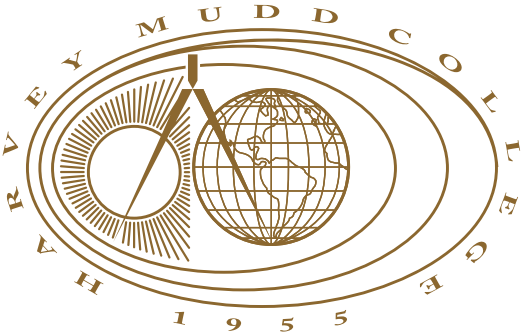
2006

Mathematical Models of Image Processing

Tyler Seacrest
Harvey Mudd College



Mathematical Models of Image Processing

Tyler Seacrest

Professor Weiqing Gu, Advisor

Professor Darryl Yong, Reader

May, 2006

HARVEY MUDD
COLLEGE
Department of Mathematics

Contents

Acknowledgments **vii**

1 Introduction **1**

2 Background **3**
2.1 Color Spaces 3
2.2 Distance in Color Space 4
2.3 Continuous Image Space 5
2.4 Discrete Image Space 6
2.5 Operations in Color Space 6
2.6 Operations in Image Space 7
2.7 QR-Decomposition 8

3 Space of Images **11**
3.1 Application 12

4 Covariance Matrix **13**
4.1 Defining the Covariance Matrix 13
4.2 Image Alignment 13
4.3 Color Alignment 16

5 Applications of the Perron-Frobenius Theorem **17**
5.1 Introduction 17
5.2 Color Clashing 17
5.3 Color Variety 19
5.4 Example 20

6 Applications of Lie Groups **23**
6.1 Introduction 23
6.2 Illumination Invariants 25

6.3 Blurring 25
6.4 Invariant Function under Gaussian Blurs 27
6.5 Application of Gaussian Blurring Invariants 29

7 Conclusion and Future Work **31**

A varietyExample.java **33**

B rot.java **37**

Bibliography **49**

List of Figures

2.1 A representation of HSL-space (figure by Alexandre Van de Sande) 5
2.2 The hypermatrix representation of a $3 \times 3 \times 3$ color image. . 6

3.1 Given two instances, a yellow and faded image can be restored using linear approximation 12

4.1 Two images rotated in accordance with decorrelating the region. Image generated by code from Appendix B. 14

5.1 The best combination of the three colors shown above according to this model. See Appendix A. 21

Acknowledgments

I would like to thank Professor Gu for making this thesis possible and for the multitudes of creative ideas she lent to the process. Also thanks to Darryl Yong for his helpful feedback, Deborah E. Berg for her editing and wonderful suggestions, and to Alexandre Van de Sande for the excellent HSL-space figure.

Chapter 1

Introduction

The purpose of this thesis is to develop various advanced linear algebra techniques that apply to image processing. With the increasing use of computers and digital photography, being able to manipulate digital images efficiently and with greater freedom is extremely important. By applying the tools of linear algebra, we hope to improve the ability to process such images.

We are especially interested in developing techniques that allow computers to manipulate images with the least amount of human guidance. In Chapter 2 and Chapter 3, we develop the basic definitions and linear algebra concepts that lay the foundation for later chapters. Then, in Chapter 4, we demonstrate techniques that allow a computer to rotate an image to the correct orientation automatically, and similarly, for the computer to correct a certain class of color distortion automatically. In both cases, we use certain properties of the eigenvalues and eigenvectors of covariance matrices. We then model color clashing and color variation in Chapter 5 using a powerful tool from linear algebra known as the Perron-Frobenius theorem. Finally, we explore ways to determine whether an image is a blur of another image using invariant functions. The inspiration behind these functions are recent applications of Lie Groups and Lie algebra to image processing.

Original work presented in this thesis includes

- a representation of color space that is also a vector space, allowing the use of linear algebra tools to apply to this image processing space.
- techniques correcting for a certain type of color distortion, using the representation of color space as a vector space.

- models to solve two problems in optimizing a choice of colors, and solutions to the models using Perron-Frobenius theorem.
- several different invariants under Gaussian image blurring, and a conjecture on the existence of infinitely many more.

Chapter 2

Background

2.1 Color Spaces

There are many ways to represent color. A standard method used for projectors and monitors is RGB, which mimics the way different colors of light combine. With a combination of red, green, and blue light, you can create any color (or shade of gray) in the spectrum. Therefore a color is represented by three numbers: a red value, a green value, and a blue value, all between 0 and 1. When combining light, the absence of any of the three is black, and the combination of all three produces white.

A color space used for printing is CMY, or CMYK. Here, colors are represented by a cyan value, a magenta value, and a yellow value. In this space, the absence of all three is white, and the combination of all three is black. Sometimes black (the "K") is a fourth value added to the mix, since printers often have a separate black ink cartridge, rather than using the other three colors.

A third color space is HSL, which stands for hue, saturation, and luminance. Hue is a quantity from on the interval $[0, 2\pi)$ which represents the color, symbolically choosing the color by specifying an angle along the color wheel. Saturation, measured from $[0, 1]$, gives the deepness of the hue. A saturation of 1 would be a bright color, while a 0 would be grey. Finally, luminance measures the lightness or darkness of a color on $[-1, 1]$, where 1 represents pure white (regardless of hue) and -1 black. HSL space is reprented in Figure 2.1.

There are other color spaces, but they are all essentially three-dimensional, and any extra channels are combinations of the first three. However, it can sometimes be hard to apply linear algebra concepts to these spaces. This

is because they are not true vectors spaces, since it is difficult to define addition, scalar multiplication, and inverses in a way that satisfy the basic axioms of a vector space, such as associativity.

Therefore, we will define color space in such a way that is a vector space, which we will denote **C**. Let (r, g, b) be a 3-tuple signifying a color in RGB space such that $r, g, b \neq 0, 1$. The corresponding color in **C** is represented by the 3-tuple:

$$\left(\frac{r - 1/2}{1/2 - |r - 1/2|}, \frac{g - 1/2}{1/2 - |g - 1/2|}, \frac{b - 1/2}{1/2 - |b - 1/2|} \right).$$

Notice that given a real red value R, we can find the RGB red value by $r = \frac{1}{2} + \frac{R}{2+2|R|}$ for any real R, and similarly for the other two color channels. Since you can go in either direction, it is essentially a bijection.

Also, note that "pure red" and other colors that involve r, g, or b to be 0 or 1 are not representable under **C**. However, you can get arbitrarily close to pure red. For example, the 3-tuple $(490, -490, -490)$ is equal to the RGB color $(.99, 0.01, 0.01)$.

Notice that $(0, 0, 0) \in \mathbf{C}$ is a middle gray color. Just as in RGB, (a, a, a) is a shade of gray for all $a \in \mathbb{R}$.

Now that we have a color space **C** that is a genuine vector space, as it is exactly like the vector space $\mathbb{R}^3$ in every respect.

2.2 Distance in Color Space

For practical application in a later chapter, it will be useful to give our intuitive idea how close or distant two colors our numeric values. A good space to use as a model is HSL-space, since it more closely models how humans perceive color than RGB-space or CMY-space. To do this, we represent HSL-space as the double cone in Figure 2.1 embedded in $\mathbb{R}^3$. We then simply measure the standard cartesian distance between two points in this space.

For two colors in this space c_1 and c_2, let h_i, s_i, l_i be the hue, saturation, and luminance values of color i. We see $h_i \in [0, 2\pi)$, $s_i \in [0, 1]$, and $l_i \in [-1, 1]$. Notice that in this space, these are somewhat like cylindrical coordinates, where h_i corresponds to θ, s_i to r, and l_i to z. Therefore, and h_i value of 0 is very near close to an h_i value of 2π. We define $d_x = (1 - |l_1|)s_1 \cos(h_1) - (1 - |l_2|)s_2 \cos(h_2)$. We define $d_y = (1 - |l_1|)s_1 \sin(h_1) - (1 - |l_2|)s_2 \sin(h_2)$. Finally, we let $d_z = l_1 - l_2$. Using these,

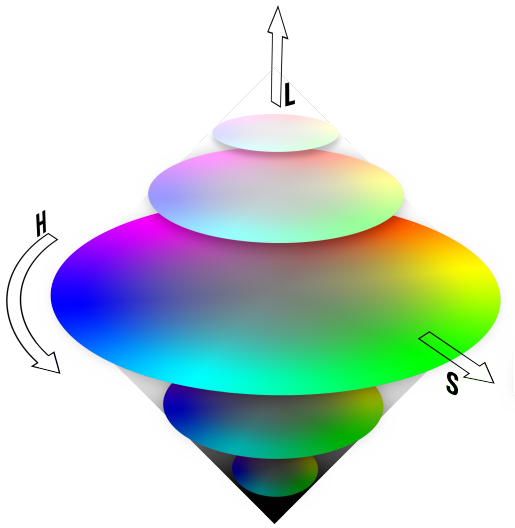


Figure 2.1: A representation of HSL-space (figure by Alexandre Van de Sande)

we let the distance $m(c_1, c_2)$ between the two colors is

$$m(c_1, c_2) = \sqrt{d_x^2 + d_y^2 + d_z^2}.$$

This will be used in Chapter 5 to help determine the variety in a given combination of colors.

