 3D object recognition framework**

1.3 Thesis overview

The main objective of this thesis, presented in this document, is to introduce an improvement on the EP learning framework, applied on the Dirichlet and Inverted Dirichlet mixture models.

The focus has been set more on a practical and efficient implementation of the EP learning with those models, than on the mathematical theory aspect.

This document is organized as follow:

- **Chapter 1** introduces the background and related work about view-based 3D objects recognition with mixture models
- **Chapter 2** presents the theory behind the EP learning framework. We then present Dirichlet and Inverted Dirichlet mixture models and EP learning based on those along with the initialization process. The whole algorithm that will be used in the implementation is summarized at the end.
- **Chapter 3** presents the implementation of what chapter 2 describes, and analyzes the experimental process and results we obtained, ending with a discussion about possible applications.
- We then give a conclusion and present potential future work ideas for improving research on the presented topics.
- At the end, appendices ("Chapter 4") provide additional useful information like formulas proofs, review of some math and algorithm points, some timing measurements on our software, and some advices for anyone interested in reimplementing the proposed process.

CHAPTER 2:
EXPECTATION PROPAGATION LEARNING OF DIRICHLET-BASED MIXTURE MODELS

This chapter will review the finite Dirichlet and Inverted Dirichlet mixture models that will be learned with the Expectation Propagation framework. The developed learning algorithms are also presented.

2.1 Finite Dirichlet and Inverted Dirichlet Mixture Models

Let's consider a positive D-dimensional vector, $\mathbf{X} = (X_1, \dots, X_D)$. First, assuming that $\mathbf{X}$ follows an Inverted Dirichlet distribution, with also positive D-dimensional parameter vector $\boldsymbol{\alpha}_j = (\alpha_{j1}, \dots, \alpha_{jD})$, then the joint PDF of $\mathbf{X}$, as per [***21***], is:

$$\mathrm{p}(\mathbf{X}|\boldsymbol{\alpha}_j) = \frac{1}{\mathrm{B}(\alpha_j)} \prod_{l=1}^{D} X_l^{\alpha_{jl}-1} \left(1 + \sum_{l=1}^{D} X_l\right)^{-|\alpha_j|} \quad (1)$$

Let's add another constraint. If $\sum_{l=1}^{D} X_l = 1$ then we can assume $\mathbf{X}$ follows a Dirichlet distribution, also with positive parameter vector $\boldsymbol{\alpha}_j$, then the joint PDF is:

$$\mathrm{p}(\mathbf{X}|\boldsymbol{\alpha}_j) = \frac{1}{\mathrm{B}(\alpha_j)} \prod_{l=1}^{D} X_l^{\alpha_{jl}-1} \quad (2)$$

with $|\alpha_j| = \sum_{l=1}^{D} \alpha_{jl}$, and for both, B is the Beta function: $\mathrm{B}(\alpha_j) = \frac{\prod_{l=1}^{D} \Gamma(\alpha_{jl})}{\Gamma(|\alpha_j|)}$.

We must note that each distribution has its own α vector parameters, but we name them both α for consistency with the general formulas that are applicable on both.

Now that we have the distributions definitions, we can explore their consideration in mixture-based frameworks.

Let's assume that we have a data set of N D-dimensional positive vectors, $\mathcal{X} = \{\boldsymbol{X_1}, \dots, \boldsymbol{X_N}\}$, generated from a mixture model with M components:

$$p(\boldsymbol{X}|\boldsymbol{\pi},\boldsymbol{\alpha}) = \sum_{j=1}^{M} \pi_j \mathrm{p}(\mathbf{X}|\boldsymbol{\alpha}_j) \tag{3}$$

$\boldsymbol{\alpha}$ being the set of distribution parameter vectors $\boldsymbol{\alpha}_j$ seen earlier, and $\boldsymbol{\pi}$ being the set of weights $\boldsymbol{\pi}_j$ (called mixing coefficients) defined such that $0 \leq \boldsymbol{\pi}_j \leq 1$ and $\sum_{j=1}^{M} \boldsymbol{\pi}_j = 1$.

The likelihood function of $\mathcal{X}$ is then:

$$p(\mathcal{X}|\boldsymbol{\pi},\boldsymbol{\alpha}) = \prod_{i=1}^{N} \left(\sum_{j=1}^{M} \pi_j \, \mathrm{p}(\mathbf{X_i}|\boldsymbol{\alpha}_j) \right) \tag{4}$$

2.2 Expectation Propagation learning

Used in Bayesian inference, EP is a deterministic approach, known for its high accuracy, based on the Assumed Density Filtering method, that yields optimal posterior distribution approximations through an iterative refinement process ([***20***], [***18***], [***22***]). When applied to mixture models, contrary to EM learning, the number of clusters does *not* need to be known beforehand as it is determined during the algorithm.

From now on, let's assume that we will be working on an observed data set of N D-dimensional positive vectors, $\mathcal{X} = \{\boldsymbol{X_1}, \ldots, \boldsymbol{X_N}\}$, that all share an unknown joint PDF $p(\boldsymbol{X}|\Theta)$, in which Θ represents the set of parameters $\{\boldsymbol{\pi}_j, \boldsymbol{\alpha}_j\}$ that have to be estimated – in this case with EP.

2.2.1 Theory

As per [***23***] and [***24***], the joint PDF mentioned above can be represented as: $p(\mathcal{X}|\Theta) = \prod_i f_i(\Theta)$, with $f_0(\Theta)$ corresponding to the prior and $f_i(\Theta) = p(\boldsymbol{X_i}|\Theta)$ the true i[th] factor.

EP approximates *('~' symbol)* the posterior $p(\Theta|\mathcal{X})$ factorized as q*:

$$q^*(\Theta) = \frac{\prod_i \widetilde{f}_i(\Theta)}{\int \prod_i \widetilde{f}_i(\Theta)\, d\Theta} \tag{5}$$

In the learning framework, all $\widetilde{f}_i(\Theta)$ factors have first to be initialized then optimized sequentially. First, for instance on factor $f_j(\Theta)$, it is removed from the approximation as:

$$q^{\setminus j}(\Theta) = \frac{q^*(\Theta)}{\widetilde{f}_j(\Theta)} \tag{6}$$

Then, by combining this with the true factor $f_j(\Theta)$, we get a new unnormalized distribution:

$$\hat{p}(\Theta) = \frac{f_j(\Theta)q^{\setminus j}(\Theta)}{\int f_j(\Theta)q^{\setminus j}(\Theta)d\Theta} \tag{7}$$

The optimal approximated posterior $q^*(\Theta)$ is obtained through the minimization of the KL-divergence: $KL(\hat{p}(\Theta) \,||\, q^*(\Theta))$, done by moment matching between the two distributions, setting the sufficient statistics of $q^*(\Theta)$ to those of $\hat{p}(\Theta)$. This yields an update for $\widetilde{f}_j(\Theta)$:

$$\widetilde{f}_j(\Theta) \propto \frac{q^*(\Theta)}{q^{\setminus j}(\Theta)} \tag{8}$$

These several steps are done on each factor until convergence.

2.2.2 Application to the Dirichlet and Inverted Dirichlet Mixture Models

As used in [***23***], because it's not feasible to get a formal conjugate prior for the mixture model, we choose a D-dimensional Gaussian distribution to approximate it, which is accurate enough with mean vector $\boldsymbol{\mu}_j$ and covariance matrix A_j:

$$p(\boldsymbol{\alpha}_j) = \mathcal{N}(\boldsymbol{\alpha}_j|\boldsymbol{\mu}_j, A_j) = \frac{|A_j|^{1/2}}{(2\pi)^{D/2}} e^{-\frac{1}{2}(\boldsymbol{\alpha}_j-\boldsymbol{\mu}_j)^T A_j(\boldsymbol{\alpha}_j-\boldsymbol{\mu}_j)} \tag{9}$$

We now focus on refining the hyperparameters $\boldsymbol{\mu}_j$ and A_j.

The first step of the algorithm is the initialization of the approximating factors $\widetilde{f}_i(\Theta)$. For this, contrary to what [***18***] proposes (one convenient set of values for all), we use the following improvement: for each vector, we calculate its mean and covariance, and assign those to the hyperparameters initial values.

Next, we have to initialize q*, the posterior approximation, product of factors as per eq. (5). Its hyperparameters are then:

$$\boldsymbol{\mu}_j^* = \frac{1}{N}\left(\sum_i A_{i,j}^{-1}\right)\left(\sum_i A_{i,j}\boldsymbol{\mu}_{i,j}\right) \tag{10}$$

$$A_j^* = \sum_i A_{i,j} \tag{11}$$

Note that in eq. (11), only the diagonal values are kept ("denoising" step), as testing showed it yielded better results. Then the factors are updated sequentially.

As per (6), $\widetilde{f}_i(\Theta)$ is removed from the posterior, and thus when computed analytically, we have:

$$\boldsymbol{\mu}_j^{\backslash i} = \left(A_{i,j}^{\backslash i}\right)^{-1}\left(A_j^*\boldsymbol{\mu}_j^* - A_j\boldsymbol{\mu}_j\right) \tag{12}$$

$$A_j^{\backslash i} = A_j^* - A_{i,j} \tag{13}$$

It is important to note that during implementation, eq. ***(13)*** *has to be done before eq.* ***(12)****, in order to rely on updated values.*

Now, for the updated posterior, we follow eq. (7). However, since $\int f_i(\Theta)q^{\backslash i}(\Theta)d\Theta$ is intractable, we will apply the technique suggested in [***23***] to use a Laplace approximation with a Gaussian distribution:

$$\mathcal{H}(\boldsymbol{\alpha}_j) = \frac{h(\boldsymbol{\alpha}_j)}{\int h(\boldsymbol{\alpha}_j)d\boldsymbol{\alpha}_j} \tag{14}$$

$$\text{with} \quad h(\boldsymbol{\alpha}_j) = p(\boldsymbol{X_i}|\boldsymbol{\alpha}_j)\,\mathcal{N}(\boldsymbol{\alpha}_j|\boldsymbol{\mu}_j^{\backslash i}, A_j^{\backslash i}) \tag{15}$$

Where p is the Dirichlet or Inverted Dirichlet – see eq. (2) and (1). Since the next steps' equations vary based on which distribution is used, they will now both be detailed separately (here as well, to each distribution its own α vector parameters, written the same for consistency).

We now have to take the logarithm of $h(\boldsymbol{\alpha}_j)$:

Dir

$$\ln h(\boldsymbol{\alpha}_j) = \ln\frac{\Gamma(|\boldsymbol{\alpha_j}|)}{\prod_{l=1}^{D}\Gamma(\alpha_{jl})} + \sum_{l=1}^{D}(\alpha_{jl}-1)\ln X_{il} - \frac{1}{2}\left(\boldsymbol{\alpha}_j - \boldsymbol{\mu}_j^{\backslash i}\right)^T A_j^{\backslash i}\left(\boldsymbol{\alpha}_j - \boldsymbol{\mu}_j^{\backslash i}\right) + cst. \tag{16 (a)}$$

Inv Dir

$$\ln h(\boldsymbol{\alpha}_j) = \ln\frac{\Gamma(|\boldsymbol{\alpha_j}|)}{\prod_{l=1}^{D}\Gamma(\alpha_{jl})} + \sum_{l=1}^{D}(\alpha_{jl}-1)\ln X_{il} - |\boldsymbol{\alpha_j}|\ln(1+|X_i|) - \frac{1}{2}\left(\boldsymbol{\alpha}_j - \boldsymbol{\mu}_j^{\backslash i}\right)^T A_j^{\backslash i}\left(\boldsymbol{\alpha}_j - \boldsymbol{\mu}_j^{\backslash i}\right) + cst. \tag{b}$$

As the Laplace method dictates, we now have to calculate the two first (partial) derivatives (used in the Taylor expansion) with respect to $\boldsymbol{\alpha}_j$:

Dir

$$\frac{\partial \ln h(\boldsymbol{\alpha}_j)}{\partial \boldsymbol{\alpha}_j} = \begin{bmatrix}\psi(|\boldsymbol{\alpha_j}|) - \psi(\alpha_{j1}) + \ln X_{i1}\\ \vdots \\ \psi(|\boldsymbol{\alpha_j}|) - \psi(\alpha_{jD}) + \ln X_{iD}\end{bmatrix} - A_j^{\backslash i}\left(\boldsymbol{\alpha}_j - \boldsymbol{\mu}_j^{\backslash i}\right) \tag{17 (a)}$$

Inv Dir

$$\frac{\partial \ln h(\boldsymbol{\alpha}_j)}{\partial \boldsymbol{\alpha}_j} = \begin{bmatrix} \psi(|\boldsymbol{\alpha}_j|) - \psi(\alpha_{j1}) + \ln X_{i1} - \ln(1 + |\boldsymbol{X}_i|) \\ \vdots \\ \psi(|\boldsymbol{\alpha}_j|) - \psi(\alpha_{jD}) + \ln X_{iD} - \ln(1 + |\boldsymbol{X}_i|) \end{bmatrix} - A_j^{\setminus i}(\boldsymbol{\alpha}_j - \boldsymbol{\mu}_j^{\setminus i}) \qquad \text{(b)}$$

Where ψ is the digamma function: $\psi(\mathrm{x}) = \frac{d}{dx}\ln(\Gamma(x))$.

