Normal Classifiers using Mathematica, Octave, and Core Graphics

Dan Beatty

December 10, 2007

1 Introduction

In the original two assignments for the pattern recognition course, the class was instructed to explore four algorithms: Principal Component Analysis (PCA), Multiple Discriminant Analysis (MDA), Expected Maximization(EM), and Independent Component Analysis (ICA). PCA, MDA, and EM are described in the mathematical overview section (section 2). ICA is given its own section as this algorithm is commonly misunderstood and clarification is warranted to understand the work presented in this report.

The specific assignment for EM, PCA, and MDA is stated as follows:

- You are given a total of 150 four dimensional samples of three Iris species. Use Multiple Discriminant Analysis (MDA) to separate the samples into three Iris species. Show the resulting misclassifications. Repeat MDA

choosing any three-dimensional features at a time and show the resulting misclassifications.

- You are given a retinal digital picture in color. Take the gradient of the monochrome image and use the Expectation-Maximization (EM) algorithm to extract the edge points from the blurred optic disc boundary.
- Use Principal Component Analysis (PCA) in combination with the Whitening transform and Linear Discriminant Analysis (LDA) to classify and segment the optic disc from the background.

For an image, segmenting any portion on the fly is a difficult thing to perform due to interface constraints. MDA for example is complied with in a prototype manner in this report as such an interface has not been built yet. MDA is described in theory in subsection 2.2, and has a Mathematica prototype for the Iris species included in section 5. The Mathematica version was chosen at the time since MATLAB was unavailable due to changes in platform (both Mac with Intel chips and Microsoft's Windows Vista). Also, for the problem in question Mathematica simply seemed more intuitive for the problem at hand.

For the EM algorithm, another two prototypes were constructed. One of them is an Octave implementation of a classic Matlab implementation of the EM algorithm, originally by [**?**]. This implementation can be directly translated into C (consequently Objective-C and C++). However, this direct translation has drawbacks to being utilized in a production

system that users will expect to be responsive. In subsection 2.3, this report describes the essence of the EM algorithm from its mathematical foundation. In section 6.1, the EM classic implementation is described as well as modifications to ensure responsiveness in a GUI based application. The prototype GUI-based implementation leads to a more general form using Quartz Composer plug-ins, and yields reasonable results.

As will be pointed out in the PCA subsection (2.1), whitening and classical estimators for PCA are nearly synonymous. Differences between PCA and whitening only have to do with the inclusion of scaling factors, which are obtained in the process of determining the PCA estimator.

While the CPU bound prototype for PCA, written for this report (in Objective-C), is typical of other implementations of PCA (most notably MATLAB implementations), it leaves much to be desired with regards to efficiency. The implementation in this example uses the Linear Algebra Package (LAPACK) libraries which are part of the Accelerate Framework in all Apple OSX systems since version 10.2. Even with these libraries, the lack of use of the many processors available on the hardware the system is running on. While this is the fastest the CPU can offer, there is another processor on the user's system that can offer. In section 6.2, a CPU bound implementation is demonstrated and attempts to squeeze as much efficiency from the CPU as possible.

ICA in the form of projection pursuit can be seen as an extension of PCA. For a single independent component, projection pursuit performs acceptably well. However, if that algorithm were to be used in deflationary pursuit, the complexity would grow.

The original instructions the pattern recognition class was provided for ICA read as follows:

> This project is to illustrate how a modified Independent Component Analysis (ICA) can be used to restore an image corrupted with additive Gaussian noise.
>
> Choose any natural image (of a reasonable size), add 30% Gaussian noise to the image, assuming a noisy image model:
>
> $$\vec{z} = \vec{x} + \vec{n} \tag{1}$$
>
> Denoise the noisy image by using the Sparse Code Shrinkage Method (closely related to ICA) as described in the following in the following reference.
>
> Compare the restored image using the above method with standard noise removal filters such the simple median filter or the Wiener filter.
>
> MATLAB codes for various versions of ICA are available from the home page of A Hyvärien and the Laboratory of Computer and Information Science at the University of Helsinki.

It turns out that the MATLAB codes provided by Hyvärien leave much to be desired in comparison with his own book. As such, it is typically downloaded in a broken state and is difficult to debug due to the complexity of the MATLAB codes. The implementation of projection pursuit and deflation pursuit presented in this report is a simplification based on

the algorithms found in Hyvärien's book. The improvements made to the implementation are covered in a separate chapter and extensively referenced in the appendix on the topics of Quartz-Composer plug-ins and Core Image Kernel Filters, which are key to the construction of the alternative form.

2 Mathematical Overviews

Expected Maximization (EM), Linear Discriminant Analysis (LDA), and Principal Component Analysis (PCA) can each be classified as a family of estimators. Classically known estimators include least squares and maximum likelihood, which differ by their method of convergence. LDA is a special case of Multiple Discriminant Analysis (MDA) where the number of discriminants is reduced to two.

Both PCA and MDA produce mappings of zero mean data to orthogonal vectors. MDA differs by using both the background scatter of the entire data and specified "within" scatter matrices defined by specific segments of the data. A scatter matrix S is defined in the following equation:

$$S = \sum_k (\vec{x_k} - \vec{m})(\vec{x_k} - \vec{m})^T \tag{2}$$

A background scatter matrix is the scatter matrix for the entire collection of data points. In practice, it is more useful to determine a scatter matrix for a reasonable sampling of the data than to use the entire data set. From such a background scatter matrix, the principal components can be determined.

2.1 Principal Components

> The principal components of the observed data are selected so that the ith principal component is the linear combination of the observed data that accounts for the ith largest portion of the variance in the observations. [**?**, 329]

Let $\vec{x}$ be a sample from the observed data whose space is denoted $\mathbf{X}$. Let $\vec{y}$ be the principal components of $\vec{x}$. Then there is a transformation matrix $\mathbf{P}$ such that equation 3 is satisfied.

$$\vec{y} = \mathbf{P}\vec{x} \tag{3}$$

In order for $\mathbf{P}$ to satisfy equation 3, it should maximize equation 4. This choice is made

to satisfy the least squares error.

$$J(\mathbf{P}) = E\{\vec{y}^2\} \tag{4}$$

Most texts show that for each column vector of $\mathbf{P}$ to have a derivation as shown in equation 5. The important feature of $\mathbf{P}$ to be noted is that the eigenvectors of the covariance matrix of $\mathbf{X}$ are ordered by matching eigenvalues.

$$J(\vec{p}_1) = E\{y_1^2\} = E\{(\vec{p}_1^T \vec{x})^2\} = \vec{p}_1^T E\{\vec{x}\vec{x}^T\}\vec{p}_1 \tag{5}$$

$$||\vec{p}_1|| = 1 \tag{6}$$

Any $\mathbf{P}$ that satisfies the constraints of PCA is a valid PCA transformation matrix. Some methods simply approximate $\mathbf{P}$, and are not a collection of eigenvectors. These methods are often called on-line methods because they approximate $\mathbf{P}$ as each $\vec{x}$ becomes available. Other methods obey equation 5 by obtaining the eigenvectors, and some texts call these methods batch methods as they work on $\mathbf{X}$ as a collected whole. Singular Value Decomposition, Karhunen-Loeve, Hotelling transforms, and Pearson all acquire the transformation matrix that determines the principal components.

In the end, the principle components are simply defined in terms of eigenvectors and eigenvalues for a scatter matrix. As stated earlier in the section, the background scatter matrix can be used to determine the principal components as well. A mixing matrix, $\mathbf{P}$, can determine each principal component as shown in equation 7. In this case, there is

an assumption that each data vector $\vec{x}$ is defined in terms of some mixing matrix, $\mathbf{P}$, a matching set of principal components $\vec{a}$, and a mean vector $\vec{\mu}$. The mean can for practical purposes be the sample mean of $\vec{x}$. Equation 8 shows the inverse relation of principal component to data vector.

$$\vec{x} = \vec{\mu} + \mathbf{P}\vec{a} \tag{7}$$

$$\mathbf{P}^T(\vec{x} - \vec{\mu}) = \vec{a} \tag{8}$$

$$\vec{x} = \vec{\mu} + \sum_i a_i \vec{e_i} \tag{9}$$

Principal components are can be used determining the independent components. The reason is that principal components inherently possess properties zero mean and of unit variance. Also, as defined in equation 9, order of the eigenvectors are irrelevant. It is important to ensure that the eigenvectors are orthonormal for this derivation and equation 9 to be correct. Some eigenvector estimators compute the eigenvectors but without any form of normalization. The result of failing to ensure normality is a scaled version of the principal components, which is wrong.

For CPU bound computation of PCA, SVD is a classic well known algorithm that is part of the LAPACK libraries and has acceleration via the Accelerate framework [**?**]. For GPU bound, both classic methods and online methods should be considered. At the time this report was written, only a scant number of implementations of PCA were made for the

GPU. Of those made, all of them were constructed for this report. Those considerations at this time are beyond the scope of this report.

In section 6.3, PCA and Independent Component Analysis (ICA) are explored in the perspective of how to apply them image processing, filters and masks. In the context of masks, it is more convenient to use the transformation matrixin the construction of a filter. In some filtering processes, it is better to use the principal components, and the inverse mixing matrix.

The following two subsection describe both Eigenvector Decomposition (EVD) and Gram-Schmidt Orthogonalization (GSO). Both procedures are well known methods for obtaining an orthogonal matrix, which will satisfy the mixing matrix. EVD is included in LAPACK, and already has a CPU bound implementation which has been optimized via the Accelerate Framework. GSO is better candidate for GPU bound computations as it can be constructed via modules that are zero-branch.

2.1.1 Gram-Schmidt Orthogonalization

Principal components are defined by a transformation matrix which is orthonormal and typically consisting of eigenvectors. One other estimate for such an orthogonal matrix is called Gram Schimidt Orthogonalization (GSO). Theorem 2.1 states that an vector space $\mathbf{V}$ can be transformed in to another vector space $\mathbf{W}$ such that each $\vec{w} \in \mathbf{W}$ is orthogonal. GSO is a procedure based on theorem 2.1 that constructs each $\vec{w}_i \in \mathbf{W}$ one right after the other, and is defined in algorithm 1.

Theorem 2.1 *Suppose $w_1, w_2, ..., w_n$ form an orthogonal set of non-zero vector in* $\mathbf{V}$*. Let v be any vector s.t. $v \in \mathbf{V}$. Define v' s.t.*

$$v' = v - c_1 w_1 - c_2 w_2 - ... - c_n w_n \tag{10}$$

$$c_i = \frac{\langle v, w_i \rangle}{||w_i||^2} \tag{11}$$

Then v' is orthogonal to $w_1, w_2, ..., w_n$.

[**?**, 211]

Algorithm 1 Gram Schmidt Orthogonalization

Require: Column Wise Source Matrix: $\mathcal{V}$
  N is the number of column vectors in $\mathcal{V}$
  Base case $\vec{v_0} = \vec{w_0}$
  initialize $\vec{\theta^0}$, T, and $i \leftarrow 0$
  $v_0 \leftarrow$ [V columnVector:1]
  [W insertVector:v_0 atColumn:1]
  for j = 2 to N **do**
    $v_j \leftarrow$ [V columnVector:j]
    for i = 1 to j **do**
      $(w_i)norm \leftarrow ||w_i||$
      innerProduct $(i_p) \leftarrow \langle w_i, v_j \rangle$
      $negsum \mathrel{+}= i_p/(w_i)norm$
    end for
    result = $v_j - negsum$
    [W insertVector:result atColumn:j]
  end for
  return W

2.1.2 Eigenvalues: Characteristic Polynomials

Definition 2.2 *Consider an n-squared matrix* $\mathbf{A}$ *over a field* $\mathcal{K}$

$$\mathbf{A} = \begin{pmatrix} a_{11} & a_{12} & ... & a_{1n} \\ a_{21} & a_{22} & ... & a_{2n} \\ ... & ... & ... & ... \\ a_{n1} & a_{n2} & ... & a_{nn} \end{pmatrix} \tag{12}$$

The matrix $t\mathbf{I}_n - \mathbf{A}$ *where* $\mathbf{I}_n$ *is the n-square identity matrix and t is an intermediate, is called characteristic matrix of* $\mathbf{A}$

[**?**, 281]

The determinant of the characteristic matrix is the characteristic equation (polynomial) of $\mathbf{A}$. The definition for eigenvalues and eigenvectors is stated in the quoted definition 2.3.

Definition 2.3 *Let* $\mathbf{A}$ *be an n-square matrix over a field* $\mathcal{K}$*. A scalar* $\lambda \in K$ *is called an eigenvalue of* $\mathbf{A}$ *if there exists a nonzero (column) vector* $\vec{v} \in \mathcal{K}^n$ *for which equation 13 holds.*

$$\mathbf{A}\vec{v} = \lambda\vec{v} \tag{13}$$

Every vector satisfying this relation is then called an eigenvector of $\mathbf{A}$ *belonging to the eigenvalue* λ*.*

[**?**, 284]

There is a family of algorithms called Eigenvalue Decomposition (EVD) that finds the eigenvalues and eigenvectors. EVD is part of the Linear Algebra Package (LAPACK) libraries and consequently part of the Accelerate framework[**?**]. Thus for CPU bound operations, the LAPACK version is the best choice, as it is already optimized for the CPU.

2.2 Multiple Discriminant Analysis

A "within" scatter matrix for each category uses selected data points $\vec{x_s}$ that are believed to be within a specific category. Each category's scatter matrix, $\mathbf{S_i}$ is summed to form the within scatter matrix $\mathbf{S_w}$, as in equation 14. There is one presumption about the data that must be taken into account regarding these types of scatter matrices, namely size.

$$S_w = \sum_i S_i \tag{14}$$

In terms of image processing, this selecting member data vectors $\vec{x_i}$ is not as trivial or randoms as is the case for principal component analysis. The size of this scatter matrix must be the same as the background scatter matrix. If it were, the columns or rows that were specific attributes to be extracted would be viable. Consequently, the size of each $\vec{x}$ is the length of a column in the image. Another approach is to use neighborhood vectors as discussed in section 4. In the case of neighborhood vectors, the characteristic the background scatter matrix should be composed of a random set of neighborhoods union any members composing the within scatter matrix.

2.3 Expected Maximization

The goal of the EM algorithm is to determine a set of parameters that define a statistical model for given a certain data set. Some estimation models are trivial, and other more complex models can only be estimated. EM can estimate sufficient statistics of the assumed distribution set by a series of two steps, called the estimation and maximization steps.