2.3 Continuous Image Space

One model for images is assigning each point in $\mathbb{R}^n$ a color value. Thus, a *continuous image* is a function f from $\mathbb{R}^n$ (or a subset of $\mathbb{R}^n$) into a color space. For example, each pair of coordinates (x, y) might be assigned a red value $f_r(x, y)$, a green value $f_g(x, y)$, and a blue value $f_b(x, y)$. This model of image space is powerful because we can use analytic tools to study images. For example, the total redness in a rectangular region of the picture could be represented by

$$\int_\alpha^\beta \int_\gamma^\delta f_r(x, y) dx dy.$$

We will make use of such a continuous representation in Chapter 6.

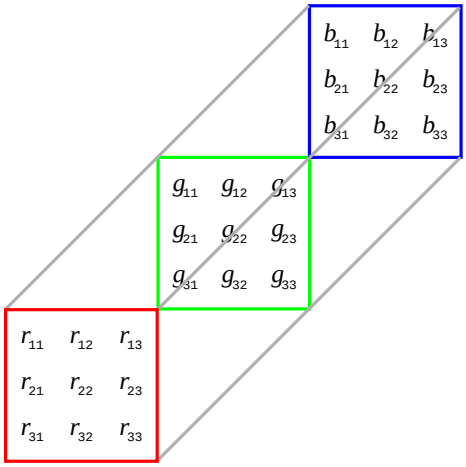


Figure 2.2: The hypermatrix representation of a $3 \times 3 \times 3$ color image.

2.4 Discrete Image Space

Sometimes, a discrete model for images is desirable. We define a *discrete image* as an $m \times n \times p$ hypermatrix C, where the ijkth entry represents the ijth pixel in color channel k.

For example, if $m = n = p = 3$, we get a hypermatrix such as the one in Figure 2.2. In general, c_{ijk} will be the ijkth entry of C. Let $\mathcal{C}(m, n, p)$ represent all hypermatrices of this type.

Another way to represent images in this manner is to model an image as a collection of vectors. Let V be a subset of vectors from R^n, and let $\mathbf{C}$ be a color space. Then a subset of $V \times \mathbf{C}$ can represent an image. Each pixel in the image is represented by a vector $\mathbf{v} \in V$ indicating the location of that pixel, and it is paired with $c \in \mathbf{C}$, which indicates the color of that pixel. For example, $((25, -50), Green)$ would represent a green pixel 25 units above and 50 units to the left of the origin.

2.5 Operations in Color Space

One common operation in image processing is adjusting the color of an image. What is often used is a linear transformation T from color space to color space, applied to each pixel of the image. For example, the linear

transformation

$$T = \begin{bmatrix} 0.2 & 0.7 & 0.1 \\ 0.2 & 0.7 & 0.1 \\ 0.2 & 0.7 & 0.1 \end{bmatrix},$$

if applied to every pixel of a color image C, produces a black and white image. Notice that this matrix is not invertible, and which shows that you cannot uniquely go from a black and white image to a color image.

Using the color space **C** defined earlier in this chapter, we see that scalar multiplication of an image has special meaning. If we multiply a color image C by a real number $a > 1$, then each pixel gets farther away from $(0, 0, 0)$, which is middle gray. Therefore, the contrast between the light (positive values) and dark (negative values) will increase. Similarly, multiplication by $a < 1$ will decrease contrast. Multiplication by -1 will turn reds into cyans, greens into magentas, and blues into yellows, so multiplication by -1 is the same as inverting the colors.

2.6 Operations in Image Space

The model of an image as a collection of vectors is especially nice for representing operations in image space. Let M be a collection of vectors and colors representing an image. To enlarge or shrink an image (known as *uniform scaling*), one simply needs to multiply each vector in M by a multiple of the identity matrix. So

$$\text{Uniform Scaling}_t(M) = \left\{ \left(\begin{bmatrix} t & 0 \\ 0 & t \end{bmatrix} \mathbf{v}, c \right) \;\middle|\; (\mathbf{v}, c) \in M \right\}.$$

Likewise, rotation is easily described as well.

$$\text{Rotation}_\theta(M) = \left\{ \left(\begin{bmatrix} \sin\theta & -\cos\theta \\ \cos\theta & \sin\theta \end{bmatrix} \mathbf{v}, c \right) \;\middle|\; (\mathbf{v}, c) \in M \right\}.$$

Given two images, seeing if one is a uniform scaling of the other is simple — simply normalize both images see if they are equal. Seeing if two images are rotations of each other is more difficult. A procedure for this is outlined in Chapter 4.

Notice that the above two operations also describe groups. The space of all rotations, for example, is the space $SO(2)$, a Lie group. Three other

important groups that describe operations in image space are the following:

$$\begin{aligned}
\text{Horizontal Scaling}_h(M) &= \left\{ \left(\begin{bmatrix} h & 0 \\ 0 & 1 \end{bmatrix} \mathbf{v}, c \right) \;\middle|\; (\mathbf{v}, c) \in M \right\} \\
\text{Vertical Scaling}_r(M) &= \left\{ \left(\begin{bmatrix} 1 & 0 \\ 0 & r \end{bmatrix} \mathbf{v}, c \right) \;\middle|\; (\mathbf{v}, c) \in M \right\} \\
\text{Shear}_s(M) &= \left\{ \left(\begin{bmatrix} 1 & s \\ 0 & 1 \end{bmatrix} \mathbf{v}, c \right) \;\middle|\; (\mathbf{v}, c) \in M \right\}.
\end{aligned}$$

Finally, we also have the group of all invertible linear transformations.

$$\text{Transformation}_{a,b,c,d}(M) = \left\{ \left(\begin{bmatrix} a & b \\ c & d \end{bmatrix} \mathbf{v}, c \right) \;\middle|\; ac - bd \neq 0, (\mathbf{v}, c) \in M \right\}.$$

It is easy to verify all these form groups under matrix multiplication. The operation labeled "Transformation" is the general linear group of order 2, $GL(2)$. Amazingly, any element of $GL(2)$ can be decomposed into the product of a rotation, a shear, a horizontal scaling, and a vertical scaling. To see this, we quickly introduce QR-decomposition.

2.7 QR-Decomposition

Let A be any non-singular linear transformation. Let $\mathbf{a}_1, \ldots, \mathbf{a}_n$ be the columns of A, which are linearly independent. By the Gram-Schmidt method found in Lax (1997), we know that we can find an orthonormal basis $\mathbf{x}_1, \ldots, \mathbf{x}_n$, such that $\mathbf{x}_i$ is a linear combination of the vectors $\mathbf{a}_1, \ldots, \mathbf{a}_i$. Suppose

$$\begin{aligned}
\mathbf{x}_1 &= r_{11}\mathbf{a}_1 \\
\mathbf{x}_2 &= r_{22}\mathbf{a}_2 + r_{12}\mathbf{a}_1 \\
\mathbf{x}_3 &= r_{33}\mathbf{a}_3 + r_{23}\mathbf{a}_2 + r_{13}\mathbf{a}_1 \\
&\vdots \\
\mathbf{x}_n &= r_{nn}\mathbf{a}_n + r_{(n-1)n}\mathbf{a}_{n-1} + \cdots + r_{1n}\mathbf{a}_1
\end{aligned}$$

Now let $Q = [\mathbf{x}_1 \cdots \mathbf{x}_n]$, and let $R^* = [r_{ij}]$ with r_{ij} defined above for $i \geq j$, and $r_{ij} = 0$ otherwise. Notice that R^* is an upper triangular matrix, and therefore its inverse, R, is also upper triangular. Then we see that $Q = AR^*$, and therefore $A = QR$, where Q is an orthonormal matrix, and R is an upper triangular matrix.

In the two dimensional case, we see that this takes a element of $GL(2)$ and writes it as a rotation times an upper triangular matrix. But a general two-dimensional upper triangular matrix is

$$\begin{bmatrix} h & sr \\ 0 & r \end{bmatrix} = \begin{bmatrix} 1 & s \\ 0 & 1 \end{bmatrix} \begin{bmatrix} h & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & r \end{bmatrix}$$

Therefore, an element A of $GL(2)$ can be written as the product of a rotation, a shear, and horizontal scaling, and vertical scaling.

Chapter 3

Space of Images

We will now consider putting Euclidean and differential structures on the space of color images $\mathcal{C}(m,n,p)$. Take a color image $C \in \mathcal{C}(m,n,p)$. We define the *standard norm* of C (written $||C||$) to be

$$||C|| = \sqrt{\sum_{ijk}(c_{ijk})^2}.$$

It is easily verified that this norm satisfies the properties $||C|| > 0$ for $C \neq 0$, $||C|| = 0$ for $C = 0$, $||\alpha C|| = |\alpha|\,||C||$ for real α, and $||C + D|| \leq ||C|| + ||D||$ for images C and D.