And regarding the second derivative, both Dirichlet and Inverted Dirichlet have the same, since $-\ln(1 + |\boldsymbol{X}_i|)$ is a constant with respect to $\boldsymbol{\alpha}_j$:

$$\frac{\partial^2 \ln h(\boldsymbol{\alpha}_j)}{\partial \boldsymbol{\alpha}_j^2} = \begin{bmatrix} \psi'(|\boldsymbol{\alpha}_j|) - \psi'(\alpha_{j1}) & \cdots & \psi'(|\boldsymbol{\alpha}_j|) \\ \vdots & \ddots & \vdots \\ \psi'(|\boldsymbol{\alpha}_j|) & \cdots & \psi'(|\boldsymbol{\alpha}_j|) - \psi'(\alpha_{jD}) \end{bmatrix} - A_j^{\setminus i} \qquad (18)$$

We now have to find the mode $\boldsymbol{\alpha}_j^*$ of the distribution – this can be done numerically by setting the first derivative to zero (eq. (17)) and solving the equation.

Instead of using traditional methods such as a gradient descent algorithm or Newton-Raphson, often used in situations as this one (see [***24***], [***25***] for instance), we proceed by dichotomy, the most efficient/fast way to solve it. Indeed, we can note that each line to solve is a monotone function and decreasing as shown by the second derivative (eq. (18)) with the parameters and constraints of the mixture models and data that we have in our case. Figure 1 shows an example of a typical equation to be solved for α_{jk}^* by the zero-finding dichotomy algorithm.

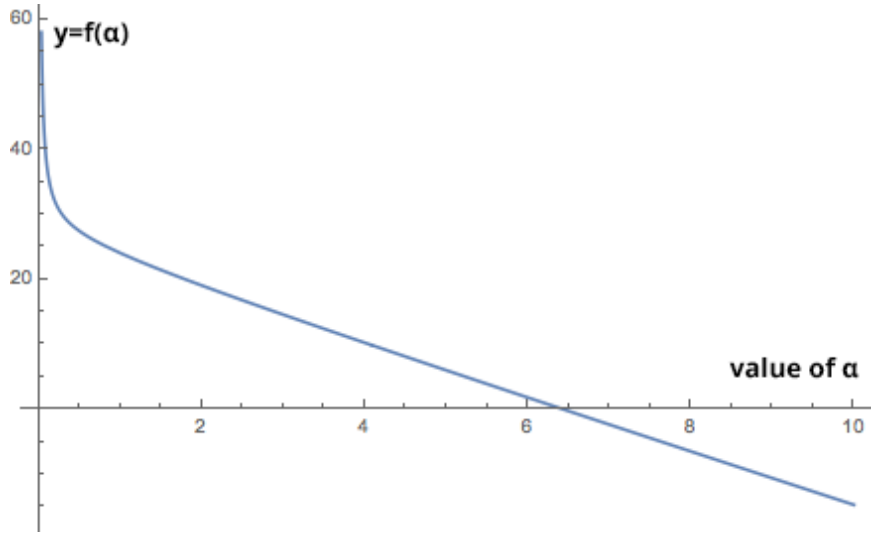


Figure 1*: Plot of a typical function to be solved for α_{jk}^* by the zero-finding dichotomy algorithm, here one line of eq. (17): f(α) = Ψ(α + 96.092) - Ψ(α) + ln(0.045) - 4*α + 26*

The dichotomy algorithm is given a minimum and a maximum value. We then take their average value, α, and use it as the value for the equation to solve. If f(α) is negative, it means the maximum was too high, and is thus it set to α. Similarly, if it is positive, the minimum value was too low, and is now set to α. This very simple process is repeated until it reaches an acceptable zero-threshold or a maximum iteration count.

Now that we have obtained the updated α^* values (the mode), we use that to approximate $h(\boldsymbol{\alpha}_j)$:

$$h(\boldsymbol{\alpha}_j) \approx h(\boldsymbol{\alpha}_j^*) e^{-\frac{1}{2}(\boldsymbol{\alpha}_j - \boldsymbol{\alpha}_j^*)\hat{A}_j(\boldsymbol{\alpha}_j - \boldsymbol{\alpha}_j^*)} \tag{19}$$

with

$$\hat{A}_j = -\frac{\partial^2 \ln h(\boldsymbol{\alpha}_j)}{\partial \boldsymbol{\alpha}_j^2}\bigg|_{\boldsymbol{\alpha}_j = \boldsymbol{\alpha}_j^*} \tag{20}$$

We now have to update q*, with moment matching, setting the sufficient statistics of $q^*(\Theta)$ to those of $\hat{p}(\Theta)$ (for the first and second order). Thus, we have, with $\boldsymbol{\alpha}_j = \boldsymbol{\alpha}_j^*$:

$$\boldsymbol{\mu}_j^* = \boldsymbol{\alpha}_j \tag{21}$$

$$A_j^* = \hat{A}_j \tag{22}$$

Finally, we can now update the hyperparameters of f_i:

$$\boldsymbol{\mu}_{i,j} = A_{i,j}^{-1}\left(A_j^* \boldsymbol{\mu}_j^* - A_j^{\setminus i} \boldsymbol{\mu}_j^{\setminus \boldsymbol{i}}\right) \tag{23}$$

$$A_{i,j} = A_j^* - A_j^{\setminus i} \tag{24}$$

Once again, eq. (24) has to be computed before eq. (23), so as to rely on updated values.

At this point, which is the end of the inner loop of the learning algorithm, we apply Bayes' decision rule [***19***] to reassign the vector, X, into the new best-fitted cluster since the hyperparameters have been updated.

$$j^* = argmax_j \, \pi_j \mathrm{p}(\mathbf{X}|\boldsymbol{\alpha}_j) \tag{25}$$

Because of vector reassignments, the weight of each cluster also has to be adjusted. This is done with the traditional formula:

$$\pi_j = \frac{number\ of\ vectors\ in\ cluster\ j}{total\ number\ of\ vectors} \tag{26}$$

When a cluster becomes only made of one vector, this vector is reassigned to the closest cluster, in terms of cluster center, with the L2 (euclidian) distance, and the old one is discarded (π_j=0). This closest-reassigning is also done after convergence on the clusters having a very low weight. As said previously, those learning steps have then to be repeated until convergence of the hyper-parameters of f_i (which corresponds to when $\Delta\, \boldsymbol{\alpha}_j^* < \boldsymbol{\epsilon}$), and for all factors.

2.3 Initialization

Initial values play an important role for optimal results. We will now see some "pre-processing" steps that take care of the input data that will be used in the EP learning detailed earlier.

2.3.1 K-Means algorithm

In order to have an initial clustering of the data before the EP learning, we have adopted the common K-Means algorithm, whose inner-workings are reviewed in the appendices (4.1).

We initially have to give the algorithm the "raw" data vectors as input, as well as the number of wanted clusters for this initialization step. We then obtain assignments of vectors to clusters. The weight of each cluster can be calculated as per eq. (26).

If it happens that a cluster is only assigned one vector (which it should not, with a reasonable number of clusters requested), the same reassignment process as in the end of the EP learning is applied: the vector is reassigned to the closest cluster, in terms of cluster center, with the L2 (euclidian) distance. The "old" cluster is then discarded (π_j=0).

2.3.2 Method of moments

The Dirichlet and Inverted Dirichlet parameters need proper initial values as they are going to be optimized in the EP learning. In order to get, on each cluster j, an initial α_j vector, we have chosen to use the method of moments, as described in [***26***] and [***17***]. Here are the equations:

$$\boldsymbol{\alpha}_{jl} = k\, x'_{1l} \quad \text{with } l=1,2,\ldots,D\text{-}1 \tag{27}$$

$$\boldsymbol{\alpha}_{jD} = k\left(1 - \frac{1}{N}\sum_{l=1}^{D-1} x'_{1l}\right) \tag{28}$$

Where

$$k = \frac{x'_{1,1} - x'_{2,1}}{x'_{2,1} - \left(x'_{1,1}\right)^2} \tag{29}$$

$$x'_{1,l} = \frac{1}{N}\sum_{i=1}^{N} x_{il} \tag{30}$$

$$x'_{2,l} = \frac{1}{N}\sum_{i=1}^{N} x_{il}^2 \tag{31}$$

2.4 Summarized complete learning algorithm

Input: Raw data vectors

1) Make a first approximate clustering of the input data vectors, as explained in 2.3.1
2) Initialize the approximate distribution parameters $\boldsymbol{\alpha}$, as explained in 2.3.2
3) Initialize the approximating factors $\tilde{f}_i(\Theta)$, i.e. the hyperparameters $\{\boldsymbol{\mu}_j, A_j\}$
4) Initialize the posterior approximation q*. The hyperparameters are given by eq. (10), (11)
5) Repeat… for all *i*
   a. Choose (sequentially or randomly) an approximating factor $\tilde{f}_i(\Theta)$ to refine
   b. Remove it from q*, as $q^{\backslash i}$. Its hyperparameters are given by eq. (12), (13)
   c. Compute the cluster's new $\boldsymbol{\alpha}$ by solving eq. (17)=0 with the dichotomy algorithm
   d. Get the new posterior by applying moment matching on the moments of q* and $\hat{p}$, with eq. (21), (22)
   e. Look for a more optimal cluster for the current vector using eq. (25) (Bayes' rule) If found, reassign the vector to it, and recompute the clusters' weight with eq. (26)

   Until convergence of the hyperparameters
6) Reassign, to the closest cluster, the vectors from clusters having a very low weight.

Output: optimally clustered data vectors and estimated mixtures parameters

CHAPTER 3: IMPLEMENTATION AND EXPERIMENTAL RESULTS

In this chapter, we present the implementation of our proposed EP learning framework presented in Chapter 2. We test it against synthetic data at first, then on real data from a 3D object database that we made. After analyzing the results that we obtained, we end with a discussion about applications.

3.1 Synthetic data

We generate synthetic data from a mixture of 3 components having a known set of weights and parameters for the distributions. Figure **2** displays both synthetic and estimated data.

	Weights (π)	**Parameters (α)**
Synthetic (original) data	0.4	{15, 65, 30}
	0.4	{65, 15, 30}
	0.2	{30, 34, 35}
Estimated data	0.389	{15.3, 65, 35.7}
	0.417	{65.5, 15.5, 35.6}
	0.194	{34.3, 36.1, 38.2}

***Table 1**: Original and estimated data from a 3-component synthetic mixture*

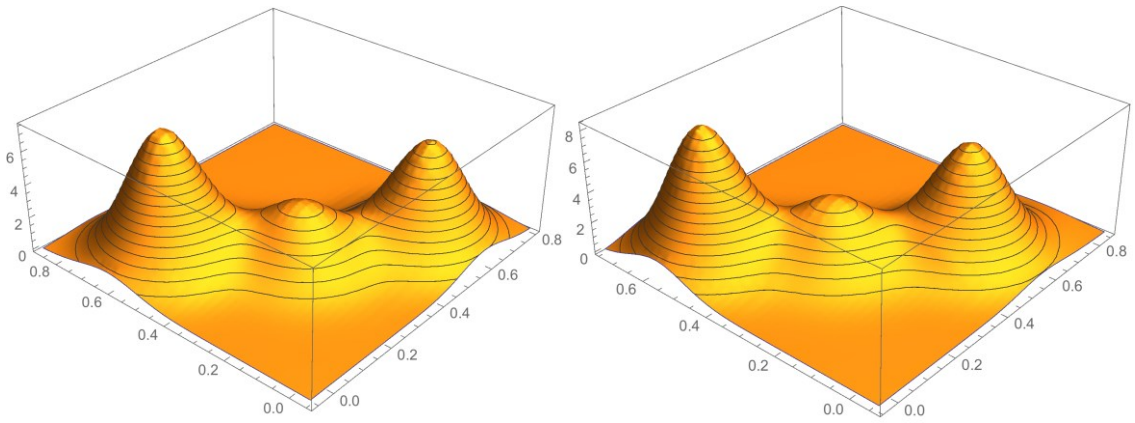


***Figure 2**: Original (left) vs. estimated (right) mixture 3D graph on sample synthetic data.*

We have noticed that if we give the actual amount of clusters to the initialization algorithms, the approximation given by the K-Means (for the first clustering) and moments method (for the alpha vectors), i.e. without any EP learning afterwards, already give exploitable results.

