Probabilistic and Statistical Methods for Target Tracking

by

Raymond Dao

B.S. in Applied Mathematics, University of Colorado at Boulder, 2015

M.S. in Applied Mathematics, University of Colorado at Boulder, 2015

A thesis submitted to the

Faculty of the Graduate School of the

University of Colorado in partial fulfillment

of the requirements for the degree of

Master of Science

Department of Applied Mathematics

2015

This thesis entitled:
Probabilistic and Statistical Methods for Target Tracking
written by Raymond Dao
has been approved for the Department of Applied Mathematics

Dr. Jem N. Corcoran

Dr. Yermal Sujeet Bhat

Date

The final copy of this thesis has been examined by the signatories, and we find that both the content and the form meet acceptable presentation standards of scholarly work in the above mentioned discipline.

Dao, Raymond (M.S., Applied Mathematics)

Probabilistic and Statistical Methods for Target Tracking

Thesis directed by Prof. Dr. Jem N. Corcoran

Accurate and efficient tracking of objects through frames of a video is important in a wide range of areas including suveillance, military, and medical imaging applications, as well the understanding of social interactions of biological populations such as swarming insects. In this thesis, we review some of the most popular deterministic template matching algorithms for tracking, including the seminal Lucas-Kanade algorithm. We also review a Monte Carlo method and introduce a simple probabilistic algorithm for parameter learning. Additionally, we offer some inprovements for existing algorithms, including a template stabilizer, formed from a principle components analysis, and an additional stopping rule for iterative attempts at matching that improves the speed of existing algorithms and in some cases results in better accuracy. Existing and new methods are compared on simulated images and on real video. In several cases, R code is provided.

Dedication

This Master's thesis is dedicated to my family. First, I would like to dedicate this to my parents for fostering the opportunity to live and enjoy my university experience with minimal distractions. In their words, when it comes down to it, your fulfillment and happiness is paramount. I would also like to dedicate this thesis to my sister, who taught me persistence, and that not all shortcuts are worth taking.

Acknowledgements

I would like to acknowledge and express my sincere gratitude to Dr. Jem Corcoran for not only being an ideal advisor, but also a fantastic teacher and mentor. I am motivated from just seeing the excitement she brings, not only for mathematics, but for all aspects of life. Her patience, perseverance, and dedication was critical in the thesis process, which surely could not have been completed without her.

I would also like to thank Dr. Anne Dougherty for her patient advising throughout my schooling here at the University of Colorado - Boulder. When I was still deciding on major, it was her that brought me to Applied Mathematics and the subsequent graduate program. Without her, I would be very lost in the intricacies of the university.

I must also acknowledge Dr. Ray Littlejohn. Besides sharing my name and unknowingly being the target of my intended self-emails, his courses formalized in my mind what I can do with the mathematical and statistical knowledge I had been acquiring. The experience I gained form him made a great impact on what I hope to get out of my career.

Finally, I am grateful to have met Dr. Sujeet Bhat. For me, his open candor made working under him and his classes much more valuable, and gave me wonderful insight onto the professor side of the spectrum.

The members of my thesis committee, Dr. Jem Corcoran, Dr. Sujeet Bhat, and Dr. Anne Dougherty, have generously donated their time to listen and comment on my research. I thank them for their affable contributions.

Contents

Chapter

Figures

Figure

Chapter 1

Introduction

Tracking an object through a sequence of video frames is a task of obvious importance in many diverse areas including biology on the micro and macro levels (for example tracking animal migration or movements of particles within a cell), sports tracking, and military applications. In this thesis we explore popular deterministic gradient descent algorithms, for which we offer some small performance enhancements, and a probabilistic algorithm using a Monte Carlo method known as a "particle filter". Comparisons for all approaches are given using real video.

We began the project seeking to improve existing algorithms for tracking multiple targets within a sequence of video frames. As we progressed, it became clear that algorithms for tracking even a single target are often inadequate and so we instead focused our attention there. Tracking a target object from frame to frame is an image alignment problem that aims to minimize the error between a template of pixels representing the object from one frame and a mapping of that template to the next frame using a "warp function" which is used to describe things like translation, rotation, and skew as the object moves from frame to frame. This thesis will explore several popular existing algorithms, including their enhancement with principle components analysis for creating more robust templates, and in some cases offer subtle improvements that leverage statistical approaches as well as analyses of empirical distributions of "error images".

On the deterministic side, a gradient descent template matching algorithm known as the Lucas-Kanade algorithm, introduced in 1981 ([9]) is arguably the mostly widely used technique in computer vision today. The Lucas-Kanade algorithm sparked the creation of many similar algo-

rithms, including the so-called "compositional alignment" and "inverse compositional alignment" algorithms [4] which in many cases are faster and more widely applicable. We discuss all three in Chapter 2. In Chapter 3 we discuss shortcomings of these gradient descent based methods when applied to real video.

In [8], Kahn, Bach, and Dellaert build a tracking model, especially suited to the tracking of multiple targets at once, using the Monte Carlo method of particle filtering which we describe in Chapter 4. We compare the performance of this algorithm to the inverse compositional alignment algorithm and to our proposed modified version of the inverse compositional alignment algorithm, which is designed to help stabilize the target template we are searching for from frame to frame, by guarding against random walks towards tracking failure caused by compounding error between frames.

In Chapter 5 we compare the performance of all of the algorithms and also introduce an additional histogram based measure of error in template matching that can be used to force an earlier declaration of a match in all of these iterative algorithms and therefore computational savings. Furthermore, we will see that this additional stopping rule will sometimes even avert tracking failure.

Chapter 2

Template Matching: The Lucas-Kanade Algorithm and Variations

Our goal in target tracking will be to follow an object or "template" through a sequence of video frame images. This is typically complicated by changes in appearance of the object over time as well as other factors such as background noise, lighting and image intensity. Our assumptions will include that the time between subsequent video frames is very small and that the object is not moving in an overly erratic manner. In other words, we expect to find the object in some neighborhood of where is was in the previous frame with a near constant template.

In this Chapter, we will first consider the ideal case where the object from "frame 2" is lifted directly from frame 2, as in Figure 2.1, so that an exact match between the template and its position within the image can be found.

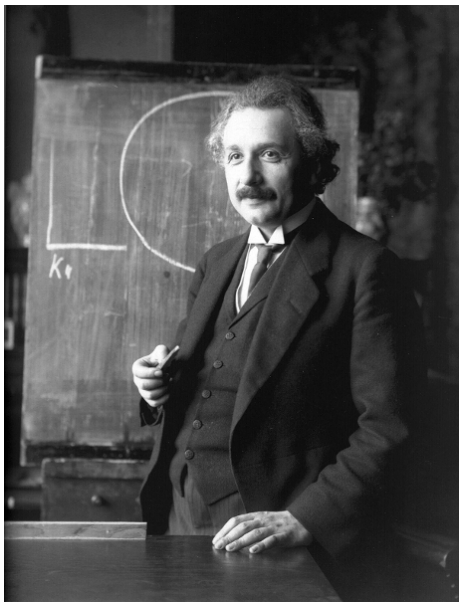
Notation

Let $\vec{x} = (x_1, x_2)^t$ be a pixel position relative to some coordinate system. For example, we may work in the Cartesian coordinate system, which has the origin in the lower left corner with the x_1-coordinate increasing to the east and the x_2-coordinate increasing to the north. Standard image processing convention, which we will call "image space", places the origin in the upper left corner with x_1 increasing to the east and x_2 increasing to the south. A third (programmatically) convenient coordinate system puts the origin in the upper left corner with the x_1-coordinate increasing to the south and the x_2-coordinate increasing to the east. We will call this "matrix space" since if the pixel locations are labeled, pixel (i, j) appears in the ith row and jth column. Pixel labeling for all

Figure 2.1: An Image and Template Pulled from the Image

(a) Template

(b) Image



three spaces are depicted for a 4 by 3 pixel "image" in Figure 2.2. Unless otherwise specified, we will use the Cartesian coordinate system in this thesis.

Figure 2.2: Pixel Locations for Different Coordinate Systems for a 4 by 3 Pixel Image

(a) Cartesian Space

(1,4)	(2,4)	(3,4)
(1,3)	(2,3)	(3,3)
(1,2)	(2,2)	(3,2)
(1,1)	(2,1)	(3,1)

(b) Image Space

(1,1)	(2,1)	(3,1)
(1,2)	(2,2)	(3,2)
(1,3)	(2,3)	(3,3)
(1,4)	(2,4)	(3,4)

(c) Matrix Space

(1,1)	(1,2)	(1,3)
(2,1)	(2,2)	(2,3)
(3,1)	(3,2)	(3,3)
(4,1)	(4,2)	(4,3)

2.1 The Lucas-Kanade Algorithm

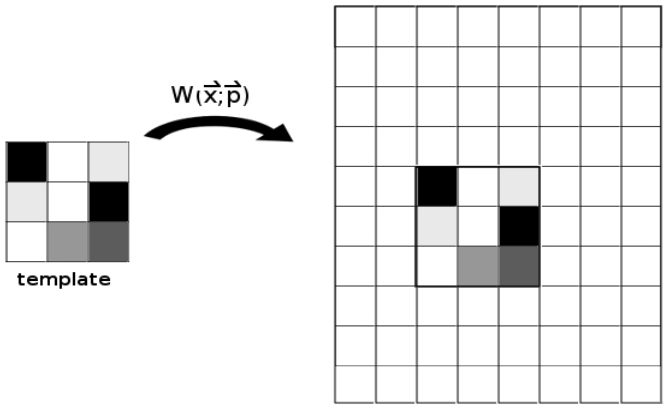
Let $T(\vec{x})$ denote the color stored (an RGB triple or a number between 0 and 1 in the case of a greyscale image) at pixel location $\vec{x} = (x_1, x_2)$ in an $h \times w$ template. (We assume a rectangular template here but it may be an arbitrary shape. All sums in this chapter, unless otherwise specified, are taken over all pixel locations in the template.) Let $I(\vec{x})$ denote the color stored at pixel location $\vec{x}$ in a larger image. The **Lucas-Kanade** (LK) algorithm [9] is a gradient descent method that attempts to align a template patch to its proper position in the image given a close initial guess. It is one of the most widely used techniques in computer vision today. In the case of video tracking, the initial guess for finding an object (template) in an image for any particular frame will be the object's location in the previous frame.

Define a **warp function**, $W(\vec{x};\vec{p})$, for a vector of parameters $\vec{p} \in \mathbb{R}^n$, that takes a pixel $\vec{x}$ in the coordinate frame of the template and maps it to pixel location $W(\vec{x};\vec{p})$ in the coordinate frame of the image. For example, a simple **translation warp** with shift parameters $\vec{p} = (p_1, p_2)$ is defined as

$$W(\vec{x};\vec{p}) = \begin{pmatrix} x_1 + p_1 \\ x_2 + p_2 \end{pmatrix} \tag{2.1}$$

and depicted in Figure 2.3. Other commonly used warp functions in target tracking include rotations, "rigid" warps (translation and rotation together), affine warps (translation, rotation, and resizing– good for tracking 3D motion or changes in perspective), and shearing warps.

Figure 2.3: A Translation Warp from Template to Image with $p_1 = 2$ and $p_2 = 3$



The specific goal of the LK algorithm is to minimize, with respect to $\vec{p}$, the sum of squared errors between the color values of the pixels in the template and those in the full image at pixels defined by the warp. That is, we aim to minimize

$$SSE := \sum_{\vec{x}} [I(W(\vec{x};\vec{p})) - T(\vec{x})]^2 \tag{2.2}$$

with respect to $\vec{p}$.

Pixel Interpolation

It is important to note that we only have image information $I(x_1, x_2)$ at integer values of x_1 and x_2 and yet $W(\vec{x};\vec{p})$ does not necessarily take values in $\mathbb{Z}^2$. In the case, for example, that we need to evaluate the image value at the point $(2.7, 4.2)$, we would really like the value somewhat northeast of pixel $(2, 4)$. To this end we use the weighted average of immediate surrounding pixels

$$I(2.7, 4.2) := (0.3)(0.8) \cdot I(2, 4) + (0.7)(0.8) \cdot I(3, 4) + (0.3)(0.2) \cdot I(2, 5) + (0.7)(0.2) \cdot I(3, 5),$$

or in general,

$$\begin{aligned} I(x_1, x_2) &= (1 - frac(x_1))(1 - frac(x_2)) \cdot I(\lfloor x_1 \rfloor, \lfloor x_2 \rfloor) \\ &+ frac(x_1)(1 - frac(x_2)) \cdot I(\lfloor x_1 \rfloor + 1, \lfloor x_2 \rfloor) \\ &+ (1 - frac(x_1))frac(x_2) \cdot I(\lfloor x_1 \rfloor, \lfloor x_2 + 1 \rfloor) \\ &+ frac(x_1)frac(x_2) \cdot I(\lfloor x_1 \rfloor + 1, \lfloor x_2 + 1 \rfloor) \end{aligned}$$

where $frac(x) := x - \lfloor x \rfloor$ is the fractional part of $x \in \mathbb{R}$.