EM extends properties of Maximum Likelihood Estimation (MLE) methods by determining via iterative convergence the governing parameters of a particular data-set. In particular, uncorrupted cases may use sufficient statistics acquired via MLE [**?**]. In general, the equation for the terminating condition of the EM algorithm [**?**] is stated in equation 15. The meaning of the terms in equation 15 are as follows:

- $\mathbf{X}$ is the true data set, $\vec{x}$ is the true data set sample,
- $\vec{\Theta}$ is the set of parameters for the statistic, and
- $\vec{y}$ is the data sample in the given data set.

$$\vec{\Theta}^* = \arg\max_{\Theta} \sum_{\vec{x} \in \mathcal{X}^n} E[\ln p(\vec{x}|\vec{\theta}, \vec{y})] \tag{15}$$

Equation 15 may be refined into equation 16 with the following notation obtained from [**?**]:

$$Q(\vec{\theta}; \vec{\theta^i}) = E_{D_b}[\ln p(D_g, D_b; \vec{\theta})|D_g; \vec{\theta^i}] \tag{16}$$

The following explains the terms used in equation 16:

- $\vec{\theta^i}$ is the current (best) estimate for the full distribution.
- $\vec{\theta}$ is a candidate vector for an improved estimate.
- $Q(\vec{\theta}; \theta^i)$ is a function of $\vec{\theta}$ and $\vec{\theta^i}$.
- D_b is the actual data set.
- D_g is the unknown and uncorrupted data set.
- $E_{D_b}[\ln p(D_g, D_b; \vec{\theta})|D_g; \vec{\theta^i}]$ is the expected value over the missing features. The expected value hinges on $\vec{\theta^i}$ which are the estimated true parameters.

One of the goals is to marginalize D_b with respect to $\vec{\theta^i}$. Another goal of the EM algorithm is to select a candidate $\vec{\theta}$ from a set of $\vec{\theta}$s which maximizes $Q(\vec{\theta}; \vec{\theta^i})$. In order for EM to work, the assumption that D_g contains independent and identically distributed (i.i.d) observations must hold.

In the Gaussian case, each class has a proportion that defines how much each class contributes as determined by their mean and covariance (denoted $\alpha, \vec{\mu}, \mathbf{\Sigma}$ respectively). If one acquires a sample as an initial set, is there an EM algorithm that will refine these sufficient statistics? In [**?**], there is an example using a matrix as the result of the expectation step. From this expectation, the sufficient statistics for the next step are computed. When an expected equation forms a matrix $\mathbf{A}$, as in equation 17, it is called an expected matrix.

$$a_{ij}^{(k)} = \frac{\alpha_j p(\vec{y_i}^{(k)}|\vec{\mu_j}^{(k)}, \Sigma_j^{(k)})}{\sum_{j=1}^{M} \alpha_j p(\vec{y_i}^{(k)}|\vec{\mu_j}^{(k)}, \Sigma_j^{(k)})} \tag{17}$$

The three sufficient statistics are computed in the maximization step. They are computed by the following equations.

$$\vec{\mu}_j^{(k+1)} = \frac{\sum_{i=1}^{N} a_{ij}^{(p)} \vec{y}_i}{\sum_{i=1}^{N} a_{ij}} \tag{18}$$

$$\mathbf{\Sigma}_j^{(k+1)} = \frac{\sum_{i=1}^{N} (\vec{y}_i \vec{\mu}_j^{(p)})^T (\vec{y}_i \vec{\mu}_j^{(p)})}{\sum_{i=1}^{N} a_{ij}^{(k)}} \tag{19}$$

$$\alpha_j^{(k+1)} = \frac{1}{N} \sum_{i=1}^{N} a_{ij}^{(k)} \tag{20}$$

The obvious question is, what reduction on $\mathbf{A}$ is used to assess the convergence of the estimations? The following equation answers this question:

$$Q(\theta^*; \theta) = \log(\prod_{i=1}^{N} a_{ii}) \tag{21}$$

Also, the initial guess for $\mathbf{A}$ can not be the zero matrix. As such, the sufficient statistics would be rendered zero, and no convergence would occur. Typically, the guesses are for μ to be scattered for each class and for the expected matrix to be

$$\mathbf{A} = \frac{1}{N}\mathbf{I}.$$

The EM Algorithm, which finds a solution to equations 16 and 15, should be implemented as stated in Algorithm 2. The expected step and maximization step are shown as separate methods and in implementation may be considered private methods and overrid-

den in each subclass. In an Objective-C, they are publicly accessible although the computation of the EM algorithm itself and any call for the resulting probability classifications are the only externally called methods.

Algorithm 2 Expectation Maximization: This algorithm satisfies the EM constraints specified in equations 16 and 15.

initialize $\vec{\theta}^0$, T, and $i \leftarrow 0$
repeat
  $i \leftarrow i + 1$
  E Step: compute $Q(\vec{\theta}; \vec{\theta^i})$
  M step: $\theta^{i+1} \rightarrow \arg\max_\theta Q(\vec{\theta}; \vec{\theta^i})$
until $Q(\vec{\theta^{i+1}}; \vec{\theta^i}) - Q(\vec{\theta^i}; \vec{\theta^{i-1}}) \leq T$
return $\hat{\vec{\theta}} \rightarrow \vec{\theta^{i+1}}$

3 Independent Component Analysis Family of Estimators

The Independent Component Analysis (ICA) family of estimators are unique regarding the items to be estimated. The standard ICA form, originally derived in [**?**, 151] , is shown in equation 22. The dataset denoted by $\vec{x}$ is the only observed part. The mixing matrix ($\mathbf{A}$), the ICs ($\vec{s}$), and the noise ($\vec{\nu}$) are all estimated. If $\vec{\nu}$ can be assumed to be normal, then it can be eliminated from the standard ICA form to give the noiseless ICA form, equation 23. The inverse of the noiseless form allows for estimators of ICA to be constructed.

$$\vec{x} = \mathbf{A}\vec{s} + \vec{\nu} \tag{22}$$

$$\vec{x} = \mathbf{A}\vec{s} \tag{23}$$

$$\vec{s} = \mathbf{A}^{-1}\vec{x} \tag{24}$$

Construction of an ICA estimator from equation 24 requires a few algebraic manipulations, which are shown in equations 25 through 28. $\mathbf{A}\vec{b}$ maximizes y's non-normal characteristics. Such a $\vec{b}$ makes $\vec{q}$ have only one non-zero component. Therefore, y is an independent component. Determining the non-normal characteristics is complicated by $\pm s_i$ which are the IC(s) determined up to a sign. The determination of $\vec{b}$ becomes an angle between vectors.

$$y = \vec{b^T}\vec{x} \tag{25}$$

$$y = \vec{b^T}\mathbf{A}\vec{s} \tag{26}$$

$$\vec{q} = \mathbf{A}^T\vec{b} \tag{27}$$

$$y = \vec{q^T}\vec{s} \tag{28}$$

One method for determining the non-normal characteristics is by kurtosis, shown in equation 29. If y is a whitened source of data, then equation 30 holds.

$$\kappa_4(y) = E\{y^4\} - 3(E\{y^2\})^2 \tag{29}$$

$$E\{y^2\} = 1 \Rightarrow \kappa_4(y) = E\{y^4\} - 3 \tag{30}$$

The measure of non-normal characteristics can be accomplished by $|\kappa_4(\dot{)}|$ or $\kappa_4(\dot{)}^2$. Kurtosis

has both additive and scaling properties, as shown in equations 31 and 32:

$$\kappa_4(x_1) + \kappa_4(x_2) \equiv k_4(x_1 + x_2) \tag{31}$$

$$\kappa_4(\alpha x) = \alpha \kappa_4(x) \tag{32}$$

The objective is to apply kurtosis to y (or to its definition as an IC element). Equation 33 shows the application of kurtosis to the characteristic, whitened equation of ICA. A consequence of the data being whitened, shown in equation 34, is a reduction of terms for the characteristic equation, by which the objective of finding each mixing vector translates to determining the unit circle of the maxima of $\kappa_4(y)$. In order to find the unit circle maxima, an assumption shown in equation 35, the kurtosis of each IC is restricted to unit length. Therefore, gradient methods can be used to determine the kurtosis.

$$\kappa_4(y) = \sum_i q_i \kappa_4(s_i) \tag{33}$$

$$E\{y^2\} = 1 \Rightarrow \sum_i q_i^2 = 1 \tag{34}$$

$$\kappa_4(s_i) = 1 \Rightarrow F(\vec{q}) = \sum_i q_i^4 \tag{35}$$

3.1 Kurtosis Gradient Maximization

Let $\vec{z}$ be $\vec{x}$ whitened and $\vec{w}$ be a whitened version of $\vec{b}$. Equations 37 through 40 show how to derive the extreme points via kurtosis. If the partial derivative in equation 40 is set to zero, then the determining the optimal $\vec{w}$ is reduced to determining the roots of $\kappa_4(y)$, which

are the maxima and minima. Note that $\vec{w}$ must be normalized in each iteration. Also, the expectation operator, $E(\cdot)$, in many cases, is treated as the classic mean of the argument inside. An algorithmic representation of Kurtosis Gradient Maximization is realized in Projection Pursuit show in Algorithm 3.

$$y = \vec{w}^T \vec{z} \tag{36}$$

$$\kappa_4(y) = \sum_i w_i \kappa_4 z_i \tag{37}$$

$$\frac{\partial \kappa_4(y)}{\partial \vec{w}} = 4\text{sign}(\kappa_4(y)) E\{\vec{z}(\vec{w}^T \vec{z})^3\} - 3\frac{\vec{w}}{||\vec{w}||} \tag{38}$$

$$\frac{\partial \kappa_4(y)}{\partial \vec{w}} \approx 4\text{sign}(\kappa_4(y)) E\{\vec{z}(\vec{w}^T \vec{z})\} \tag{39}$$

$$\frac{\partial \kappa_4(y)}{\partial \vec{w}} \to 0 \tag{40}$$

Algorithm 3 Projection Pursuit

Require: data source $\vec{x}$
Require: Dot Product Expectation
Require: Random Vector producer
  Center data to make its mean zero
  Whiten the data to provide $\vec{z}$
  $\vec{w} \leftarrow$ random vector
  Provide initial value for γ
  repeat
    $y = E\{\vec{w}^T \mathbf{Z}\}$
    $\Delta\vec{w} = E\{y^3 \mathbf{Z}\}$
    $\vec{w} += \Delta\vec{w}$
    $\vec{w} \leftarrow \frac{\vec{w}}{||\vec{w}||}$
  until $\Delta\vec{w} \to 0$
  return $\vec{w}$

3.2 Non-normality by Negentropy

One characteristic noted in [**?**, 94] is that normal variables have the largest entropy of all random variables. Appendix G.2.1 shows the theory of neg-entropy and its use in maximizing entropy. Two equations from the theory of neg-entropy, equations 93 through 94, can be used to approximate non-normality. If $\vec{y}$ is used in these equations, then the fact that $\vec{y}$ possesses properties of a zero mean and unit variance (whitened) causes the neg-entropy estimation to derive an estimation identical to the one via kurtosis as shown equation 41. A variation to Projection Pursuit is considered using non-polynomial functions and is motivated by neg-entropy.

$$J(y) \approx k_1(E\{G_1(y)\})^2 + k_2[(E\{G_2(y)\}) - (E\{G_2(v)\})]^2 \tag{41}$$

3.2.1 Negentropy Gradient Algorithm

A non-polynomial approximation is shown in equation 42. Assuming that $G(\cdot)$ is defined as in G_1 or G_2, then the application constraints in equation 45 through equation 47 lead to algorithms 4 and 5. Note that v in these equations is a standardized normal variable, of equal mean and covariance to the observed source $\vec{z}$.

$$J(y) \approx [(E\{G(y)\}) - (E\{G(v)\})] \tag{42}$$

$$G_1(y) = \frac{1}{a_1} \log \cosh a_1 y \tag{43}$$

$$G_2(y) = -\exp(\frac{-y^2}{2}) \tag{44}$$

$$E\{(\vec{w}^T\vec{z})^2\} \rightarrow ||\vec{w}||^2 = 1 \tag{45}$$

$$\Delta\vec{w} \propto \gamma E\{\vec{z}(\vec{w}^T\vec{z})\} \tag{46}$$

$$\vec{w} = \frac{\vec{w}}{||\vec{w}||}\gamma = E\{G(\vec{w}^T\vec{z}) - E\{G(v)\}\} \tag{47}$$

Algorithm 4 FastICA Stochastic Negentropy Projection Pursuit

Require: data source $\vec{x}$
  Center data to make its mean zero
  Whiten the data to provide $\vec{z}$
  $\vec{w} \leftarrow$ random vector
  Provide initial value for γ
  repeat
    $\Delta\vec{w} \propto \vec{z}g(\vec{w}^T\vec{z})$
    $\vec{w} += \Delta\vec{w}$
    $\vec{w} \leftarrow \frac{\vec{w}}{||\vec{w}||}$
    $\Delta\gamma \propto (G(\vec{w}^T\vec{z})E\{G(v)\}) - \gamma$
  until $\Delta\vec{w} \rightarrow 0$
  return W

Algorithm 5 FastICA Negentropy Projection Pursuit

Require: data source $\vec{x}$
  Center data to make its mean zero
  Whiten the data to provide $\vec{z}$
  $\vec{w} \leftarrow$ random vector
  repeat
    $\vec{w}_{old} \leftarrow \vec{w}$
    $\vec{w} \leftarrow E\{\vec{z}g(\vec{w}^T\vec{z}) - E\{g'(\vec{w}^T\vec{z})\}\}$
    $\vec{w} \leftarrow \frac{\vec{w}}{||\vec{w}||}$
    $\Delta\vec{w} = \vec{w} - \vec{w}_{old}$
  until $\Delta\vec{w} \rightarrow 0$
  return W

4 Sparse Coding Shrinkage

Images also have characteristic equations, such as equation 48. The goal of Sparse Coding Shrinkage is to estimate image mixing functions a_i and independent image components s_i by forcing the sparseness constraint. If the image values are treated as the $\vec{x}$ in ICA, and a_i and s_i are their ICA equivalents, then ICA can estimate this characteristic equation.

$$I(x,y) = \sum_{i=1}^{n} a_i(x,y)s_i \tag{48}$$

Sparse Coding Shrinkage represents basis vectors so that only a small number of them need to be activated at the same time. [**?**, 397]. Sparse Coding Shrinkage is good for compression and de-noising. In compression, Sparse Coding Shrinkage endeavors to find the rare samples and allow them to be coded differently from other more mundane samples. In de-noising, Sparse Coding Shrinkage provides the independent components, and a selection of interest determines the especially active components. It is the de-noising part that is the interest of this report.

From all accounts of Sparse Coding Shrinkage, it is an application of ICA such that the local mean component is removed and the local variance is normalized. Furthermore, PCA is applied to the data. Sparse Coding Shrinkage does not specify that the procedure be restricted in the arrangement of data obtained from the mixing matrix.

In [**?**, 391-400], Hyv*ä*rien uses a neighborhood patch scheme, sampling $l \times l$ sections

and arranging their values as one vector. The complete set of samples composes the source matrix, and the mixing matrix with their independent components may be estimated. Some of Hyvärien's experiments used a sliding neighborhood window rather than discrete, evenly distributed neighborhoods.