This means that in situations when we know in advance how some objects could be clustered (at least how many there could be), the whole learning is able to start with good initial values. This allows then to reduce the amount of iterations needed later on.

For instance, this has been tested on some objects (not the whole database), and while the results were not as good (lack of "contrast" between classes) as with the EP, unsurprisingly, the overall recognition still worked for objects matching the forced number of clusters (and less so for others with a different optimal clustering, on weights and number).

This, once again, shows the importance of the initialization values, and in general, of any kind of knowledge about the input that would be valuable to the learning framework.

3.2 Objects database

In order to test our new approach and compare it to existing ones, experiments have been made on a set of more than 100 3D objects images, manually created from 3D objects model files (*.3ds*, *.obj*, etc.) gathered from different sources such as the INRIA "Gamma" project's 3D meshes research database [***27***] (itself having put them together from different places), "TF3DM"'s free models [***28***], "3dmodelfree.com"'s models [***29***]. Initially, the work was tested with a focus on different types of cars, so those are more present than other kinds of objects.

3.2.1 Object images

For the creation of the view-based object images, the 3D models are loaded into a 3D viewer software, then the object model's position, angle, zoom, and lighting settings are adjusted for an optimal view, and using a built-in feature of the software[1], screenshots are taken every 5° of rotation on the Z axis, counter-clockwise. We thus get 72 images per object.

[1] Initially, a custom HTML+JavaScript-based software was created (based on an existing 3D-viewer code), and used in order to rotate the model and export screenshots, but a desktop software, *GLC_Player* [***29***], was later found, and had this feature built-in, giving faster export performance.

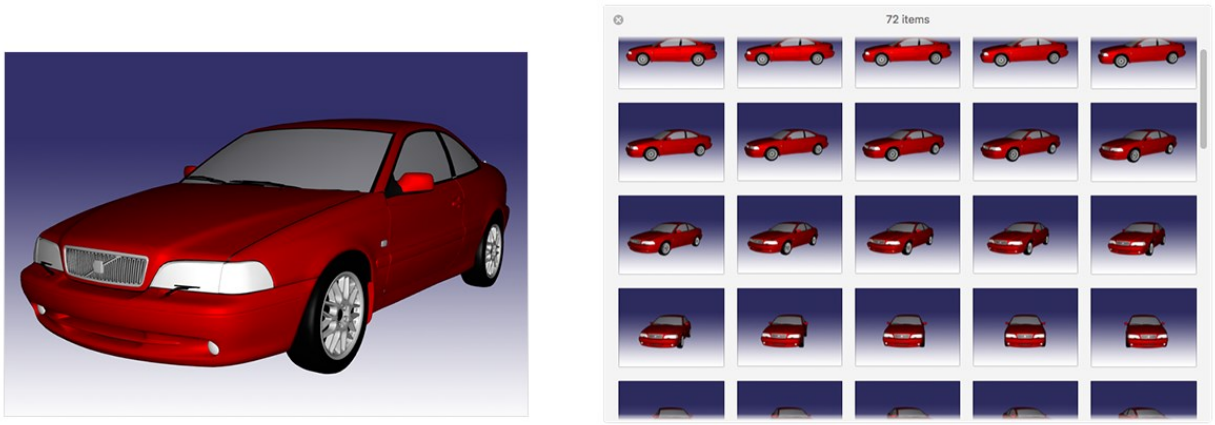


***Figure 3**: Example 3D model in the viewer software, and its 72 extracted rotation images*

3.2.2 Initial classification

Because we need some initial classification of the objects to later compare our algorithm's results against the expected outcome, a software (web-based for practical reasons, programmed in HTML/JS/CSS/PHP) was developed, allowing to have an overview of the whole objects database, to visualize, in both color and greyscale, each view/rotation of an object, and to edit its class attribution. The metadata are dynamically loaded/saved in JSON files, which will be read by the main C++ program afterwards. Figure **4** shows the webapp with some of the 3d objects (here, vehicles) classified when the background is green. One car in the upper right (object #94) is being shown being rotated (the angle is based on the mouse position).

***Figure 4**: Screenshot of a car part of the database in the viewer/editor webapp*

3.3 Implementation

3.3.1 Introduction

The implementation of the algorithm was done by programming, as efficiently as possible, the algorithm summarized in part 2.4.

For this, we wrote a software taking the object images database as input (note that they are made greyscale by the software), with other additional and optional tuning parameters, and outputting, as CSV, the results (recognition ranks, recognized class, confidence, etc.) of each object treated as a test one by removing it from the trained set.

This allows us to test all the database at once, very quickly, and import the results inside a spreadsheet application for further analysis and comparisons between the different parameters.

The software is programmed in C++ for performance reasons. It relies on the OpenCV library for some math-related functions and structures, as well as for our test implementation of the EM-GMM framework. Additionally, we have leveraged the OpenMP library to get multi-threaded parallelized computations. This brought a ≈ 4.2x speedup in average. On our computer[2], the program takes around 20 seconds on a 100-object database to both do the whole learning and try each object as a test one. More time measurements are given in the appendices (4.4).

The final program executable binary, when compiled with speed optimization, is around 320 KB (including logging code and related strings), and the actual EP code itself around 1/5th of that. A test framework, done in the Bash scripting language, was also created, to launch our program sequentially with various parameters in order to determine what works best.

3.3.2 Feature extraction

A critical point in computer vision is the appropriate choice of a way to "reduce" an image to a set of optimal features, which are in fact "point of interests".

After having tested several common features such as Hu, SURF, SIFT, BRIEF, ORB, we have determined that for our implementation[3], Zernike-moments based features were the appropriate candidate in our case, in terms of accuracy and speed (although SURF, for instance, was faster).

---

[2] Apple MacBook Pro, 2.5 GHz Intel Core i7-4870HQ – program running with 8 threads

[3] In our software, the implementation of the Zernike feature extraction is a modified version (edited to go well with the rest of our code) of the freely-available source from [***30***].

The math behind it is reviewed in the Appendices, but what is worth noting is that Zernike features are especially interesting for several reasons:

In addition to being quite accurate (see [***30***]), their rotational invariance (moment magnitude is the same) as shown in [***31***], is very useful where inputs in this context may not always be perfectly adjusted. Moreover, scale and translation invariance are achieved by having already pre-processed inputs (images as pixel matrices), as done in our database: they always are sized 320x240px and with a properly positioned centroid. This is visible in **Figure 4**.

With Zernike features, contrary to some other methods, we can "choose" how many columns our final feature vector will have, by picking an appropriate moments order n. In our case, n=6 was chosen empirically (by testing and doing a size/speed/accuracy trade-off), which leads to a vector of dimension 16. Because we use 72 angles to "describe" one object, it means that each one will be characterized, as input to the training algorithm, by a matrix made of 72 rows of 16 columns.

Here are examples of the output of the Zernike feature extractor for a car object, in **Figure 6** shown with both angles (72) on the x axis and the column index. Y axis is always the magnitude.

What we directly notice (and even more depending on which model is viewed) is the somewhat symmetrical aspect, especially on the angle-major view (top part), which isn't very surprising for a car. Here is a superimposed view of that with its object:

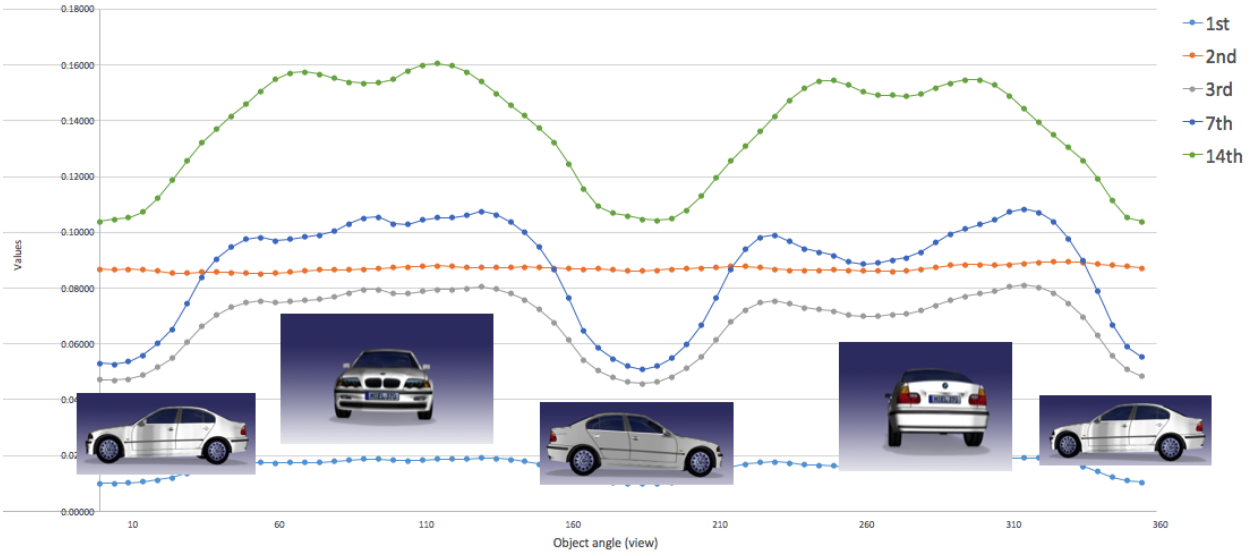


Figure 5*: Some Zernike moments of a car across 72 views, with corresponding pictures*

The downside of having precise details (even with n=6) is that a small disturbance on the image can make a very visible difference in one (or several) column.

For instance, we have noted that one car model had a broken texture for one of the back lights, and the image thus showed some high contrast over that area. This was very noticeable in the Zernike features, especially when represented with angles on the x axis: an unusual "hole" became part of the curve, and later resulted in misclassification.

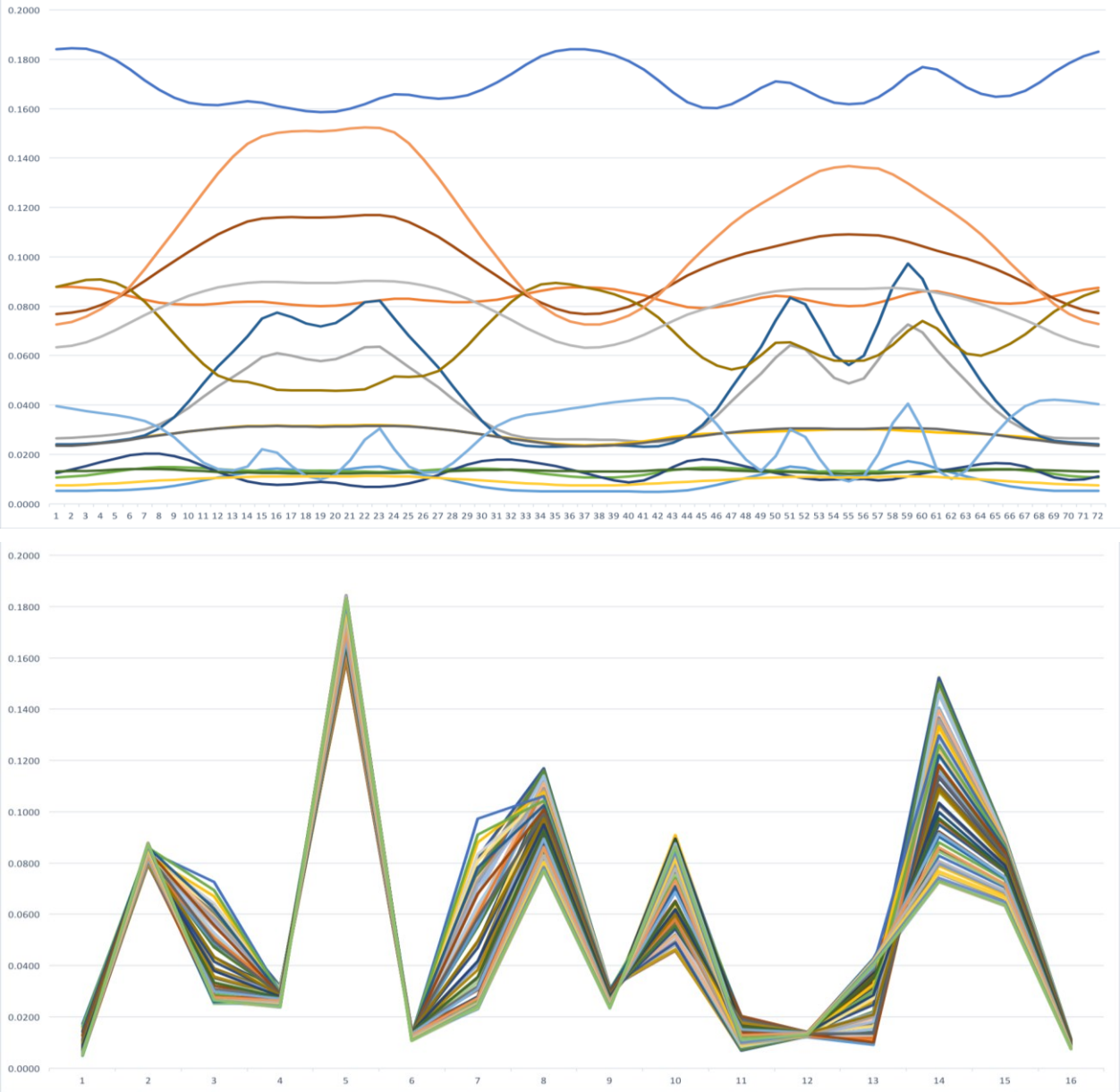


***Figure 6**: View (angle)-major and order-major 16D Zernike features of the same car*

Here are more examples of Zernike feature visualizations, that show peculiar behavior:

- Depending on the model, the end results can be extremely symmetrical (and in which case, it's a waste of precision, see the end of 3.4 about some mitigation), as shown in **Figure 7**.
- When a model doesn't show any remarkable point of interest, the Zernike features will be flat, and can thus be problematic afterwards depending on the training algorithms, as shown in **Figure 8**

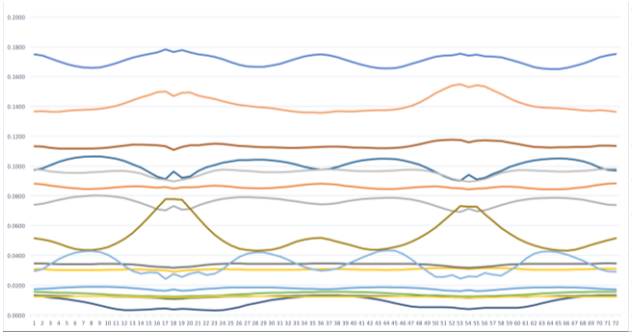


***Figure 7**: Symmetrical 16D Zernike features of a bike*

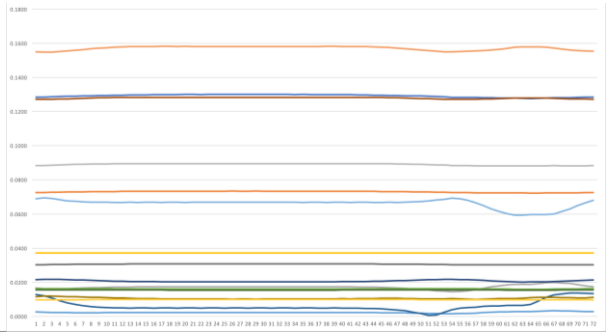


***Figure 8**: "Flat" 16D Zernike features of a ball*

3.3.3 Initialization

We apply the processes described in part 2.3. However, we modify the generic formula (29) from [***17***], by one that is more useful in our case:

$$k = \frac{x'_{1,14} - x'_{2,14}}{x'_{2,14} - \left(x'_{1,14}\right)^2} \tag{32}$$

Now, it slightly differs in the indices, since we have made a specific improvement following an observation of feature vectors' values: we have noted that the variance (denominator of k) taken on the 1st x value (the usual formula) can be too low, which yields a high k, thus making α values too high for the initializations. Instead, we take the 14th value (out of 16 Zernike vector dimensions, in our case), since we can see empirically that it is most often this one that presents a higher variation/range (while still being relatively low). In fact, the 7th one also presents interesting variations, and sometimes better, but has however lower values than the 14th, thus decreasing its interest for k.

This particular phenomenon can be seen in **Figure 6**: in the top half (by angle), the orange curve (second from top). This is also clearly visible in **Figure 5** (top two curves, green and blue).

3.3.4 Main computation

This is a direct implementation of steps 3 to 6 of the algorithm in 2.4. Although for a better accuracy due to possible technical issues such as floating-point approximations and underflows & overflows, the computations of the distribution PDFs, for instance in equation (25), are done logarithmically. See appendices for the adapted formulas and details. Numerical issues are a source of many problems on programs like this. Signaling NaNs and features like floating point interrupts/exceptions (overflows, underflows, division by zero…) are very useful. On the actual implementation side, [***32***] (Mac) and [***33***] (Linux) provide code samples to help with this.

One more change regarding the implementation, compared to the Bayes rule steps around eq. (25), is that a threshold is applied, i.e. a vector only gets reassigned if the chosen-best cluster actually gives a probability that is k times (with $k > 1.25$ in our tests) higher than with the current one. This "control" over a less limited reassignment allows for a safer vector flow between clusters, giving at least equal or better accuracy, and decreases the amount of iterations of the main loop.

3.3.5 Post-processing

Comparing mixtures with the KL-divergence

In order to establish a "distance" between the models, we have to properly compare the obtained mixtures. For this, we have chosen the well-known Kullback-Leibler divergence, also named relative entropy. We will now call it D_{KL}. With $f(\alpha)$ and $g(\alpha)$ the two PDFs, we have by definition:

$$D_{KL}(f||g) = \int f(\alpha) \log \frac{f(\alpha)}{g(\alpha)} \, d\alpha \tag{33}$$

It follows three properties: positivity ($D_{KL}(f||g) \geq 0$), self-similarity ($D_{KL}(f||f) = 0$), and self-identification ($D_{KL}(f||g) = 0$ only if f=g).

For the Dirichlet, we have ([***34***]):

$$D_{KL}(f||g) = \log \frac{B(|\alpha_g|)}{B(|\alpha_f|)} + \sum_{k=1}^{D} \left[(\alpha_{f_k} - \alpha_{g_k}) \left(\psi(\alpha_{f_k}) - \psi(|\alpha_f|) \right) \right] \tag{34}$$

For the Inverted Dirichlet, we have ([***16***]):

$$D_{KL}(f||g) = \log \frac{B(|\alpha_g|)}{B(|\alpha_f|)} + \sum_{k=1}^{D} \left[(\alpha_{f_k} - \alpha_{g_k}) \left(\psi(\alpha_{f_k}) - \psi(|\alpha_f|) \right) \right] + (|\alpha_f| - |\alpha_g|) \psi(|\alpha_f|) \tag{35}$$

It's noteworthy that the KL-Divergence is not symmetric, and generally, *f* serves as the reference on which *g* is compared. However, it can be made symmetric by computing, for instance (others exist, see [***35***]), the following:

$$\mathrm{D}_{\mathrm{KL}_{\mathrm{sym}}}(f||g) = \mathrm{D}_{\mathrm{KL}}(f||g) + \mathrm{D}_{\mathrm{KL}}(g||f) \tag{36}$$

This symmetric version is the one used in the implementation.

A practical issue, which didn't seem to be talked about in papers, was how to handle comparing mixtures that did not have the same amount of clusters (referred to "cluster number mismatch" later on). This definitely happens often because objects do not get clusterized the same way, especially when they are of a different class. The approach taken in our implementation is the following:

1) Sort both mixtures' clusters by their weight (from highest to lowest)
2) minSize := min(nclusters_mixt1, nclusters_mixt2)
3) For *i=0 ; i<minSize ; i++*
   a. Compute the desired value for the i^{th} cluster of each mixture
   EndFor

In our case, step 3.a would be here the KL-Divergence.

One can note that there is indeed information loss from the mixture where extra clusters had to be discarded, However, this can be compensated (see eq. (37) and the paragraph above it).

Let's note that in the implementation, for performance's sake, the sorting step is done globally on all mixtures before the looped KL-Divergence computations explained in the next paragraph.

Actual class recognition process

In order to measure and analyze how good the recognition is, different factors have to be taken into account, and several numerical values per test object are evaluated.

Now that we have a way to get a "distance" between two mixtures, we can compute the KL-divergence matrix (or rather, only half of it, then mirror it, since we use eq. (36)), between all the objects that we have including the test/unknown one. That is because we want to test all the objects at once; of course if we want to recognize *one* "new" object, we will not have a matrix, but rather a single vector, of dimension *number of objects in the database + 1* (itself, 0).

An improvement over simply filling the matrix with the KL-Divergence values is to actually artificially add more contrast to the numbers. In fact, this step also happens to make up for the loss

of precision when ignoring extra clusters due to a cluster number mismatch. In order to do that, we make the very-probable hypothesis (which is confirmed experimentally) that objects pertaining to the same class will most likely be represented by mixtures having the same number of clusters. As such, each number in the matrix is actually modified as:

$$\mathrm{mat}_{j,i} = \mathrm{mat}_{j,i} * \left(1 + \left|nclusters_j - nclusters_i\right|\right) \quad (37)$$

The figure below shows a color-coded display of the KL-Divergences matrix, from lowest (green) to highest (red), between a few dozen mixtures of cars from our database *(note that the predominance of greener values for some rows/columns is actually normal, as this part of the database had more "sedan" and "coupe" classified objects).*

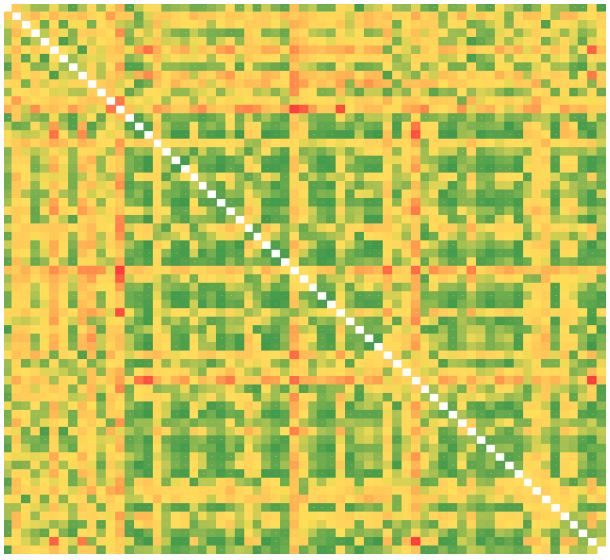


***Figure 9**: Color-coded ("temperatures") matrix of KL-Divergences between object mixtures*

As said, for one test object, we only have a vector. Colors, when grouped, indicate recognized / suggested classes. See figure below:

***Figure 10**: Color-coded ("temperatures") vector of KL-Divergences between object mixtures*

After obtaining this KL-divergence matrix (or vector), we must analyze it, taking into account the initial classification on the training database. For this, we "group" the matrix values into their object's class and attribute it 4 main statistics:

- Minimum value (and associated class)
- Average value (and associated class)
- Median value (and associated class)

- Difference between first and second median values ("Δnext_med"), which is a confidence

For the first three, the values are given by taking the set of matrix values from columns that correspond to each class known from the database.

For instance, let's consider a 3D model of a car to classify (with the goal being to recognize it of type/class "sedan"). In the matrix, we group the values by their known classes, and compute, for these classes values: the minimum, the average, and the median. Then, in theory, the object can be considered to be part of the class that shows the lowest values. Sample lowest values and class are given in **Table 2**.

min	avg	med	Δnext_med
2.255 hatchback	28.609 sedan	22.397 sedan	24.637

***Table 2**: Sample lowest values and class for an object to be recognized as a "sedan" car.*

In this case, the unknown object would be (correctly) classified as a "sedan".

We tend to give more weight to the median value since it is a rather good indicator of the class' results, and possibly not disturbed by outliers that can happen in minimum and average.

Improving the accuracy

We are also interested in the difference between first and second median values because that gives an idea about how contrasted the recognized classes were, i.e. if the first class happened to be very distinct from the next one (which means it is a very clear recognition), or close to it (meaning the two class "look" the same from this object's mixture's "point-of-view", thus not giving a result as reliable as one with a higher difference).