The LK algorithm, which is a Gauss-Newton gradient descent non-linear optimatization algorithm, assumes that a current estimate of $\vec{p}$ is known and iteratively solves for increments $\Delta\vec{p}$ by minimizing

$$SSE_1 := \sum_{\vec{x}} [I(W(\vec{x}; \vec{p} + \Delta p)) - T(\vec{x})]^2 \tag{2.3}$$

with respect to $\Delta\vec{p}$ and updating $\vec{p} \leftarrow \vec{p} + \Delta\vec{p}$ until $||\Delta p|| \leq \varepsilon$ for some predesignated ε. The non-linear SSE_1 is first linearized by performing a first order Taylor series expansion on $I(W(\vec{x}; \vec{p}+\Delta\vec{p}))$ to give

$$SSE_1 \approx \left[I(W(\vec{x}; \vec{p})) + \nabla I \frac{\partial W}{\partial \vec{p}} \Delta\vec{p} - T(\vec{x}) \right]^2. \tag{2.4}$$

Here, $\nabla I = (I_{x_1}, I_{x_2}) = \left(\frac{\partial I}{\partial x_1}, \frac{\partial I}{\partial x_2} \right)$ is the gradient of image I evaluated at $W(\vec{x}; \vec{p})$ and $\partial W / \partial \vec{p}$ is the warp Jacobian

$$\frac{\partial W}{\partial \vec{p}} = \begin{pmatrix} \frac{\partial W_1}{\partial p_1} & \frac{\partial W_1}{\partial p_2} & \cdots & \frac{\partial W_1}{\partial p_n} \\ \frac{\partial W_2}{\partial p_1} & \frac{\partial W_2}{\partial p_2} & \cdots & \frac{\partial W_2}{\partial p_n} \end{pmatrix} \tag{2.5}$$

where W_1 and W_2 are the two components of the warp function.

Approximating the Image Gradient

We estimate components of the image gradient in this discrete pixel setting as

$$\frac{\partial I}{\partial x_1} = \frac{\partial I(x_1, x_2)}{\partial x_1} = \lim_{\Delta x_1 \to 0} \frac{I(x_1 + \Delta x_1, x_2) - I(x_1, x_2)}{\Delta x_1} \approx \frac{I(x_1 + 1, x_2) - I(x_1 - 1, x_2)}{2} \tag{2.6}$$

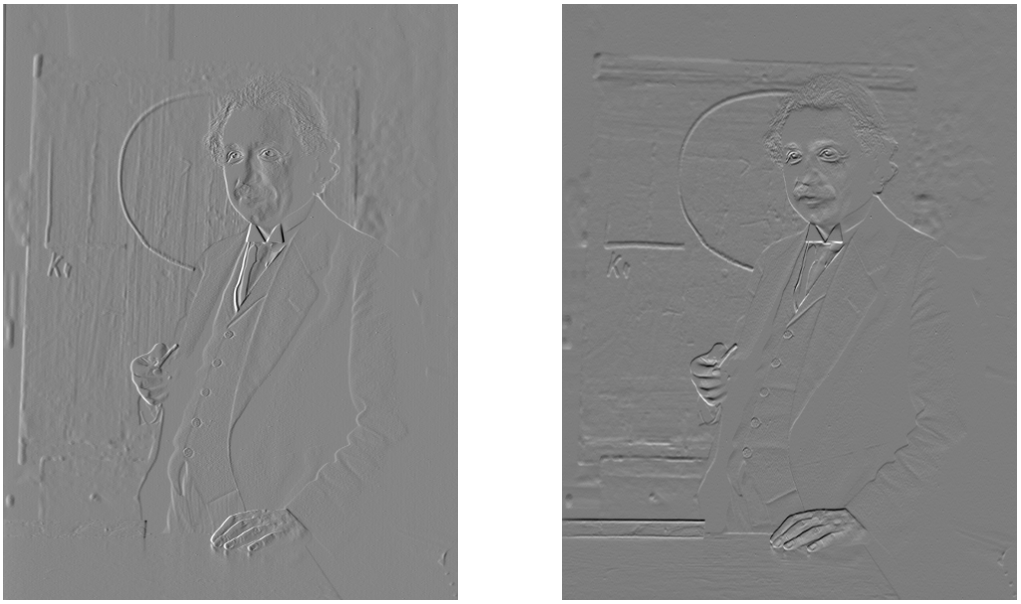
and

$$\frac{\partial I}{\partial x_2} = \frac{\partial I(x_1, x_2)}{\partial x_2} = \lim_{\Delta x_2 \to 0} \frac{I(x_1, x_2 + \Delta x_2) - I(x_1, x_2)}{\Delta x_1} \approx \frac{I(x_1, x_2 + 1) - I(x_1, x_2 - 1)}{2}. \quad (2.7)$$

To compute the image gradient at border pixels, we define a repeating border with $I(0, x_2) := I(1, x_2)$, $I(x_1, 0) := I(x_1, 1)$, $I(h + 1, x_2) := I(h, x_2)$, and $I(x_1, w + 1) := I(x_1, w)$. Image gradients for Figure 2.1(b) are shown in 2.4.

Figure 2.4: Image Gradients

(a) I_{x_1} (b) I_{x_2}



Minimizing (2.4), with respect to $\Delta\vec{p}$, is a least squares problem with a closed form solution. Taking the derivative with respect to Δp, we get

$$2\sum_{\vec{x}} \left[\nabla I \frac{\partial W}{\partial \vec{p}}\right]^T [I(W(\vec{x}; \vec{p})) + \nabla I \frac{\partial W}{\partial \vec{p}} \Delta\vec{p} - T(\vec{x})] \stackrel{set}{=} \vec{0}.$$

Solving for $\Delta\vec{p}$ gives

$$\Delta\vec{p} = H^{-1} \sum_{\vec{x} \in T} \left[\nabla I \frac{\partial W}{\partial \vec{p}}\right]^T [T(\vec{x}) - I(W(\vec{x}; \vec{p}))] \quad (2.8)$$

where H is the (Gauss-Newton approximation to the) Hessian matrix defined as

$$H = \sum_{\vec{x} \in T} \left[\nabla I \frac{\partial W}{\partial \vec{p}} \right]^T \left[\nabla I \frac{\partial W}{\partial \vec{p}} \right]. \tag{2.9}$$

From (2.8) we see that the LK algorithm requires that the chosen warp must be differentiable with respect to the warp parameters $\vec{p}$.

In summary, the Lucas-Kanade algorithm is as follows.

Algorithm 1 Lucas-Kanade Algorithm

- Start with an initial estimate of $\vec{p}$.
- Set an error tolerance $\varepsilon > 0$.
- Set $||\Delta \vec{p}||$ to an arbitrary value greater than ε.

while $||\Delta \vec{p}|| > \varepsilon$ **do**
  Compute $I(W(\vec{x}; \vec{p}))$.
  Compute $T(\vec{x}) - I(W(\vec{x}; \vec{p}))$.
  Compute $\nabla I = \nabla I(W(\vec{x}; \vec{p}))$.
  Compute $S := \nabla I \frac{\partial W}{\partial \vec{p}}$.
  Compute $\sum_{\vec{x}} S^T [T(\vec{x}) - I(W(\vec{x}; \vec{p}))]$.
  Compute the Hessian $H = \sum_{\vec{x} \in T} S^T S$.
  Compute $\Delta \vec{p}$ using (2.8).
  Update the warp parameters as $\vec{p} \leftarrow \vec{p} + \Delta \vec{p}$.
end while

$I(W(\vec{x}; \vec{p}))$ is an $h \times w$ matrix containing pixel colors from the image at the current guess for the template position. $I(W(\vec{x}; \vec{p})) - T(\vec{x})$ (or $T(\vec{x}) - I(W(\vec{x}; \vec{p}))$ when convenient) is known as the **error image**. $\nabla I(W(\vec{x}; \vec{p}))$ is a 1×2 vector of $h \times w$ matrices containing values from the gradient images (as in Figure (2.4)) at the current guess for the template position. S is a 1×2 vector of $h \times w$ matrices representing **steepest descent images**. For the translation warp given by (2.1), the Jacobian $\partial W / \partial \vec{p}$ is the identity matrix and $S = (S_1, S_2)$ where $S_1 = I_{x_1}$ and $S_2 = I_{x_2}$ are simply the image gradients.

The Hessian is the 2×2 matrix

$$H = \sum_{\vec{x}} \begin{bmatrix} (S_1(\vec{x}))^2 & S_1(\vec{x}) S_2(\vec{x}) \\ S_1(\vec{x}) S_2(\vec{x}) & (S_2(\vec{x}))^2 \end{bmatrix} = \begin{bmatrix} \sum_{\vec{x}} (S_1(\vec{x}))^2 & \sum_{\vec{x}} S_1(\vec{x}) S_2(\vec{x}) \\ \sum_{\vec{x}} S_1(\vec{x}) S_2(\vec{x}) & \sum_{\vec{x}} (S_2(\vec{x}))^2 \end{bmatrix}.$$

So, for example, the $(1, 1)$ entry of the Hessian matrix is the sum of the squared pixel color values in the first steepest descent image.

The Lucas-Kanade algorithm is illustrated for a simple translation warp to find the template in Figure 2.1a in the image in Figure 2.1b starting with a close initial guess. R code is given in Appendix A. Results are shown in Figure 2.5.

2.2 The Compositional Alignment Algorithm

The **compositional alignment** (CA) algorithm, which first appeared in [5], is a template matching algorithm that is similar to the Lucas-Kanade algorithm but instead, updates the warp function directly as the composition

$$W(\vec{x};\vec{p}) \leftarrow W(\vec{x};\vec{p}) \circ W(\vec{x};\Delta\vec{p}) = W(W(\vec{x};\Delta\vec{p});\vec{p}) \tag{2.10}$$

as opposed to additively updating the parameters $\vec{p}$. Specifically, the goal is to minimize

$$SSE_2 := \sum_{\vec{x}} [I(W(W(\vec{x};\Delta\vec{p});\vec{p})) - T(\vec{x})]^2 \tag{2.11}$$

with respect to $\Delta\vec{p}$ and update the warp function via (2.10), $||\Delta p|| \leq \varepsilon$,f for some predesignated ε.

Due to (2.10), we require that the set of warps used must be closed under composition. As with the LK algorithm, SSE_2 is estimated and linearized using a first order Taylor series expansion, this time on $I(W(W(\vec{x};\Delta\vec{p});\vec{p}))$, about $\Delta\vec{p} = \vec{0}$. Specifically,

$$SSE_2 \approx \sum_{\vec{x}} \left[I(W(W(\vec{x};\vec{0});\vec{p})) + \nabla I(W(\vec{x};\vec{0}))\frac{\partial W}{\partial \vec{p}}\Delta\vec{p} - T(\vec{x}) \right]^2. \tag{2.12}$$

Here, $\nabla I(W)$ is the gradient of the image in the current template region defined by the warp. We will estimate this using (2.6) and (2.7) on $I(W)$ as with the Lucas-Kanade Algorithm. $\partial W/\partial\vec{p}$ is evaluated at $(\vec{x};\vec{0})$.

To proceed in describing the CA algorithm, we need to make the assumption that $W(\vec{x};\vec{0})$ is the identity warp. That is, we need to assume that $W(\vec{x};\vec{0}) = \vec{x}$. This may restrict us from using certain warp functions but can often (if it is not naturally the case) be achieved by a simple reparameterization. The goal then is to minimize

$$SSE_2 \approx \sum_{\vec{x}} \left[I(W(\vec{x});\vec{p})) + \nabla I(W(\vec{x};\vec{0}))\frac{\partial W}{\partial \vec{p}}\Delta\vec{p} - T(\vec{x}) \right]^2. \tag{2.13}$$

Figure 2.5: Iterations Through Lucas-Kanade Algorithm; Pure Translation

with respect to $\Delta\vec{p}$.

Taking the derivative with respect to Δp, we get

$$2\sum\left[\nabla I(W)\frac{\partial W}{\partial \vec{p}}\right]^T [I(W(\vec{x};\vec{p})) + \nabla I(W)\frac{\partial W}{\partial \vec{p}}\Delta\vec{p} - T(\vec{x})] \stackrel{set}{=} \vec{0}.$$

Solving for $\Delta\vec{p}$ gives

$$\Delta\vec{p} = H^{-1}\sum\left[\nabla I(W)\frac{\partial W}{\partial \vec{p}}\right]^T [T(\vec{x}) - I(W(\vec{x};\vec{p}))] \tag{2.14}$$

where H is the Hessian matrix given by

$$H = \sum\left[\nabla I(W)\frac{\partial W}{\partial \vec{p}}\right]^T \left[\nabla I(W)\frac{\partial W}{\partial \vec{p}}\right]. \tag{2.15}$$

In the LK algorithm, the gradient ∇I is evaluated at the warp $W(\vec{x};\vec{p})$ which is changing as $\vec{p}$ is updated with $\vec{p} \leftarrow \vec{p} + \Delta\vec{p}$. Here, it is evaluated only once at $W(\vec{x};\vec{0})$. Overall, the two algorithms can be shown to be equivalent in the case that the warp is invertible. We refer the reader to the very comprehensive summary provided in [4].

The Compositional Alignment algorithm is summarized as follows.

Algorithm 2 Compositional Alignment Algorithm

- Start with an initial estimate of $\vec{p}$. (And thereby an initial estimate of the warp function.)
- Set an error tolerance $\varepsilon > 0$.
- Set $||\Delta\vec{p}||$ to an arbitrary value greater than ε.
- Evaluate the Jacobian, $\partial W/\partial\vec{p}$ at $(\vec{x};\vec{0})$ at all pixels $\vec{x}$ in the template.