4.1 Insert and Remove Noise on Patch Oriented Sparse Coding Shrinkage

Algorithm 6 Insert and Remove Noise on Neighborhood Oriented Sparse Coding Shrinkage

Require: Image source I
Establish neighborhood matrix
Insert a random sample of neighborhoods column wise into data source $\mathcal{X}_p$
Estimate $\tilde{\mathbf{W}}$ for $\tilde{\mathbf{S}} = \tilde{\mathbf{W}}^T \mathcal{X}_p$
Insert noise
Construct $\hat{\mathbf{S}}$ using $\tilde{\mathbf{W}}$ s.t. $\hat{\mathbf{S}} = \tilde{\mathbf{W}}^T \mathcal{X}$
Zero out whole rows of $\hat{\mathbf{S}}$ and construct $\hat{\mathbf{I}}$ with it.
Show I, noisy I and $\hat{\mathbf{I}}$.

Algorithm 7 was built for both the original Distributed Computing Group (DCG)-Renaissance frameworks and also the DC-Quartz-Renaissance frameworks. The concept of using smaller matrices, suggested by this algorithm, provides a glimpse into another useful method in the GPU-bound implementations.

The reason for using such a construction of neighborhoods is that a whole image may be too large for standard PCA and/or ICA to determine an accurate mixing image. Hyvärien showed in [**?**, 260-271] that it is not practical or necessary to determine a mixing matrix for the entire sample; rather an estimate will suffice in practical cases. This practice is followed

Algorithm 7 Neighborhood Oriented Source De-constructor

Require: Image source I_m
  M is the number of rows in I_m
  N is the number of columns in I_m
  T is the total number of patches
  M_p is the number of row patches. $M_p = \text{ceiling}(M/l)$
  N_p is the number of column patches. $N_p = \text{ceiling}(N/l)$
  $T = M_p N_p$
  for each $\vec{h}_i \in \mathbf{H}$ **do**
    for $x \in L(\vec{h}_i)$ **do**
      for $y \in L(\vec{h}_i)$ **do**
        $\vec{h}_i(x\%l + y) = I_m(x, y)$
      end for
    end for
  end for
  return H

in the CPU-bound implementation of both ICA and PCA prepared for this report.

4.2 Connection between Independent Component Analysis and Sparse Code Shrinkage

The constraints on Sparse Code Shrinkage are included in the following list. Accompanying these requirements are comments that show the connection of Sparse Code Shrinkage to ICA.

1. First, using a noise-free training set of $\vec{x}$, use some sparse coding method for determining the orthogonal matrix $\mathbf{W}$ so that the components s_i in $\vec{s} = \mathbf{W}\vec{x}$ and so that $\vec{s}$ has as sparse a distribution as possible. Estimate a density model $p_i(s_i)$ for each sparse component.

- Note that if $\vec{x}$ is already whitened, then the ICA mixing matrix is a transformation matrix satisfying the constraints on $\mathbf{W}$.
- The exponential and Laplace density functions are specific components to the non-polynomial projection pursuit family.

2. Invert the relation $\vec{s} = \mathbf{W}\vec{x}$ to obtain estimates of the noise-free $\mathbf{x}$, given by $\hat{\mathbf{x}}(t) = \mathbf{W}^T\hat{\mathbf{s}}(t)$.

The matrices $\mathbf{W}$ and $\mathbf{A}$ are mixing matrices. Under the constraint that both $\vec{s}$ and $\vec{x}$ are zero mean and unit variant, then $\mathbf{W} = \mathbf{A}^{-1} = \mathbf{A}^T$.

5 Mathematica implementation of Multiple Discriminant Analysis

The Mathematica version is a straight-forward implementation and possess a copy of the data set supplied via a worksheet. The structure allows for a classifier to be constructed on the assumption that the data is Gaussian. A listing of the Mathematica implementation to solve the Iris data-set is included in Appendix A.

The variables setosa, viriginica, and versicolor contain the original measurements. Each observation is defined in a row matrix, and is named barSetosaArray. This array is used to generate $\vec{x_i} - \mu$ for row vector $\vec{x_i}$ in the setosa collection, which is misnnamed normSetosa. This allows the mean to construct the scatter matrix for setosa.

The same thing was done for versicolor and virginica. As stated in the preliminary sub-section on MDA (subsection 2.2) $\mathbf{S_w} = \sum_i S_i$ where S_i are the scatter matrices for each of the classes. The background set is obtained by combining the flowers using the join method, and obtaining the mean of the combined whole. From there, the mean array concept is used to reduce the action of finding the background scatter to matrix addition, scalar multiplication, matrix multiplication, and a transpose. Thus, mapping the original data source via the $\mathbf{W}$ obtained is simply a matter of matrix multiplication. The results of equation 49 are shown in Figure 1.

This implementation demonstrates some characteristics inherent to MDA that make its implementation for practical production image processing systems. First and for most is the need to identify each member of the classification prediction set. These elements construct the scatter matrices for each classification. For classical data analysis, this is practical as observations are supplied as collection of measurements. Therefore each measurement set can be collected into a vector, and each vector used to construct the scatter matrix pertaining to its classification. In image analysis, scatter matrices imply that portions of the image comprise a similar observation vector. One such concept is presented in the patch world section, but any arrangement of the elements of an observed image may be used to perform such a set of analyses. Another characteristic, defined in equation 49 where S_B is the background scatter matrix, is that the combined points belonging to the classifications and points not belonging to them at all comprise the background data-set used to construct the background scatter matrix, as is the case in PCA. This second characteristic is the easier

to accommodate, generally. What happens the background is selected at random, and the within scatter matrix is selected notably fewer from known points? How do these data-sets get selected. It is one thing to gather the points and encode them into a MDA algorithm. For a practical system it does require a sufficient interface so that users need not rewrite the system just to use it.

$$\mathbf{S_w}\vec{e_i} = \lambda_i \mathbf{S_B}\vec{e_i} \tag{49}$$

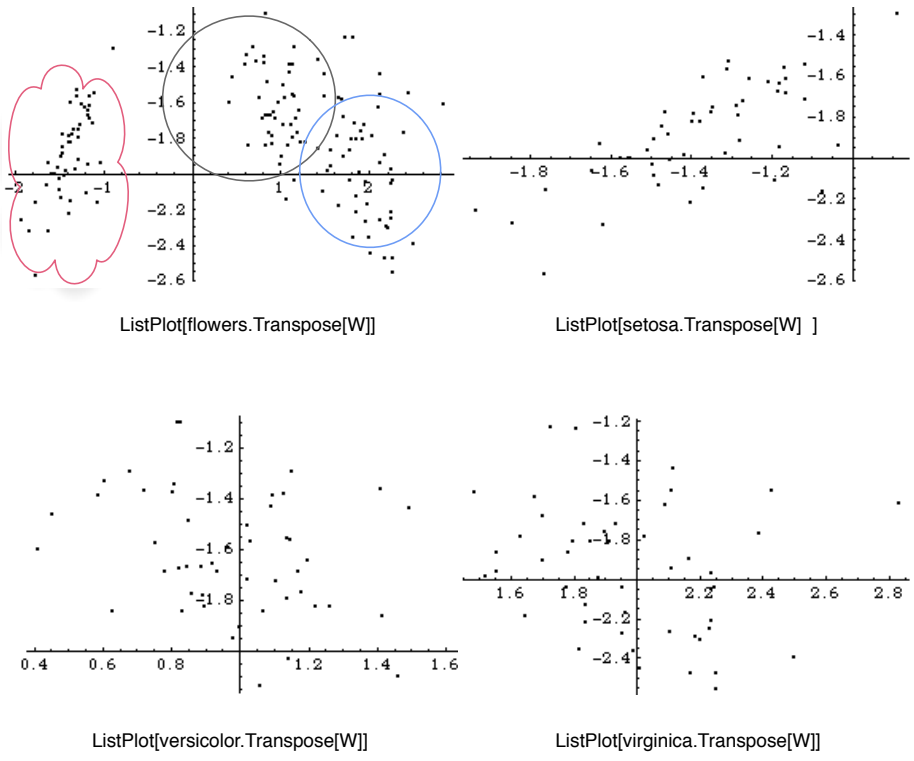

Figure 1: Mathematica Plots using a manual MDA

6 Evolution of CPU Bound Solutions

6.1 Octave and Core Graphics Implementation of Expected Maximization

The first experiment conducted for this paper implements a product using Octave (an open source product similar to Matlab). In this example, there is an original image of a human optic disc. From this image, a histogram of normal distribution functions is computed to assist in determining a model, shown in equation 50. The results are seen in Figures 2 and 3. Significant in this implementation of the EM segmentation is the use of histograms as opposed to the generation of a structure with normal distribution structure for the sample data.

$$p(x|\vec{\theta}) = \sum_i p_i(x|\vec{\theta}_i) \tag{50}$$

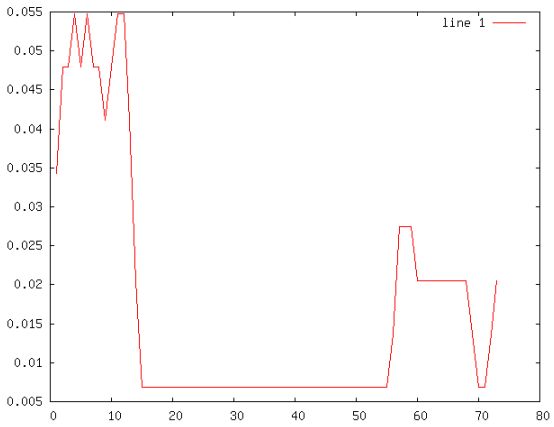

Figure 2: This figure is a plot of Guassian sums for the optic disc.

The Octave prototype supplied some insight that led to the designing of an Objective-

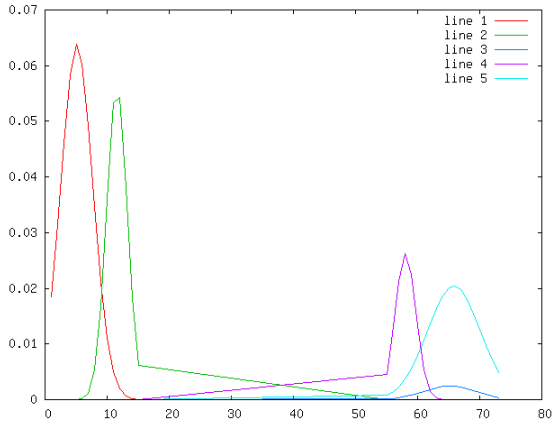

Figure 3: This is a plot of sub-Gaussian components for the optic disc.

C version. During the development of the Objective-C with Core Graphics, unit tests were used to verify each component developed. Each of these tests used a small data set. Once the data set became large, rapid allocation and deallocation becomes a constraint on performance.

6.1.1 Core Graphics Version

All of the EM implementations generate a set of statistical classes defined by a set of sufficient statistics. Each collection provides a method of classifying every subsequent element. In the case of image processing, these classifications represent a collection of masks. These masks identify segments to either be retained or eliminated.

A consequence of this need is that the interface consists of an additional panel. The panel contains a slider to control the number of normal distributions constructing the estimate of the image. The original design is shown in Figure 4. In the final model

demonstrated in figure 5, a table inside the panel shows a selection of specific distributions and displays them by mean and variance. At the time this report was written, only one distribution could be selected at a time. The reason for the limitation was to allow the mask to be produced and rendered in reasonable time. Selecting a component causes a mask to to be constructed for the univariate component chosen. Changing the number of distributions causes the distributions estimations to be computed via EM.

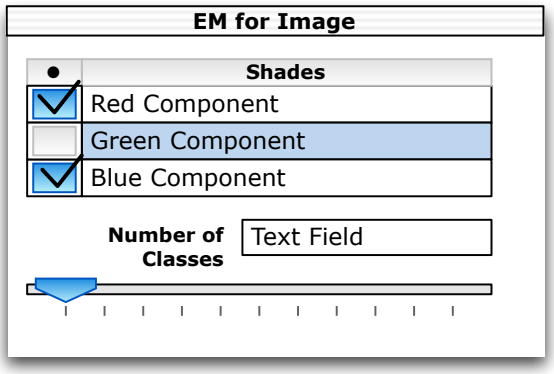

Figure 4: The results of an "EM for Image" are a set of segmenting classifiers and the masks that result from the classification

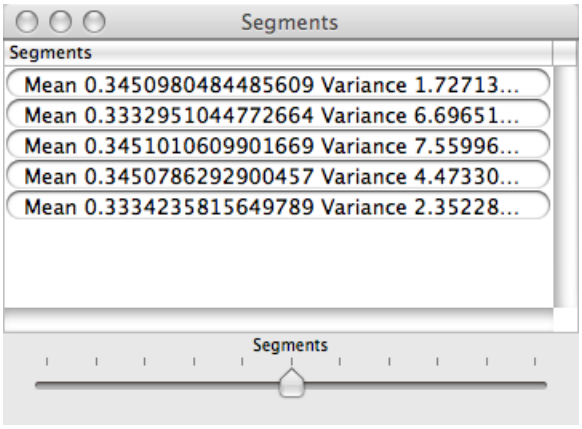

Figure 5: EM Segments shown by mean and variance for a photo of a human iris.

A mask has the sole purpose of blending sections of the image. In this case, either the color point belongs, or it does not. It is in constructing these masks that Core Graphics Data Providers and Image Providers become useful. Also, in this construction DCGMatrix, obtained from the Grey-Scale Bitmap, is reused and fed into the classifier. If the point belongs, a white value is inserted into the mask. The mask is black otherwise. Note that a true color filter would use a multivariate variety of the classifier, as it is a vector (color vector). Using this technique, the need to construct a new form of the image itself is eliminated as the mask can be fed into a Core Image filter with the original image for display and rendering.

6.1.2 First Successful GUI Model

The first successful version used a subclass of NSOpenGLView and model controller to test the concept of an EM filter. A better version should move the computation to a CIFilter and use threads to move the computation away from the main display thread. Otherwise, the intensity of the computation can overwhelm the main thread and waste the existence of multiple-core CPUs and computational grids. With the first version, the controller includes a panel to show the segments (displayed by mean and variance shown in Figure 5) and view window that shows the image masked with the selected segment. If no segment is selected, the mask is clear and shows the original image, as shown in Figure 6. There is the possibility of computing the EM directly from the Graphics Card, but that implies providing an expected step kernel, a maximization step kernel, and a termination

test kernel.

It is this hybrid Core Image - Core Graphics approach introduced in earlier versions of this report that offers the ability use classic CPU oriented computation with GPU oriented filters. Core Image is itself a GPU bound computation mechanism for computing powerful filters, with a skeleton running CPU bound to provide control. Mask blending is a rudimentary GPU bound practice, and does not need to be computed CPU bound. This technique does save time, enough to make the difference between a responsive GUI based application and an unresponsive one. Another approach that is even more efficient is to use the mean and variance parameters as control methods to a Core Image filter designed specifically for such classifications. This causes the CPU bound computation to be limited to computing the summed normal model itself. But that implementation is scheduled for another report.

6.1.3 Design Details and Considerations

The two steps for EM are explicitly stated in listing **??** . The estimation step is computed implicitlybefore the loop and again at the end of the loop. The reason for computing the estimation step is to determine the estimation termination value. Another issue of implementation, revealed in listing 3, is the need for careful zeroing of summing values. The previous mean is needed in computing the maximization steps variance. Therefore, careful application of order ensures maximum efficiency, for CPU bound computations, of both expectation and maximization steps.