With that, we use the minimum and average values and ranks for balancing "guesses" and improve the recognition: if we note that a class suggestion is close to the next one (as per the median difference), we may want to also check if the second suggested class happens to have better (lower) values for the minimum and average – if that is the case, we tend to prefer that new suggestion. This uncertainty could have probably been lifted with a bigger database (as said later in the "Future work" part). A typical example of this, for an unknown car that should have to be detected as "convertible", is shown in part 3.4.

3.4 Results on real data

We have used our custom-made database to form several tests using lots of various sets of classes to try different combinations and see what is optimal, what works, what doesn't etc.

3.4.1 Initial tests

To be sure that the mixture comparison algorithm and implementation are correct, we have tested the recognition of test objects that were themselves part of the training database.

Indeed, 100% of the objects were recognized correctly with the best possible theoretical score (KL-Divergences of 0, ranks #1) and highest confidence.

After that, tests were made on a database that was made of only images from car models, which were classified, manually with our webapp, into several classes like « SUV », « convertible », « coupe », « sedan », « truck », « hatchback », « van », « old », « sport/tuned », « limousine ».

The initial results were not very good, with a rather high percentage of false positives for classes that had mode objects in the database. Over time, we've been able to determine that those poor results were due to:

- The will to classify a test object rather than to tell if it pertains to a specific class (or, as per the next point, several well-distinct ones) or not.
- Not respecting enough the two rules detailed in the next part, 3.4.2 .
- Not having proper tuning values specialized for at least general car classes, or a specific class, as explained towards the end of 3.4.3 .

3.4.2 The importance of a well done training database

It is important that classes are distinct enough between each other, especially when the database doesn't have thousands of items. For instance, let's consider those three models, in **Figure 11**:

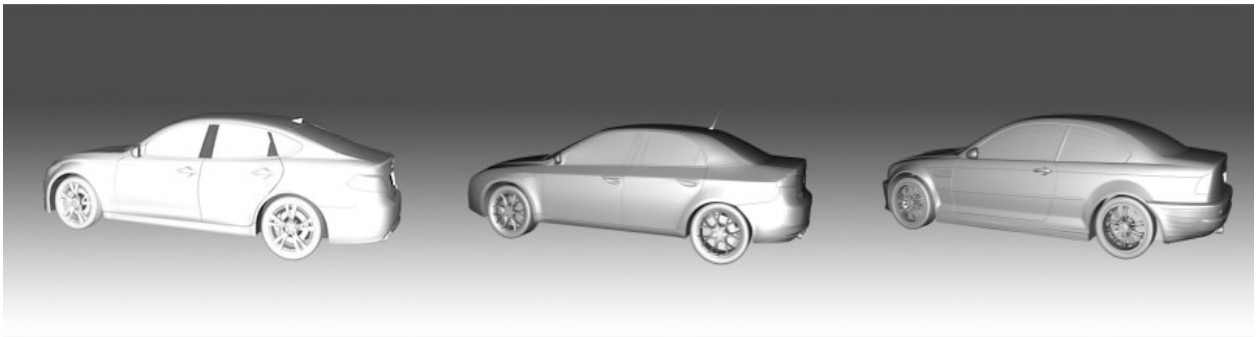


***Figure 11**: Three very similar cars, but not theoretically in the same class (sedan, sedan, coupe)*

Ignoring the color/lighting difference, it is not obvious at all at first sight that those three cars should be in two different classes – in fact, in theory it should be Sedan for the first two, and Coupe for the last one, because of the number of doors (4 vs. 2). The problem is that they are still very close visually, and making such a distinction ends up causing confusion for the training, as many coupes will be recognized as sedans, for example.

Another example, though less pronounced, is for certain sport carts getting mixed with convertible cars. Here are some recognition results with classes as they were set initially (separate sedan/coupe classes) then with those classes merged since they are enough similarly-looking to justify this:

	Object classes	**Approximate Recognition rate**	**Confidence (see *Δnext_med*)**
Before merging	**Coupe**	46%	Low
After merging	**Sedan+Coupe**	87%	High

***Table 3**: Some optimal cluster numbers of classes after EP Learning*

Another "mistake" is to put, in the same class, several objects that don't directly expose the same visual features. For instance, in **Figure 12**, initially both cars were put in the "old car" class. Indeed, a human will recognize that common characteristic, and the fact that both show a third wheel. However, the non-visual property of the age of the car or the different place of the fifth wheel will not be good for the feature extractor, and will create a great heterogeneity of objects among the same class objects, which is definitely detrimental to the training.

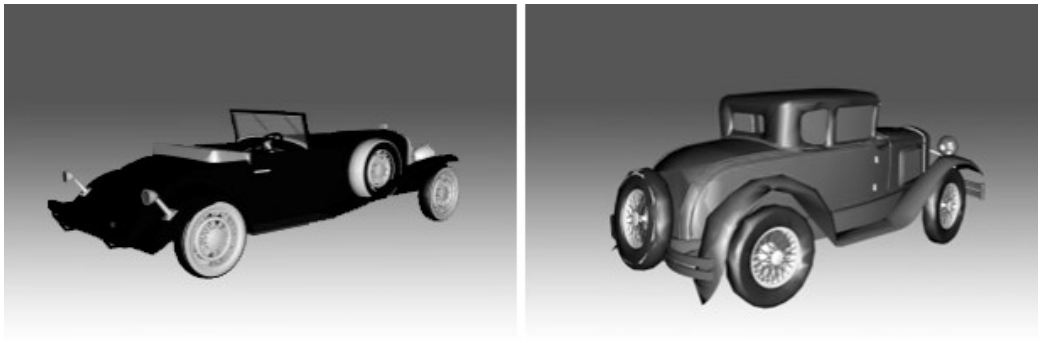


***Figure 12**: Two cars humans may put in the same "old cars" class but a computer would not*

A solution to the problem exposed in the above figure is to either split the class into several distinct types of "old cars" and populate them with similar looking models, or to reclassify them in other existing class.

For instance, the first one could probably fit in the "convertible" or "sport" one, which is exactly what our software outputted (two first classes suggestions, with low Δnext_med).

In summary, here are two important "rules" for training databases we've learned experimentally:

1. The classes must be "far" enough from each other (i.e. not like **Figure 11**)
2. The classes must contain objects that look very alike, and as many as possible.

In fact, regarding this second rule, it'd be even better if the similarity was not only visual, but also feature-wise, meaning that the feature extractor used in the implementation also considers the class objects to be very similar to each other (since humans can "misclassify" objects from a feature extractor's point of view, as per **Figure 5**).

Applying those rules as much as possible, even on our "modest" database, improved the results.

3.4.3 Tests with training on cars + other objects

In order to directly see how resilient our algorithms are to "abnormal conditions", a foreign object (also from the INRIA website [*27*]), not a car, has been tested against a car-only database.

When analyzing the KL-divergence matrix between all objects, we can see that this new object is definitely recognized as an outlier with values far greater than the ones corresponding to car classes, especially for the minimum ranks. The test is also successful when injecting 2 and 3 foreign classes, isolating them as well.

Figure 13 and **Figure 14** show an example of these tests. Note that even better results/contrast can be achieved when tuning the parameters to indicate we should be looking at "cars", for instance – in fact, differences as high as 175x as were achieved.

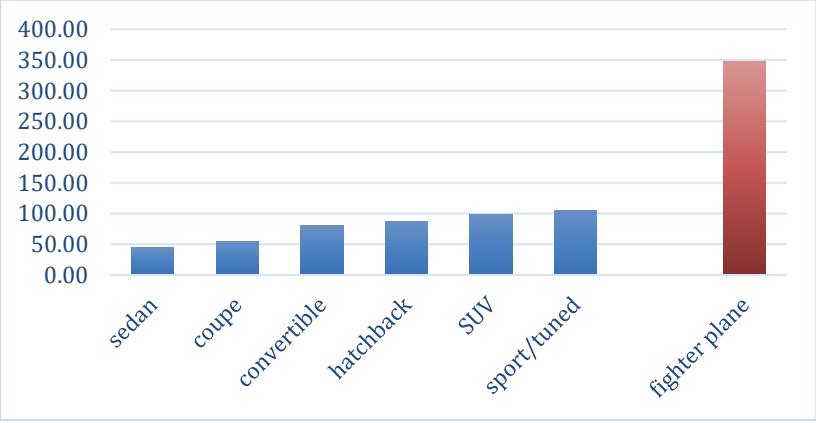


***Figure 13**: Some KL-divs of a successful EP-InvDMM foreign object robustness test*

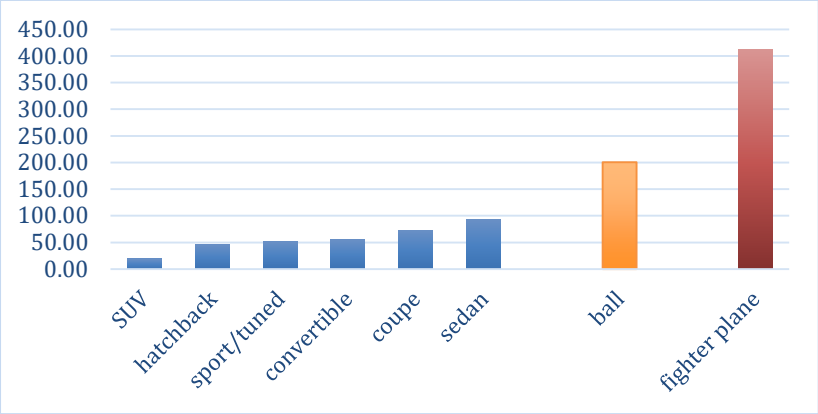


***Figure 14**: Other KL-divs of a successful EP-InvDMM foreign objects (2) robustness test*

For this test, let's note that we have also tried the EM-GMM learning approach, and it does perform better than EP-DMM/InvDMM, in terms of getting very different KL-divergence values without specific tuning, and the classes hardly ever get mixed. **Figure 15** shows this.

We can even note that it is superior almost regardless of the cluster number, though trying somewhat "extreme" numbers (low & high Gaussian components number) may give closer values.

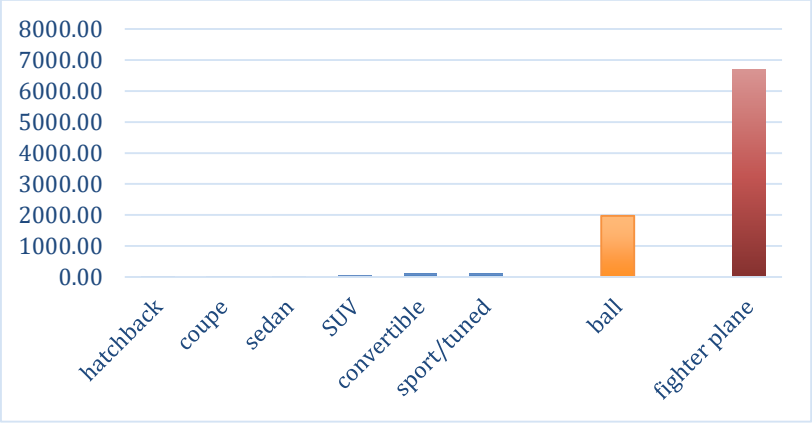


***Figure 15**: EM-GMM showing extreme KL-divs heterogeneity with foreign objects classes*

Over time, we have added several classes to the training database. It is interesting to compare the final (and in theory optimal) mixtures cluster number at the end of the EP learning algorithm; **Table 4** shows a few classes and their average optimal cluster number.

Moreover, all have a low standard deviation, and this observation led to the KL-divergence contrast improvement method based on the compensation of the cluster number mismatch ((37) and above).

Class	Average optimal cluster number
Face	5
Most cars	4
Airplane	3
Bike	2
Ball	1

***Table 4**: Some optimal cluster numbers of classes after EP Learning*

What we have noted initially is that recognition false positives started to get higher as object from new classes were added. To overcome that, we have started to try different tuning parameters that would be able to improve the performance on identifying particular classes so that the software can clearly (with a high confidence value) tell if a test object is of a given class or not.

It's interesting to note that for faster "bulk" testing, we didn't hardcode tuning parameters directly in the source code, but instead made the software take command-line arguments. Indeed, those don't require a rebuild, thus is a great time-saver when trying a lot of different values.