1: **while** $||\Delta\vec{p}|| > \varepsilon$ **do**
2: Compute $I(W(\vec{x};\vec{p}))$.
3: Compute $T(\vec{x}) - I(W(\vec{x};\vec{p}))$.
4: Compute $\nabla I(W)$, the gradient of the warped image.
5: Compute $S := \nabla I(W)\frac{\partial W}{\partial \vec{p}}$.
6: Compute $\sum_{\vec{x}} S^T[T(\vec{x}) - I(W(\vec{x};\vec{p}))]$.
7: Compute the Hessian $H = \sum_{\vec{x}\in T} S^T S$.
8: Compute $\Delta\vec{p}$ using (2.14).
9: Update the warp as $W(\vec{x};\vec{p}) \leftarrow W(\vec{x};\vec{p}) \circ W(\vec{x};\Delta\vec{p}) = W(W(\vec{x};\Delta\vec{p});\vec{p})$.
10: **end while**

R code is given in Appendix A for a simple translation warp. Note that the Jacobian is the identity matrix and is not featured in the code. Furthermore, for the warp (2.1), $W(W(\vec{x};\Delta\vec{p});\vec{p}) = W(\vec{x};\vec{p}+\Delta\vec{p})$ so it appears that we are just updating the parameters as in the LK algorithm.

2.3 The Inverse Compositional Alignment Algorithm

A third closely related, and rather groundbreaking algorithm due to [3], is known as the **inverse compositional alignment** (ICA) algorithm. The ICA algorithm (a generalization of an algorithm due to Hager and Belhumeur [7]) utilizes a change of variables to invert the roles of the template and image. A major advantage is that one only needs to compute a single Hessian matrix and can bypass computing the inverse Hessian more than once. Not only is this a significant savings computationally, but it can be useful in images where the "moving" Hessian is often not invertible such as in pure black and white (as opposed to more nuanced greyscale) images that we will consider later in this thesis.

Consider (2.11) as an approximation to

$$\int_T [I(W(W(\vec{x};\Delta\vec{p});\vec{p})) - T(\vec{x})]^2 d\vec{x} \tag{2.16}$$

where T defines the template region continuously in $\mathbb{R}^2$.

Using the change of variables $\vec{y} := W(\vec{x};\Delta\vec{p})$, (2.16) becomes

$$\int_{W(T)} [I(W(\vec{y};\vec{p})) - T(W^{-1}(\vec{y};\Delta\vec{p}))]^2 \left|\frac{\partial W^{-1}}{\partial \vec{y}}\right| d\vec{y}. \tag{2.17}$$

where $W(T) = \{W(\vec{x};\Delta\vec{p}) : \vec{x} \in T\}$.

With the same compositional alignment assumption that $\vec{p} = \vec{0}$ gives the identity warp, one can show ([3]) that

$$\left|\frac{\partial W^{-1}}{\partial \vec{y}}\right| = 1 + O(\Delta\vec{p}).$$

Under the assumption that $I(W(\vec{y};\vec{p})) - T(W^{-1}(\vec{y};\Delta\vec{p}))$ is $O(\Delta\vec{p})$, (which is basically saying that the current parameter vector is approximately correct), the first order term in the Jacobian can essentially be ignored. Furthermore, the integration domain $W(T)$ is approximately T to a zeroth order approximation. Thus, (2.17) is approximately

$$\int_T [I(W(\vec{y};\vec{p})) - T(W^{-1}(\vec{y};\Delta\vec{p}))]^2 \, d\vec{y}. \tag{2.18}$$

The specific goal of the inverse compositional alignment algorithm is to iterate and to minimize the sum of squares error

$$SSE_3 := \sum_{\vec{x}} [I(W(\vec{x};\vec{p})) - T(W(\vec{x};\Delta\vec{p}))]^2 \tag{2.19}$$

with respect to $\Delta\vec{p}$ in each iteration, at the end of which, the warp is updated as

$$W(\vec{x};\vec{p}) \leftarrow W(\vec{x};\vec{p}) \circ W(\vec{x};\Delta\vec{p})^{-1}. \tag{2.20}$$

SSE_3 is almost the discrete version of (2.18) except that the inverse on the warp function in (2.18) is missing in (2.19). Baker and Matthews ([3]) argue that $I(W(\vec{y};\vec{p})) - T(W^{-1}(\vec{y};\Delta\vec{p}))$ and $I(W(\vec{y};\vec{p})) - T(W(\vec{y};\Delta\vec{p}))$ are equivalent to a zeroth order approximation, which is not entirely surprising under the assumption that $W(\vec{x};\vec{0})$ is the identity warp.

From (2.20), we see that it is necessary that the family of warps under consideration form a group with respect to composition. As for the Lucas-Kanade and Compositional Alignment algorithms, one proceeds by performing a first order Taylor series expansion, this time on $T(W(\vec{x};\nabla\vec{p}))$ which gives

$$SSE_3 \approx \sum_{\vec{x}} \left[I(W(\vec{x};\vec{p})) - T(W(\vec{x};\vec{0})) - \nabla T \frac{\partial W}{\partial \vec{p}} \Delta\vec{p} \right]^2. \tag{2.21}$$

Minimizing (2.21) with respect to $\Delta\vec{p}$ gives

$$\Delta\vec{p} = H^{-1} \sum_{\vec{x}} \left[\nabla T \frac{\partial W}{\partial \vec{p}} \right]^T [I(W(\vec{x};\vec{p})) - T(\vec{x})] \tag{2.22}$$

where $\partial W/\partial\vec{p}$ is evaluated at $(\vec{x};\vec{0})$, ∇T evaluated at $W(\vec{x};\vec{0})$, and H is the Hessian matrix given by

$$H = \sum_{\vec{x}} \left[\nabla T \frac{\partial W}{\partial \vec{p}} \right]^T \left[\nabla T \frac{\partial W}{\partial \vec{p}} \right]. \tag{2.23}$$

Algorithm 3 Inverse Compositional Alignment Algorithm

- Start with an initial estimate of $\vec{p}$. (And thereby an initial estimate of the warp function.)
- Set an error tolerance $\varepsilon > 0$.
- Set $||\Delta\vec{p}||$ to an arbitrary value greater than ε.
- Evaluate the Jacobian, $\partial W/\partial\vec{p}$ at $(\vec{x};\vec{0})$ at all pixels $\vec{x}$ in the template.
- Compute the steepest descent images, $S = \nabla T\frac{\partial W}{\partial\vec{p}}$.
- Compute the Hessian $H = \sum_{\vec{x}\in T} S^T S$.

1: **while** $||\Delta\vec{p}|| > \varepsilon$ **do**
2: Compute $I(W(\vec{x};\vec{p}))$.
3: Compute $I(W(\vec{x};\vec{p})) - T(\vec{x})$.
4: Compute $\sum_{\vec{x}} S^T[I(W(\vec{x};\vec{p})) - T(\vec{x})]$.
5: Compute $\Delta\vec{p}$ using (2.22).
6: Update the warp as $W(\vec{x};\vec{p}) \leftarrow W(\vec{x};\vec{p}) \circ W^{-1}(\vec{x};\Delta\vec{p}) = W(W^{-1}(\vec{x};\Delta\vec{p});\vec{p})$.
7: **end while**

Chapter 3

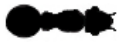
Template Alignment Algorithms for Video Frames

In this Chapter, we begin to address the issue of tracking an object through a sequence of video frames. The basic starting point is to use one of the algorithms from Chapter 2 from frame to frame using the last tracked object as the template for the next frame. A potential problem with the gradient descent methods from Chapter 2 is that they assume that the template exists as a sub-patch of the larger image. Even when the time between subsequent frames is very small, one would be hard pressed to find an exact copy of an object of interest lifted from one frame in the next due to, for example, changes in target appearance (such as changes in expression when tracking a face), lighting, perspective, and background noise. Inspired by Kahn, Bach, and Dellaert [8], we will consider motion tracking for ants, using both ideal simulated ants and real video available at `http://www.cc.gatech.edu/~borg/biotracking/experimental-data.html` as of the writing of this thesis.

3.1 The Simulated Ant

We simulated frames of a simple rigid legless ant, depicted in Figure 3.1 using a modified random walk in a two dimensional box with a plain white background. The ant was more likely to go forward than not and initiated smooth turns when coming in contact with the boundary of the box. Due to the extreme dichotomy of the pixels (black or white with no shades of gray) it was necessary to use the inverse compositional alignment algorithm since the "moving" Hessian matrices used in the Lucas-Kanade and compositional alignment algorithms often became nonsingular.

Figure 3.1: Ant Template



The warp used, consisting of both translation and rotation, was

$$W(\vec{x};\vec{p}) := \begin{pmatrix} \cos\theta & -\sin\theta & a \\ \sin\theta & \cos\theta & b \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ 1 \end{pmatrix} = \begin{pmatrix} x_1\cos\theta + x_2\sin\theta + a \\ x_1\sin\theta + x_2\cos\theta + b \\ 1 \end{pmatrix}$$

where $\vec{p} = (\theta, a, b)$.

It was necessary to introduce a third row in order to ensure that the warps form a group with respect to composition. The inverse and warp update are

$$W^{-1}(\vec{x};\vec{p}) = \begin{pmatrix} x_1\cos\theta - x_2\sin\theta - a\cos\theta - b\sin\theta \\ -x_1\sin\theta + x_2\cos\theta + a\sin\theta - b\cos\theta \\ 1 \end{pmatrix}$$

and

$$W(\vec{x};\vec{p}) \leftarrow W(\vec{x};\vec{p}) \circ W(\vec{x};\Delta\vec{p}) = W(\vec{x};\vec{p}\,')$$

where

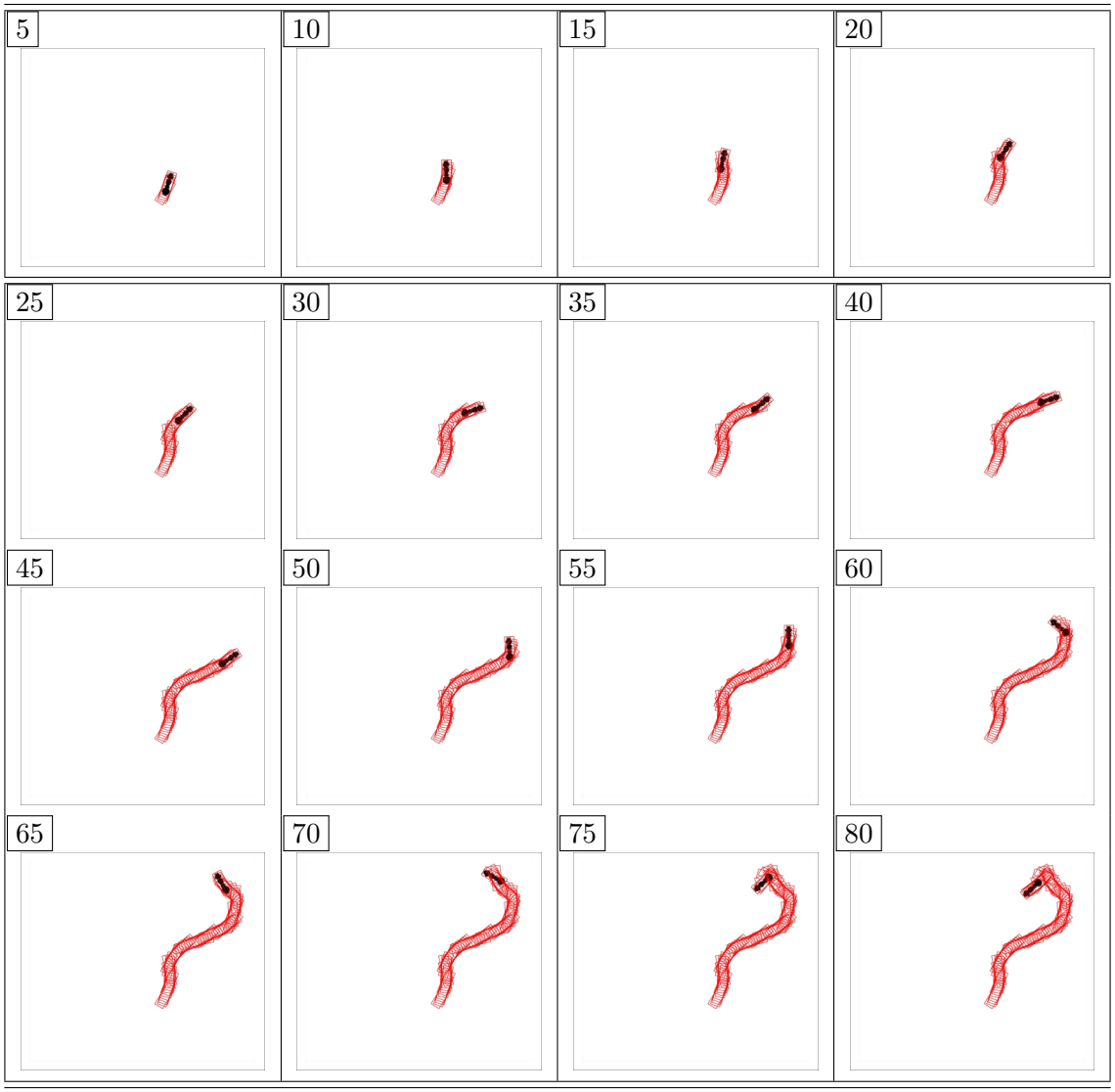
$$\vec{p}\,' = \begin{pmatrix} \theta - \Delta\theta \\ a - \Delta a\cos(\theta - \Delta\theta) + \Delta b\sin(\theta - \Delta\theta) \\ b - \Delta a\sin(\theta - \Delta\theta) - \Delta b\cos(\theta - \Delta\theta)) \end{pmatrix}.$$

Obviously, our simulated ant is ideal to track as it never undergoes any sort of deformation or change in lighting. Results are shown for the ICA algorithm in Figure 3.2. Note that, in this thesis, we are only concerned with frame-to-frame tracking and not the problem of locating the target in the first frame. Hence, we chose the tracking rectangle for the first frame simply by "eyeballing".