The univariate case has the possibility that the EM algorithm will require many iterations. One design decision is whether to retain the results of each iteration of the EM algorithm. In some cases, these steps may be discarded. Others they must be retained. In this implementation, the results of the maximization and estimation steps are recycled in each subsequent iteration. The presumption is that the sufficient statistics computed in both the last and next to last maximization step have in fact converged. Convergence of the result is measured by the values of the estimation function computed in each estimation step. While it is a waste to compute the maximization upon convergence, it is also harmless.

Another presumption needed in any implementation of the EM algorithm is initial values. These initial values are produced upon initializing the EM object. The object containing the EM algorithm is initialized with the samples to build the estimator and the number of distributions in the summation equation. In listing 1, mean, variance, and proportions are each of the class dcgVector, which is an Objective-C wrapper around a C-array with helper methods typical for vector operations. The mixture member is a dcgMatrix, which is a wrapper around a C two dimensional array with methods suitable for matrix operations. Other components are needed in the Objective-C version to make it more viable, and this is where design decisions must be considered. In the initial design, obtaining the image data itself was a matter of drawing to a Core Graphics Grey-Scale Bitmap and copying the data out. This forces the computation to be CPU bound. Either CPU bound or Graphics Processing Unit (GPU) bound versions are acceptable. This

subsection illustrates the CPU bound version. A future report may review the GPU bound equivalent.

In order to continue with a DCGMatrix obtained from a Core Graphics Grey-Scale Bitmap, it must be transformed into a vector of equal number of elements. This vector and the number of desired classifications must then initialize an object to compute the EM Univariate Normal. Once initialized, it is convenient to extract the results into an NSArray containing the sufficient statistics. Such an array need not retain the proportions value.

It should be noted that obtaining the DCGMatrix from the Core Graphics Grey Scale Bitmap is quite tricky. It is one section that is anything but universal between PowerPC (PPC) and Intel x86 architectures. Care has to be taken to ensure byte ordering on the bitmap. Otherwise, the results are garbage, and the EM will compute nonsense.

6.2 Principal Component Anaylsis

In the first steps of constructing the PCA segment filter, two methods were considered, SVD and EVD, as many texts reference SVD as an efficient means of determining PCA. SVD is available in LAPACK and software packages that use it. Building a filter for segmentation using SVD is another matter. It is more intuitive to generate the mixing matrix for PCA, transform the data with it, filter the transformed data, and use the transpose of the mixing matrix to obtain the results. Also, this straight-forward method of constructing PCA yields as byproducts the matrices needed to constitute a whitening agent to be used with Projection Pursuit.

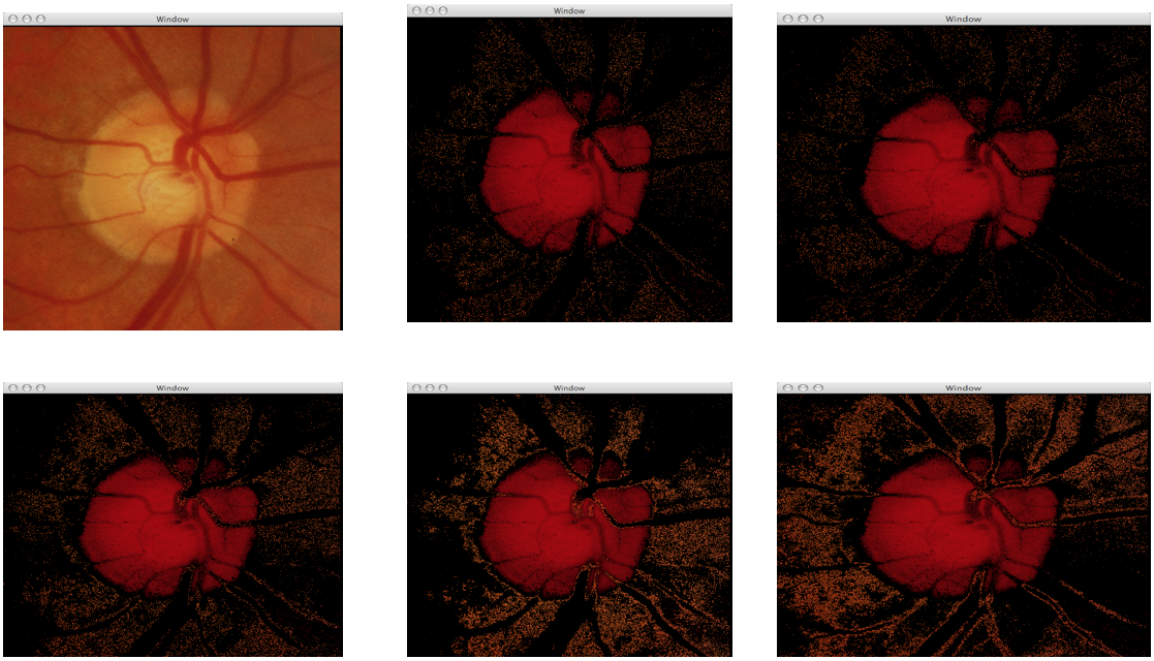

Figure 6: Both the original image and 5 EM Segments of a human iris.

Also, using either EVD or SVD from LAPACK has debugging side-effects that tend to be difficult to trace. Both decompositions implemented in LAPACK have an advantage in the fact they have optimized implementations such as the Accelerate framework (as called in OSX 10.3 to 10.5). Thus, there is a trade-off between the ability to debug and determine which input to these functions was in error, versus speed of actual execution due to the vector pipeline acceleration.

Observations made during the first set of tests of the EVD and SVD implementations using Quartz Composer plug-ins conducted for this report show the operations themselves to be efficient, but in order to be effective the multiplication elements have to use the C version of the Basic Linear Algebra Sub-routine (C-BLAS) implementation of matrix

multiply (cblas_dgemm in the Vector Library framework which is part of the Accelerate framework on OSX). In these cases the maximum size of the neighborhood still can not go above six by six pixels. Without the C-BLAS version of matrix multiplication the maximum pixel patch size is five by five. The advantage of not using the C-BLAS version is that mistakes are easier to find and remove.

Size of the data-set in tests conducted using Quartz-Composer as the facilitating tool have also been limited due to the time required to perform computations on the data. If computation time exceeds certain limits, then the performance of the Quartz Composer editor becomes unstable and unusable. Images in excess of 256 by 256 pixels are therefore scaled to a size within this limit. While stable, Quartz Composer is an excellent tool for observing how each component works in constructing principal components. Such observations clarify subtle changes to the filter, which include the number of neighborhoods, which eigenvectors to use during reconstruction, and the appearance of the principal components themselves. All of these observations were intuitively available within a few mouse clicks.

6.2.1 Unit Test of PCA

The experiment requested for this project is significantly more complex than what a unit test should test. Any unit test should have a simple input, a simple output, and a known comparison for the output. An example of this would be, say, a matrix shown in the following equation:

$$\mathbf{A} = \left\{ \begin{array}{cccc} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{array} \right\} \tag{51}$$

$$\mathbf{U} = \left\{ \begin{array}{cccc} -0.13 & 0.82 & -0.54 & 0.03 \\ -0.34 & 0.42 & 0.75 & 0.36 \\ -0.54 & 0.03 & 0.13 & -0.8 \\ -0.75 & -0.36 & -0.341 & 0.42 \end{array} \right\} \tag{52}$$

$$\mathbf{\Lambda} \left\{ \begin{array}{cccc} 38 & 0. & 0. & 0. \\ 0. & 2.07 & 0. & 0. \\ 0. & 0. & 0. & 0. \\ 0. & 0. & 0. & 0. \end{array} \right\} \tag{53}$$

$$\mathbf{V}^T = \left\{ \begin{array}{cccc} -0.42 & -0.71 & -0.16 & -0.52 \\ -0.47 & -0.27 & 0.60 & 0.57 \\ -0.52 & 0.17 & -0.72 & 0.41 \\ -0.56 & 0.61 & 0.28 & -0.47 \end{array} \right\} \tag{54}$$

$$\text{SVD}(\mathbf{A}) = U \Lambda V^T \tag{55}$$

$$V = \text{eigenvectos}(\mathbf{A}\mathbf{A}^T) \tag{56}$$

$$\Lambda = \text{eigenvalues}(\mathbf{A}\mathbf{A}^T) \tag{57}$$

The values for $\mathbf{U}$, $\mathbf{\Lambda}$, and $\mathbf{V}$ are computed using Mathematica's readily available SVD method. A comparison between this and the unit test can be valuable.

6.2.2 A Quartz Composer Model - CPU Bound

There are methods to using Quartz Composer Plug-ins which allow them to rapidly prototype a PCA filter (or other filters of such consideration). The original intention was to use the Quartz Composer plug-in model to construct a GPU-bound solution to PCA, and that solution is still under consideration. Even in a CPU-bound solution, Quartz Composer is useful for determining which handlers belong and how each feature contributes.

As a multi-threaded environment, explicitly constructing new threads to explore and compute parts of a model is unnecessary as this is taken care of by the Quartz Composer development tool. One detriment to this approach is that Quartz Composer requires each patch be responsive. A patch is a computational unit which is either a plug-in or system defined patch and consists of inputs, outputs, and specifications as to when to execute. A few overhead plug-in patches were constructed for the handlers constituting the PCA model. The composition, a collection of patches that constitute a graphical program, was constructed for this exercise as shown in Figure 11. The two images used to test this composition were the human iris image, used in the EM section (6.1), and a photo of Noe Lopez-Benitez from the Computer Science Department (Texas Tech University) web site. Noe's photo was chosen for the size of the photo, which reduced the size of the samples and thus allowed unscaled tests of the composition. Due to the size of the human iris photo, a

scaling patch was inserted to maintain the 256 by 256 limit.

Another design consideration needed to comply with the instructions for the project was how the classifier worked on the principal components themselves. The scatter matrix is defined in equations 2. This scatter matrix is determined by the mean matrix and correlation patches. Covariance is a special case of correlation known as zero-mean correlation, and in this instance is convenient to keep in its more general form. One improvement to be made to the method used in constructing the correlation patch could have been to use the Basic Linear Algebra Subroutines (BLAS) implementation of matrix multiply to implement the patch (specifically the DGEMM included in the BLAS dense library). With standard matrix multiplication, the patch does tend to consume excess computation time which takes away from the amount of computation that can be done responsively.

The EVD patch has a handler object that computes the EVD via the LAPACK implementation. The two outputs are the supplied matrix eigenvalues and eigenvectors. Because the patch feeding the EVD patch is the correlation patch, the resulting eigenvectors are the mixing matrix for determining the principal components defined in equation 8. The subsequent patches are used to multiply the mixing matrix and the neighborhood samples to obtain the principal components. At this point, any common data-oriented filtering technique may be applied. In the first demonstration, a simple one-vector selection was applied. This had an unintended result. Figure 11 shows sections of the image removed, which is consistent with the assumption that the component constitutes the classification. Such a technique causes the image to look as though it had been struck with a barrage of

shotgun pellets.

This method does provide insight into a GPU-bound version. It is anticipated that size of the image will not be as much of a problem in a GPU-bound version as it is in the CPU equivalent. Another useful insight gained by this implementation is that some of the filters, which would otherwise require many iterations of construction, can be defined. A result of this insight is the possibility that a fixed size filter can be constructed once and used many times for this type of filtering in the GPU-bound version.

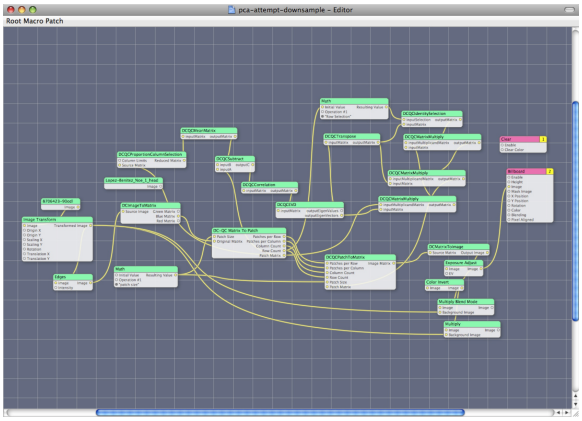

Figure 7: A Quartz Composer patch for computing PCA via EVD.

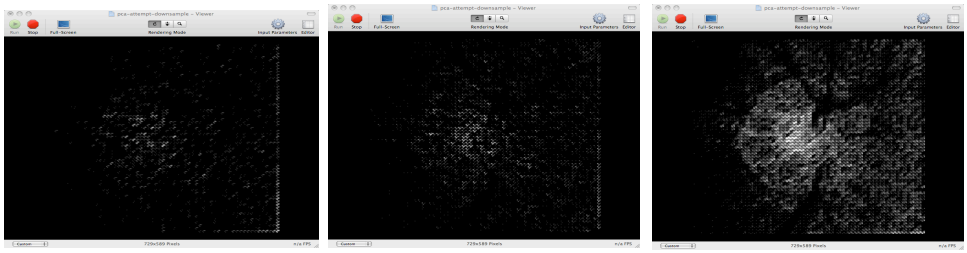

Figure 8: This is a set of results of a Quartz Composition PCA-Downsample of PCA via EVD. In this sample, a few of the upper half principal components are shown.

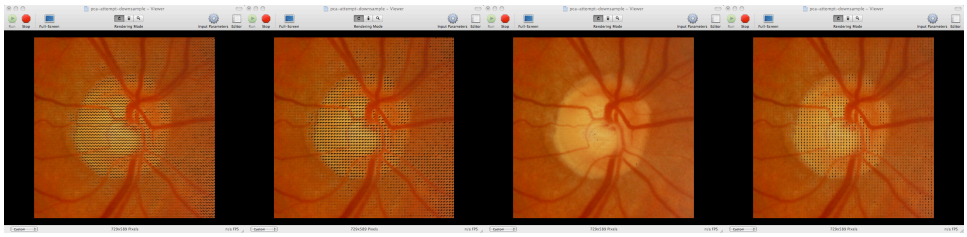

Figure 9: This is a set of results of a Quartz Composition PCA-Downsample of PCA via EVD. In this sample, the green channel is used to provide the filter, and an enhancement is made on the filter elements themselves.