We can now define a curve in the space of color images. Consider $F : [0,1] \to \mathcal{C}$, a function from the real line onto the space of color images. We say F is *continuous at a given point* t_0 if, given some $\epsilon > 0$, there exists a δ such that for all $|x| < \delta$,

$$||F(t_0 + x) - F(t_0)|| < \epsilon$$

We say F is *continuous* if it is continuous at every point t.

A *line* in the space of color images is defined by images A and B, such that

$$F(t) = At + B$$

Theorem 3.1. *Given any two images C and C^*, there is exactly one linear function F such that $F(0) = C$ and $F(1) = C^*$.*

The ijkth pixel of C is some real number c_{ijk}, and we define c^*_{ijk} similarly. We know there is exactly one linear function $f_{ijk}(t)$ such that $f_{ijk}(0) = c_{ijk}$

and $f_{ijk}(1) = c^*_{ijk}$. Let the ijkth entry of $F(t)$ be defined by $f_{ijk}(t)$. Then, it is easily verified $F(0) = C$ and $F(1) = C^*$. Furthermore, the uniqueness of $F(t)$ follows from the uniqueness of each $f_{ijk}(t)$. □

We can now define the derivative for color spaces. The *derivative* of $F(t)$ is the image

$$F'(t) = \lim_{h \to 0} \frac{F(t+h) - F(t)}{h}$$

Given an unknown curve in $\mathcal{C}$ and the derivative at a point, we can approximate the curve at that point with a line.

3.1 Application

For a photograph that is yellowing or fading over time, using the linear approximation, we can approximate how the photograph would have looked in the past. For example, if examine the picture at $t = 20$ and $t = 21$, we can extrapolate the changes between these two time periods to find what the picture looked like at $t = 0$. Look at Figure 3.1 for an example.

Given three data points (say $t = 20.5$ in addition to $t = 20$ and $t = 21$), we could make a quadratic approximation. However, it may be that such an approximation would not work as well as the linear one, depending on the physical realities of the deterioration of the picture.

Figure 3.1: Given two instances, a yellow and faded image can be restored using linear approximation

Chapter 4

Covariance Matrix

4.1 Defining the Covariance Matrix

We can account mathematically for the effects of rotation using a method found in Gonzalez and Woods (2002) . Suppose each pixel of an image is given by a k-dimensional vector $\mathbf{x}$. We can then treat $\mathbf{x}$ as being a vector picked from the population of vectors, with expected value $\mathbf{m_x}$. We let $C_\mathbf{x}$ be the matrix of covariances between each pair of elements of $\mathbf{x}$, which is given by

$$C_\mathbf{x} = E\{(\mathbf{x} - \mathbf{m_x})(\mathbf{x} - \mathbf{m_x})^T\}.$$

To estimate $\mathbf{m_x}$, we can average over a collection of K values for $\mathbf{x}$:

$$\mathbf{m_x} = \frac{1}{K}\sum_{k=1}^{K} \mathbf{x}_k.$$

Combining these two equations, we see

$$C_\mathbf{x} = \frac{1}{K}\sum_{k=1}^{K} \mathbf{x}_k\mathbf{x}_k^T - \mathbf{m_x}\mathbf{m_x}^T.$$

4.2 Image Alignment

Given a set of pixels that defines a region of the image, we can assign each pixel a vector $\mathbf{x}$ in $\mathbb{R}^2$, and then find the covariance matrix over this set of pixels. Because the covariance matrix is a real symmetric matrix, it has a set of n orthonormal eigenvectors. These eigenvectors will tell us how to rotate the image so that the vectors in the given region are decorrelated.

For example, take Figure 4.1. A simple computer program we wrote for image alignment was given two smiley images, rotated differently (the two images on the left). Using only the data of the locations of the black pixels, the computer program could automatically determine how to rotate the two images so the output in both cases was approximately the same (the two images on the right).

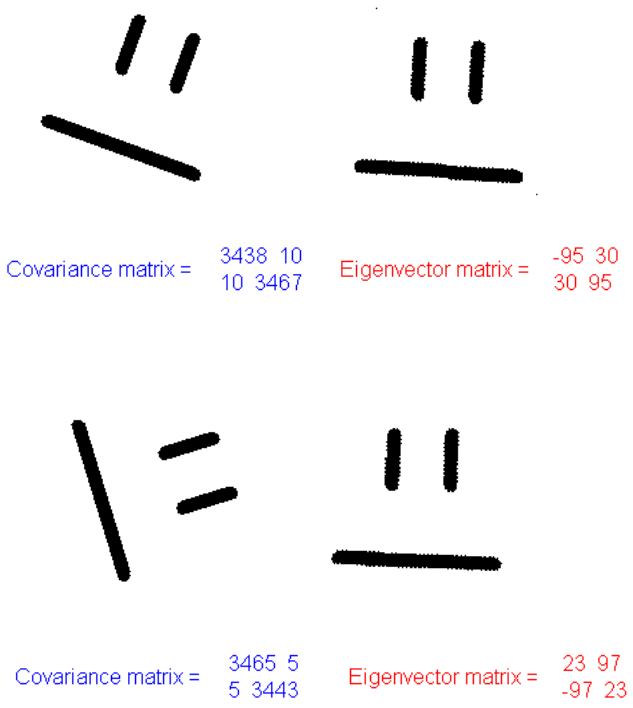


Figure 4.1: Two images rotated in accordance with decorrelating the region. Image generated by code from Appendix B.

To see how this works, let $A : \mathbb{R}^n \to \mathbb{R}^n$ be a rotation and reflection transformation, which is by definition an orthonormal matrix. If $\{\mathbf{x}_1, \ldots, \mathbf{x}_K\}$ are the position vectors of some region in the image, then $\{A\mathbf{x}_1, \ldots, A\mathbf{x}_K\}$ is the rotated version of the image (with possible reflection as well). Let $C_{\mathbf{x}}$ be the covariance matrix of the original image and let $C_{A\mathbf{x}}$ be the covariance matrix of the new image. First we will establish the following lemma.

Theorem 4.1. *For any matrix A, $C_{A\mathbf{x}} = AC_{\mathbf{x}}A^T$.*

Proof Let us use the coordinate axis so that $\mathbf{m}$ is the zero vector. Then we see, by definition,

$$C_{\mathbf{x}} = \frac{1}{K}\sum_{k=1}^{K}\mathbf{x}_k\mathbf{x}_k^T.$$

Therefore, if we apply A to each x, we see that the resulting covariance matrix $C_{A\mathbf{x}}$ is as follows:

$$\begin{aligned} C_{A\mathbf{x}} &= \frac{1}{K}\sum_{k=1}^{K}(A\mathbf{x}_k)(A\mathbf{x}_k)^T \\ &= \frac{1}{K}\sum_{k=1}^{K}(A\mathbf{x}_k)\mathbf{x}_k^T A^T \\ &= A\left(\frac{1}{k}\sum_{k=1}^{K}\mathbf{x}_k\mathbf{x}_k^T\right)A^T \\ &= AC_{\mathbf{x}}A^T, \end{aligned}$$

as desired. □

Notice that this allows very fast computation of $C_{A\mathbf{x}}$ given $C_{\mathbf{x}}$ and A. It also implies the next theorem.

Theorem 4.2. *For an orthonormal matrix A, the eigenvalues of $C_{\mathbf{x}}$ and $C_{A\mathbf{x}}$ are the same. Furthermore, if $\mathbf{v}$ is an eigenvector of $C_{\mathbf{x}}$ corresponding to eigenvalue λ, then $A\mathbf{v}$ is an eigenvector of $C_{A\mathbf{x}}$ corresponding to λ.*

Proof Let λ be an eigenvalue of $C_{\mathbf{x}}$ and $\mathbf{v}$ the corresponding eigenvector. Then we know $C_{\mathbf{x}}\mathbf{v} = \lambda\mathbf{v}$. From Theorem 4.1, we know that $C_{A\mathbf{x}} = AC_{\mathbf{x}}A^T$. From Lax (1997), we know that $A^T = A^{-1}$, so this implies $C_{A\mathbf{x}}A = AC_{\mathbf{x}}$. This means, $C_{A\mathbf{x}}A\mathbf{v} = AC_{\mathbf{x}}\mathbf{v}$

$$\begin{aligned} C_{A\mathbf{x}}A &= AC_{\mathbf{x}} \\ C_{A\mathbf{x}}A\mathbf{v} &= AC_{\mathbf{x}}\mathbf{v} \\ C_{A\mathbf{x}}A\mathbf{v} &= A\lambda\mathbf{v} \\ C_{A\mathbf{x}}A\mathbf{v} &= \lambda A\mathbf{v} \end{aligned}$$

This means that $A\mathbf{v}$ is a eigenvector of $C_{A\mathbf{x}}$ associated with eigenvalue λ. This proves the theorem. □

This allows us to determine A given only $C_{\mathbf{x}}$ and $C_{A\mathbf{x}}$.