We have concluded that certain classes have specific parameters that allow for more accurate recognition, in addition to optimizing the initial cluster number to the ones in **Table 4** ; for instance the sedan class is better recognized when using a Bayes vector reassignment PDF threshold (as explained at the end of part 3.3.4) of 1.4, while the convertible tends to yield better results when using a higher threshold, 1.6. Another parameter to tune, is rotation skipping (explained in more details in part 3.4.7), for instance most car models can be differentiated from other models by training on the first 90° or 180° angles, while differently symmetrical objects would require the whole rotation range for better accuracy (or simply no catastrophic feature loss).

One peculiar issue we have consistently witnessed over the course of the tests is that sometimes, objects of one specific class (a car one: SUV) were particularly bad at being correctly recognized, with extremely low recognition rate, and for no apparent reason at first. Moreover, we weren't apparently able to determine a particular set of tuning values that improved recognition rates higher significantly, which is usually an easy task, as we've been doing for other classes in earlier tests.

One thing we noted, however, is that the objects did not get recognized as any other car classes in our tests, which is a good result in itself, but rather were sometimes mixed with other object classes that also presented the same kind of symmetries, and with low confidence levels. It's almost as if the class was an "outsider" for the model, and got treated as such. After more investigation, it

became apparent that the objects were being heavily influenced by another class that was a minority in quantity (in this case, some of the "airplane" objects) during the training, so much than when carefully removing the disturbing elements (and only those, such that the database was still made of about 100 objects), the SUV class went from a nil-like recognition rate to actually 100%[4] (although object #105 has a low recognition confidence), as shown in **Table 5**, giving a sample of the results before and after cleansing the training database from the disturbing objects.

It seems that a way to properly solve this kind of issue, rather than removing the problematic elements and thus losing some training objects, is to build a bigger database with more models, and respecting the two "rules" detailed in the previous part. Indeed, we have not come across such an issue with other non-car objects being present in higher quantity in the training database.

obj_id	class	rank_min	rand_avg	rank_med	Δ_next_med
58	SUV	2	2	4	33.6781
84	SUV	1	2	2	3.11188
89	SUV	2	3	3	5.43049
102	SUV	1	3	3	7.79237
104	SUV	1	2	2	13.3934
105	SUV	1	3	3	7.37101

obj_id	class	rank_min	rand_avg	rank_med	Δ_next_med
58	SUV	1	1	2	2.42768
84	SUV	1	1	1	69.5897
89	SUV	1	1	1	64.3906
102	SUV	1	1	1	59.0777
104	SUV	1	1	1	72.7513
105	SUV	1	2	1	7.33812

***Table 5**: Recognition stats of a few SUV cars before (0%) and after (100%) cleansing the training database from problematic models influencing negatively on this class*

3.4.4 Recognition with a hierarchy in mind

Because we have a database with several "sub-classes", at least for the cars (see the list in the "Initial tests" part), we can see that if we replace all car classes occurrences in our tests results by the parent category "car", for the class detection (the rest, like "plane", "bike", etc. don't change),

[4] In **Table 5**, we see that there is still a "2" rank for the object #58, indicating that the recognition failed (SUV not being the best class suggestion), and thus not yielding a 100% recognition rate. However, it actually becomes recognized correctly as SUV when applying the results analysis method detailed in 3.4.7

then the results for the car classes are great rate for tests on a mixed database of 110 objects (although with a majority of car objects)

- around 95% car recognition (and 100% if based on the minimum values instead of median)
- around 80% recognition for the other classes

This result isn't surprising, and if for some application the goal is to detect what general kind of object it is, then this is a good approach. The problem with more sub-classes is that they have to be very well done and with sufficient homogeneity within the classes, but sufficient heterogeneity across them as well at the same time – see the previous parts.

3.4.5 Dirichlet vs. Inverted Dirichlet

Regarding the differences on the results between the two mixture models, it is not actually *very* pronounced. We note that both perform globally the same way in terms of accuracy.

However, across several tests, objects presenting difficult features conditions (very high symmetry, flat Zernikes, see **Figure 8** for instance, although this is the extreme - see the next part "Recognition failures") are handled better by the Dirichlet than the Inverted Dirichlet. Here is an example (where we group classes by object quantity of this test database for simplicity's sake):

	Object classes	**Approximate Recognition rate**	**Confidence (see *Δnext_med)***
Dirichlet	Sedan cars	100%	Very High
	Other objects (balls, planes, faces, bikes…)	86%	Medium to High
Inverted Dirichlet	Sedan cars	92%	High to Very high
	Other objects (balls, planes, faces, bikes…)	71%	Medium

***Table 6**: Some recognition rates & confidences in suboptimal conditions, on a test database*

For some more difficult conditions on some classes (less training objects than in some other classes), the Inverted Dirichlet tends to performs better on the majority of tests that we have done.

3.4.6 Recognition failures

We have hit some cases where objects were very often not recognized correctly, and contrary to the SUV issue detailed in part 3.4.3 , without apparent solutions or workarounds on the algorithm or results analysis side:

- Certain objects do not present enough characteristics to later have workable Zernike features (cf. the flat one). We have even tried a different MM and learning algorithm, EM-GMM, and it turns out that it was even worse for those particular objects (thus apparently confirming that it's not an internal issue with our implementation).

  In this case, since this directly affects the data given to the EP learning algorithm, there is not much we can do about it. We could think of using a different feature extractor when we detect an extremely low variance across vectors, though.

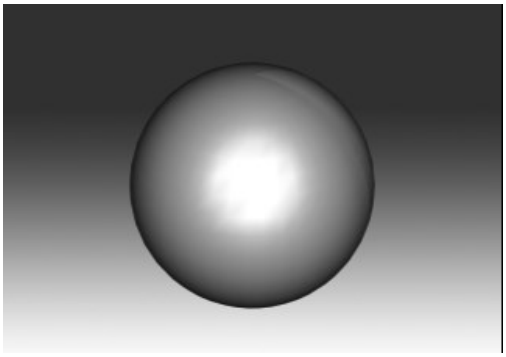


***Figure 16**: A ball yielding very poor Zernike features across its angles (flat)*

- Some other objects were manually assigned a class because they belonged there according to human reasoning, but clearly the Zernike features were very different.

  Some explanations could be:

  - particularly bad lighting on this specific model
  - background interference (the models' textures play a role)
  - low-resolution 3D object model source (which tends to create artificial noise)

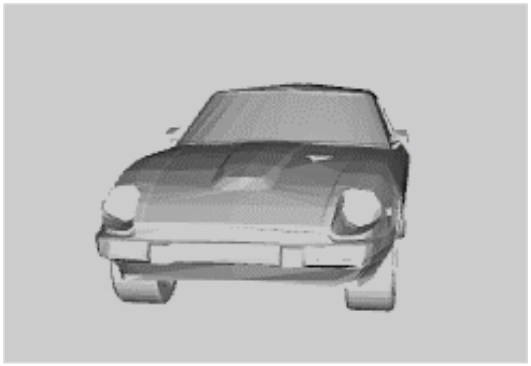


***Figure 17**: Example of suboptimal lighting and low-poly 3D object model*

3.4.7 Further analysis of the results

Here is a typical example of a low Δnext_med as explained in 3.3.5 Improving the accuracy:

min	avg	med	Δnext_med
6.099 convertible	52.373 convertible	61.474 sport	0.28

Table 7: Example of an uncertainty in recognition clearly shown by a low Δnext_med

As we can see, for some reason, the sport class had a better median than the convertible, but the Δnext_med is extremely low, showing that this class suggestion isn't very reliable (low confidence). But the minimum and average however "agreeing" on the same class, we will then consider that the unknown object is of class "convertible", which is the correct class of this object.

We have noticed an improvement of the recognition rankings when we decide to restrict the views of the objects to ones that show less symmetry over the angles. For instance on a car-only database, since there are roughly 4 very different viewpoints (front, rear, left, right - see **Figure 5**), or even possibly 2 (front/rear, side) depending on the model and precision, we have tried only training on the first 90° and first 180° view angles.

Percent changes for some test rankings are given in **Table 8**.

Count of..	% Change
Best rankings	≈ **0%** in average
Worst rankings	≈ **-50%** in average *(best seen: **-84%**)*

Table 8: % change in rankings of recognition on a restricted view-set

So, while it doesn't improve the best recognitions, it does so on the worst ones. The consequence is that objects that previously failed to be recognized properly can now be considered as some "unsure" recognitions (over different test objects, they may give either correct or close to correct results – for instance, cars that look alike).

On a more contrasted database (more objects than cars), testing for cars will naturally improve results for cars (and any objects that happen to show similar symmetries), and tests have shown that almost all uncertainties (low Δnext_med) can decrease or even disappear with proper tuning. This restricts the usage of the software to specific data to be looked for, but this is not a real problem as we can simply launch differently tuned instances of the program if needed.

Moreover, such partial training has a huge positive speed impact, as there is much less data to process, so the whole program is faster.

Partial conclusion

With our proposed method, we can achieve very good recognition results as long as the training is meticulously done: sufficiently separated classes, and within the same class, a good homogeneity. This is crucial in order to have minimal false positives and low KL-divergences. And of course, the bigger the database is, the better. From our tests, it is clear that we achieve much more reliability when we set our goal not to classify unknown objects, but rather to be able to tell whether they belong to a certain class, by leveraging, if needed, several instances of our program tuned specifically for different class, in order to have more accurate recognitions.

3.5 Impact of our improvements

Our improvements have mostly been done along the way (not thought of from the beginning) and based on results analysis over time, trying out different tuning parameters, experimenting on some algorithms, etc.

Pre-processing / initialization

Improvement…	…on speed	…on accuracy
Dynamic choice of the initial values of mean vector and covariance matrix	✗ *(negligible)*	✓
Optimal choice of vector for the constant in the moment method to initialize alphas	unchanged	✓

EP learning / training

Improvement…	…on speed	…on accuracy
Finding the zeros of the alphas equations using a dichotomy-based method instead of Newton-Raphson or Gradient descent, etc.	✓	unchanged
Covariance matrices denoising after they have been updated	✗ *(negligible)*	✓
Computing PDFs in log instead of the traditional formulas	✗ *(negligible)*	✓
Constrained (minimum threshold) Bayes-rule-based reassignment of vectors	✓ *(small improvement)*	✓

Post-processing / recognition

Improvement…	…on speed	…on accuracy
Dynamic compensation of a cluster number mismatch when calculating the KL-Div.	✗ *(negligible)*	✓

3.6 Failed improvement attempts

We believe it is also interesting to mention some improvements attempts that have failed to actually make the results better. Here are some.

3.6.1 Input images histogram equalization

It's generally not a bad idea to try to optimize properties like this on images before further processing, however we've noticed that, in addition to taking a small but non-negligible time, applying a simple histogram equalization pass on our input images often does not improve the results, and can even make them worse.

Indeed, we already may have lighting issues, on both the background and the object itself, and thus "noise" was made more prominent, as shows in the **Figure 18** comparison where artificial boundaries get created on the object, which has a negative impact on the feature extraction step.

Let's however note that more advanced equalization techniques such as Contrast Limited Adaptive Histogram Equalization ([***36***]) could probably give better results. We have not tested this, though.

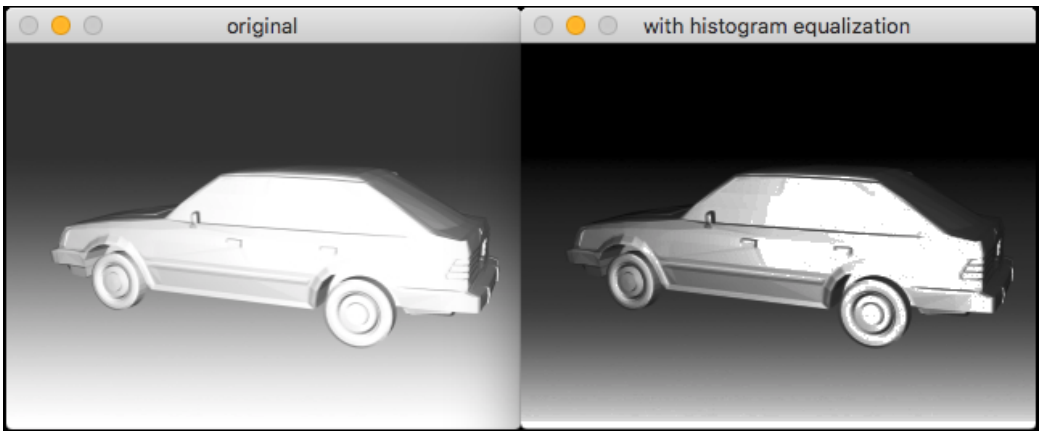


Figure 18*: Artificial boundaries and noise introduced by trivial histogram equalization*

3.6.2 Additional EP pass after the final reassignment

At the end of the algorithm described in 2.4, step 6), we reassign vectors whose clusters have a very low weight. An idea was then to call, once, the main loop of the EP algorithm. It turns out that results were a bit worse when doing that (the convergence now being disturbed), or identical at best. In any case, there was a non-negligible time impact too.