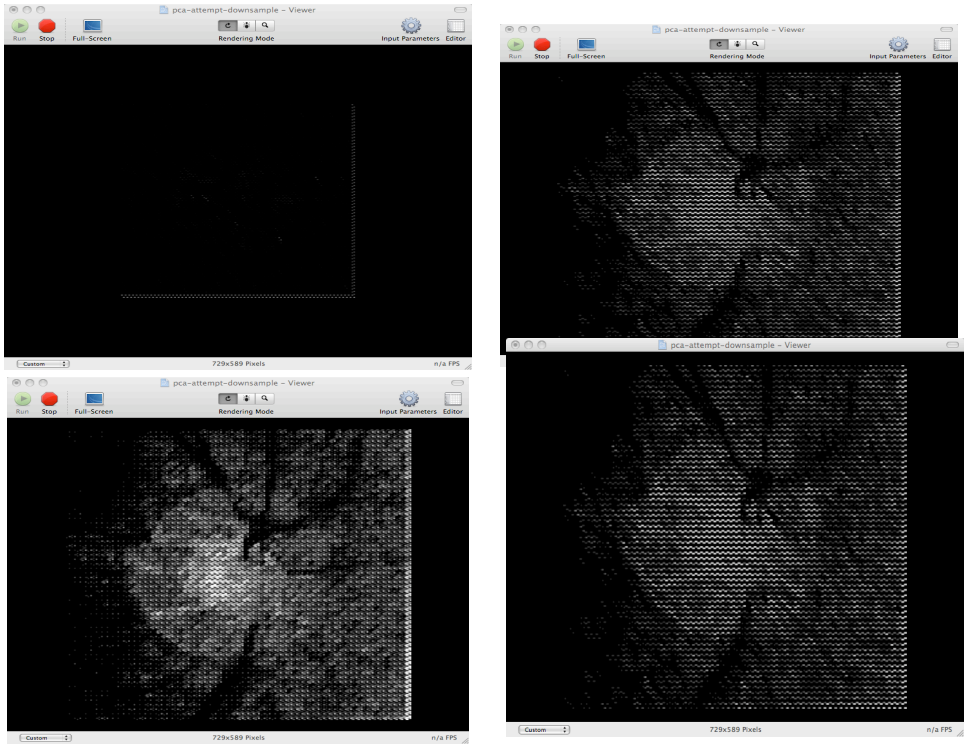

Figure 10: This is a set of results of a Quartz Composition PCA-Downsample of PCA via EVD. In this sample, the red channel is used with an enhancement. These elements come from the lower principal components.

6.3 Filters using ICA and PCA

Adapting a PCA filter to become an ICA filter is specified in the Algorithm 6. All of the steps of obtaining the neighborhoods and ensuring the data are whitened, or relatively whitened, are part of the mathematical definition of PCA. Therefore, a projection pursuit patch can be inserted after the principal component stage. Further, as stated, the transpose of the PCA mixing matrix is included with the ICA mixing matrix to obey the following equation:

$$\mathbf{X} = \mathbf{PAS} + \mathbf{M} \tag{58}$$

where P is the PCA mixing matrix and $\mathbf{A} = \mathbf{W}^T$ is the ICA mixing matrix. Because projection pursuit is used in this case to determine the Sparse Coding Shrinkage, the size of $\mathbf{W}$ is $l^2 \times 1$, where l is the size of the observation neighborhood vector. One observation from applying the ICA filter to the human irisis that it has the effect of sharpening certain pixels within each neighborhood. A neighborhood size of 2 by 2 tends to have the fewest side-effects from excessive information. The 4 by 4 neighborhood size shows signs of excessive information loss.

7 A Core Image Innovation

As mentioned in the mathematical background section, there is another way to compute the mixing matrix for determining principal components, namely Gram-Schmidt Orthogonalization. In CPU form, the iterative nature of EVD and SVD.

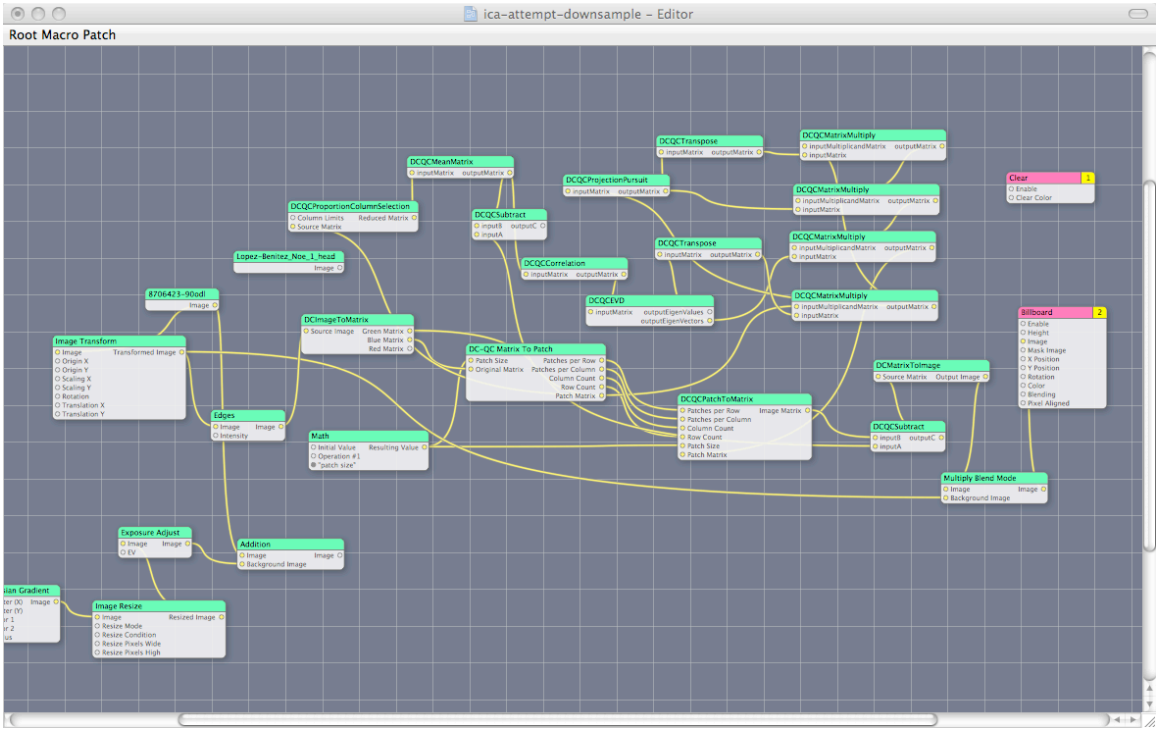

Figure 11: A Quartz Composer patch for computing PCA via EVD.

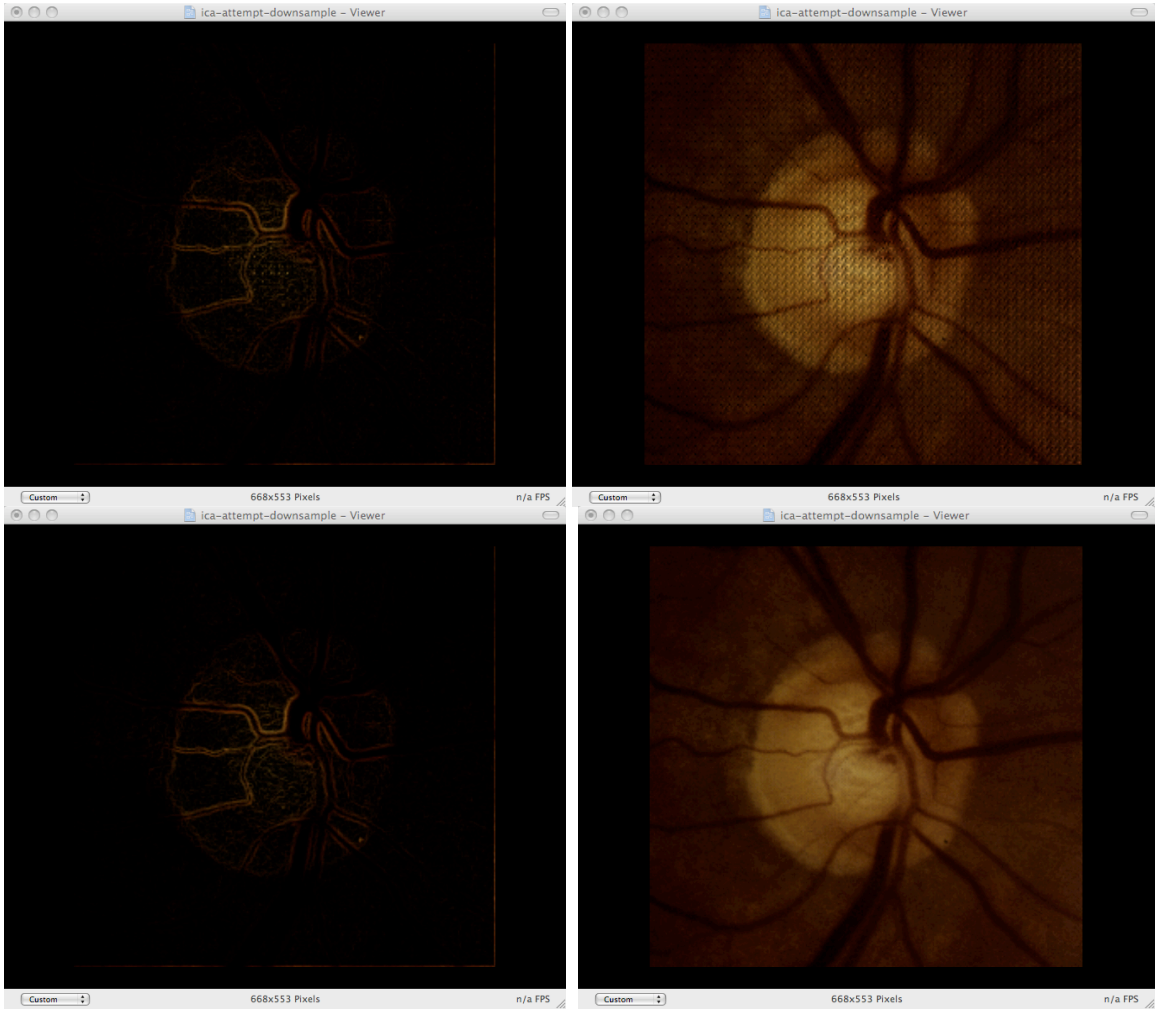

Figure 12: This is a set of results of a Quartz Composition of a ICA-Downsample. In this sample, the red channel is the primary source for the PCA-ICA estimator. On the left, an edge filter is applied before the image is channelled into the PCA-ICA estimator. The results are multiply blended with the original image. On the right, the original image is multiply blended with the PCA-ICA estimation.

7.1 Pseudo Zero Branch Program Model

Turing's theorem inherently requires that a computational models include branches or more precisely have a form of recursion. In the simplest form, recursion is defined in terms of multiple stacks. The concept of recursion allows for a finite number of instructions to be repeated many times to accomplish an otherwise complex result. Models for recursion were developed that specify that any construct conforming to a safe recursion model will in fact complete its task.

Vector engines have trouble with branches. Trouble occurs in the construction of such engines and is rooted in the key engine mechanism, which is the pipeline. Pipelines are designed with the same philosophy as a commonly known concept called the assembly line. Each stage of the pipeline carries out a portion of the work, and each stage works simultaneously. The consequence of this model is that one result is produced per stage execution (after the pipeline is filled). Many CPU models have this concept, and there are compilers for these models that optimize programs for these pipelines. Because CPU models are general purpose machines there are limits of how large pipelines can be while satisfying the branch requirement.

A GPU favors the pipeline model over branches by necessity. In order to provide the billions of operations per second necessary for responsive renderings, a GPU must include pipelines optimized for determining each two and three dimensional points. In these cases, a program that has no branches of any kind is called zero-branch.

There is a hybrid approach between CPU and GPU models that can generate modules which are zero-branch. The hybrid uses modules which are themselves zero-branch and are optimized for the GPU. The CPU furnishes the branches necessary to combine each module into a structure satisfying the desired calculation. The result of this hybrid approach is that the CPU builds zero-branch programs to be executed on the GPU. Optimization of this hybrid approach becomes a function of well chosen modules, the reuse of constructed modules, and a parallel construction of reused modules.

7.2 Core Image Kernel Model

The Core Image (CI) Kernel Model (CIKM) satisfies the requirement for modules that are themselves zero-branch programs. A Core Image Kernel is a zero-branch program which is defined in terms of each pixel in a defined region of interest (ROI). The language used to construct Core Image Kernels, Core Image Kernel Language (CIKL), is a subset of Silicon Graphic's OpenGL's Shading Language (GLSL), which is built on a similar philosophy. CIKL restricts branches to those which can be determined at compile time. The output of each kernel is an image, and the input types include images, scalar numbers and four element vectors.

A ROI is specific set of points (pixels) by which a kernel may operate on. There may be multiple ROI, one for the destination image and one for each input image. A CI Filter object provides a region of interest which may be overwritten to specify a ROI for each input image. The ROI for the destination is specified on the call of the CI Kernel itself,

which by convention of the construction of CI Filters is invoked inside of the output image method.

CIKL includes a compare instruction. This compare instruction is not a branch instruction. It simply provides one of the two provided values based on the less than operation included in the instruction. The compare operation compares, in the scalar case, an input denoted x, and returns either the second input, y, if $x < 0$ and the third input, z, otherwise. The vector version performs the selection on each element of the vectors $\vec{x}, \vec{y}$, and $\vec{z}$ respectively.

Each kernel function is defined in terms of each of the input vectors, scalars, image samplers, and destination pixel, whose coordinate is given by the destination coordinate function (destCoord()). Any of point in an input image sampler may be accessed through the "sample" function. It is typical to define a kernel which isolates the specific point in the input samplers and performs some fundamental mathematical operation on these points. Mathematical operations include scalar and dot vector versions of trigonometric operations (cosine and sine), addition subtract, multiply, division, and modulus.

No destination pixel can be reused as an input pixel for the kernel itself. This would force an order dependency which is not supported in the CIKM. In order to perform such operations, one kernel may supply its output image as an input image to another, as shown in sub-section 7.3. Each of the points are determined in parallel via the vector pipelines, which are specified in the GLSL specification. The kernel then becomes a pipelined structure, and provides the efficiency of the CIKM.

7.3 Constructing Where Size and Position Matters

As stated the results of a CI Kernel can be used as an input to another CI Kernel. The CI Filter object encapsulates each respective CI Kernel. In principal, any loop can be used to insert images, both original and outputs of CI Filters, into more CI Filters. This process can encapsulate and define complex filters using what is called a CI Filter Generator. A CI Filter Generator provides a connect object method to connect one CI Filter, number or vector to another CI Filter, and form a structure of these connections. The structure itself is capable of forming an archive which can be saved and passed on to other processes.

This structure can also be used to construct another CI Filter. Such a CI Filter is the compilation of the CI Filter Generator's member filters. The definition of the CI Filter is determined by the way each member filter is connected. The resulting filter is a stored object which the CIKL compiler constructs into a concise filter. The resulting filter constructed by a CI Filter Generator can be used by another CI Filter Generator.

Thus a CPU-bound loop can assemble any complex CI Filter through a CI Filter Generator instance from less complex filters. It is typical to define the inner loop of a particular process as a CI Filter (constructed by a CI Filter Generator instance), then use a CPU bound loop to connect each inner loop filter to each other. The just-in-time compiler determines the dependencies and how many pipelines are needed to perform the computation optimally.

There are special items in addition to the connecting filters that define a CI Filter

Generator's output filter. Each CI Filter Generator supplies a dictionary defining the name, description and other special attributes of the constructed filter. Also, each CI Filter Generator supplies export keys to define the input ports and output image of the resulting CI Filter.

7.4 GSO version of PCA using the Core Image Kernel Model

The CIKM solution for determining PCA is solved by solving GSO through a CIKM model. This solution requires eight filters. Of the eight filters, three are kernels supplied by the DC lab, six are generated as a combination of the three kernels, and three of the six can be computed in parallel without difficulty.