3.2 Tracking in Real Video

In this section we will attempt to track the lone ant in the upper left corner of Figure 3.3 through several frames pulled from a video. As in Section 3.1, we assign a first bounding rectangle

Figure 3.2: Tracking a Simulated Ant Using the ICA Algorithm



informally. Even with the fairly uncomplicated background, a target that is relatively stable in appearance, and restriction to frames where the ant of interest is isolated from the others, we will see that the inverse compositional alignment will have trouble due to moving legs and shadows. Simply put, the ant found in any given frame does not exist unchanged in the next frame.

Figure 3.3: First Frame of Ant Video

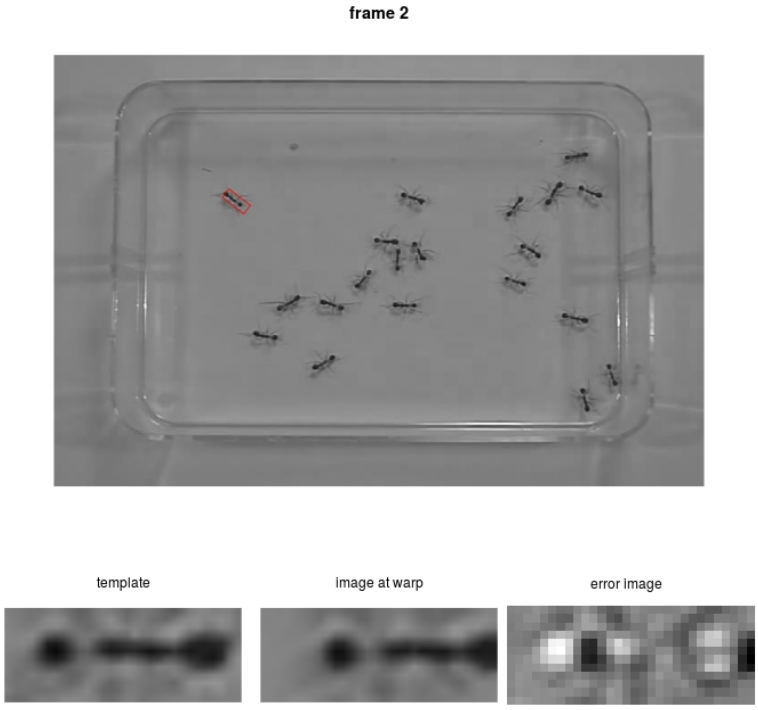
In Figure 3.4, we again show the first frame of the video, this time with the initial tracking rectangle in place and three sub-images. The second sub-image is the current rectangle which has been pulled directly from the bounding rectangle. The first sub-image, labeled "image at warp", shows the part of the image that is in the current tracking rectangle. At this point it is identical to the template by definition of the template. The third sub-image shows the error image which is computed as the second sub-image minus the first sub-image. (Note: The main image and first two sub-images are rendered in greyscale with values in $[0, 1]$ representing colors from black to white. The error image will consist of values between -1 and 1 and thus has been rendered differently with -1 representing black and 1 representing white and grays in between.)

In Figure 3.5, we show the second frame of the video before tracking. That is, the ant has moved but the tracking rectangle has not.

Figure 3.6 shows the eight iterations of the ICA algorithm that were ultimately required to reposition the tracking rectangle for frame 2 using $\varepsilon = 0.05$. Note that $||\Delta\vec{p}||$ is not monotonically

Figure 3.4: Tracking: First Frame of Ant Video

Figure 3.5: Tracking: Second Frame of Ant Video



decreasing. This gradient descent "failure" is due to the fact that the template pulled from frame 1 does not actually occur in frame 2. Indeed, the ICA algorithm starts to break down around frame 12 and completely loses the ant at frame 23. Baker, Gross, and Matthews [2] have suggested a method using principle components analysis (PCA) to improve ICA alignment results under frame to frame object appearance variation. We review this in the next Section before going on to describe our own approach to improve object tracking which we will ultimately compare to the PCA approach.

3.3 Improving ICA for Video Tracking Using Principle Components Analysis

In order to improve the ICA algorithm results in video tracking, Baker, Gross, and Matthews [2] relax the assumption that the template image $T(\vec{x})$ from any given frame appears in the next frame. Instead, they assume that

$$T(\vec{x}) - \sum_{i=1}^{m} \lambda_i A_i(\vec{x})$$

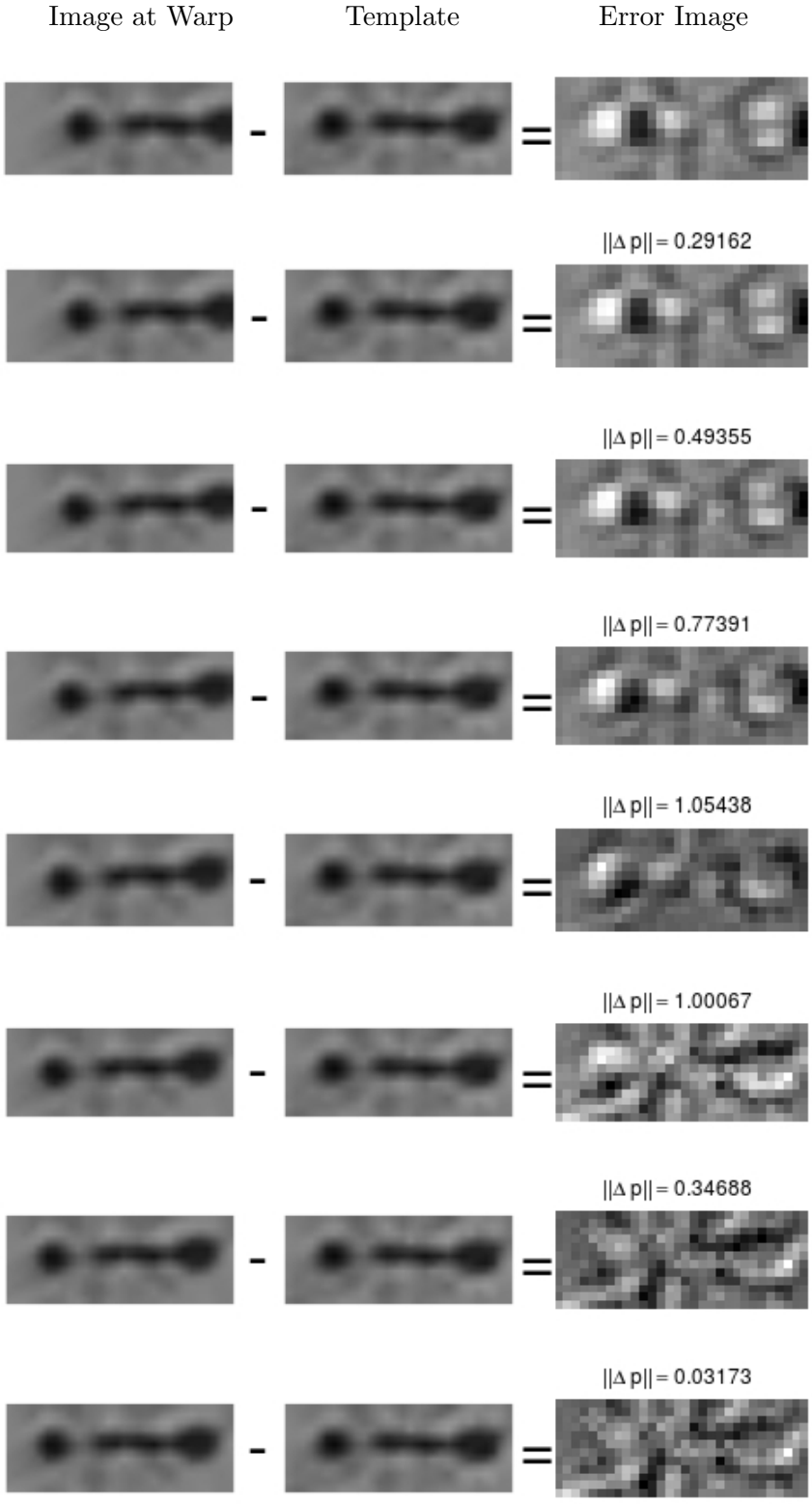
appears in the next frame where $\{A_i\}_{i=1}^m$ are a sequence of known "appearance variation images" and $\{\lambda_i\}_{i=1}^m$ are "appearance parameters". These appearance variation images are often taken as the most significant images in a principle components analysis of target template images pulled from several frames of the video.

3.3.1 Principle Components Analysis

Principle components analysis (PCA) is a method of transforming a data set to convert a collection of (possibly) correlated variables into a collection of linearly uncorrelated variables. The uncorrelated variables, known as **principle components**, form a basis for reconstructing the data set. The principle components can be ranked according to their importance in explaining variability in the data and often lower ranked components can be dropped, creating a smaller basis that can recreate the original data set with varying degrees of accuracy. For example, suppose we have $n = 10$ observations of $m = 2$ variables.

Variable 1:	0.85	−1.47	−0.51	−0.61	−1.20	−0.55	−0.03	0.05	−1.13	0.28
Variable 2:	1.41	0.81	1.20	1.06	0.71	1.00	1.19	1.31	0.70	1.21

Figure 3.6: Frame 2 ICA Iterations



Arrange the data into an $m \times n$ matrix X, with with entries recentered to have row means of 0.

$$X = \begin{bmatrix} 1.282 & -1.038 & -0.078 & -0.178 & -0.768 & -0.118 & 0.402 & 0.482 & -0.698 & 0.712 \\ 0.35 & -0.25 & 0.14 & 0.00 & -0.35 & -0.06 & 0.13 & 0.25 & -0.36 & 0.15 \end{bmatrix}$$

A scatterplot of the recentered variables (called X_1 and X_2) is shown in Figure 3.7.

The covariance matrix for X is

$$Var(X) = \frac{1}{n-1} XX^T.$$

Our goal is to find an $m \times m$ matrix P such that $Var(PX)$ is a diagonal matrix. $N := PX$ will then be our new data set of linearly uncorrelated variables. Write

$$Var(PX) = \frac{1}{n-1}(PX)(PX)^T = \frac{1}{n-1} PXX^T P^T = \frac{1}{n-1} PAP^T$$

where $A := XX^T$. Rewrite A as $A = EDE^T$ where E is an $m \times m$ orthogonal matrix whose columns are the orthonormal eigenvectors of A arranged in descending eigenvalue order and D is a diagonal matrix of corresponding eigenvalues. Then, since $E^{-1} = E^T$, if we choose $P = E^T$, we have that the covariance matrix of the transformed data matrix $N = PX$

$$Var(N) = \frac{1}{n-1} PEDE^T P^T = \frac{1}{n-1} D$$

is diagonal. Thus, the transformed principle components variables that are realized as rows of N are linearly uncorrelated and are arranged in order of decreasing variance. For our two-dimensional example,

$$A = \begin{bmatrix} 4.75056 & 1.5040 \\ 1.50400 & 0.5622 \end{bmatrix}$$

which decomposes into eigenvectors and eigenvalues

$$E = \begin{bmatrix} -0.95190 & 0.30640 \\ -0.30640 & -0.95190 \end{bmatrix} \quad \text{and} \quad D = \begin{bmatrix} 5.234675 & 0 \\ 0 & 0.078084 \end{bmatrix}.$$

The eigenvectors give the directions of the principle components and are plotted in Figure 3.8a.