Theorem 4.3. *For an orthonormal matrix A and an orthonormal basis of eigenvectors $\mathbf{v}_1, \ldots, \mathbf{v}_n$ of $C_{\mathbf{x}}$, there exists an orthonormal basis of eigenvectors of $C_{A\mathbf{x}}$, $\mathbf{u}_1, \ldots, \mathbf{u}_n$, such that $[\mathbf{u}_1 \cdots \mathbf{u}_n][\mathbf{v}_1 \cdots \mathbf{v}_n]^T = A$.*

Proof Let $[\mathbf{u}_1 \cdots \mathbf{u}_n] = [A\mathbf{v}_1 \cdots A\mathbf{v}_n] = A[\mathbf{v}_1 \cdots \mathbf{v}_n]$. We know from Theorem 4.2 that each $\mathbf{u}_i$, defined in this way, is an eigenvector of $C_{A\mathbf{x}}$. Since the $[\mathbf{v}_1 \cdots \mathbf{v}_n]$ is an orthogonal matrix, $[\mathbf{v}_1 \cdots \mathbf{v}_n]^T = [\mathbf{v}_1 \cdots \mathbf{v}_n]^{-1}$. So we see

$$\begin{aligned} [\mathbf{u}_1 \cdots \mathbf{u}_n] &= A[\mathbf{v}_1 \cdots \mathbf{v}_n] \\ [\mathbf{u}_1 \cdots \mathbf{u}_n][\mathbf{v}_1 \cdots \mathbf{v}_n]^T &= A \end{aligned}$$

as desired. □

Given two images where one is a rotation of the other, we can find the angle of rotation. We simply compute the covariance matrix of the first image $C_{\mathbf{x}}$, compute the covariance matrix of the second image $C_{A\mathbf{x}}$, and find the orthogonal matrix of eigenvectors for both. We then take the second eigenvector matrix and multiply by the transpose of the first eigenvector matrix, and get rotation matrix A.

4.3 Color Alignment

In Chapter 2, we developed the space of colors as a vector space. One way to transform the colors of an image is to take the colors, represented as vectors, and transform them using a matrix A that is an orthonormal matrix (which rotates and reflects the vector in color space). Given the image and the transformed image, we can use Theorem 4.3 to find the matrix A that was used to transform one image into the other. Notice further if A is the composition of a rotation and a uniform scaling, then this technique can still be used to align the images. Finding the coefficient of uniform scaling is a simple task once the images are aligned.

Chapter 5

Applications of the Perron-Frobenius Theorem

5.1 Introduction

In this section, we explore applications of the Perron-Frobenius theorem to image processing. The Perron theorem states that any positive matrix G has, as its eigenvalue of largest absolute value, a positive eigenvalue, and a corresponding strictly positive eigenvector. The Perron-Frobenius theorem says the same is true for irreducible non-negative matrices G, though the positive eigenvalue in question this case need not be strictly the largest eigenvalue in absolute value. By irreducible, we mean a matrix A where there is no permutation matrix S such that

$$SAS' = \begin{pmatrix} B & 0 \\ C & D \end{pmatrix}$$

for square matrices B and D, and matrix C. Clearly, a strictly positive matrix is irreducible, as is a matrix with strictly positive entries everywhere but the diagonal. We can now use this theorem to help us guarantee solutions to certain image processing models.

5.2 Color Clashing

To the human eye, some color combinations look better than others. For example, one might like the look of blue and red together, where orange and teal just do not belong. While this is certainly subjective, survey data

could be used to creating an index quantifying how well two colors go together on a scale from 0 to 1 (1 being the that two colors go together the best, 0 being they clash horribly). Given a set of n colors, let the index value between the ith and jth colors be g_{ij}.

It is natural to want to find the best combination of colors. Of course, just one color does not clash at all, but it is fairly boring. Having a variety of colors is also important to a good picture. Therefore, let λ be the importance of having a variety of colors.

We can model this in matrix form. Let $\mathbf{v}$ be the vector of color intensities with entries for each of the i colors. Given a color intensity v_i, we want the benefit from having that color intensity (λv_i) to be greater than the color clash. If we let $G = [g_{ij}]$, then we can represent this problem in matrix form:

$$\lambda \mathbf{v} \geq G\mathbf{v}. \tag{5.1}$$

Here, $G\mathbf{v}$ can be seen as being a vector whose ith entry represents how much color i clashes with the other colors.

When can we find such a vector $\mathbf{v}$? The Perron-Frobenius theorem can help! Let $\lambda_{\max}$ be the largest eigenvalue of G. We see that G in general is a nonnegative, irreducible matrix (it is only reducible in unlikely, degenerate cases). Therefore, Perron-Frobenius tells us that there exists at least one solution to Equation 5.1 with equality, where $\lambda = \lambda_{\max}$ is the largest eigenvalue (which Perron-Frobenius tells us will be positive) and $\mathbf{v}$ is the associated positive eigenvector. Furthermore, we can use Gersgorin's theorem to get an upper bound on $\lambda_{\max}$. Gersgorin's theorem states all the eigenvalues of G lie on unit discs on the complex plane that follow the inequality $|\lambda - g_{ii}| \leq \sum_{j \neq i} g_{ij}$ for all $i = 1, \ldots n$. Since $g_{ii} = 0$, all these discs are centered at the origin. Therefore, we see that

$$\lambda_{\max} \leq \max_i \sum_{j \neq i} g_{ij}.$$

We can therefore say that for $\lambda \geq \max_i \sum_{j \neq i} g_{ij}$, we can find a solution to Equation 5.1.

Does there exist a $\lambda < \lambda_{\max}$ such that Equation 5.1 has a solution with that particular λ? As we see from the following theorem, the answer is no.

Theorem 5.1. *Given a non-negative, irreducible matrix A with largest eigenvalue λ_{max}, $\lambda = \lambda_{max}$ is the smallest positive solution to*

$$A\mathbf{x} \leq \lambda \mathbf{x}$$

for any nonnegative vector $\mathbf{x}$.

Proof Let $\beta < \lambda_{\max}$ be a solution to $A\mathbf{y} \leq \beta\mathbf{y}$ for some non-negative vector $\mathbf{y}$. By the Perron-Frobenius theorem, A can have no eigenvalues such that the corresponding eigenvector is non-negative other than $\lambda_{\max}$, so the inequality must be strict. Therefore, $A\mathbf{y} < \beta\mathbf{y}$. Also by the Perron-Frobenius theorem, we have that $A\mathbf{x} = \lambda_{\max}\mathbf{x}$ for a nonnegative vector $\mathbf{x}$. Since multiplication by a constant does not matter, let us normalize the vector $\mathbf{x}$ such that $\mathbf{x} \leq \mathbf{y}$. Then we see that $(A - \beta I)\mathbf{y} < 0 = (A - \lambda_{\max} I)\mathbf{x}$ and yet $(A - \beta I)\mathbf{y} \geq (A - \beta I)\mathbf{x} \geq (A - \lambda_{\max} I)\mathbf{x}$, a contradiction. Therefore, we see that there is no $\beta < \lambda_{\max}$, and therefore $\lambda \geq \lambda_{\max}$ for all λ. □

Therefore, $\lambda_{\max}$ is the smallest value of λ that satisfies Equation 5.1. For $\lambda > \lambda_{\max}$, the same eigenvector $\mathbf{x}$ corresponding to $\lambda_{\max}$ will still satisfy Equation 5.1. Therefore, $\mathbf{x}$ is the best combination of the set of n colors in regards to clashing information G.

5.3 Color Variety

Suppose someone is designing a webpage using a certain set of colors. In what ratios should the colors be used to make the webpage as colorful as possible? To model this situation, again assume n colors are under consideration. Some shades of color are very close to each other, while others are radically different. Let g_{ij} be some measure of how far the ith color is from the jth color. Then $G = [g_{ij}]$ is a matrix that represents these distances. Let $\mathbf{v}$ be a vector of the amount of each color. Then, ideally, each color will contribute enough variety to justify its presence. To model this, we require $\mathbf{v}$ to satisfy

$$\lambda\mathbf{v} \leq G\mathbf{v}. \tag{5.2}$$

Notice this is identical to Equation 5.1, except that the inequality is reversed. We see G meets the same requirements as before, so again we can apply the Perron-Frobenius Theorem to determine that G must have a positive eigenvalue $\lambda_{\max}$ with a corresponding positive eigenvector $\mathbf{v}$, and $\lambda_{\max}$ and $\mathbf{v}$ satisfy Equation 5.2. Clearly, any $\lambda \leq \lambda_{\max}$ will also satisfy it with the same $\mathbf{v}$. Can the inequality be satisified with $\lambda > \lambda_{\max}$? Just as before, the answer is no.

Theorem 5.2. *Given a non-negative, irreducible matrix A with largest eigenvalue λ_{max}, $\lambda = \lambda_{max}$ is the largest positive solution to*

$$A\mathbf{x} \geq \lambda\mathbf{x}$$

for any non-negative vector $\mathbf{x}$.