3.6.3 Parallelized EP learning

Being able to add multi-threading to the main loop of the learning algorithm could have probably brought a significant improvement on the speed. While it was assumed that a trivial parallelization without taking care of adapting the algorithm was probably not going to work since it is an iterative process, we have confirmed it – not only the convergence criterion also wasn't applicable anymore "as is", but threading issues arose, too. When working with multi-threading, we have to make sure that all the results are equal to those when the program run on a single thread, and we saw that here. Issues such as race conditions can cause a lot of troubles, too. However, let's note that there exists research to properly parallelize EP training, like [*37*], based on further data partitioning.

3.7 Discussion

3.7.1 Automatic training

An important optimization is probably possible on the implementation part regarding the actual database making process: instead of having a human-made classification of each object initially, we could make the software itself group the objects it trains into clusters of objects that have low KL-Divergence between each other. It would in fact be comparable to "K-Means" resulting on some sort of "meta-clustering" on classes.

Of course, the bigger the object database, the better, and it would be an option to choose how many classes the software would put together. Once the objects grouping is done, then a human can try to assign a tag/name for each one, for features relevant to what the computer "saw".

This process has an incredible positive impact on the initial database making. However, a downside would be that what "looks" similar in a feature-extractor way for the computer (Zernike, etc.) may not always be representative of actual different object types that a human would distinguish.

3.7.2 Applications and real-time processing

Obvious applications of such recognition technology could be:

- **Security/Surveillance** for cameras, possibly coupled with tracking
- **Navigation** for robots, autonomous vehicles, etc.
- **Industrial processing/manufacturing** for factory robots interacting with objects

While some applications may not require real-time processing (a surveillance camera's data could be processed later on a computer, daily for instance), some others do, as they could be on critical systems. The most striking example would be for the emerging market of autonomous / self-driving vehicles, that would need to constantly monitor the environment, detect what's around it and recognizes the relevant "objects" to be able to make decisions, at a high frequency.

The question of processing power for real-time quickly arises: on a standard computer processor, it's an accuracy (input images resolution, etc.) vs. speed trade-off. It all comes down to how fast one object / a set of images can be passed through the whole algorithm and compared to the trained data (the database training is assumed to have been done beforehand, with readily accessible data).

This is becoming more and more possible even on embedded devices, where specialized SoC take the lead for these applications. For instance, NVIDIA has recently developed its "DRIVE PX" product [***38***], especially designed for the autonomous vehicles market. On the other hand, traditional small boards, like Arduino, Raspberry Pi etc. are much cheaper, but don't have enough processing power for such real-time applications.

On our desktop computer, as detailed in 4.4, we could achieve around 10 car models per second.

CONCLUSION & FUTURE WORK

The goal of this thesis was to develop Expectation Propagation learning frameworks, applied to the Dirichlet and Inverted Dirichlet mixture models, to recognize objects from a view-based 3D models database which we have manually assembled and classified. The EP Learning allowed us to experiment various enhancements to recognize unknown objects. Improvements include both accuracy and speed. For instance, automatic optimal choice of initialization values, dichotomy-based solving of a central equation needed to refine mixture parameters (much faster than techniques such as Newton-Raphson), dynamic compensation of a cluster number mismatch when comparing mixtures, etc.

The promising results that we have obtained have shown the importance of proper conditions on the database for the training classification: we are able to get good recognition rates as long as two rules are followed: the objects of a class must have a good homogeneity, especially in terms of the feature descriptor/extractor used, and the classes must be well distinguishable so that they don't get mixed during the learning or recognition. In addition, it is natural that the bigger the database is, the better the training and final results will be. We can note that results with the proposed EP learning with Dirichlet mixture model are relatively similar to the Inverted Dirichlet one: both good when we apply the rules stated above; although the latter tends to yield a better recognition rate on “difficult” shapes (a ball, for instance). The results also show that the performance is better when we try to detect if an unknown object is of a specific class or not, rather than directly classify it in general among several classes. Indeed, our implementation has shown the ability to be tuned for certain classes, thus with an improved specialized recognition rate. This is a very acceptable behavior as it often corresponds to practical applications in real-life, already existing nowadays but which will be even more developed in the near future: security/surveillance for cameras, industrial processing/manufacturing for factory robots, and of course, and possibly the most important, navigation, for robots and autonomous vehicles, etc.

Here are some ideas for potential improvements on our approach:

- **On a database level**, having a bigger object database for the training would considerably help obtaining a bigger matrix of KL-divergences with more data allowing finer comparisons. The issue in our case was that making the database was a manual and tedious process. Automating it, from the 3D models to the image files, would be helpful. If/When that is done, trying what is discussed in 3.7.1 would be very interesting.

Moreover, for instance focusing on a car-database perspective, with a lot more training objects, it would probably be better to give more importance to the minimum ranks (compared to the median), and possibly removing the first match (in case it is an outlier – although few next values), as it is possible that for a large amount of cars, a very similar one got trained, thus the learning will get very close mixtures and low KL-Divergence.

- **On a view level**: detect the object symmetries, similar angle sights, etc. and reduce the set of views that would yield redundant features, allowing the data to be clustered in a more efficient way for the same amount of feature columns (this was started in 3.4).
- Regarding the overall **learning process**, for a more unsupervised approach with a goal to classify multiple unknown objects of different classes, it would be possible, for each object, to try out sequentially several parameters from known classes and use the results that give the lowest mixture distances for all the result ranks, as it would probably mean that it corresponds to an object being detected with its specific class parameters.
- [***39***] presents improvements on Expectation Propagation learning that would be worth looking into, although the given details and example are more towards the Gaussian Process.

REFERENCES

[1] Daniel Lowd and Pedro Domingos, "Naive Bayes Models for Probability Estimation," in *ICML*, 2005.

[2] Xiangfei Qian and Cang Ye, "3D Object Recognition by Geometric Context and Gaussian Mixture-Model-Based," in *IEEE International Conference on Robotics & Automation*, Hong-Kong, 2014.

[3] Kah-Kay Sung and Tomaso Poggio, "Example-Based Learning for View-Based Human Face Detection," *IEEE Transactions On Pattern Analysis And Machine Intelligence*, vol. 20, no. 1, pp. 39-51, January 1998.

[4] Meng Wang, Yue Gao, Ke Lu, and Yong Rui, "View-Based Discriminative Probabilistic Modeling for 3D Object Retrieval and Recognition," *IEEE TRANSACTIONS ON IMAGE PROCESSING*, vol. 22, no. 4, pp. 1395-1407, Avril 2013.

[5] Christophe Andrieu, Nando De Freitas, Arnaud Doucet, and Michael Jordan, "An Introduction to MCMC for Machine Learning," in *Machine Learning*., 2003.

[6] Zhihua Zhang et al., "Learning a multivariate Gaussian mixture model with the reversible jump MCMC algorithm," *Statistics and Computing*, vol. 14, pp. 343-355, 2004.

[7] Sun Min, Su Hao, Savarese S., and Fei-Fei L., "Min, Sun ; Hao, Su ; S., Savarese ; L., Fei-Fei," in *Computer Vision and Pattern Recognition*, Miami, 2009, pp. 1247-1254.

[8] Cordelia Schmid and Liebelt Joerg, "Multi-View Object Class Detection with a 3D Geometric Model," in *Computer Vision & Pattern Recognition*, 2010, pp. 1688-1695.

[9] Gu Chunhui and Ren Xiaofeng, "Discriminative Mixture-of-Templates for Viewpoint Classification," *Computer Vision - ECCV*, vol. 6315, pp. 408-421, 2010.

[10] Derek Hoiem and Silvio Savarese, *Representations and Techniques for 3D Object Recognition and Scene Interpretation*.: Morgan & Claypool, 2011.

[11] Lian Zhouhui, Godil Afzal, and Sun Xianfang, "Visual Similarity based 3D Shape Retrieval Using Bag-of-Features," in *Shape Modeling International Conference (SMI)*, Aix-en-Provence, 2010, pp. 25-36.

[12] Chen Ding-Yun, Tian Xiao-Pei, Shen Yu-Te, and Ouhyoung Ming, "On Visual Similarity Based 3D Model Retrieval," *Computer Graphics Forum*, vol. 22, no. 3, pp. 223-232, 2003.

[13] Krishnan Ramnath, Dufipta N Sinha, Richard Szeliski, and Edward Hsiao, "Car Make and Model Recognition using 3D Curve Alignment," in *Applications of Computer Vision (WACV), 2014 IEEE Winter Conference*, 2014, pp. 1-8.

[14] Louis-Philippe Morency, Ali Rahimi, and Trevor Darrell, "Adaptive view-based appearance models," in *Computer Vision and Pattern Recognition - IEEE conference*, 2003, pp. 803-810.

[15] Qiong Liu, "A Survey of Recent View-based 3D Model Retrieval Methods," Research report arXiv:1208.3670 [cs.CV], 2012.

[16] Taoufik Bdiri and Nizar Bouguila, "Bayesian learning of inverted Dirichlet mixtures for SVM kernels generation," *Neural Computing and Applications*, vol. 23, no. 5, pp. 1443-1458, October 2013.

[17] Nizar Bouguila, Djemel Ziou, and Jean Vaillancourt, "Unsupervised Learning of a Finite Mixture Model Based on the Dirichlet Distribution and Its Application," *IEEE TRANSACTIONS ON IMAGE PROCESSING*, vol. 13, no. 11, pp. 1533-1543, Nov. 2004.

[18] Wentao Fan and Nizar Bouguila, "Non-Gaussian Data Clustering via Expectation Propagation Learning of Finite Dirichlet Mixture Models and Applications," *Neural Process Letters*, vol. 39, no. 2, pp. 115-135, Avril 2014.

[19] Wentao Fan and Nizar Bouguila, "Expectation propagation learning of a Dirichlet process mixture of Beta-Liouville distributions for proportional data clustering," *Engineering Applications of Artificial Intelligence*, vol. 43, no. C, pp. 1-14, August 2015.

[20] Tom Minka, "Expectation Propagation for Approximate Bayesian Inference," *Proceedings of the conference on uncertainty in artificial intelligence (UAI)*, pp. 362-369, 2001.

[21] George G. Tiao and Irwin Cuttman, "The Inverted Dirichlet Distribution with Applications," *Journal of the American Statistical Association*, vol. 60, no. 311, pp. 793-805, Sept. 1965.

[22] Manfred Opper, Andre Manoel, and Jack Raymond, Expectation Propagation, 2013.

[23] Zhanyu Ma and Arne Leijon, "Expectation propagation for estimating the parameters of the beta distribution," *Proceedings of IEEE international conference on acoustics speech and signal processing (ICASSP)*, pp. 2082-2085, 2010.

[24] Tom Heskes and Onno Zoeter, "Expectation propagation for approximate inference in dynamic Bayesian networks," *Proceedings of the conference on uncertainty in artificial intelligence (UAI)*, pp. 216-223, 2002.

[25] Miguel A. Carreira-Perpiñán, "Mode-Finding for Mixtures of Gaussian Distributions," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 22, no. 11, pp. 1318-1323, November 2000.

[26] Bruce Fielitz and Buddy Myers, "Estimation of parameters in the Beta distribution," *Decision Sciences*, vol. 6, pp. 1-13, 1975.

[27] INRIA GAMMA Group. (2013, November) 3D Meshes Research Database. [Online]. https://www.rocq.inria.fr/gamma/gamma/download/download.php

[28] Various authors for "The Free 3D Models". (2015) Vehicles 3D Models - Free 3D Vehicles download. [Online]. http://tf3dm.com/3d-models/vehicles

[29] Various authors for 3dmodelfree.com. (2015) Transport 3D Models Downloads. [Online]. http://www.3dmodelfree.com/3dmodel/list420-1.htm

[30] Stephen Yoo and Vorobyov Vorobyov. (2011, August) Binary Shape Clustering via Zernike Moments. [Online]. http://www.math.uci.edu/icamp/summer/research_11/vorobyov/ZM.pdf

[31] P. F. Krekel, "Zernike Moments and Rotation Invariant Object Recognition: A Neural Network Oriented Case Study," *TNO-Physics and Electronics Laboratory*, 1992.