In the process of generating a CIKM solution, the orientation of the solution is kept in neighborhood form, which alters the orientation of the operations needed to produce the GSO super-kernel. The add image and dot multiply stay the same, as there are no cross products in either of these kernels. The computations of the norms, inner products, column scales, and column subtractions take on a new form.

7.4.1 Neighborhood Orientation to Linear Algebra

The mathematics associated with this neighborhood orientation is called, in this report, neighborhood algebra. A neighborhood is treated as analogous to a column vector in standard linear algebra. A neighborhood is a $k \times k$ image extracted from the image itself. Neighborhoods are arranged horizontally forming a $k \times kN$ image where N is the quantity

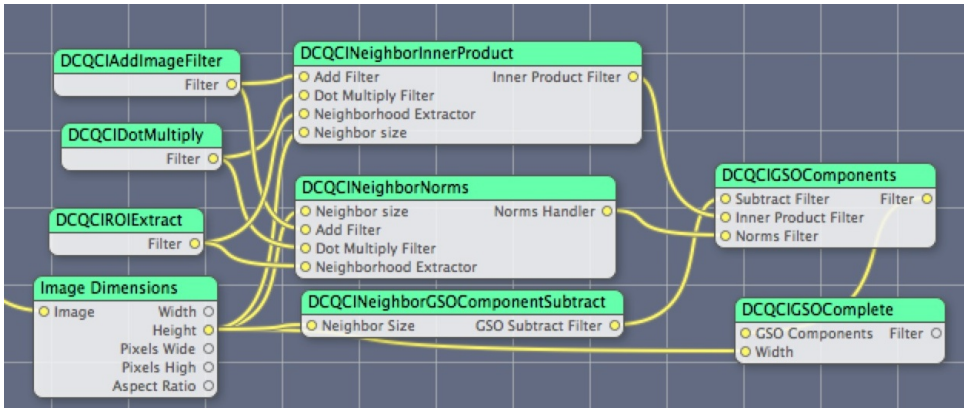

Figure 13: These are the core filters required for GSO. They are computing in this case by a pseudo-algebra called in this report neighborhood algebra.

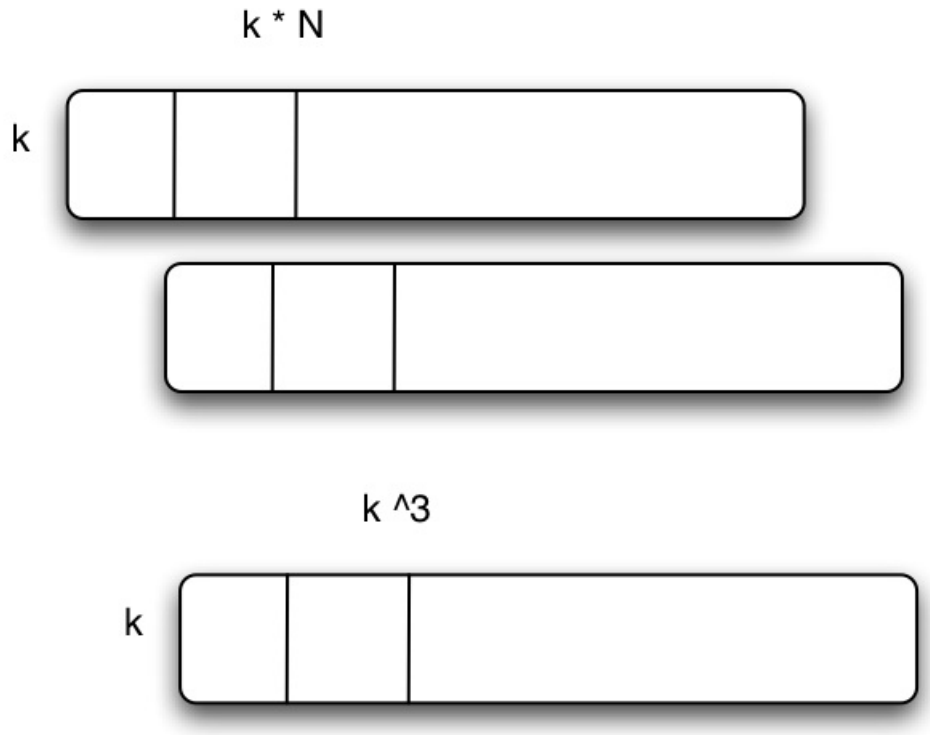

Figure 14: This diagram shows the arrangement of neighborhoods in a neighborhood mixing image.

of neighborhoods, which is shown in Figure 14. A third kernel filter produced for this report, which crops and translocates specific neighborhoods, is ROI extract, which copies out a region of interest as the name implies. This neighborhood orientation has advantages in image processing operations. Dot multiply and addition are simpler operations than full matrix multiply, which is an example of the motivation for this neighborhood orientation.

GSO is an example of a linear algebra operation that benefits from neighbor orientation, and becomes a means of achieving parallelism in the construction of the CIKM generated filter for GSO. The generated filters are shown in Figure 13. The inner product, norms, and neighborhood scale with subtract (GSOComponentSubtract in the figure) handlers can each be determined independent of each other. Their generator connection diagrams are shown in Figures 17, 18, and 19.

7.4.2 Reductions resulting from "Neighborhood Algebra"

Inner products computed in these cases reduces to one dot multiply between two neighborhoods, followed by k sums. Norms of the neighbors is one dot multiply, and k sums of the elements inside each neighborhood. The scale and subtract is even more straight forward. The scaling operation dot divides the inner product image by the norm image. The resulting image is used to scale each neighborhood via the neighborhood scale kernel filter. The final subtract filter subtracts a selected neighborhood from the working neighborhood and repeats this process for each neighborhood in the mixing image. The caveat of the scale and subtract filter is that the neighborhoods in the scaled mixing image from the working

neighborhood to the right are zeroed. This allows for the scale and subtract to be general for each inner loop.

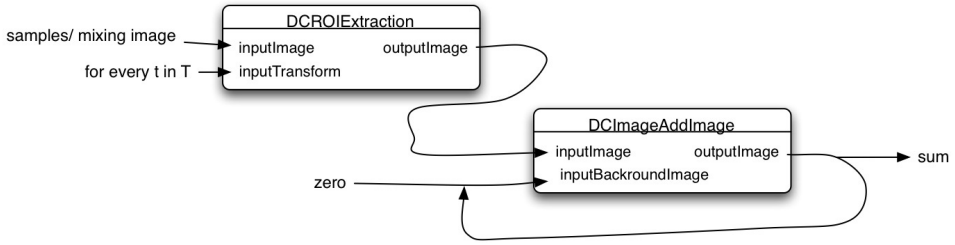

Figure 15: This diagram shows the connection of fundamental kernels to form a neighborhood sum kernel.

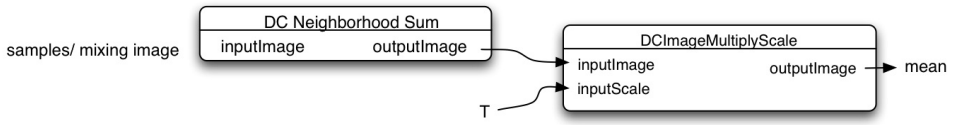

Figure 16: This diagram shows the connection of fundamental kernels to form a neighborhood mean kernel.

7.5 Debugging GPU-bound products using Quartz Composer

Quartz Composer (QC) is a rapid development environment tool that allows a developer or user to use concise units, called patches, to construct powerful graphics oriented effects. QC provides the user with a set of filters, generators, input patches, composite patches and plug-ins to construct effects while contributing very little program code. The plug-in patches and SDK provides developers the ability to test their own code, which they provide in an intuitive manner. QC was shown in the CPU-bound implementations of PCA and ICA in this report.

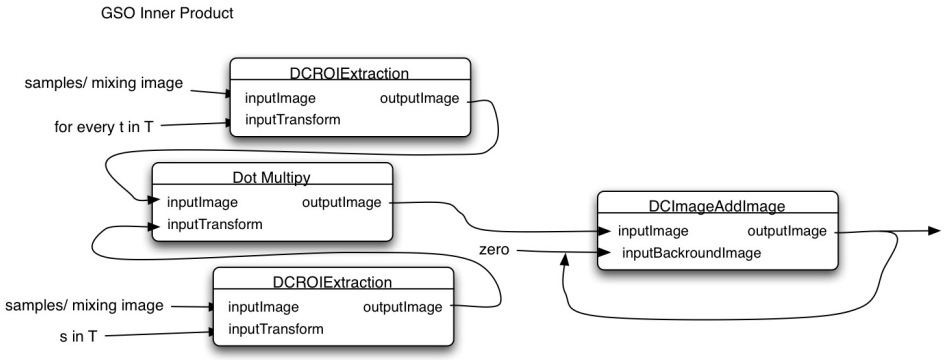

Figure 17: This diagram shows the connection of fundamental kernels to form a neighborhood inner product kernel.

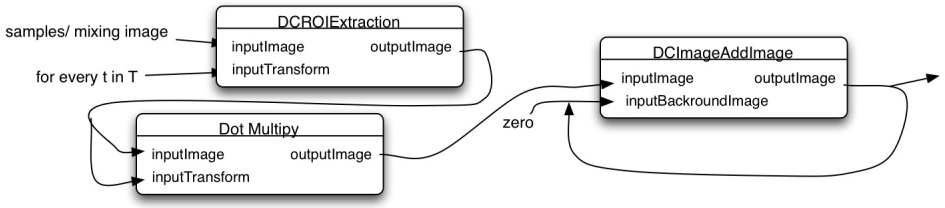

Figure 18: This diagram shows the connection of fundamental kernels to form a neighborhood norms kernel.

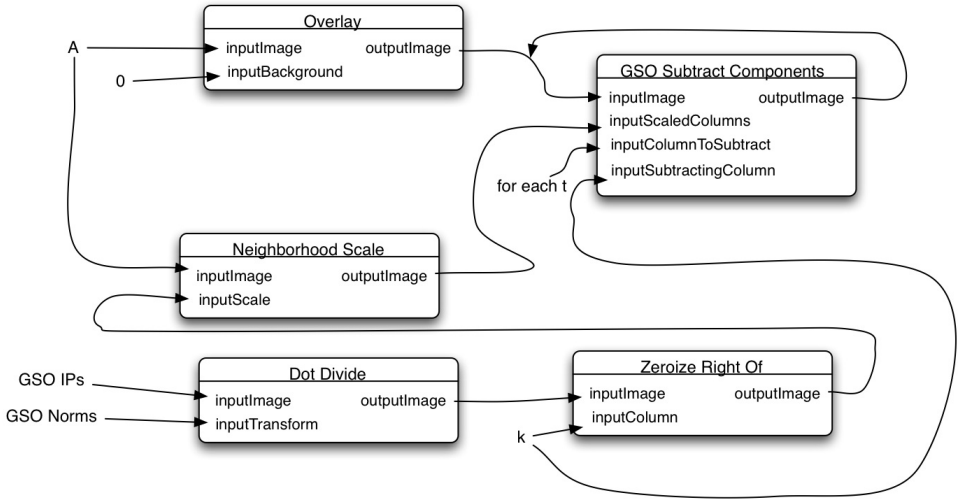

Figure 19: This diagram shows the connection of fundamental kernels to form a neighborhood subtract kernel.

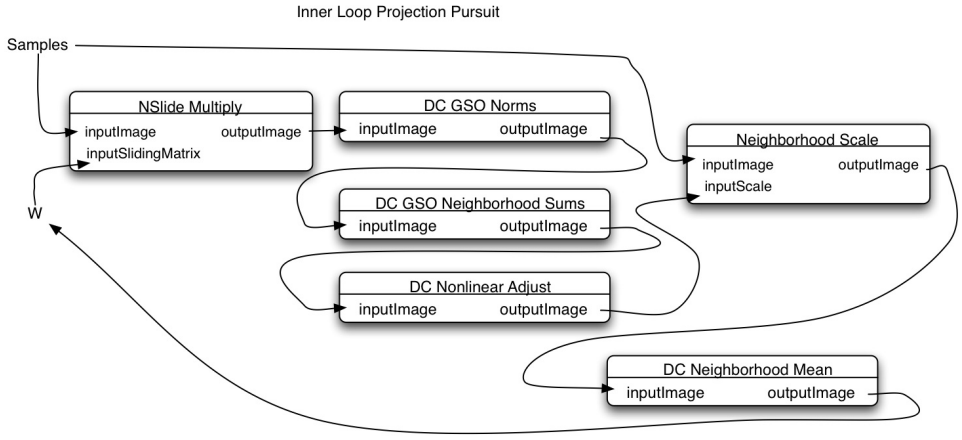

Figure 20: This diagram shows the connection of fundamental kernels to form a neighborhood ICA kernel.

In this sub-section, QC plug-ins are further elaborated to aid in understanding QC's prototype assisting power. In both cases, developing a CPU-bound filter and a CIKM filter, the dictionary type (NS-Dictionary) is key to moving these data-types from patch to patch. This particular dictionary type can store any object type, and it is useful to pair that object with a standardized key. In the case of CIKM types, the key used is "CIFilter," which is the precise data type used and produced.

Also in this sub-section an incremental approach is made by reducing the QC Plug-in to a loop controller with feedback. This concept allows kernel level CI Filters to be connected directly in QC without using a dictionary as a wrapper. The advantage is that the prototype is reduced in complexity. The disadvantage is rooted in the selection and copy of either the feedback or initial images.

7.5.1 Excessive Handlers: Debugging the generator and filters

Each QC plug-in constructing a CIKM filter has the same output type, a CI Filter wrapped in a dictionary. The input for these plug-ins are dictionaries containing composing filters, and double-floating point real numbers representing the constants needed to construct the filter. The execution of the plug-in determines if there has been a change in the input, and proceeds if there has been. If the inputs are appropriate for the filter handler, which is an object that contains the method and objects necessary to construct the combined filter, then the handler returns an object, and nil otherwise. If the result is nil, then the output is also assigned nil and the execution is reported with a value of "NO". If the handler is not nil, then the handler is instructed to determine the filter, and the filter is assigned to the output dictionary.

Each handler initialization method takes in the input parameters necessary to construct the desired filter. There are four filter types that handler can use. A filter produced by another handler is one of the types, and by convention, is one that should be a property of the handler. The second type is a standard CI filter, which is included with the CI package for which there are many supplied by Apple, Inc. Another set of CI filters are those supplied by the Distributed Computing Lab (Texas Tech University) as part of the DC frameworks. Lastly, more handlers and filters could be constructed by third parties.

Each handler contains a method, called "determine," which uses the CI Filter Generator to connect each of the supporting filters. There is a safeguard in the event that the

"determine" method fails. If the output dictionary is not nil, then the execution is reported as "YES," otherwise "NO" is reported. The "NO" tells Quartz Composer that something is wrong and causes any patches in the execution path to stop.