Proof Let $\beta > \lambda_{\max}$ be a solution to $A\mathbf{y} \geq \beta\mathbf{y}$ for some non-negative vector $\mathbf{y}$. By the Perron-Frobenius theorem, A can have no eigenvalues such that the corresponding eigenvector is non-negative other than $\lambda_{\max}$, so the inequality must be strict. Therefore, $A\mathbf{y} > \beta\mathbf{y}$. Also by the Perron-Frobenius theorem, we have that $A\mathbf{x} = \lambda_{\max}\mathbf{x}$ for a nonnegative vector $\mathbf{x}$. Since multiplication by a constant does not matter, let us normalize the vector $\mathbf{x}$ such that $\mathbf{x} \geq \mathbf{y}$. Then we see that $(A - \beta I)\mathbf{y} > 0 = (A - \lambda_{\max} I)\mathbf{x}$ and yet $(A - \beta I)\mathbf{y} \leq (A - \beta I)\mathbf{x} \leq (A - \lambda_{\max} I)\mathbf{x}$, a contradiction. Therefore, we see that there is no $\beta > \lambda_{\max}$, and therefore $\lambda \leq \lambda_{\max}$ for all λ. □

So despite reversing the inequality, we see that $\lambda_{\max}$ is still the best possible weight to use when trying to maximize the variety of colors.

5.4 Example

To do a specific example, we need to find some metric on color space to determine the matrix G. Any metric will work, but metric that corresponds more closely to how humans see colors will work better. We will use the one described in Chapter 2, $m(c_1, c_2)$, based one based on HSL-color space. Given colors $c_1, \ldots, c_n$, then we get matrix G made up of entries

$$g_{ij} = m(c_i, c_j).$$

Then finding an optimal combination of colors according to this model is as easy as finding the positive eigenvector associated with G. The results of an example with $n = 3$ is given in Figure 5.1.

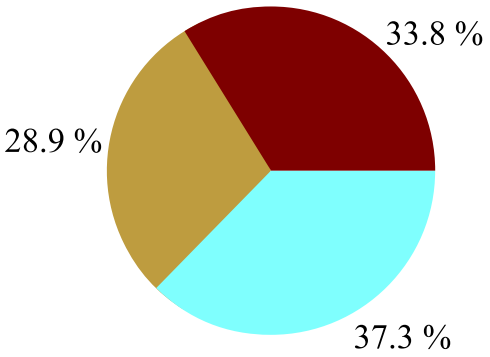


Figure 5.1: The best combination of the three colors shown above according to this model. See Appendix A.

Chapter 6

Applications of Lie Groups

6.1 Introduction

Let G be a group of matrices under multiplication. A function $f(x) : \mathbb{A} \rightarrow K$ (for vector space $\mathbb{A}$ with field K) is invariant under the group G if

$$f(g\langle x\rangle) = f(x) \qquad \forall x \in \mathbb{A} \qquad \forall g \in G$$

Another way to look at a functional invariant is as a set of points in $\mathbb{A}$ that is closed under the action of the group G. Finding invariant functions such as these has important applications to image processing. For example, suppose one wanted a computer to be able to recognize an object from a video feed. The object may appear different due to lighting conditions, positioning relative to the camera, etc. Since changing conditions can be modelled through the use of transformation groups, functions that are invariant under the actions of transformation groups are useful.

How can we find invariant functions f on a given group G? Putting a differential structure on the group can help. Define $M(t_1, \ldots, t_k)$ to be

$$M(t_1, \ldots, t_k) = e^{t_1 X_1 + \cdots + t_k X_K}$$

for matrices $X_1, \ldots, X_k$. $M(t_1, \ldots, t_k)$ is called a *k-parameter subgroup* of G. For example, for $G = SO(2)$, we can define a one-parameter subgroup to be

$$M(t) = exp\left(t \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix}\right) = \begin{bmatrix} \cos t & -\sin t \\ \sin t & \cos t \end{bmatrix}$$

Given any group element, we can now create a homeomorphism between a neighborhood of the group element and a neighborhood in $\mathbb{R}^k$, simply by

using the map M. We can even define k differential operators

$$\frac{d}{dt_i} f(M(t_1, \ldots, t_k)x),$$

one for each i, given any differentiable $f : \mathbb{R}^k \to \mathbb{R}$.

If we require f to be invariant, then $f(Mx) = f(x)$ for all $M(t_1, \ldots, t_k)$, and therefore

$$\frac{d}{dt_i} f(Mx) = 0. \tag{6.1}$$

Solving this system of differential equations gives f. In general, this method will find a set of all the independent functional invariants of a group G. To see how this works, consider $G = SO(2)$ with the corresponding $M(t)$ described above. Then Equation 6.1 becomes

$$\frac{d}{dt} f \begin{pmatrix} x_1 \cos t - x_2 \sin t \\ x_1 \sin t + x_2 \cos t \end{pmatrix} = 0.$$

To solve this equation, we let $y_1 = x_1 \cos t - x_2 \sin t$ and $y_2 = x_1 \sin t + x_2 \cos t$. Then the equation becomes $\frac{d}{dt} f(y_1, y_2) = 0$. By the chain rule, $\frac{\partial f}{\partial y_1}\frac{dy_1}{dt} + \frac{\partial f}{\partial y_2}\frac{dy_2}{dt} = 0$. This means that

$$\left(\frac{\partial f}{\partial y_1}, \frac{\partial f}{\partial y_2}\right) \cdot \left(\frac{dy_1}{dt}, \frac{dy_2}{dt}\right) = 0$$

holds, where $* \cdot *$ indicates the usual dot product. We know that two vectors have a dot product of zero if and only if they are perpendicular. The vector perpendicular to $\left(\frac{dy_1}{dt}, \frac{dy_2}{dt}\right)$ is $\left(\frac{dy_2}{dt}, -\frac{dy_1}{dt}\right)$. Therefore,

$$\begin{aligned}
\begin{bmatrix} \frac{\partial f}{\partial y_1} \\ \frac{\partial f}{\partial y_2} \end{bmatrix} &= k \begin{bmatrix} \frac{dy_2}{dt} \\ -\frac{dy_1}{dt} \end{bmatrix} \\
\begin{bmatrix} \frac{\partial f}{\partial y_1} \\ \frac{\partial f}{\partial y_2} \end{bmatrix} &= k \begin{bmatrix} x_1 \cos t - x_2 \sin t \\ x_1 \sin t + x_2 \cos t \end{bmatrix} \\
\begin{bmatrix} \frac{\partial f}{\partial y_1} \\ \frac{\partial f}{\partial y_2} \end{bmatrix} &= k \begin{bmatrix} y_1 \\ y_2 \end{bmatrix}.
\end{aligned}$$

Now we have $\frac{\partial}{\partial y_1} f(y_1, y_2) = k y_1$, and we see that $f(y_1, y_2) = k y_1^2/2 + g(y_2)$ for some function g. We also have $\frac{\partial}{\partial y_2} f(y_1, y_2) = k y_2$, so $\frac{d}{dy_2} g(y_2) =$

ky_2, so $g(y_2) = ky_2^2/2$. Arbitrarily letting $k = 2$, we see that $f(y_1, y_2) = y_1^2 + y_2^2$ solves Equation 6.1.

This tells us that $h(x_1^2 + x_2^2)$ is invariant under the rotation group $SO(2)$ for any differentiable function h. For example, the length of a vector $\sqrt{x_1^2 + x_2^2}$ is invariant under $SO(2)$.

In many cases, this is a very nice way of finding functional invariants. For example, for the group of horizontal scalings, $f(x_1, x_2) = x_1$ is an invariant, and this is also an invariant for the group of shears. The group of vertical scalings has invariant $f(x_1, x_2) = x_2$. Finally, the group of uniform scalings has invariant $f(x_1, x_2) = x_1/x_2$.

It appears that each important subgroup of $GL(2)$ has exactly one linear invariant. This is due to the dimension of the underlying Lie Algebra of each Lie Group. There is a result (see Lenz et al. (2003)) that states that if the dimension of the space $\mathbb{A}$ we are working in is N, and the dimension of the Lie algebra of G is k, then there are $N - k$ independent functional invariants under G. In our examples, we are working in $\mathbb{R}^2$, so $N = 2$, and each is a one-parameter group, $k = 1$, which implies there is $2 - 1 = 1$ independent functional invariant.

6.2 Illumination Invariants

In Lenz et al. (2003) and Lenz and Bui (2003), the techniques from the previous section have been used to find functional invariants of images under different illumination. This helps make it possible to describe a scene visually in a way that is independent of the illumination present. Using mathematical models of the physical process of light reflection and absorbtion, they develop matrix Lie groups that describe the effects of illumination on an image. Then using the above process, they set up a system of differential equations, and they used Maple to solve the equations and determine the set of invariant functions.

6.3 Blurring

Given these results in finding illumination invariants, we decided to try to apply similar techiques to image blurring. Given two images, a clear one and a blurred one, how might we verify that the first is just a blurry version of the second? One way would be to find a function f that is invariant under blurring. Therefore, by evaluating f on both images and getting the

same value, we have some evidence that the two images are in fact of the same object. We develop five functional invariants under image blurring, and conjecture that there exist infinitely many invariants of a similar type.

The type of blurring we study is Gaussian blurring, because it is both widely used in image processing and lends itself to a group-like structure.