The new data, given by $N = PX$ are shown in Figure 3.8b. Obviously, the original data can be recovered as $X = P^{-1}N = P^T N$. However, if we want to compress the data, we could use less

Figure 3.7: Scatterplot of Centered X_1 and X_2

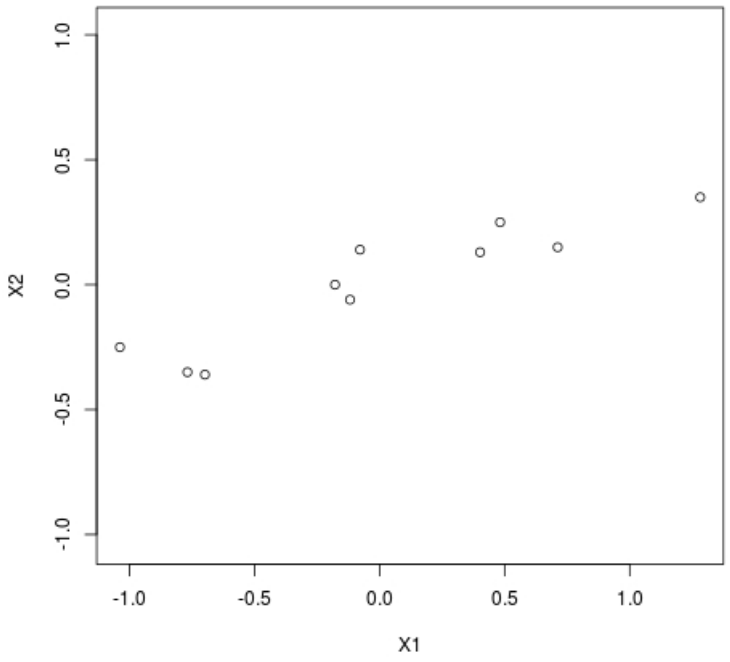
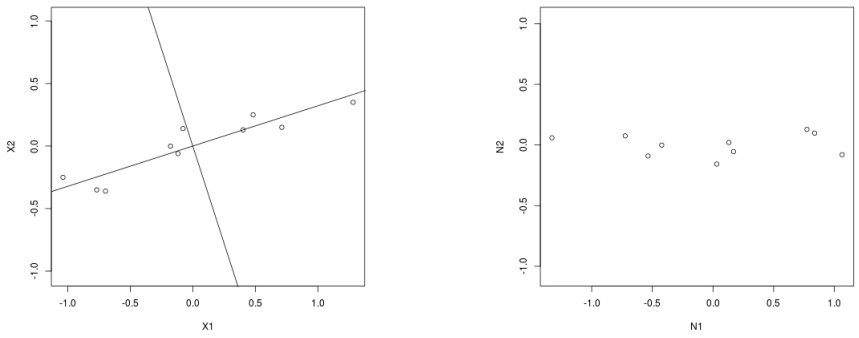


Figure 3.8: Principle Components Analysis of X

(a) Axes in the Direction of the Eigenvectors

(b) Rotated "New" Data



eigenvectors columns from E in creating P. In this example, if we define $P' = [-0.95190, -0.30640]$, our new data is defined by $N' = P'X$. Rotated back, the "recovered" 1×10 data set is $(P')^T N'$ which is the data from Figure 3.8a projected on to the positively sloped eigenvector as shown in Figure 3.9. In order to determine how many components of the data to use in compression, one usually looks for a sharp drop off in the standard deviations of the principle components which are given by the square root of the diagonal entries of D.

3.3.2 Principle Components Analysis on Images

Principle components analysis can be used on images. Consider the sequence of six faces, taken from [6], and shown in Figure 3.10. Each face is a 211×240 image. Let X be a 6×50640 matrix where the ith row contains the color values for the pixel in the ith face pulled out column by column. From here, a principle components analysis is done exactly as in Section 3.3.1 and produces a new sequence of images, stored as the rows of $N = PX$, and shown in Figure 3.11.

The principle components in Figure 3.11 are arranged in order of descending importance as determined by variances given along the diagonal of D. The corresponding standard deviations are shown in Figure 3.12.

One would hope for a more dramatic drop off of bar heights in Figure 3.12 in order to choose components for compression. Compression and reconstruction of images in Figure 3.10 using different numbers of components are shown in Figure 3.13. R code is given in Appendix B.

3.3.3 Inverse Compositional Alignment with Principle Components Analysis

In Section 3.2 we saw that the ICA algorithm encountered some difficulty when tracking a single ant due to the fact that the image of the ant in any one frame does not exist unchanged in the next frame. Baker, Gross, and Matthews [2] change the assumption that the template image $T(\vec{x})$ from any given frame appears in the next frame into the assumption that

$$T(\vec{x}) - \sum_{i=1}^{m} \lambda_i A_i(\vec{x})$$

Figure 3.9: Recovered Data based on One Eigenvalue

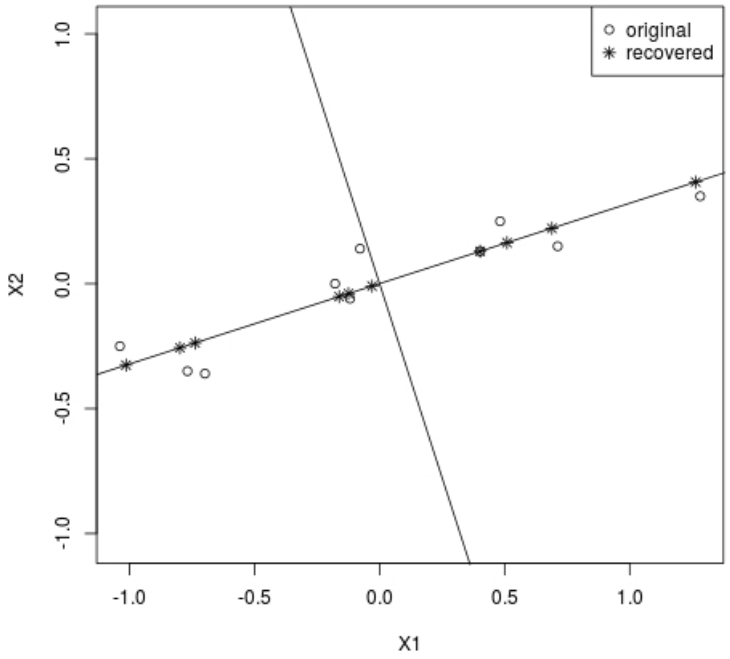


Figure 3.10: Sequence of Faces

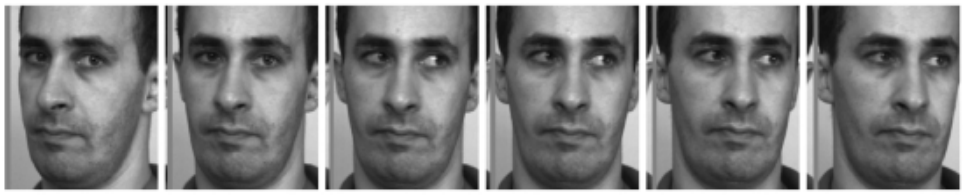


Figure 3.11: Principle Components for Figure 3.10

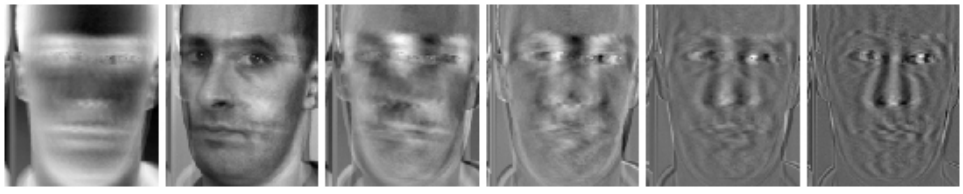


Figure 3.12: Standard Devations of Principle Components

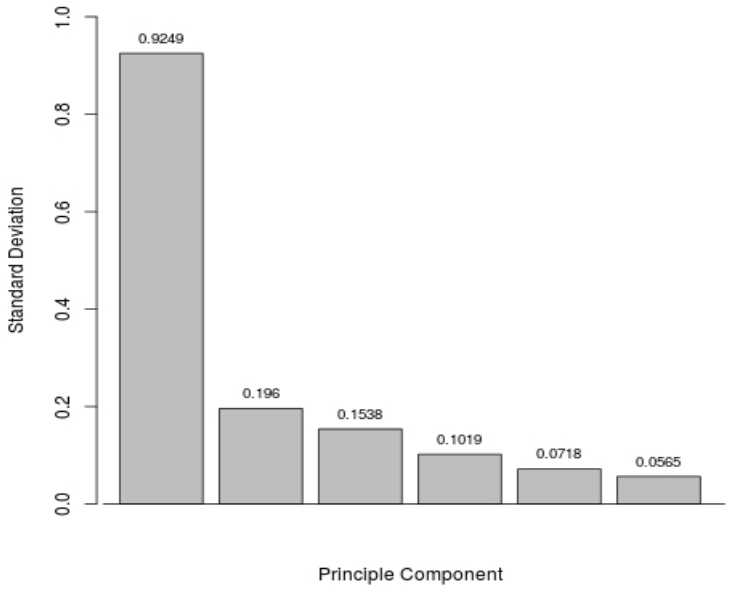


Figure 3.13: Image Recovery After Compression with the ith Row Using i Principle Components

appears in the next frame where $\{A_i\}_{i=1}^m$ is a sequence of known appearance variation images. The sequence of appearance variation images is typically formed by applying PCA to a set of training images and keeping the components that correspond to the top 90-95% of total variance. The goal then would be to minimize

$$\sum_{\vec{x}} \left[I(W(\vec{x};\vec{p})) - T(\vec{x}) - \sum_{i=1}^{m} \lambda_i A_i(\vec{x}) \right]^2$$

with respect to $\vec{p}$ and $\vec{\lambda} = (\lambda_1, \lambda_2, \cdots, \lambda_m)^T$. Baker, Gross, and Matthews [2] suggest three ICA based algorithms and here we review one of them, called the **simultaneous inverse compositional algorithm**, which uses inverse compositional alignment updates on the warp parameters along with additive updates for the appearance parameters given by $\vec{\lambda}$.

The goal of the simultaneous inverse compositional alignment algorithm is to minimize

$$SSE_4 := \sum_{\vec{x}} [I(W(\vec{x};\vec{p})) - T(W(\vec{x};\Delta\vec{p})) - \sum_{i=1}^{m} (\lambda_i + \Delta\lambda_i) A_i(W(\vec{x};\Delta\vec{p}))]^2 \tag{3.1}$$

with respect to $\Delta\vec{p}$ and $\vec{\lambda}$ in each iteration, at the end of which the warp is updated as

$$W(\vec{x};\vec{p}) \leftarrow W(\vec{x};\vec{p}) \circ W(\vec{x};\Delta\vec{p})^{-1} \tag{3.2}$$

and the appearance parameters as

$$\vec{\lambda} \leftarrow \vec{\lambda} + \Delta\vec{\lambda}. \tag{3.3}$$

As with all of the sum of squared errors in Chapter 2, SSE_4 is approximated using first order Taylor series expansions on $T(W(\vec{x};\Delta\vec{p}))$ and now also on $A_i(W(\vec{x});\Delta\vec{p})$ for $i = 1, 2, \ldots, m$. For details of the derivation of the algorithm which we now summarize, please see [2].

3.3.3.1 Alignment Images Example

In Chapter 5, we will see how this "ICA with PCA" algorithm performs. Here, we discuss our choice of appearance variation images for tracking the one upper left ant in the video discussed in this Chapter. First, we pulled 30 templates from still frames. Ideally, in order to capture a good range of variation in appearance, these templates would be pulled from several different points in

Algorithm 4 Simultaneous Inverse Compositional Alignment Algorithm

- Start with an initial estimate of $\vec{p}$. (And thereby an initial estimate of the warp function.)
- Set an error tolerance $\varepsilon > 0$.
- Set $||\Delta\vec{p}||$ to an arbitrary value greater than ε.
- Evaluate the Jacobian, $\partial W/\partial\vec{p}$ at $(\vec{x};\vec{0})$ at all pixels $\vec{x}$ in the template.
- Compute ∇T and ∇A_i for $i = 1, 2, \ldots, m$.

1: **while** $||\Delta\vec{p}|| > \varepsilon$ **do**
2: Compute $I(W(\vec{x};\vec{p}))$.
3: Compute the error image $I(W(\vec{x};\vec{p})) - T(\vec{x}) - \sum_{i=1}^{m} \lambda_i A_i(\vec{x})$.
4: Compute the steepest descent images, S, which is a $1 \times (n+m)$ vector of $h \times w$ images where the first n images are given by

$$(\nabla T + \sum_{i=1}^{m} \lambda_i \nabla A_i)\frac{\partial W}{\partial \vec{p}}$$

and the last m by $(A_1, A_2, \ldots, A_m)$.
5: Compute

$$\sum_{\vec{x}} S^T[I(W(\vec{x};\vec{p})) - T(\vec{x}) - \sum_{i=1}^{m} \lambda_i A_i(\vec{x})].$$

6: Compute the Hessian matrix $H = \sum_{\vec{x}} S^T S$.
7: Compute $(\Delta\vec{p}, \Delta\vec{\lambda})$ using

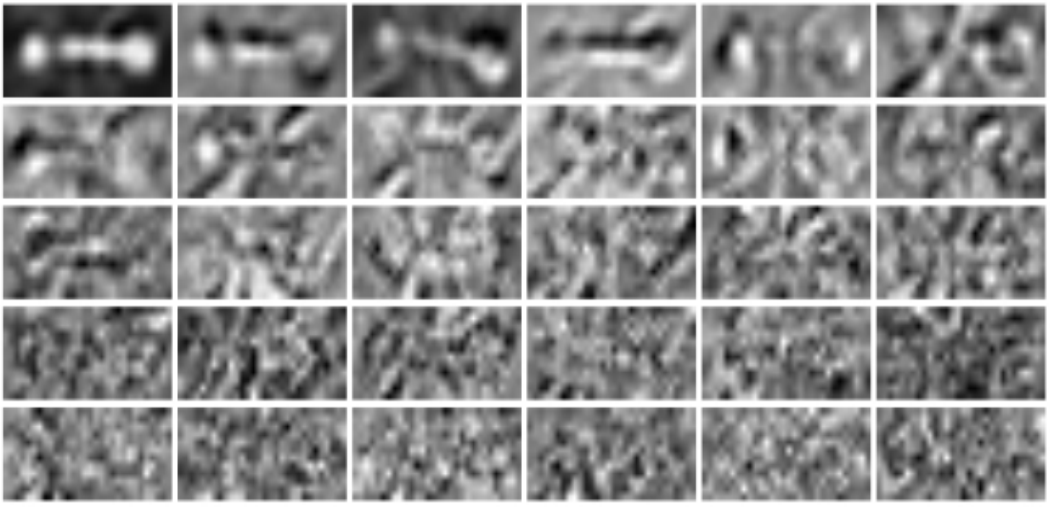
$$\begin{pmatrix} \Delta\vec{p} \\ \Delta\vec{\lambda} \end{pmatrix} = H^{-1} \sum_{\vec{x}} S^T[I(W(\vec{x};\vec{p})) - T(\vec{x}) - \sum_{i=1}^{m} \lambda_i A_i(\vec{x})].$$

8: Update the warp as $W(\vec{x};\vec{p}) \leftarrow W(\vec{x};\vec{p}) \circ W^{-1}(\vec{x};\Delta\vec{p}) = W(W^{-1}(\vec{x};\Delta\vec{p});\vec{p})$ and $\vec{\lambda} \rightarrow \vec{\lambda} + \Delta\vec{\lambda}$.
9: **end while**

the video. However, in this case it is too difficult to distinguish the particular ant in randomly selected frames. Hence, we pulled the templates from the first 30 frames manually by inspection. Due to the similarity among the different ants, we considered using all of the ants in the first frame as appearance variation images for tracking the one particular ant but ultimately decided against this since it will likely make our future goal of simultaneous multiple ant tracking difficult by averaging out distinguishing factors, subtle as they may be.