[32] Geoffrey Irving. (2008, December) Stack Overflow - Enabling floating point interrupts on Mac OS X Intel. [Online]. http://stackoverflow.com/a/340683/378298

[33] Jay Conrod. (2010, May) jayconrod.com - Trapping floating point exceptions in Linux. [Online]. http://jayconrod.com/posts/33/trapping-floating-point-exceptions-in-linux

[34] Manuel Gil, Fady Alajaji, and Tamas Linder, "Rényi divergence measures for commonly used univariate continuous distributions," *Information Sciences*, vol. 249, pp. 124-131, November 2013.

[35] Don H. Johnson and Sinan Sinanovic, "Symmetrizing the Kullback-Leibler Distance," *IEEE Transactions on Information Theory*, 2000. [Online]. http://www.ece.rice.edu/~dhj/resistor.pdf

[36] Karel Zuiderveld, "Contrast limited adaptive histogram equalization," *Graphics gems IV*, pp. 474-485, 1994.

[37] Aki Vehtari, Pasi Jylnki, Christian Robert, Nicolas Chopin, John P. Cunningham Andrew Gelman. (2014, December) arxiv - Expectation propagation as a way of life. [Online]. http://arxiv.org/abs/1412.4869

[38] NVIDIA. (2016) NVIDIA DRIVE PX. [Online]. https://www.nvidia.com/object/drive-px.html

[39] Manfred Opper, Ulrich Paquet, and Ole Winther, "Improving on Expectation Propagation," *Advances in Neural Information Processing Systems*, no. 21, pp. 1241-1248, 2008.

[40] Vladimir Sacek. (2006, July) Telescope Optics. [Online]. http://www.telescope-optics.net/monochromatic_eye_aberrations.htm

[41] Laurent Ribon. (2011) GLC_Player -- Documentation : Multi Captures. [Online]. http://glc-player.net/doc.php?page=multiCaptures

[42] Michael Boland, Tom Macura, Lior Shamir, and Ilya Shamir. (2012, December) Zernike moment generating function. [Online]. https://wnd-charm.googlecode.com/svn-history/r230/wndchrm/branches/feature-timing/textures/zernike/zernike2.cpp

CHAPTER 4: APPENDICES

4.1 Review of the K-Means clustering method

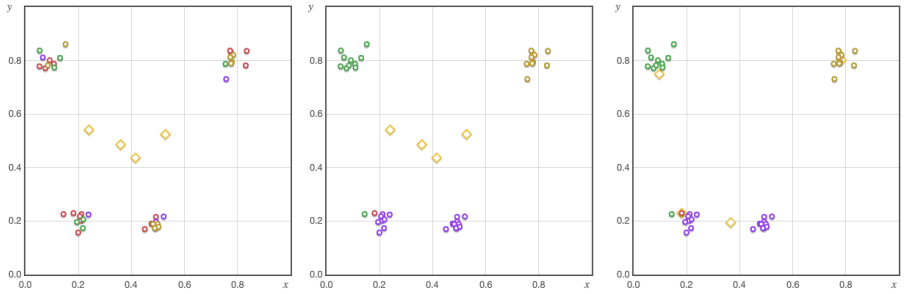
The goal of k-means clustering is to partition n observations into k clusters such that each observation is assigned to the cluster with the closest mean.

The most common algorithm to implement that, also called Lloyd's algorithm, is simple and done by alternating two steps, given an initial set of k means $m_1^{(1)}, \ldots, m_k^{(1)}$:

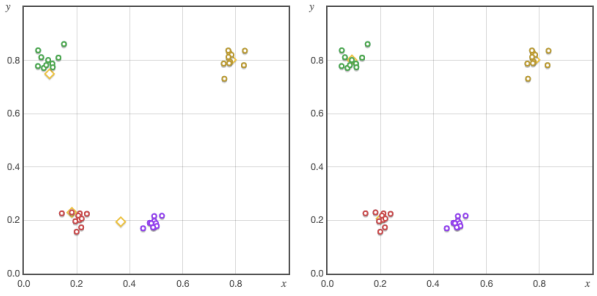
- Assignment: each observation is assigned to the cluster having the nearest mean as per the Euclidian distance): $S_i^{(t)} = \left\{x_p : \left\|x_p - m_i^{(t)}\right\|^2 \leq \left\|x_p - m_j^{(t)}\right\| \; \forall j, 1 \leq j \leq k\right\}$ where each x_p is assigned to only one $S_i^{(t)}$.
- Update: the new means is calculated, as to be the centroids of the observations in the new clusters: $m_i^{(t)} = \frac{1}{\left|S_i^{(t)}\right|} \sum_{x_j \in S_i^{(t)}} x_j$. This minimizes the nearest mean.

who

Here is a visualization of k-means clustering iterations with random initial centroids and 4 clusters:



1: initialization, 2: assignment step, 3: update step



4: assignment step, 5: update step

At the end, the centroids are well positioned directly in the clusters' points.

4.2 Zernike moments and polynomials

We have seen that the Zernike moments show interesting properties in terms of geometric invariances. Let's investigate the math behind them. They are a sequence of polynomial forming a complete orthogonal set on the unit disk. There are even and odd polynomials, respectively: $Z_n^m = (\rho, \phi) = R_n^m(\rho)\cos(m\,\phi)$ and $Z_n^{-m} = (\rho, \phi) = R_n^m(\rho)\sin(m\,\phi)$. With m and n positive integers and $n \geq m$. ϕ is the "azimuthal angle", ρ the radial distance ($0 < \rho < 1$), and R_n^m:

$$R_n^m(\rho) = \sum_{k=0}^{\frac{n-m}{2}} \frac{(-1)^k (n-k)!}{k!\left(\frac{n+m}{z}-k\right)!\left(\frac{n-m}{2}-k\right)!}\rho^{n-2k}$$

Here are some Zernike polynomials:

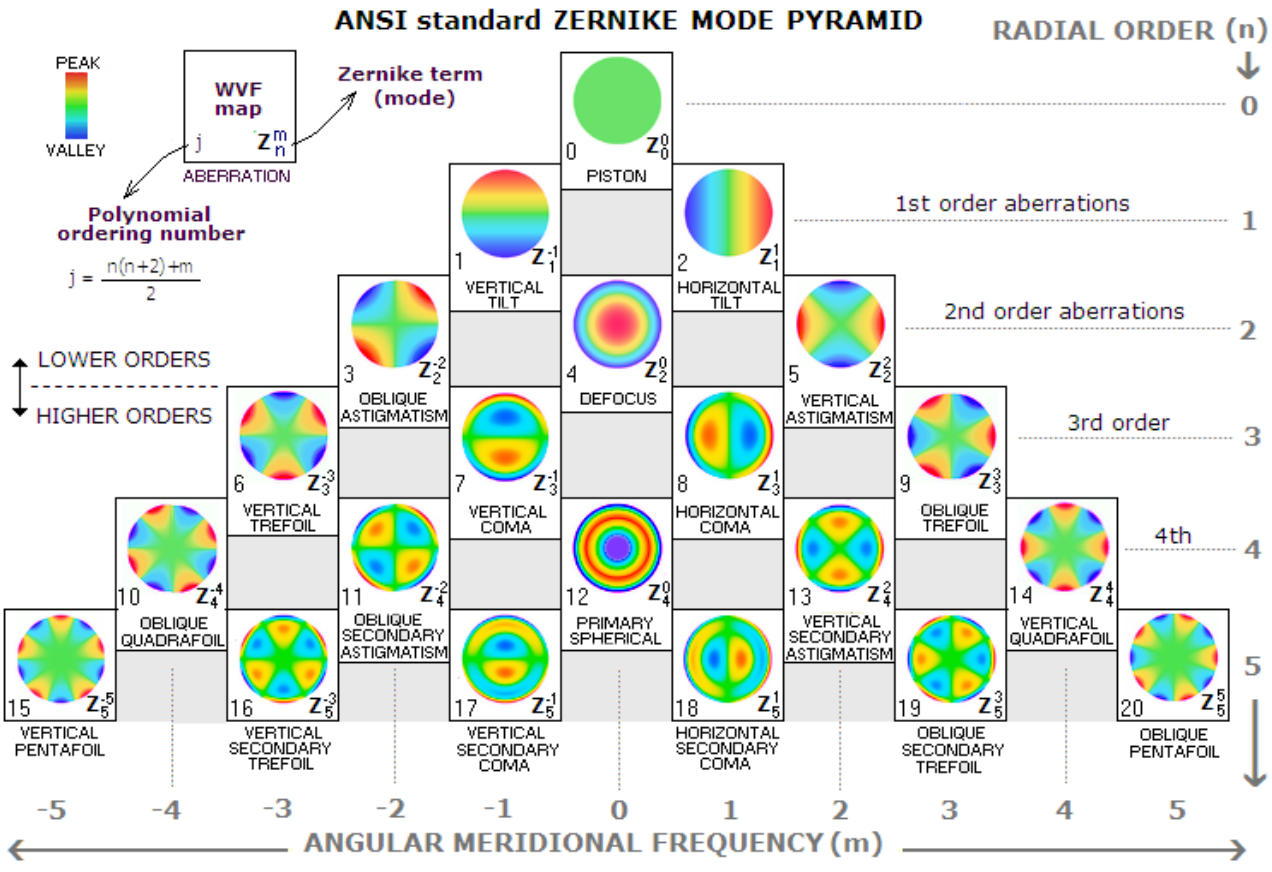


Image source: [*40*]

4.3 Formulas used in their logarithmic versions

As said, mathematical formulas involving "big" numbers and several multiplications, that may result in underflows or overflows in some implementation (even if 64bit variables are used), can instead be calculated logarithmically. This changes multiplications into sums and numbers get much more "manageable" in term of how large they are. Here are some of the log formulas used:

The **Dirichlet PDF**:

$$\log \mathrm{p}(\mathbf{X}|\boldsymbol{\alpha}_j) = \log\left(\frac{1}{\mathrm{B}(\boldsymbol{\alpha_j})}\prod_{l=1}^{D} X_l^{\alpha_{jl}-1}\right)$$

$$= -\log B(\boldsymbol{\alpha_j}) + \sum_{l=1}^{D}(\alpha_{jl}-1)\log X_l$$

The **Inverted Dirichlet PDF**:

$$\log \mathrm{p}(\mathbf{X}|\boldsymbol{\alpha}_j) = \log\left(\frac{1}{\mathrm{B}(\boldsymbol{\alpha_j})}\prod_{l=1}^{D} X_l^{\alpha_{jl}-1}\left(1+\sum_{l=1}^{D} X_l\right)^{-|\boldsymbol{\alpha}_j|}\right)$$

$$= -\log B(\boldsymbol{\alpha_j}) + \sum_{l=1}^{D}(\alpha_{jl}-1)\log X_l - |\boldsymbol{\alpha_j}|\log\left(1+\sum_{l=1}^{D} X_l\right)$$

Both use $\boldsymbol{\log B(\alpha_j)}$ which can be computed as:

$$\log B(\boldsymbol{\alpha_j}) = \log\frac{\prod_{l=1}^{D}\Gamma(\alpha_{jl})}{\Gamma(|\boldsymbol{\alpha_j}|)}$$

$$= \log\prod_{l=1}^{D}\Gamma(\alpha_{jl}) - \log\Gamma(|\boldsymbol{\alpha_j}|)$$

$$= \sum_{l=1}^{D}\log\Gamma(\alpha_{jl}) - \log\Gamma(|\boldsymbol{\alpha_j}|)$$

If the result of the actual original formula is needed, then we can simply take the exponential of the result, otherwise, if we only need the result to be compared to another one, then we can compare both logarithm-based results.

4.4 Some time measurements of our software

Code profiling is a great help to understand and locate bottlenecks; it includes measuring the time spent in functions. Here are some noteworthy timings, done with the full 72 views, on a database of 104 objects, on the same configuration as in as in footnote 2. Some of the timings overlap, since we are doing a lot of multithreading.

Total time spent	**200 ms per object**
Reading the object's images	33 ms per object
Feature extraction (Zernike)	150 ms per object
EP learning	13 ms per object
Building the KL-Div matrix	20 ms for all objects

For specific applications, as we've seen on cars, views could be reduced to half or even a fourth of the 72 images, that means the 150ms could be reduced to about 38ms, and the total time spent for an object recognition would take less than 100ms, thus at least a 10fps processing could be done.