Let $g_t(x)$ be the one-dimensional Gaussian function

$$g_t(x) = \frac{1}{\sqrt{2\pi t}} e^{\frac{-x^2}{2t}}.$$

The most natural realm in which to work with Gaussian blurring is one-dimensional continuous, grey scale images. To further simplify the problem, assume the image extends infinitely in both directions, and therefore f is defined on all of $\mathbb{R}$. Finally, let $\int_{-\infty}^{\infty} e^{|sx|} f dx$ be finite and converge absolutely for some s. Let f_t, a Gaussian blur of the image f_0, be defined by

$$f_t(x) = \int_{-\infty}^{\infty} g_t(s-x) f_0(s) ds.$$

(Let the Gaussian blur for $t = 0$ be the identity function.) Notice this is simply the convolution of $f_0(s)$ with the Gaussian function, denoted $f_0 * g_t$.

We can verify that the Gaussian blur with parameter t_1 combined with a Gaussian blur with parameter t_2 is simply a Gaussian blur with parameter $t_1 + t_2$. Let $(f_{t_1})_{t_2}$ be a Gaussian blur of f_{t_1} with parameter t_2. Then we see

$$\begin{aligned}
(f_{t_1})_{t_2}(x) &= \int_{-\infty}^{\infty} g_{t_2}(s-x) f_{t_1}(s) ds \\
&= \int_{-\infty}^{\infty} g_{t_2}(s-x) \int_{-\infty}^{\infty} g_{t_1}(u-s) f_0(u) du ds \\
&= \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} g_{t_2}(s-x) g_{t_1}(u-s) f_0(u) du ds \\
&= \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} g_{t_2}(s-x) g_{t_1}(u-s) f_0(u) ds du \\
&= \int_{-\infty}^{\infty} f_0(u) \int_{-\infty}^{\infty} g_{t_2}(t-x) g_{t_1}(u-s) ds du \\
&= \int_{-\infty}^{\infty} f_0(s) \int_{-\infty}^{\infty} g_{t_2}(s) g_{t_1}(u-x-s) ds du.
\end{aligned}$$

We now take advantage of the fact that the convolution of two Gaussian functions is a Gaussian function, and therefore

$$\int_{-\infty}^{\infty} g_{t_2}(s) g_{t_1}(u-x-s) ds = g_{t_1+t_2}(u-x).$$

We then have

$$(f_{t_1})_{t_2}(x) = \int_{-\infty}^{\infty} g_{t_1+t_2}(u-x) f_0(u) du.$$

Thus, the operation of Gaussian blurring is closed in this sense, and it is easy to see that the operation commutes. We also have an identity element with parameter $t = 0$. However, it does not form a group because there are no inverses. Therefore, we have a monoid. We can determine invariant functions by putting a differential structure on the monoid, and, as we will see in the next section, there is a very natural one with parameter t.

6.4 Invariant Function under Gaussian Blurs

Let h be a function from infinite, one-dimensional, grey images to $\mathbb{R}$. We would like h to be constant under the action of of Gaussian blurs, so $h(f_t) = h(f_0)$ for all t. As in the case of rotation matrices, we can define a differential operator

$$\frac{d}{dt} h\left(\int_{-\infty}^{\infty} g_t(s-x) f_0(s) ds \right).$$

Invariants are functions h that satisfy

$$\frac{d}{dt} h\left(\int_{-\infty}^{\infty} g_t(s-x) f_0(s) ds \right) = 0. \tag{6.2}$$

One h that satisfies Equation 6.2 is

$$h_0(f) = \int_{-\infty}^{\infty} f(x) dx.$$

This is analogous to a "total mass" function, and the fact that it is invariant under blurring means the total mass of the image does not change. It is easy to verify that h_0 is in fact a functional invariant under blurring.

$$\begin{aligned}
\frac{d}{dt} h_0\left(\int_{-\infty}^{\infty} g_t(s-x) f_0(s) ds \right) &= \frac{d}{dt} \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} g_t(s-x) f_0(s) ds dx \\
&= \frac{d}{dt} \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} g_t(s-x) f_0(s) dx ds \\
&= \frac{d}{dt} \int_{-\infty}^{\infty} f_0(s) \int_{-\infty}^{\infty} g_t(s-x) dx ds \\
&= \frac{d}{dt} \int_{-\infty}^{\infty} f_0(s)(1) ds \\
&= 0
\end{aligned}$$

Thus, h_0 is a functional invariant.

Another invariant function is given by

$$h_1(f) = \int_{-\infty}^{\infty} xf(x)dx.$$

This is analogous to a "center of mass" function. Again, it is easy to verify this is a functional invariant.

$$\begin{aligned}
\frac{d}{dt}h_1\left(\int_{-\infty}^{\infty} g_t(s-x)f_0(s)ds\right) &= \frac{d}{dt}\int_{-\infty}^{\infty} x\int_{-\infty}^{\infty} g_t(s-x)f_0(s)dsdx \\
&= \frac{d}{dt}\int_{-\infty}^{\infty}\int_{-\infty}^{\infty} xg_t(s-x)f_0(s)dxds \\
&= \frac{d}{dt}\int_{-\infty}^{\infty} f_0(s)\int_{-\infty}^{\infty} xg_t(s-x)dxds \\
&= \frac{d}{dt}\int_{-\infty}^{\infty} f_0(s)\int_{-\infty}^{\infty} (s-x)g_t(x)dxds \\
&= \frac{d}{dt}\int_{-\infty}^{\infty} f_0(s)sds - \int_{-\infty}^{\infty} f_0(s)\int_{-\infty}^{\infty} (x)g_t(x)dxds \\
&= \frac{d}{dt}\int_{-\infty}^{\infty} f_0(s)(s)ds - 0 \\
&= 0.
\end{aligned}$$

Define h_i analogously to h_0 and h_1. Based upon the previous two results, it seems as if h_i might be an invariant under Gaussian blurring for all i. However, when one computes the case of h_2, the result is

$$\frac{d}{dt}h_2\left(\int_{-\infty}^{\infty} g_t(s-x)f_0(s)ds\right) = h_0(f_0).$$

This means the h_2 case grows proportional to the constant $h_0(f_0)$. Computing the case of h_3 yields

$$\frac{d}{dt}h_3\left(\int_{-\infty}^{\infty} g_t(s-x)f(s)ds\right) = 3h_1(f_0).$$

It also grows as a constant. This allows use to create yet another invariant as a linear combination of these two. Let $i_0 = h_0$ and $i_1 = h_1$ be the first two invariants. Then the third invariant, i_2, is given by

$$\begin{aligned}
i_2(f) &= h_0(f)h_3(f) - 3h_1(f)h_2(f) \\
&= \left(\int_{-\infty}^{\infty} f(x)dx\right)\int_{-\infty}^{\infty} x^3f(x)dx - 3\left(\int_{-\infty}^{\infty} xf(x)dx\right)\int_{-\infty}^{\infty} x^2f(x)dx
\end{aligned}$$

Likewise, we compute

$$\frac{d}{dt}h_4\left(\int_{-\infty}^{\infty} g_t(s-x)f_0(s)ds\right) = 6h_2(f_0) + 6th_0(f_0).$$

With this we can make yet another invariant. We notice

$$\begin{aligned}\frac{d}{dt}h_0(f_t)h_4(f_t) - 3[h_2(f_t)]^2 &= h_0(f_t)\frac{d}{dt}h_4(f_t) - 3\frac{d}{dt}[h_2(f_t)]^2 \\ &= h_0\frac{d}{dt}h_4(f_t) - 6[h_2(f_t)]\left(\frac{d}{dt}h_2(f_t)\right) \\ &= h_0\frac{d}{dt}h_4(f_t) - 6[h_2(f_0) + th_0]h_0 \\ &= 0\end{aligned}$$

So we have a fourth invariant,

$$i_3(f) = h_0(f)h_4(f) - 3[h_2(f)]^2.$$

Likewise, one can verify that the following is also an invariant:

$$i_4(f) = 3h_1(f)h_5(f) - 5[h_3(f)]^2.$$

It appears that these invariants are independent, and that using h_j for larger and larger j, one could construct any number of such invariants, leading to the following conjecture.

Conjecture 6.1. *There are an infinite number of functionally independent invariants under Gaussian bluring of an infinite continuous one-dimensional image of the form*

$$\alpha(h_0, h_1, \ldots, h_k)$$

for some k and some polynomial $\alpha : \mathbb{R}^k \to \mathbb{R}$.

6.5 Application of Gaussian Blurring Invariants

The results above deal with a very abstract concept of infinite, contiuous, grey scale, one-dimensional images. However, discrete images containing many pixels can be closely approximated with a contiuous image. Also, a two-dimensional image can be viewed as a stack of one dimensional images. Implementation of a Gaussian blur in two dimensions is often done by first doing a one-dimensional, horizontal blur for each row of pixels, and

then doing a vertical blur for each column. Therefore, the one-dimensional case is applicable to two-dimensional images. Finally, these techniques can be applied to RGB color images by simply applying them separately to each one-dimensional color channel. Applying some combination of discrete approximations of the invariants given above could be a useful test in determining whether one image is simply a blurred version of another.

Chapter 7

Conclusion and Future Work

Currently, one of the most important questions in image processing is how to recognize and analyze features of an image with as little human intervention as possible. To find simple algorithms that accomplish this task, often powerful tools from mathematics are needed.