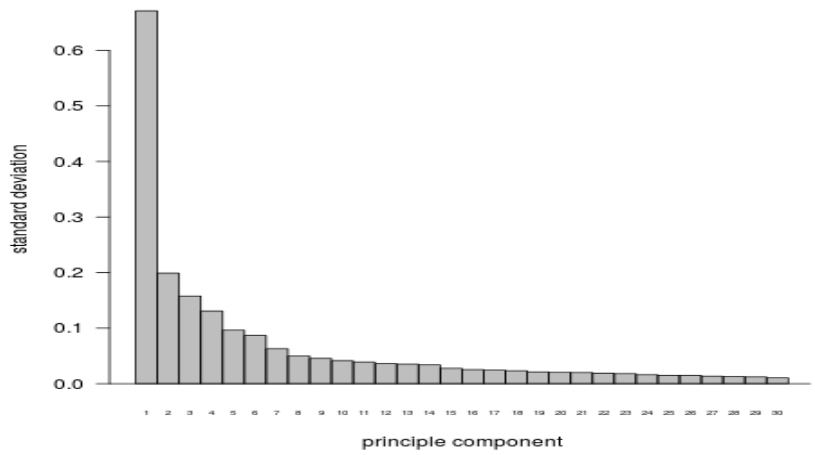
The principle components analysis basis, ordered across rows in decreasing importance, is shown in Figure 3.14.

Figure 3.14: Principle Components for 30 Ant Templates



Though the last two and a half rows of Figure 3.14 may appear to be noise, inspection of the standard deviations (Figure 3.15) again does not reveal an obvious dramatic drop-off. In [2], the authors suggest choosing the basis images corresponding to the top 95% of variance. Here, that would amount to 22 images. In Chapter 5 we consider several different numbers for selection of the principle components.

Figure 3.15: Standard Deviations of Principle Components in Figure 3.14



3.4 What's New?

We show some results for the inverse compositional alignment algorithm and inverse compositional alignment with principle components analysis in Chapter 5. These algorithms have been presented so far as finding a sub-image from one frame in the next image. In video tracking, the (presumably) located target then becomes the template for the next frame since a target in frame 10, for example, is more likely to look more like it did in frame 9 than it did in frame 1. The problem is that small errors in tracking will accumulate over time. That is, if the target starts to veer off towards a side of the tracking rectangle, we are then searching for this "crooked" ant the next time through. In the ant tracking problem, while it seems intuitive that adding a linear combination of "basis ants" should help to correct for this, we will see in practice that it is not always the case. We explore the novel idea of using the ICA with PCA algorithm with an extra PCA step where rather than the template being the last tracked ant, it is the reconstruction of the last tracked ant from the first principle component found using two images. One is the last tracked ant and the other is a "typical" looking ant pulled from the video at a place where it was not particularly contorted. In the case of the upper left ant of Figure 3.3, we used the template that was manually pulled from the first frame. While this "PCA within ICA with PCA" approach only makes it a few frames further after the ICA with PCA approach fails, we will see in Chapter 5 that it really helps to stabilize the tracked image all the way through until failure whereas the ICA with PCA algorithm goes through several frames of veering off and recovering before ultimate failure.

Currently, the only stopping rule for the gradient descent tracking algorithms discussed in this Chapter is that the change in parameters between iterations is becoming smaller than a user defined tolerance. That is, we continue through iterations of the ICA algorithm until $||\Delta\vec{p}|| \leq \varepsilon$ where ε has been specified in advance. As we have seen in Figure 3.6, for real video $||\Delta\vec{p}||$ is not necessarily decreasing due to the fact that the template from frame n does not actually exist unchanged in frame $n+1$. Furthermore, stopping the repositioning of the template just because

it is no longer moving does not mean that we have necessarily found the target. Indeed, if we lose the target completely so that the image at warp consists entirely of background pixels, the search will stop as it is no longer sensing any improvement through further iterations. On the other hand, we have often observed what seemed like an adequate template match even though $||\Delta\vec{p}|| > \varepsilon$ and the ICA algorithm is still iterating. These "wasted iterations" not only take up computation time but they sometimes resulted in the object being lost further down the line.

To formalize this idea of an "apparent match", we examine a histogram of the intensity values in the error image $E(\vec{x}) := I(W(\vec{x};\vec{p})) - T(\vec{x})$ and stop the algorithm (be it the LK algorithm, the ICA, ICA with PCA, PCA within ICA within PCA, etc...) at the first time that **either** $||\Delta\vec{p}|| \leq \varepsilon$ **or**

$$\frac{1}{2}\left(\left|\max_{\vec{x}} E(\vec{x})\right| + \left|\min_{\vec{x}} E(\vec{x})\right|\right) < c$$

for some user defined cutoff c. For most of our examples we had good performance when $c = 0.2$. Results are shown in Chapter 5.

Chapter 4

Probabilistic and Statistical Methods for Target Tracking

Probabilistic methods can be used in image alignment to increase both speed and robustness of the algorithm. Speed may be improved, for example, if we have a sense of the most likely next movement of the ant. As for robustness, deterministic tracking will often fail due to things such as target occlusion, inconsistencies in image brightness, and in the specific case of the video from Figure 3.3, the existence of multiple similar targets. For example, in frames where the target ant is in close proximity to another ant, the algorithms of Chapter 2 will likely become confused and fail. Additionally, the ant motion changes in somewhat predictable ways when encountering another ant or a container wall.

There are two main probabilistic/statistical methods used for target tracking through a sequence of images. The first is known as the **Mean Shift tracker** which uses the target's **color histogram**. A color histogram is just as it sounds. In greyscale, it is a histogram of the values in $[0, 1]$ used to make up the target. For color images, it involves histograms for each of the red, green, and blue channels used. The Mean Shift tracker involves searching the full next frame image for a region with a similar color histogram.

The second probabilistic/statistical method involves a **particle filter** which is a sequential Monte Carlo method that we detail in Section 4.2. In this Chapter, we also explore some subtle variations which appear to result in marginal improvements in speed. In Chapter 5, we will compare these algorithms to the deterministic alternative of Chapter 3.

4.1 Simple Estimation of the Distribution of Motion Parameters

In this Section, we revisit the simulated ant from Section 3.1. Recall that we were tracking three motion parameters: rotation, horizontal translation, and vertical translation. Let $\Delta\vec{p}_n = (\Delta\theta_n, \Delta a_n, \Delta b_n)^T$ be the change in these parameters for our ant between frames $n-1$ and n. We will assume that the ant's motion is time homogeneous and drop the subscripts on $\Delta\vec{p}_n$ and its individual components. By using training data from manual inspection of the ant location in the first N frames or possibly using one of the deterministic algorithms from Chapter 2 along with manual verification of tracking, we can give method of moments, or equivalently in this case, maximum likelihood estimators for the parameters of the normal distributions used to generate the ant's motion. (Indeed, the more sophisticated probabilistic tracking algorithms such as, for example, the particle filter discussed in Section 4.2 all involve some sort of manual learning phase.) More generally though, we can forgo the inside knowledge that the ant's motion parameters are governed by normal distributions and instead, generate purely empirical distributions for $\Delta\theta$, Δa, and Δb.

Once parametric or non-parametric distributions for the components of $\Delta\vec{p}$ are estimated we can, at each frame, generate a collection of likely positions and orientations for the next frame. Each proposed triple is used in the update $\vec{p} \leftarrow \vec{p} + \Delta\vec{p}$ and produces its own error image. Naturally, we wish to favor the better proposal, in terms of the error image, to form a finalized composite guess. Recall that the error image is defined, at pixel $\vec{x}$, as the difference $I(W(\vec{x};\vec{p})) - T(\vec{x})$ between the image at the proposed warp and the template from the previous frame. We will ultimately define the warp function as a weighted average of the proposed moves where the weights are inversely proportional to the sum over pixels of the absolute error image values. That is, the weight for the jth proposed move will be W_i, defined as

$$W_j \propto \frac{1}{\sum_{\vec{x}} |I(W(\vec{x};\vec{p} + \Delta\vec{p}_j)) - T(\vec{x})|}$$

where $\Delta\vec{p}_j$ is the jth sampled change in parameters. We propose to use the weighted average of proposed tracking rectangles not as our next position in tracking but as the first proposed move

for the gradient descent methods described in Chapter 2. The "compositional guess" parameter, which we denote by $\vec{p}_C$ is

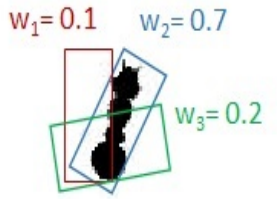
$$\vec{p}_C = \sum_j W_j \vec{p}_j \tag{4.1}$$

where $\vec{p}_j$ is the jth proposed parameter.

Example:

We illustrate this compositional parameter guess to produce a first guess for $\Delta\vec{p}$ for the ICA algorithm using only 3 template guesses (illustrated in Figure 4.1) that were proposed after training the data on the first 30 frames. By inspection, it is apparent that the second guess is most useful,

Figure 4.1: Three proposed Rectangles



which is reflected in its relatively high assigned weight compared to the others. If the three guesses had the following parameters, then we could generate the composite guess using weighted sum. For example, if the proposals are

$$\vec{p}_1 = \{1.5, 0.4, 0.5\}^T$$

$$\vec{p}_2 = \{1, 0.5, 0.5\}^T$$

$$\vec{p}_3 = \{0.7, 0.4, 0.3\}^T$$

The composite guess, $\vec{p}_C$, is

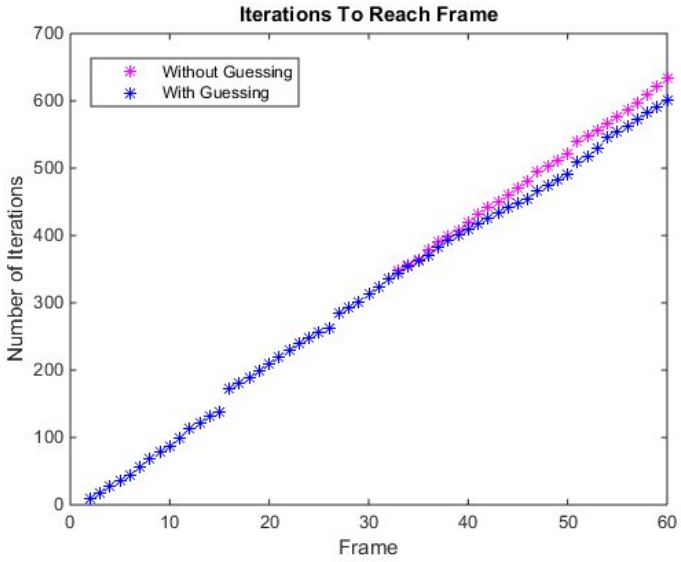
$$\vec{p}_C = \sum_i w_i \vec{p}_i = 0.1 \cdot \{1.5, 0.4, 0.5\}^T + 0.7 \cdot \{1, 0.5, 0.5\}^T + 0.2 \cdot \{0.7, 0.4, 0.3\}^T$$

$$\vec{p}_C = \{0.47, 0.46, 0.99\}^T$$

which is, as expected, closest to the second guess.

The ICA algorithm was run with composite guesses based on only 3 proposals for $\vec{p}$ as a starting parameter beginning after training through the first 30 frames. The performance of the algorithm with and without the probabilistic predictor is shown in Figure 4.2. As expected, for frames 31 and beyond, the composite guess algorithm begins to process frames quicker than the usual ICA algorithm and gives an approximate 5% increase in performance. Obviously, there is an efficiency trade-off to generate the composite guesses, but overall run time was still slightly improved.