Lastly, there is one special QC Plug-in that can only use the handler for connecting attributes to an input filter. This QC Plug-in family is called an imposition plug-in. Its purpose is to apply the generated filter to an actual image. This type of plug-in is built for a specific type of QC patch, which is for the final filter. As such, the patch possesses as input ports the filter itself and the filter's inputs (images, numbers and vectors). The output must be a QC image.

7.5.2 Reducing debugging problems by using QC-Plug-in to Control the Loop

There are benefits from the control loop model. First, the prototype reduces all CI Filters to kernel level. Each patch can be reduced to unit purpose. Each unit purpose patch has a direct resemblance to its neighborhood algebra model. Lastly, the native CI patch can be used to reduce the CI Filter plug-in's, reducing the number of filters necessary to distribute.

The inner loop of GSO in this feedback control loop model obeys the neighborhood algebra model. The inner product and norms results are fed into the subtraction composition. The input to both inner product and norms compositions are either the initial image or feedback from the subtract composition. The loop and determination of the input to the inner loop is controlled by the feedback and control plug-in.

The inner product composition also obeys the neighborhood algebra model and has a feedback control patch. The same can be said about norms and the subtraction composition. As the neighborhood model specifies the number of neighborhoods is the ratio of width to height, the dimensions' patch in combination with a math patch determines these values. These patches can be observed in Figure 21.

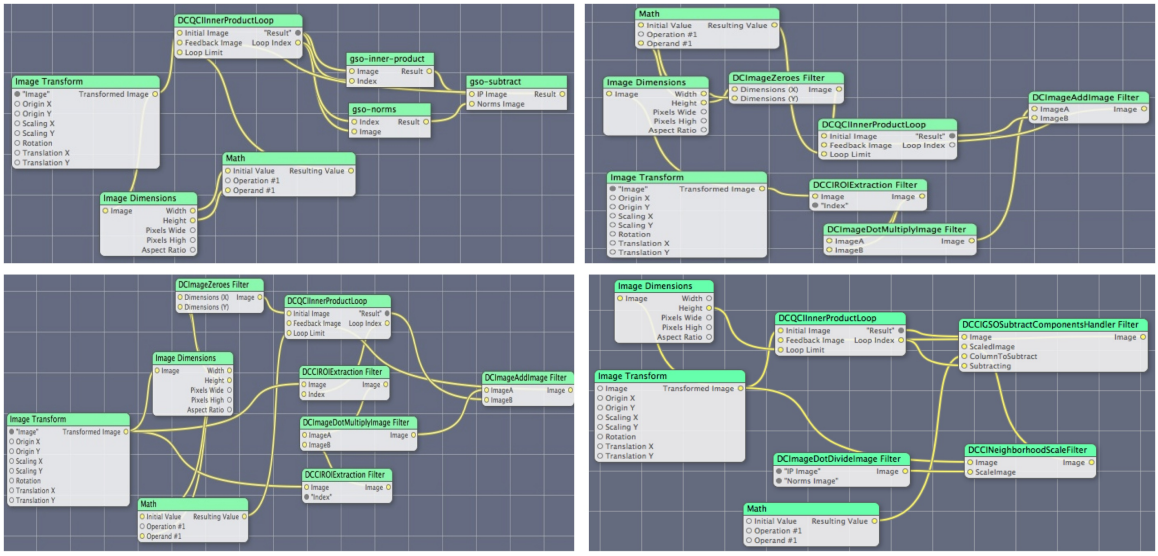

Figure 21: This diagram shows the collection of QC patches computing GSO using a feedback control patch. Called DCQCIInnerProductLoop in these diagrams, the control patch provides a mechanism for feedback to build a solution for the PCA mixing matrix. This approach empowers QC to provide prototyping tool for GPU bound solutions.

The feedback control loop itself has three inputs. The two images, initial and feedback, supply the control with a selection to start and finish the loop. The loop limit controls how many loops may be executed. This works well with the neighborhood algebra model as it is typically the height of the image being computed. The outputs of the feedback control loop patch are the result image and the loop index. The result image is fed into the

compositions components, which define the unit's inner loop. The loop index, as the name implies, is the index for the inner loop's components to work on. This is a consequence of the neighborhood algebra model and does have a design pattern to it.

The GSO version of PCA produces a similar mixing matrix to the EVD-PCA method. In this case, neighborhood expansion and selection can solve segmentation problems. Also, use of neighborhood expansion and scaling can solve noise reduction problems.

There are some caveats with using the QC Plug-in control loop implementation of the neighborhood algebra model of GSO-PCA. The Loop controller itself has to copy the image from input to output protocols. A limit exists in these protocols for texture and buffer resolution. Perpetual build and stop means that the solution is eventually reached, and the intermediate values are seen too.

7.6 Accelerate Construction using NSOperation

There is a method for altering a handler, originally made for a Quartz Composer plug-in, to inherit multi-threaded properties by making it a NS-Operation subclass. There are three key differences which do not effect the logic, but supply parallelism to the computation. First the algorithm specified in the "determine" method becomes the "main" method. Second, the properties passed by other handlers must be accessed from the other handlers and the other handlers become dependent properties of the operating handler. These handlers must be specified as dependencies. Third, the handlers become operations themselves, and those operations must be loaded into an NS-Operation Queue while the queue is suspended.

Once loaded, the queue may be reactivated, which causes the handlers to be determined in parallel fashion.

Figure 13 shows the Quartz Composer patch connection method of combining these filter generating handlers. The filters can be converted to NS-Operation subclasses with relative ease. This conversion allows for a NS-Operation Queue to determine the final filter. There are two possibilities of deployment via this queue. The queue can be inserted into a CI-Filter plug-in as a CPU component of the filter. The advantage of the pseudo filter is that it is usable in Quartz Composer and is treated as a kernel filter. This is sort of pseudo filter can have one disadvantage. The disadvantage is the filter may not be reused without being regenerated. Also, the filter does have to wait for the queue to finish computations. On single CPU systems, that queue could take more time than the security model allows.

A A Listing Mathematica Implementation for the Iris Dataset

```
barSetosa = Mean[setosa]
barSetosaArray = PadLeft[{barSetosa}, First[Dimensions[setosa]], {barSetosa}]
normSetosa = setosa - barSetosaArray
ScatterSetosa = Transpose[normSetosa].normSetosa/First[Dimensions[setosa]]


barVersicolor = Mean[versicolor]
```

```
barVersicolorArray = PadLeft[{barVersicolor}, First[Dimensions[
  versicolor]], {barVersicolor}]

normVersicolor = versicolor - barVersicolorArray
ScatterVeriscolor =
        Transpose[normVersicolor].normVersicolor/First[Dimensions[versicolor]\
]

barVirginica = Mean[virginica]
barVirginicaArray = PadLeft[{barVirginica}, First[Dimensions[virginica]], \
{barVirginica}]
normVirginica = virginica - barVirginicaArray
ScatterVirginica = \
Transpose[normVirginica].normVirginica/First[Dimensions[virginica]]
whiteningScatter = ScatterVirginica + ScatterSetosa + ScatterVeriscolor
flowers = Join[setosa, versicolor, virginica]
combinedMean = Mean[flowers]
flowers = Join[setosa, versicolor, virginica]
combinedMean = Mean[flowers]
meanArray = Join[{barSetosa}, {barVersicolor}, {barVirginica}]
```

```
combineMeanArray = PadLeft[{combinedMean}, Dimensions[meanArray], \
{combinedMean}]
meanDifference = (meanArray - combineMeanArray)
scatterBackground = Transpose[meanDifference] .(50*meanDifference)
{lambda, W} = Eigensystem [{scatterBackground, whiteningScatter}, 2]
ListPlot[flowers.Transpose[W]]
```

B A Listing of the Objective-C version of EM

Listing 1: Initialize EM

```
-(id) initWithSamples:(dcgVector *) someSamples
                                    numberOfGaussians:(int) M
{
        [super init];
        numberOfClasses = M;
        numberOfSamples = [someSamples vecLength];
        samples = [someSamples retain];

        double max = [samples max];
```

```
        double min = [samples min];


        double scale = (max - min)  / M ;
//      scale *= 0.5;
        int r;


        mean = [[dcgVector alloc] initWithLength:numberOfClasses];
        variance = [[dcgVector alloc] initWithLength:numberOfClasses];
        proportions = [[dcgVector alloc] initWithLength:numberOfClasses];
        mixture = [[dcgMatrix alloc] initWithRows:numberOfSamples
                                          columns:numberOfClasses
                                          ];
        double *mu = [mean localVector];
        double *sigma = [variance localVector];
        double *alpha = [proportions localVector];


        for ( r = 0; r < M; r++) {
                mu[r] = (max - min ) * r * scale + min ;
                sigma[r] = scale;
                alpha[r] = scale;
```

```
        }


        [self computeEM];
        return self;
}
```

Listing 2: Estimation step of EM, computed explicitly in Objective C

```
-(void) estimationStep
{
        double *mu = [mean localVector];
        double *sigma = [variance localVector];
        double *alpha = [proportions localVector];
        double **A = [mixture localMatrix];
        double *D = [samples localVector];

        double amp, sumAlpha = 0;
        Q = 0;
        int r, c;

        for ( r = 0 ; r < numberOfSamples; r++)
```

```
                for ( c = 0; c < numberOfClasses; c++)
                {
                        amp = alpha[c];
                        amp /= sqrt(2 *M_PI * sigma[c]);
                        A[r][c] = amp * exp (-0.5 * (D[r] - mu[c])
* (D[r] -mu[c]) / sigma[c]) ;
                }

        for ( r = 0 ; r < numberOfSamples; r++)
        {
                sumAlpha = 0;
                for ( c = 0; c < numberOfClasses; c++)
                        sumAlpha += A[r][c] ;
                for ( c = 0; c < numberOfClasses; c++)
                        A[c][r] /= sumAlpha ;
        }

        for ( r = 0; r < numberOfSamples; r++)
        {
                 sumAlpha = 0.0;
```

```
                for ( c = 0; c < numberOfClasses; c++)
                    sumAlpha += A[r][c] ;
                Q += fabs(log(sumAlpha));
        }

}
```

Listing 3: Maximization step of EM, computed explicitly in Objective C

```
-(void) maximizationStep
{
        double *mu = [mean localVector];
        double *sigma = [variance localVector];
        double *alpha = [proportions localVector];
        double **A = [mixture localMatrix];
        double *Y = [samples localVector];

        int r, c;
        double meanDifference;
```

```
[proportions zeroize];
for ( r =0; r < numberOfSamples; r++)
        for ( c = 0; c < numberOfClasses; c++)
                alpha[c] += A[r][c];


[variance zeroize];
for ( r =0; r < numberOfSamples; r++)
        for ( c = 0; c < numberOfClasses; c++)
        {
                meanDifference = Y[r] - mu[c];
                sigma[c]  += A[r][c] * meanDifference * meanDifference;
        }
for ( c = 0; c < numberOfClasses; c++)
                sigma[c] /= alpha[c];



[mean zeroize];
for ( r =0; r < numberOfSamples; r++)
        for ( c = 0; c < numberOfClasses; c++)
```

```
                mu[c]  += A[r][c] * Y[r];

        for (c = 0; c < numberOfClasses; c++)
                mu[c] /= alpha[c];

        for (c = 0; c < numberOfClasses; c++)
                alpha[c] /= numberOfSamples;

}
```

C Implementation of Projection Pursuit

Listing 4: Projection Pursuit in Objective C

```
-(void) apply
// Override on how the ICA algorithm is
// computed.  Namely, this one is a
// straight computation of projection
// pursuit.
{
[self prepareData];
int N = [samples colSize];
```

```
[randomAgent setLength:[NSNumber numberWithInt:N]];
[randomAgent apply];
dcgVector *w = [randomAgent randomVector];
dcgVector *wold;
dcgVector *deltaW;
double y;
Q = [NSNumber numberWithDouble:[w L2norm]];
do
 {
  lastQ = Q;
  wold = w;
  y = [self dotProductExpectation:w];
  [meanVector setSourceColumnWise:
   [blasWork multiplyMatrix:whitenedData
        byScalar:pow(y, 3.0)]];
        deltaW = [meanVector apply];

  w = [self addMixVector:w withDeltaVector:deltaW];
  Q = [NSNumber numberWithDouble:
```

```
                    [blasWork dotProduct:w
                    byVector:wold]];
 }
 while (![self epsilonReached]);
}
```

Listing 5: Zero Mean Multivariate in Objective C

```
-(void) apply
{
 dcgBlasService *blasWork;
 blasWork = [[dcgBlasService alloc]
   init];
 dcgMatrix *workingMatrix;
 workingMatrix = [dcgMatrix zeroMatrix:
  [sourceColumnWise rowSize]
  columns:[sourceColumnWise colSize]];
 int N = [sourceColumnWise colSize];
 int i;
 DCGMeanVector *meanVectorAgent;
 meanVectorAgent = [[DCGMeanVector alloc] init];
 [meanVectorAgent
```

```
  setSourceColumnWise:sourceColumnWise];
 [self setMean:[meanVectorAgent apply]];
 for ( i = 0; i < N; i++)
 {
  [workingMatrix insertVector:mean atColumn:i];
 }
 [self setZeroMeanMultivariate:
   [blasWork subtractMatrix:sourceColumnWise
   bMatrix:workingMatrix]];
 [blasWork release];
 [meanVectorAgent release];
}
```

D Core Image Filters Developed for Conventional Linear Algebra

D.1 Image Add Image

D.2 Image Add Scalar

D.3 Image Column Dot Product No Scale

D.4 Image Column Multiply

D.5 Image Determinant Unit

D.6 Image Determinant Column

D.7 Image Divide Scalar

D.8 Image Dot Divide Image

D.9 Image Dot Multiply Image

D.10 Image Dot Square Root

D.11 Image Identity

D.12 Image Zeroes

D.13 Image Inner Product and Norm Multiplication

D.14 Image Maximum Unit

D.15 Image Multiply Scalar

D.16 Image Multiply Image Unit (Standard)

D.17 Image Subtract Image

D.18 Image Subtract Scalar

D.19 Image Trace Unit

D.20 Image Transpose

E Neighbor Algebra Notes

E.1 Equivalent Transpose

A special case of transpose exist for match neighborhood algebra transpose to standard linear algebra, namely the case of a square matrix. The neighborhood equivalent renders $T = k^2$ such that the dimensions of the neighborhood collection is $k \times k(k^2) = k \times k^3$. In this case, the location equations have the following conversions:

$$j = dc.x/k \tag{59}$$

$$i = dc.y \cdot k + dc.x\%k \tag{60}$$

$$sdc.x = \frac{j}{k} \tag{61}$$

$$sdc.y = ik + j\%k \tag{62}$$

$$A = \text{sample (src, sdc)} \tag{63}$$

Second case, if $\tau = \sqrt{T}$, then τ must be a natural number. This forces constrictions on what linear algebra equivalents can be guaranteed through neighborhood algebra. In these case, this still give a generic transpose operation that is a one kernel operation.

$$j = dc.x/\tau \tag{64}$$

$$i = dc.y \cdot \tau + dc.x\%\tau \tag{65}$$

$$sdc.x = \frac{j}{k} \tag{66}$$

$$sdc.y = ik + j\%k \tag{67}$$

$$A = \text{sample (src, sdc)} \tag{68}$$

E.2 Region Of Interest Extractor

E.3 Outer Product

E.4 Samples Every So Many

E.5 Dot multiply and Add

F Theory of Estimators

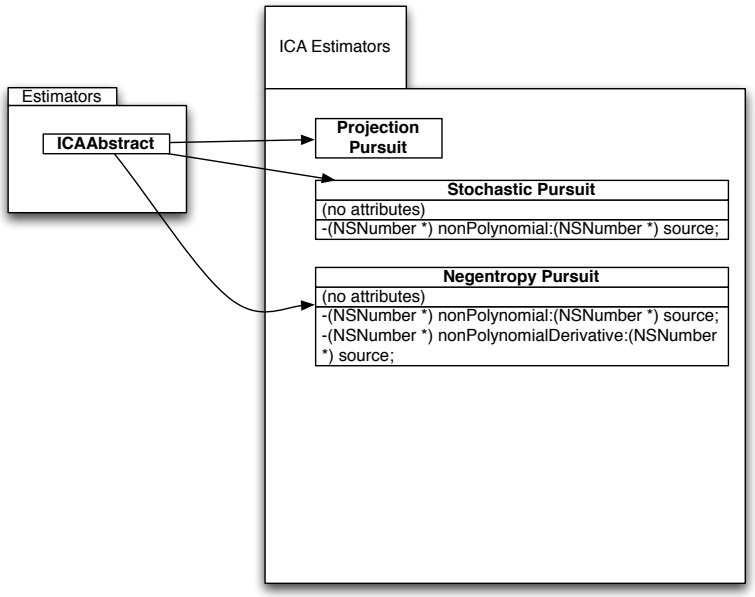

Figure 22: Estimators (ICA) Framework

The theory of estimators is based on the Definition F.1 and includes many linear and nonlinear methods to obtain the parameters. Estimators that achieve the estimated parameters close to the actual parameters are called constant.