For example, we know from Gonzalez and Woods (2002) that properties of the covariance matrix can be used to find invariants under rotations in both pixel space. In Chapter 4, we examined the theoretical underpinnings of this technique, and applied these techniques to color space. Similarly, invariants can be found for groups acting on images. Using results from lumination invariants as a model, we developed invariants under Gaussian blurring. Finally, in Chapter 5, we examined problems dealing with optimizing the choice of a combinations of colors, and we found using the Perron-Frobenius theorem that optimal solutions could be guaranteed and easily computed.

The main unfinished task in this thesis is a proof of Conjecture 6.1. A proof would consist of two parts:

- First, proving there exist invariants of the form $\alpha(h_0, \ldots, h_k)$ for infinitely many k. Such a proof would probably proceed by strong induction, using previous invariants to construct more and more.

- Second, proving that $\alpha(h_0, \ldots, h_{k_0})$ is functionally independent from $\alpha(h_0, \ldots, h_{k_1})$ for any $k_0 < k_1$. To prove this, one would first need to find a pair of images f_0 and f_1, such that the first invariant could not distinguish between them, but the second could. This would prove that the second is not the composition of the first and some real continuous function. Likewise, another pair would need to be found, f_2

and f_3, to prove the first is not the composition of the second and some continuous function.

If the first step of the proof does involve construction, then not only will it prove the conjecture, but it will also tell us how to find arbitrarily many invariants. That would be extremely useful in providing evidence that one picture is just a blur of another.

Appendix A

varietyExample.java

```
// varietyExample.java
//
// This java program takes as input some number of
// colors in HSL space, and then computes a matrix
// of their distance between each pair of colors.
// Finally, using the Jama matrix solving package,
// finds the eigenvectors of this matrix.  By
// identifying the positive eigenvector, one can
// compute the optimal combination of the inputed
// colors.


import java.io.*; // needed for input
import java.lang.Math;
import Jama.*;

class varietyExample
{

private static BufferedReader stdin = new BufferedReader(
        new InputStreamReader( System.in ) );


public static void main(String[] args)  throws IOException
        {


        println("How many colors?");
```

```
// n is the number of colors to be inputed
int n = getInt();

int[][] colorHSL = new int [n][3];
double[][] dcolorHSL = new double[n][3];



// This for-loop gets the HSL values for each of
// the n colors
for(int i = 0; i < n; i++)
        {
        println("Hue for color " + i + "?");
        colorHSL[i][0] = getInt();

        println("Saturation for color " + i + "?");
        colorHSL[i][1] = getInt();

        println("Luminace for color " + i + "?");
        colorHSL[i][2] = getInt();
        }


// This for-loop converts the HSL values into a form
// for computing the distances between colors
for(int i = 0; i < n; i++)
        {
        dcolorHSL[i][0] = convertHue(colorHSL[i][0]);
        dcolorHSL[i][1] = convertSat(colorHSL[i][1]);
        dcolorHSL[i][2] = convertLum(colorHSL[i][2]);
        }



// The next block of code creates a nxn array
// distances, whose ijth entry is the distance
// between colors i and j.  This array is then
// turned into matrix G.
double[][] distances = new double[n][n];

for(int i = 0; i < n; i++)
        for(int j = 0; j < n; j++)
                {
                distances[i][j]
                        = distance(dcolorHSL[i],
```

```
                                        dcolorHSL[j]);
                    }

        Matrix G = new Matrix(distances);

        println("G: ");

        G.print(n, n);

        // Using the Jama package, we create a new
        // matrix E consisting of Eigenvectors of G,
        // and then print E to the screen.
        EigenvalueDecomposition E =
            new EigenvalueDecomposition(G);

        Matrix eigs = E.getV();

        println("Eigenvectors: ");

        eigs.print(n, n);

        }


public static double convertHue(int hue)
        {
        return (2*Math.PI*hue)/255;
        }

public static double convertSat(int sat)
        {
        return ((double)sat)/255;
        }


public static double convertLum(int lum)
        {
        return (2*(double)lum)/255 - 1;
        }

public static double distance(double [] colori,
                              double [] colorj)
        {
        double hi = colori[0];
        double si = colori[1];
```

```
            double li = colori[2];

            double hj = colorj[0];
            double sj = colorj[1];
            double lj = colorj[2];

            double dx = (1 - Math.abs(li))*si*Math.cos(hi)
                    - (1 - Math.abs(lj))*sj*Math.cos(hj);

            double dy = (1 - Math.abs(li))*si*Math.sin(hi)
                    - (1 - Math.abs(lj))*sj*Math.sin(hj);

            double dz = li - lj;

            return Math.sqrt(dx*dx + dy*dy + dz*dz);
            }

    public static void print(String s)
            {
            System.out.print(s);
            }

    public static void println(String s)
            {
            System.out.println(s);
            }

    public static int getInt() throws IOException
            {
            String s = stdin.readLine();
            return Integer.parseInt( s);
            }


    }
```