Figure 4.2: Performance Difference With Probabilistic Tracking



4.2 Particle Filtering

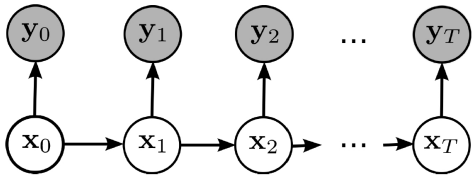
4.2.1 General Particle Filtering

Particle filters are important sampling based sequential Monte Carlo methods that are often employed to sample from and provide estimates of the distribution of a set or subset of latent variables in a hidden Markov model given observations. They are constructed specifically to provide

updated sampling and estimation when additional observations become available without reprocessing all observations.

Consider a hidden Markov model with underlying and unobserved states $\{X_n\}_{n=0}^{\infty}$, transition density $\pi(x_n|x_{n-1})$, and an initial distribution with density $\pi(x_0)$. Suppose that $\{Y_n\}_{n=0}^{\infty}$ represents a sequence of observable variables that are conditionally independent when the unobserved states are given and where each Y_n is related to the unobserved process through X_n and a "measurement model" density $\pi(y_n|x_n)$. Such a model is depicted in Figure 4.3.

Figure 4.3: A Hidden Markov Model



The goal of particle filtering is to sample from the density $\pi(x_{0:n}|y_{0:n})$, where $u_{i:j}$ denotes the vector $(u_i, u_{i+1}, \ldots, u_j)$ for $j \geq i$, sequentially in the sense that samples from $\pi(x_{0:n-1}|y_{0:n-1})$ will be used along with a new observation y_n to produce the desired points. The sampled points (called "particles") can then be used to approximate, for example, expectations of the form $\mathsf{E}[f(X_{0:n})|y_{1:n}]$ and marginal distributions $\pi(x_i|y_{0:n})$ for $i = 0, 1, \ldots, n$.

The Markov and conditional independence assumptions allow us to write our "target density" for the model in Figure 4.3 in the recursive form

$$\begin{aligned} \pi(x_{0:n}|y_{0:n}) &\propto \pi(y_0|x_0)\pi(x_0)\textstyle\prod_{i=1}^{n}\pi(y_i|x_i)\pi(x_i|x_{i-1}) \\ &\propto \pi(y_n|x_n)\cdot\pi(x_n|x_{n-1})\cdot\pi(x_{0:n-1}|y_{0:n-1}) \end{aligned} \tag{4.2}$$

so that draws (sampled values) from $\pi(x_{0:n}|y_{0:n})$ can be recursively related to draws from $\pi(x_{0:n-1}|y_{0:n-1})$.

4.2.1.1 Sequential Importance Sampling (SIS)

When one can not sample directly from $\pi(x_{0:n}|y_{0:n}) \propto h(x_{0:n}|y_{0:n})$, importance sampling can be used to instead sample points from a more tractable **importance sampling density**, q, and these points can be used to estimate the target density. Sequential importance sampling (SIS) is importance sampling for $\pi(x_{0:n}|y_{0:n})$ in such a way where n-dimensional draws from $\pi(x_{0:n-1}|y_{0:n-1})$ are "extended" to $(n+1)$-dimensional points that are then reweighted to produce draws from $\pi(x_{0:n}|y_{0:n})$. To this end, the importance sampling density is chosen to have the form

$$q(x_{0:n}|y_{0:n}) = q(x_0|y_0) \prod_{i=1}^{n} q(x_i|x_{i-1}, y_i) \tag{4.3}$$

so that it may be sampled from recursively. A draw from $q(x_{0:n}|y_{0:n})$ may be reweighted to a draw from $\pi(x_{0:n}|y_{0:n})$ via the weight $w(x_{0:n}|y_{0:n}) := \pi(x_{0:n}|y_{0:n})/q(x_{0:n}|y_{0:n})$ (or unnormalized weight $h(x_{0:n}|y_{0:n})/q(x_{0:n}|y_{0:n})$) as

$$\pi(x_{0:n}|y_{0:n}) = w(x_{0:n}|y_{0:n})\, q(x_{0:n}|y_{0:n}).$$

Due to the forms of (4.2) and (4.3), the weights can also be computed recursively as

$$\begin{aligned} w(x_{0:n}|y_{0:n}) &= \frac{\pi(x_{0:n}|y_{0:n})}{q(x_{0:n}|y_{0:n})} = \frac{\pi(x_{0:n}|y_{0:n})}{q(x_{0:n-1}|y_{0:n-1})q(x_n|x_{n-1}, y_n)} \\ &= \frac{\pi(x_{0:n-1}|y_{0:n-1})}{q(x_{0:n-1}|y_{0:n-1})} \cdot \frac{\pi(x_{0:n}|y_{0:n})}{\pi(x_{0:n-1}|y_{0:n-1})q(x_n|x_{n-1}, y_n)} \\ &=: w(x_{0:n-1}|y_{0:n-1}) \cdot \alpha(x_{0:n}|y_{0:n}) \end{aligned} \tag{4.4}$$

where $\alpha(x_{0:n})$ is an **incremental weight function** that is defined as

$$\alpha(x_{0:n}|y_{0:n}) := \frac{\pi(x_{0:n}|y_{0:n})}{\pi(x_{0:n-1}|y_{0:n-1})q(x_n|x_{n-1}, y_n)}$$

for $n \geq 1$.

The SIS algorithm is detailed in Algorithm 5. At any time n, one can estimate $\pi(x_{0:n}|y_{0:n})$ using

$$\widehat{\pi}(x_{0:n}|y_{0:n}) = \frac{1}{N} \sum_{i=1}^{N} w(X_{0:n}^{(i)})\, \mathbb{1}[X_{0:n}^{(i)} = x_{0:n}] \tag{4.5}$$

where $\mathbb{1}[X_{0:n}^{(i)} = x_{0:n}]$ is the indicator function that takes on the value 1 when $x_{0:n} = X_{0:n}^{(i)}$, and zero otherwise.

One can also obtain approximate dependent draws from $\pi(x_{0:n}|y_{0:n})$ by sampling from (4.5). That is, by sampling from the set of values $\{X_{0:n}^{(1)}, X_{0:n}^{(2)}, \ldots, X_{0:n}^{(N)}\}$ using respective weights $\{W_n^{(1)}$, $W_n^{(2)}$, ..., $W_n^{(N)}\}$ which is a resampling of values that were sampled from q.

Algorithm 5 Sequential Importance Sampling (SIS) Algorithm

- Sample $X_0^{(1)}, X_0^{(2)}, \ldots, X_0^{(N)} \overset{iid}{\sim} q(x_0|y_0)$.
- Compute weights $w(X_0^{(i)})$ for $i = 1, 2, \ldots, N$ as

$$w(X_0^{(i)}) = \frac{\pi(X_0^i|y_0)}{q(X_0^{(i)}|y_0)}.$$

1: **for** $n \geq 1$ **do**
2: Sample $X_n^{(1)}, X_n^{(2)}, \ldots, X_n^{(N)}$ independently with $X_n^{(i)} \sim q(x_n|X_{n-1}^{(i)}, y_n)$.
3: Compute weights $w(X_{0:n}^{(i)}|y_{0:n}) = w(X_{0:n-1}^{(i)}|y_{0:n}) \cdot \alpha(X_{0:n}^{(i)}|y_{0:n})$ for $i = 1, 2, \ldots, N$.
4: **end for**

4.2.1.2 Sequential Importance Sampling with Resampling (SIR)

In practice, iteration of the SIS algorithm leads to a "degeneracy of weights" problem (see, for example, [1]) where the weights of all but one particle will approach zero, causing the method to break down and give meaningless results. One way to avoid the issue of degenerate weights is to implement a resampling scheme at each time step. This sequential importance sampling **with resampling** (SIR) algorithm is described in Algorithm 6.

(Note that step 4 is not an error. While the particles $\{X_{0:n-1}^{(i)}\}_{i=1}^N$ have respective weights $\{W_{n-1}^{(i)}\}_{i=1}^N$, the resampled particles $\{\widetilde{X}_{0:n-1}^{(i)}\}_{i=1}^N$ have equal weight $1/N$. That is, $w(\widetilde{X}_{0:n-1}^{(i)}) = 1/N$. This constant factor in the recursive weight formulation is not important as the weights will be normalized to $W_n^{(i)}$.)

Algorithm 6 Sequential Importance Sampling with Resampling (SIR) Algorithm

- Sample $X_0^{(1)}, X_0^{(2)}, \ldots, X_0^{(N)} \overset{iid}{\sim} q(x_0|y_0)$.
- Compute weights $w(X_0^{(i)})$ for $i = 1, 2, \ldots, N$ as

$$w(X_0^{(i)}) = \frac{\pi(X_0^i|y_0)}{q(X_0^{(i)}|y_0)}.$$

- Compute normalized weights

$$W_0^{(i)} = \frac{w(X_1^{(i)})}{\sum_{j=1}^N w(X_1^{(j)})}$$

for $i = 1, 2, \ldots, N$.

- Sample N points, $\widetilde{X}_0^{(1)}, \widetilde{X}_0^{(2)}, \ldots, \widetilde{X}_0^{(N)}$, with replacement, from the set $\{X_0^{(1)}, X_0^{(2)}, \ldots, X_0^{(N)}\}$ with respective probabilities $\{W_0^{(1)}, W_0^{(2)}, \ldots, W_0^{(N)}\}$.

1: **for** $n \geq 1$ **do**
2: Sample $X_n^{(1)}, X_n^{(2)}, \ldots, X_n^{(N)}$ independently with $X_n^{(i)} \sim q(x_n|X_{n-1}^{(i)}, y_n)$.
3: Extend each "particle" $\widetilde{X}_{0:n-1}^{(i)}$ to particles $(\widetilde{X}_{0:n-1}^{(i)}, X_n^{(i)})$.
4: Compute associated weights $w(\widetilde{X}_{0:n-1}^{(i)}, X_n^{(i)}) := \alpha(\widetilde{X}_{0:n-1}^{(i)}, X_n^{(i)})$ for $i = 1, 2, \ldots, N$.
5: Compute the normalized weights

$$W_n^{(i)} = \frac{w(\widetilde{X}_{0:n-1}^{(i)}, X_n^{(i)})}{\sum_{j=1}^N w(\widetilde{X}_{0:n-1}^{(j)}, X_n^{(j)})}$$

for $i = 1, 2, \ldots, N$.
6: Sample N n-dimensional points, $\widetilde{X}_{0:n}^{(1)}, \widetilde{X}_{0:n}^{(2)}, \ldots, \widetilde{X}_{0:n}^{(N)}$, with replacement, from the set $\{(\widetilde{X}_{0:n-1}^{(1)}, X_n^{(i)}), (\widetilde{X}_{0:n-1}^{(2)}, X_n^{(2)}), \ldots, (\widetilde{X}_{0:n-1}^{(N)}, X_n^{(N)})\}$ with respective probabilities $\{W_n^{(1)}, W_n^{(2)}, \ldots, W_n^{(N)}\}$.
7: Note that $\widetilde{X}_{0:n}^{(1)}, \widetilde{X}_{0:n}^{(2)}, \ldots, \widetilde{X}_{0:n}^{(N)}$ are now equally weighted particles, each with weight $1/N$, so assign weights $w(\widetilde{X}_{0:n}^{(i)}) = 1/N$ for $i = 1, 2, \ldots, N$.
8: **end for**

4.2.2 Particle Filtering for Ants

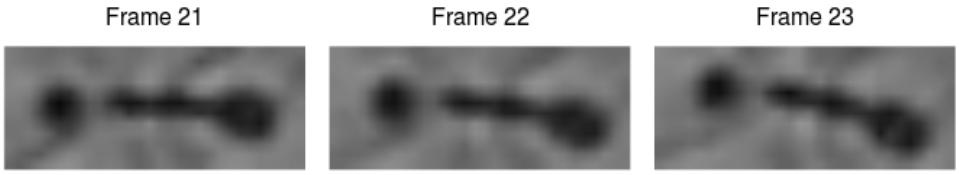
In [8], Kahn, Bach, and Dellaert use a particle filter to simultaneously track all 20 ants in the video depicted in Figure 3.3. In this context, X_n, is a 60 dimensional vector containing the x coordinates, y coordinates, and rotation angle for each ant. $X_{0:n}$ is a $60 \times n$ dimensional vector containing all of this information through $n+1$ frames of video. The observed variables Y_n are large arrays containing all of the pixel information for the entire full image in the nth frame. Conditional on the ant positions and orientations being known, each pixel of the image in the nth frame can easily be classified as either a foreground pixel or a background pixel. Under the assumption that the foreground pixels are independent of the background pixels, one only needs to evaluate the **appearance likelihood model** density $\pi(y_n|x_n)$ over small (in this case rectangular) templates defined by the template to image warp function as terms outside of templates are presumed background and therefore will be constant and will cancel out of calculations. The authors in [8] assume that the intensity values of the pixels in the rectangular templates, organized into one long column vector, have a multivariate t-distribution for which parameters were estimated from a set of training images. For the **motion model** with density $\pi(x_n|x_{n-1})$, they assumed that marginally each ant had a $\Delta\vec{p}$ that consisted of independent mean zero normal components whose variances were also estimated from the training images. Furthermore, $\pi(x_n|x_{n-1})$ included ant interaction modeling that we will not discuss here as we are only considering tracking a single target.