Definition F.1 *An estimator, $\hat{\theta}$, of the parameter vector, $\vec{\theta}$, is the mathematical expression that estimates the parameters of an analytical solution for a given set of measurements (samples).*

[**?**, 78]

There are a few constraints that any estimator should satisfy. Estimators for the equations 69 and 70 are called unbiased.

$$E\{\hat{\theta}\} \approx E\{\vec{\theta}\} \tag{69}$$

$$E\{\hat{\theta}|\vec{\theta}\} = \vec{\theta} \tag{70}$$

Estimators not meeting this criteria are called biased. The bias vector is defined as follows:

$$\vec{b} = E\{\tilde{\theta}\} \tag{71}$$

$$\vec{b} = E\{\tilde{\theta}|\vec{\theta}\} \tag{72}$$

If $\vec{b} \to \vec{0}$ as the number of measurements goes up, then the estimator is called asymmetrically unbiased. [**?**, 79]

Estimators often require discrete items, which are called bins when forced on continuous values, in order to generate statistical models for data-sets. One important concept is the loss function as defined in Definition F.2

Definition F.2 *A loss function $h(\tilde{\theta})$ is a metric of the relative importance of specific estimation errors.*

[?, 79]

Another important concept for estimators is their metric of accuracy. The article efficient is used as a ranking of an estimator as defined in Definition F.3.

Definition F.3 *An efficient estimator provides the smallest error covariance matrix among all unbiased estimators.*

[?, 81] Symmetry is also a means of ordering a matrix. If $\mathbf{A}$ and $\mathbf{B}$ are both symmetrical matrices and $\mathbf{B} - \mathbf{A}$ is positive definite, then $\mathbf{A} < \mathbf{B}$. One theorem, Cramer-Rao Lower Bound, provides for partial ordering amongst symmetric matrices. This theorem is defined in Theorem F.4.

Theorem F.4 *If $\hat{\theta}$ is any unbiased estimator of $\vec{\theta}$ based on the measurement data $\vec{x}$, then the covariance matrix of error in the estimator is bounded below by the inverse of the Fisher information matrix:*

$$J^{-1} \leq E\{(\vec{\theta} - \hat{\theta})(\vec{\theta} - \hat{\theta})^T | \vec{\theta}\} \tag{73}$$

$$J = E\{[\frac{\partial}{\partial \vec{\theta}} \ln p(\vec{x_t}|\vec{\theta})][\frac{\partial}{\partial \vec{\theta}} \ln p(\vec{x_t}|\vec{\theta})]^T | \vec{\theta}\} \tag{74}$$

[?, 83]

The last quality of an estimator is robustness. Robustness is defined in Definition F.5. Robustness determines an estimator's ability to accurately model a data-set despite the presence of outlier errors.

> **Definition F.5** *Robustness is an insensitivity to gross measurement errors, and errors in specification of the parametric models.*
>
> [**?**, 83]

Maximum-likelihood (ML) is covered in another report. ML may be reused in the pursuit of ICA, but only in specific derivations thereof. As this report does not cover the ML version of ICA, this report only alludes to it as an alternative estimator used to determine ICA.

F.1 Least Squares

Least Squares (LS) regression is a linear method used to determine specific parameters of an assumed analytical solution. LS is a classic curve fitting method, which constructs a matrix that is solved by matrix solution methods like Gauss Jordan Elimination. From this algorithm, there are a family of least squares estimators. Each one has their own assumed model and parameters. There is a general class of LS for polynomials. Other models such as Gaussian or other statistical models should be developed with the following rules:

- The desired property is to have more measurements than parameters (known or unknown).

Algorithm 8 Least Squares Estimation (Regression) derived from definition found at [**?**].

Require: DCGVector $\vec{x}$ consisting of m observations
Require: Assumed model, parameters, and derived functions.
Take an initial guess at the model parameters $\lambda_i \ \forall_i \in [1, n]$.
repeat
Form matrix A such that

$$\begin{pmatrix} \frac{\partial f(x)}{\partial \lambda_1}|x_1 & \dots & \frac{\partial f(x)}{\partial \lambda_n}|x_1 \\ \dots & \dots & \dots \\ \frac{\partial f(x)}{\partial \lambda_1}|x_m & \dots & \frac{\partial f(x)}{\partial \lambda_n}|x_m \end{pmatrix} \tag{75}$$

Compute $R = A^T A$
Compute dB such that

$$B = \sum_{i=1}^{m} (y_i - f(x)) \tag{76}$$

Determine $L = A^T dB$
Determine $\partial\vec{\lambda}$ from $R(\partial\vec{\lambda}) = L$
Adjust $\vec{\lambda}$ by $\partial\vec{\lambda}$
until $d\vec{\lambda} \to 0$

- The goal is to choose an estimator that minimizes the effects of the error.

F.2 Maximum a Posteriori

Maximum a Posteriori (MAP) is based on Bayes Theorem, theorem 77. The characteristic equation for a given set of data is the sum of a set of distributions. Each parameter of this characteristic equation can be determined by LS regression. This principal comes from the Optimal Estimator Theorem.

$$p_{\vec{\theta}|\vec{x_t}}(\vec{\theta}|\vec{x_t}) = \frac{p_{\vec{x_T}|\vec{\theta}}(\vec{x_T}|\vec{\theta}) p_{\vec{\theta}}(\vec{\theta})}{p_{\vec{x_T}}(x_T)} \tag{77}$$

$$p_{\vec{x_T}}(x_T) = \int_{-\infty}^{\infty} p_{\vec{x_T}|\vec{\theta}}(\vec{x_T}|\vec{\theta}) p_{\vec{\theta}}(\vec{\theta}) d\vec{\theta} \tag{78}$$

Theorem F.6 ***Optimal Estimator Theorem:*** *Assume that the parameters* $\vec{\theta}$ *and the observations* $\vec{x_T}$ *have the joint probability density function* $p_{\vec{\theta},\vec{x_t}}(\vec{\theta}, \vec{x_t})$. *The minimum mean-square estimator* $\hat{\theta}_{MSE}$ *of* $\vec{\theta}$ *is given by the conditional expectation:*

$$\hat{\theta}_{MSE} = E[\vec{\theta}\vec{x_T}] \tag{79}$$

[**?**, 94]

The goal of MAP is to find the parameter vector $\vec{\theta}$ that maximizes the posterior density of the Bayes Theorem equation. The parameters are also based on an analytically defined model. In this case, $\vec{\theta}$ needs to maximize the numerator, as the denominator does not depend on $\vec{\theta}$.

$$p_{\vec{\theta}|\vec{x_t}}(\vec{\theta}|\vec{x_t}) \approx p_{\vec{x_T}|\vec{\theta}}(\vec{x_T}|\vec{\theta}) p_{\vec{\theta}}(\vec{\theta}) d\vec{\theta} \tag{80}$$

$$\ln p_{\vec{\theta}|\vec{x_t}}(\vec{\theta}|\vec{x_t}) \approx \ln(p_{\vec{x_T}|\vec{\theta}}(\vec{x_T}|\vec{\theta}) p_{\vec{\theta}}(\vec{\theta}) d\vec{\theta}) \tag{81}$$

$$\approx \ln(p_{\vec{x_T}|\vec{\theta}}(\vec{x_T}|\vec{\theta})) + \ln(p_{\vec{\theta}}(\vec{\theta}) d\vec{\theta}) \tag{82}$$

$$\approx \frac{\partial}{\partial\vec{\theta}} \ln(p_{\vec{x_T}|\vec{\theta}}(\vec{x_T}|\vec{\theta})) + \frac{\partial}{\partial\vec{\theta}} \ln(p_{\vec{\theta}}(\vec{\theta}) d\vec{\theta}) \tag{83}$$

Given these equations the solution can be defined by setting equation 83 to zero.

$$\frac{\partial}{\partial\vec{\theta}}\ln(p_{\vec{x_T}|\vec{\theta}}(\vec{x_T}|\vec{\theta})) + \frac{\partial}{\partial\vec{\theta}}\ln(p_{\vec{\theta}}(\vec{\theta})d\vec{\theta}) = 0 \tag{84}$$

Like ML and EM, MAP is mentioned here as an alternative means of determining ICA. MAP itself does not provide independent components or its mixing matrix. Rather it provides an intermediate step from which ICA may be determined.

G Entropy and Mutual Information: Determining Independence

One of the most crucial features of the ICA estimator is the ability determine independence. Two fundamental concepts that lead to deterministic methods of independence are entropy and mutual information. As these two concepts are very related, they are presented in the following sub-sections.

G.1 Measures of Information

The concept of entropy is best stated in its differential forms, both discrete and continuous. Entropy can be treated as a measure of the amount of information contained in a specific data set. In the discrete case, this concept can determine the minimum encoding necessary

to represent the data.

$$H(X) = \sum_i f(P(X = a_i)) \tag{85}$$

$$H(x) = -\int p_x(\eta) \log p_x(\eta) d\eta = \int f(p_x(\eta)) d\eta \tag{86}$$

Note that transforming entropy increases and decreases monotonically.

A measure of information that one random variable has on other random variables in a set can be defined in terms of a single random variable x_i.

$$I(\mathbf{X}, \mathbf{Y}) = H(\mathbf{Y}) - H(\mathbf{Y}|\mathbf{X}) = H(\mathbf{X}) - H(\mathbf{X}|\mathbf{Y}) \tag{87}$$

> Maximizing the joint entropy $H(Y_1, Y_2)$ is accomplished by maximizing the individual entropies while minimizing the mutual information $I(Y_1, Y_2)$. When $I(Y_1, Y_2)$ is zero, then Y_1 and Y_2 are statistically independent. [**?**, 662]

The purpose of Kullback-Leibler Divergence is provide a pseudo metric in terms of distance between two density functions, and therefore determine mutual information. It is not a true metric since it is not symmetrical. Yet, the Kullback-Leibler Divergence does increase monotonically with mutual information.

$$\Delta(p|q) = \sum_i p(i)(\log p(i) - \log q(i)) \tag{88}$$

$$\Delta(p|q) = \int p(i)(\log p(i) - \log q(i)) \tag{89}$$

G.2 Theory of Maximizing Representation

The goal of maximum-entropy is to determine the probability density function that satisfies equation 90 such that the c_i constants are maximum. One example under some regularity conditions is equation 91 (under the constraint in equation 91).

$$\int p(\eta)F_i(\eta)d\eta = c_i \tag{90}$$

$$p_0(\eta) = A\exp(\sum_i a_i F_i(\eta)) \tag{91}$$

$$\int p_0(\eta)d\eta = 1 \tag{92}$$

G.2.1 Negentropy

Negentropy is a concept derived from maximum entropy specifically for determining Gaussian properties. The objective is to have "a measure that is zero for a Gaussian variable and always nonnegative can be simply obtained from differential entropy." Negentropy, denoted $J(\vec{x})$, is defined in equation **??**. The entropy of x_{Gauss} is defined in terms the covariance matrix of $\vec{x}$ (Σ) and the dimension of $\vec{x}$.

$$J(\vec{x}) = H(\vec{x}_{Gauss}) - H(\vec{x}) \tag{93}$$

$$H(\vec{x}_{Gauss}) = \frac{1}{2}\log|\det\Sigma| + \frac{n}{2}[1 + \log 2\pi] \tag{94}$$

Properties of negentropy include:

- $J(\vec{x}) = 0$ if and only if $\vec{x}$ has a Gaussian distribution.
- The maximum Gaussian distribution satisfying equation 90 forces $J(\vec{x}) \geq 0$
- $J(\vec{x})$ is invariant and is an invertible linear transformation as proven in [**?**, 113].

G.2.2 Approximation by Cummulants

Polynomial density expansion approximation of negentropy, maximum entropy, and mutual information is defined for a standardized Gaussian model (zero mean and unit variance) as in equation 95, and its derivative in equation 95. The polynomial system is chosen for the practicality that they are orthogonal.

$$\phi(\eta) = \frac{1}{2\pi} \exp(\frac{-\eta}{2}) \frac{\partial^i \phi(\eta)}{\partial \eta} = (-1)^i H_i(\eta) \phi(\eta) \tag{95}$$

Another approximation that maximizes representation is the log of the cumulants. This approximation depends on the cumulants being relatively small.

$$\log(1 + \epsilon) \approx \epsilon - \frac{\epsilon^2}{2} \tag{96}$$

Therefore, cumulants can be used to define a characteristic equation for the information

carried in a data-set x.

$$p_x(\eta) \approx \phi(\eta)(1 + \kappa_3(x)\frac{H_3(\eta)}{3!} + \kappa_4(x)\frac{H_4(\eta)}{4!}) \tag{97}$$

$$J(x) \approx \frac{1}{12}\kappa_3(x)^2 + \frac{1}{48}\kappa_4(x)^2 \tag{98}$$

Thus a sum of the skewness squared and the kurtosis squared are both computable from the random framework. The drawback is that this method is sensitive to outliers and it measures the tails and not the center. It is thus worth considering methods of determining entropy by non-polynomial functions. The constraints on such methods are as follows:

- Non-polynomial functions should be based on maximum entropy methods.
- Non-polynomial functions confine the distribution from the infinite which will satisfy to a specific set of constraints.