Appendix B

rot.java

```
// rot.java
//
// Based on code by Ran Libeskind-Hadas
// and Zachary Dodds
//
// Takes image "rot1.gif" or "rot2.gif"
// (black and white images)
// and rotates them according to the
// eigenvectors of the covariance
// matrix.  If "rot1.gif" and "rot2.gif"
// are the same image, only rotations
// of each other, then this process
// rotates them to the same position!
// This program requires the Jama
// matrix solving package.


import java.io.*;
import Jama.*;
import java.applet.*;
import java.awt.*;
import java.awt.image.PixelGrabber;
import java.awt.image.BufferedImage;
import java.awt.image.ImageObserver;
import java.awt.event.*;
import java.net.URL;
import java.net.URLConnection;
import java.io.BufferedReader;
import java.io.InputStreamReader;
```

```
import java.lang.Thread;
import java.lang.Math;

public class rotation extends Applet
  implements ActionListener, ItemListener,
        KeyListener, Runnable
{

  Image image;              // off-screen buffer
  Graphics g;               // graphics tool

  Image smilie;             // Image to be rotated
  BufferedImage bsmilie;    // Buffered image

  int sleepTime = 25;       // 25 milsecs between updates
  int cycleNum = 0;         // cycles so far


  // these are AWT components in the applet
  //
  // AWT: java's Abstract Windowing Toolkit, is a library
  //      of graphical classes and methods -- for example,
  //      Button and Choice are two available classes.


  private Button bButton;
  private Button pauseButton;
  private Button startButton;
  private Choice choiceInput;
  private Color  sqColor;

    //Width and height of the java applet
  int width;
  int height;

    //Width and height of the image
  int wimage;
  int himage;

    //Used when determining whether
    //"rot1.gif" or "rot2.gif" is used
  String type;

    //Stores pixels color values
  int[] pixels;
```

```
    //Stores pixel color values
    //in two dimensional array
  int[][] newpixels;

    //Store pixel color values
    //after the image is changed
    //in some way
  int[] verynewpixels;

  double[][] data;

  int n;

    //Covariance matrix
  Matrix C;
    //Matrix of Eigenvectors
  Matrix eigs;



public void init()
 {

    // Create an off-screen image for drawing
    image = createImage(getSize().width, getSize().height);
    g = image.getGraphics();
    clear();


    // create each component and add it to the applet


    //Based on input parameter of the Java applet,
    //Decides whether to load rot1.gif or rot2.gif
        type = getParameter( "type" );

        if(type.equals("0"))
                smilie = getImage(getCodeBase(), "rot1.gif");
        if(type.equals("1"))
                smilie = getImage(getCodeBase(), "rot2.gif");



        wimage = smilie.getWidth(this);
```

```
    himage = smilie.getHeight(this);


// The pause button
pauseButton = new Button("Pause");
pauseButton.addActionListener(this);
pauseButton.addKeyListener(this);
add(pauseButton);

// The start buttton
startButton = new Button("Start");
startButton.addActionListener(this);
startButton.addKeyListener(this);
add(startButton);


width = getSize().width;
height = getSize().height;


bsmilie = new BufferedImage(wimage, himage,
                    BufferedImage.TYPE_INT_RGB);


Dimension d = size();

pixels = new int[wimage*himage];
newpixels = new int[wimage][himage];

for(int i = 0; i<(wimage*himage); i++)
            pixels[i] = 0;


// Takes the pixels of smilie and stores
// them in array pixels
PixelGrabber pg = new PixelGrabber
        (smilie, 0, 0, wimage, himage,
            pixels, 0, wimage);
    try {
        pg.grabPixels();
                    }
            catch (InterruptedException e) {
                    System.err.println
                    ("interrupted waiting for pixels!");
                    return;
```

```
                    }
        if ((pg.getStatus() &
                ImageObserver.ABORT) != 0) {
            System.err.println
                ("image fetch aborted or errored");
            return;
            }

        // data is the array that lists the black pixels
        // of image as two dimensional vectors.  This
        // initializes the array
        data = new double[200*200][2];

        for(int i = 0; i< (200*200); i++)
                {
                data[i][0] = 0;
                data[i][1] = 0;
                }

        // Sets pixels and verynewpixels the same
        for(int i = 0; i < wimage*himage; i++)
            verynewpixels[i] = pixels[i];


        }

void updateEnvrionment()
        {
        Dimension d = size();

        // Turns the dark pixels into vectors, and stores
        // the vectors in data
        getRegion(pixels);

        // Computes covariance matrix from vectors in data
        C = covariance(data);

        // Uses Jama package to compute eigenvector matrix eigs
        EigenvalueDecomposition E =
            new EigenvalueDecomposition(C);
        eigs = E.getV();


        double angle = 0;
```

```
            // Dramatic pause
            if(cycleNum > 150)
                {

                // Computes angle needed to rotate image
                // from eigenvector matrix eigs
                angle = Math.atan
                            (eigs.get(1, 0)/eigs.get(0, 0));

                // Rotates image stored in pixels by angle
                rotate(pixels, angle);
                }
            }


        void drawEnvironment()
         {
                Dimension d = size();

                // Sets buffered image to the image stored in
                // pixels
                bsmilie.setRGB
                        (0, 0, wimage, himage, pixels, 0, wimage);

                // Draws buffered image to the screen
                g.drawImage(bsmilie, (d.width - wimage)/2,
                        (d.height - himage)/2, this);

              g.setColor(Color.blue);
                Font font = new Font(null, 0, 16);
                g.setFont(font);

                // Draws Covariance matrix
                g.drawString("Covariance matrix = ", 20, 450);
                g.drawString("" + (int)C.get(0,0) + "  "
                        + (int)C.get(0, 1), 180, 440);
                g.drawString("" + (int)C.get(1,0) + "  "
                        + (int)C.get(1, 1), 180, 460);

                // Draws eigenvector matrix
                g.setColor(Color.red);
                g.drawString("Eigenvector matrix = ", 270, 450);
                g.drawString("" + (int)(eigs.get(0,0)*100) + "  "
                              + (int)(eigs.get(0, 1)*100), 430, 440);
                g.drawString("" + (int)(eigs.get(1,0)*100) + "  "
```

```
                    + (int)(eigs.get(1, 1)*100), 430, 460);


  }

  void cycle()
  {
        if (cycleNum % 10 == 0) // one way to slow things down
                {
                updateEnvrionment();
                }


    drawEnvironment();
    repaint();      // the board should get redrawn each time!
    cycleNum++;
  }



 public void actionPerformed(ActionEvent evt)
    {
        Object source = evt.getSource();

        if (source == pauseButton)
            pause();

        if (source == startButton)
            go();
    }

 void clear()
  {
    g.setColor(Color.white);
    g.fillRect(0, 0, getSize().width, getSize().height);
  }


// The following two methods are overriding those from
// the Applet base class. You will not need to call these
// explicitly, but they will be used when you do call


  public void update(Graphics gr)
  {
```

```
    paint(gr);
  }

  public void paint(Graphics gr)
  {
    gr.drawImage(image, 0, 0, null); // double-buffering
  }



    // The following methods and data members are used
    // to implement the Runnable interface and to
    // support pausing and resuming the applet.
    //

  Thread thread;
  boolean threadSuspended;
  boolean running;


    // This is the method that calls the "cycle()"
    // method every so often
    //(every sleepTime milliseconds).


  public void run()
  {
    while (running) {
      try {
        if (thread != null) {
          thread.sleep(sleepTime);
          synchronized(this) {
            while (threadSuspended)
              wait();
          }
        }
      }
      catch (InterruptedException e) { ; }

      cycle();
    }
    thread = null;
  }
```

```
    // This is the method attached to the "Start" button

  public synchronized void go()
  {
    if (thread == null)  {
      thread = new Thread(this);
      running = true;
      thread.start();
      threadSuspended = false;
    } else {
      threadSuspended = false;
    }
    notify();
  }

    // This is the method attached to the "Pause" button

  void pause()
  {
    if (thread == null) {
      ;
    } else {
      threadSuspended = true;
    }
  }


   // This is a method called when you leave the page
   //   that contains the applet. It stops the thread altogether.


  public synchronized void stop() {
    running = false;
    notify();
  }

// This method takes the image stored in pixels, rotates it
// angle, and puts the new image in array pixels.

public void rotate(int pixels[], double angle)
        {
        for(int i = (int)(-wimage/2); i < (int)(wimage/2); i++)
        for(int j = (int)(-himage/2); j < (int)(himage/2); j++)
                {
                double tanner;
```

```
                if(i >= 0)
                        tanner = Math.atan(((double)j)/i);
                else
                        tanner = Math.atan(((double)j)/i)
                            +Math.PI;
                double lenth = Math.sqrt(i*i + j*j);
                int newx = i
                    + (int)(lenth*Math.cos(tanner-angle)
                     - lenth*Math.cos(tanner));
                int newy = j
                    + (int)(lenth*Math.sin(tanner-angle)
                            - lenth*Math.sin(tanner));
                newx = newx+(int)(wimage/2);
                newy = newy+(int)(himage/2);
                if(newx >= 0
                   && newy >= 0
                   && newx < wimage
                   && newy < himage)
                        newpixels[i+(int)(wimage/2)]
                                 [j+(int)(himage/2)]
                            = pixels[newy*wimage+newx];
                else
                        newpixels[i+(int)(wimage/2)]
                                 [j+(int)(himage/2)]
                            = 0xFFFFFFFF;
                }

        for(int i = 0; i<wimage; i++)
        for(int j = 0; j<himage; j++)
                pixels[j*wimage+i] = newpixels[i][j];
        }



    // Finds the first component of the mean vector of
    // a set of two dimensional vectors stored in data.
    // Used in the covariance method.
        public double findMeanVect0(double data[][])
                {
                double mean = 0;

                for(int i = 0; i<n ; i++)
                        mean += (1/(double)n)*data[i][0];

                return mean;
```

```
            }


    // Finds the second component of the mean vector of
    // a set of two dimensional vectors stored in data.
    // Used in the covariance method.
        public double findMeanVect1(double data[][])
                {
                double mean = 0;

                for(int i = 0; i<n ; i++)
                        mean += (1/(double)n)*data[i][1];

                return mean;
                }


    // Finds the mean vector of a set of two dimensional
    // vectors stored in data.  Used in the covariance
    // method
        public Matrix findMeanVect(double data[][])
                {
                Matrix m = new Matrix(2, 1);

                m.set(0, 0, findMeanVect0(data));
                m.set(1, 0, findMeanVect1(data));

                return m;
                }

    // This compute the covariance matrix given a set of
    // two dimensional vectors stored in data
        public Matrix covariance(double data[][])
                {
                Matrix m = findMeanVect(data);

                Matrix C = new Matrix(2, 2);

                for(int i = 0; i<n; i++)
                        {
                        Matrix x = new Matrix(2, 1);

                        x.set(0, 0, data[i][0]);
                        x.set(1, 0, data[i][1]);
```

```
                        C.plusEquals
                            (x.times
                              (x.transpose()).minus
                             (m.times(m.transpose())).
                             times(1/(double)(n+1)));

                        }

                    return C;
                    }


        // This method takes an image stored in pixels,
        // and stores the location of all dark pixels
        // (defined by the color being less than zero)
        // in array data, as two dimensional vectors.
            public void getRegion(int pixels[])
                    {
                    n = 0;

                    for(int i = 0; i < wimage; i++)
                            for(int j = 0; j< himage; j++)
                                    if(pixels[j*wimage+i] < 0)
                                            {
                                            data[n][0] = j;
                                            data[n][1] = i;
                                            n++;
                                            }

                    }


}
```

Bibliography

2003. *PICS 2003: The PICS Conference, An International Technical Conference on The Science and Systems of Digital Photography, including the Fifth International Symposium on Multispectral Color Science, May 13, 2003, Rochester, NY, USA*. IS&T - The Society for Imaging Science and Technology.

Gel'fand, I. M., M. M. Kapranov, and A. V. Zelevinsky. 1992. Hyperdeterminants. *Adv Math* 96(2):226–263.

Gonzalez, Rafael C., and Richard E. Woods. 2002. *Digital Image Processing*. Prentice Hall, 2nd ed.

Lax, Peter D. 1997. *Linear Algebra*. John Wiley and Sons, 1st ed.

Lee, Seung-Yong, Kyung-Yong Chwa, James Hahn, and Sung Yong Shin. 1996. Image morphing using defomation techniques. *J Visualization Comput Anim* 7:3–23.

Lenz, Reiner, and Thanh Hai Bui. 2003. Illumination invariants. In DBL (2003), 506–511.

Lenz, Reiner, Linh Viet Tran, and Thanh Hai Bui. 2003. Group theoretical invariants in color image processing. In *IS&T/SID 11th Color Imaging Conference*, vol. 11. Society for Image Science and Technology and Society for Information Display.

Pillai, Unnikrishna, Torsten Suel, and Seunghun Cah. 2005. The perron-frobenius theorem: Some of its applications. *IEEE Signal Processing Mag* 62–75.

Wolberg, George. 1998. Image morphing: A survey. *The Visual Computer* 14:360–372.