Chapter 5

Results and Conclusions

In this Chapter, we give some results and comparisons for the algorithms discussed in this thesis. We will see tracking problems in the majority of the methods at or around frame 22. This is because it is the first time in the video where the ant is really starting to bend. In Figure 5.1, we have manually pulled these templates to show this.

Figure 5.1: A Trouble Spot: Templates Manually Pulled from Frames 21-23

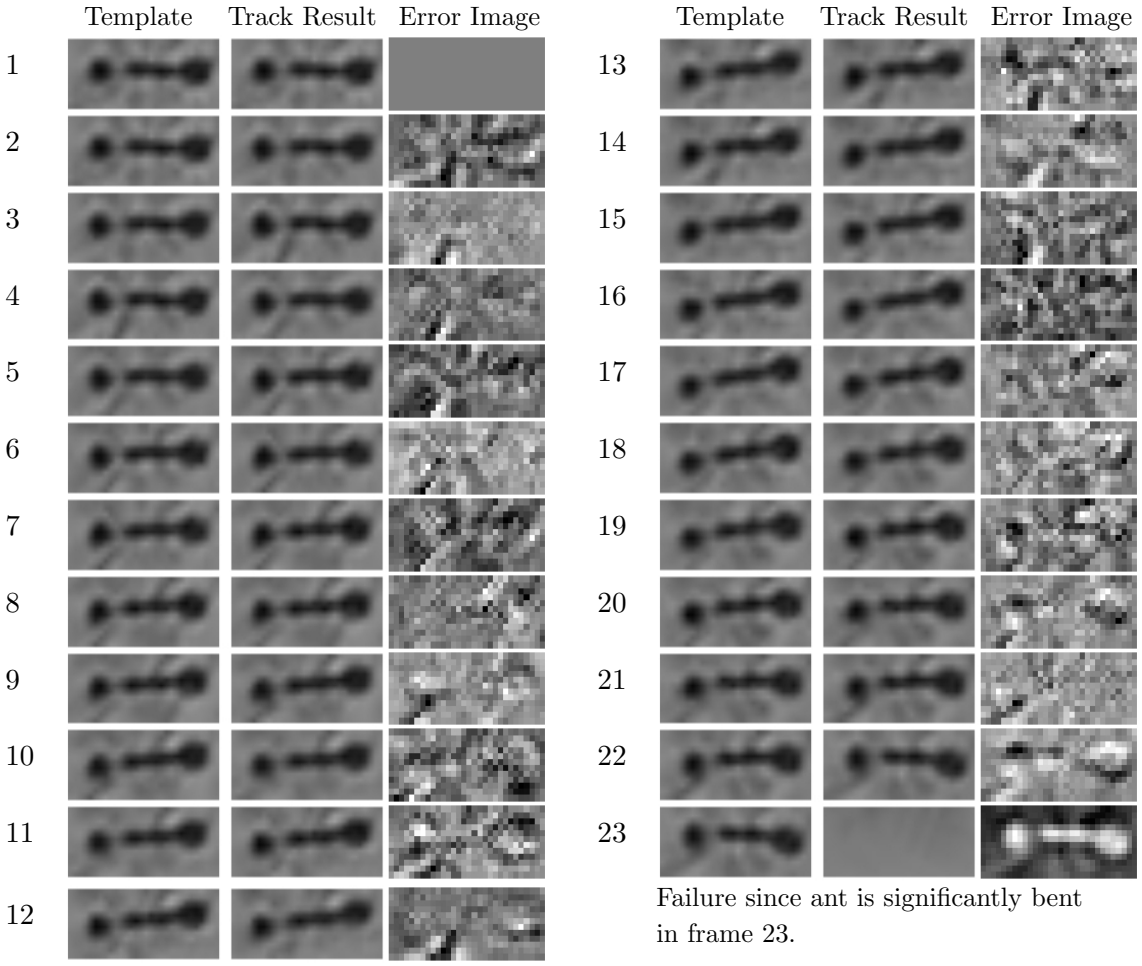


5.1 ICA Algorithm

In Figure 5.2, we show, for each frame, the current template, the end result of tracking which becomes the next template, and the error image of the ICA algorithm which fails at frame 23. This is likely due to the ant starting with an apparently significant bend around this time as shown in Figure 5.1.

In Figure 5.2, we can see that the template is becoming more and more angled. This is the compounding error due to the ant found in one frame becoming the template for the next frame. This updating is necessary since the ant in frame 15, for example, is much more likely to be similar to the ant in frame 14 as opposed to the ant in frame 1. Just as an experiment, we ran the ICA

Figure 5.2: ICA Algorithm Tracking Frame to Frame with Failure on Frame 23



Failure since ant is significantly bent in frame 23.

algorithm without updating the template at all. That is, the template pulled from frame 1 is used throughout. The results are shown in Figure 5.3. Here, we lose the ant at frame 18 even though the ant seems to maintain (Figure 5.4) a pretty constant appearance between frames 17 and 19.

5.2 ICA with Error Histograms

The ICA and other gradient descent algorithms stop when the parameter change between iterations gets very small– specifically when $||\Delta\vec{p}|| \leq \varepsilon$ for some user defined ε. Throughout this thesis, we use $\varepsilon = 0.05$ which was a number arrived at by trial and error. At no point do they account for when the warped image actually "looks good". In fact, sometimes intermediate iterations appear to offer a decent match, but because $||\Delta\vec{p}|| > \varepsilon$, the search continues, sometimes, even to the point of failure. For example, we see in Figure 3.6 that the ICA algorithm goes through 7 iterations to get down to $||\Delta\vec{p}|| \leq 0.05$, but by the 6th image (5th iteration), the tracked ant (image at warp) looks respectable. This is also reflected in the error image, which is starting to look more irregular and noisy. Hence, we propose a new stopping rule that ends the search the first time **either** $||\Delta\vec{p}|| \leq \varepsilon$ **or** a histogram of the intensity values in the histogram gets "narrow enough". Specifically, we defined "narrow enough" as

$$\frac{1}{2}\left(\left|\max_{\vec{x}} E(\vec{x})\right| + \left|\min_{\vec{x}} E(\vec{x})\right|\right) < c$$

for some user defined cutoff c. Here, $E(\vec{x})$ is the error image defined as $E(\vec{x}) = I(W(\vec{x}); \vec{p}) - T(\vec{x})$ and the maximum is taken over all pixels in the template region. We generally saw good results, cut down on the number of iterations, and in some cases, avoided a tracking failure, with $c = 0.2$.

For tracking from frame 1 to frame 2, this shaved off 2 iterations. Intermediate histograms of the intensities of the error image are shown in Figure 5.5.

Final tracking results after adding this error image criterion are given in Figure 5.6. Even though we were only able to track the target for 2 more frames, comparisons of Figure 5.6 and Figure 5.2 show that the tracked image was more stable throughout. Figure 5.7 compares how many iterations were used to track each frame when running the ICA algorithm with and without

Figure 5.3: ICA Algorithm with Constant Template Tracking Frame to Frame with Failure on Frame 18

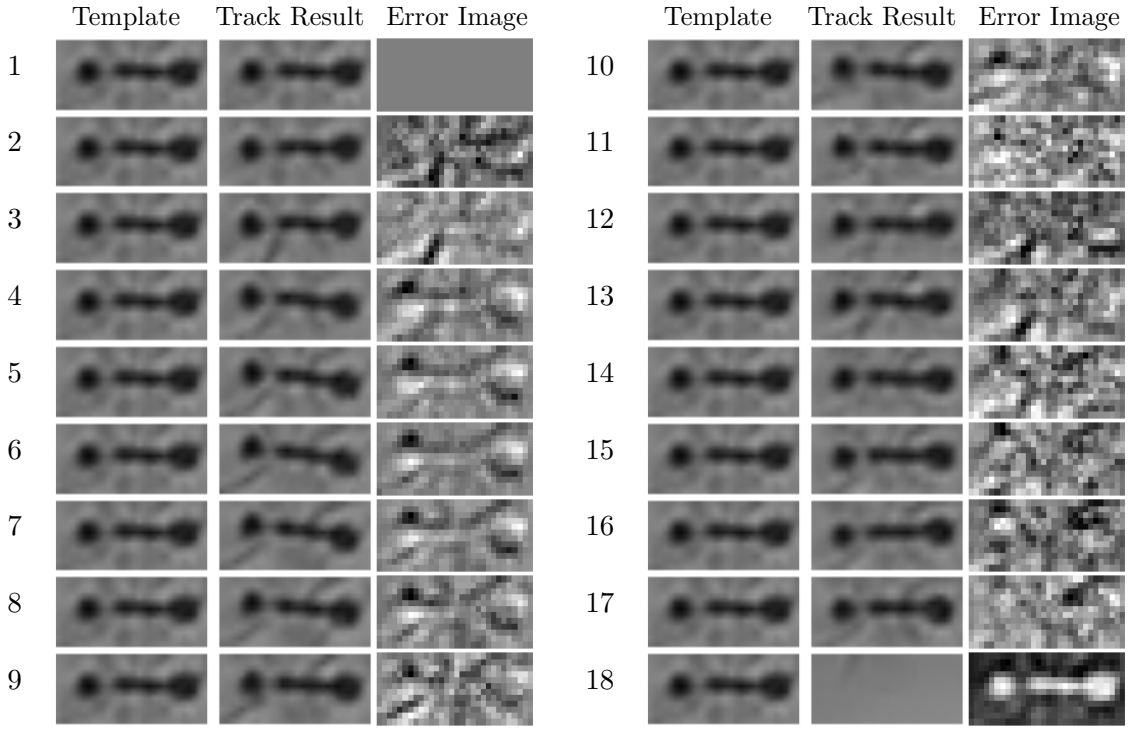


Figure 5.4: A Trouble Spot: Templates Manually Pulled from Frames 17-19

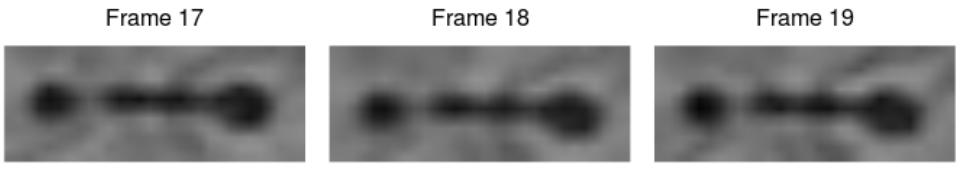
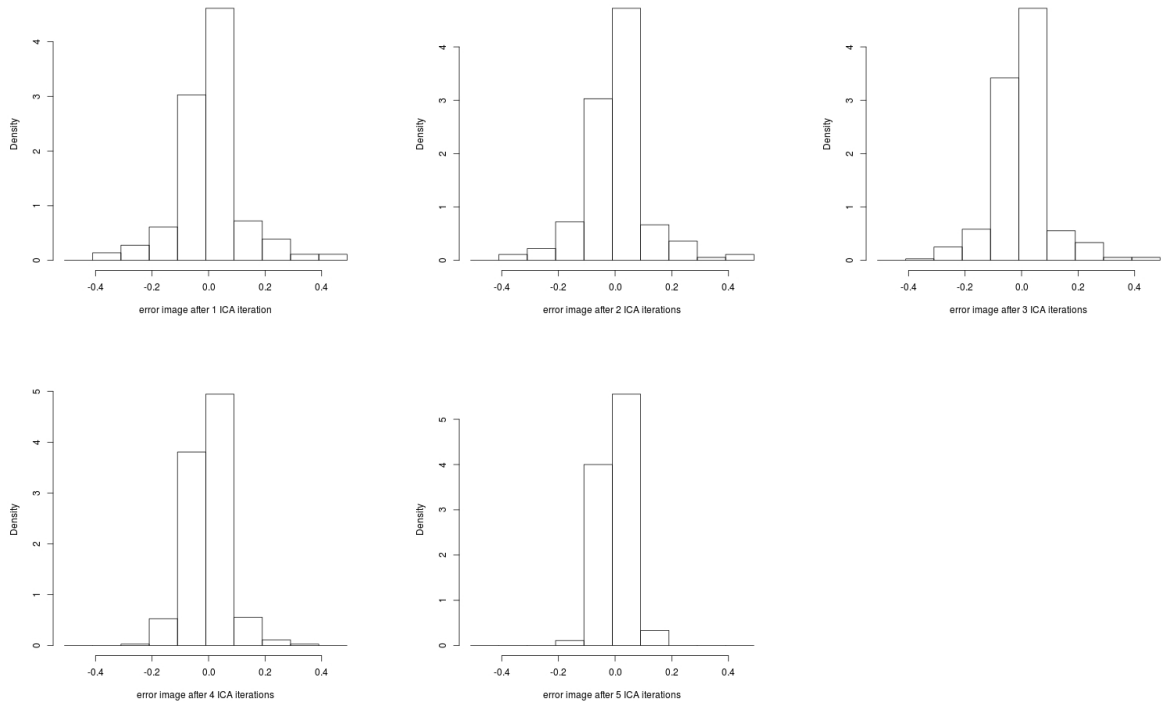


Figure 5.5: ICA Algorithm Tracking from Frame 1 to Frame 2: Error Histograms



the error image stopping rule, and it is clear that using the new stopping rule saves some iterations.

5.3 ICA with Principle Components Analysis

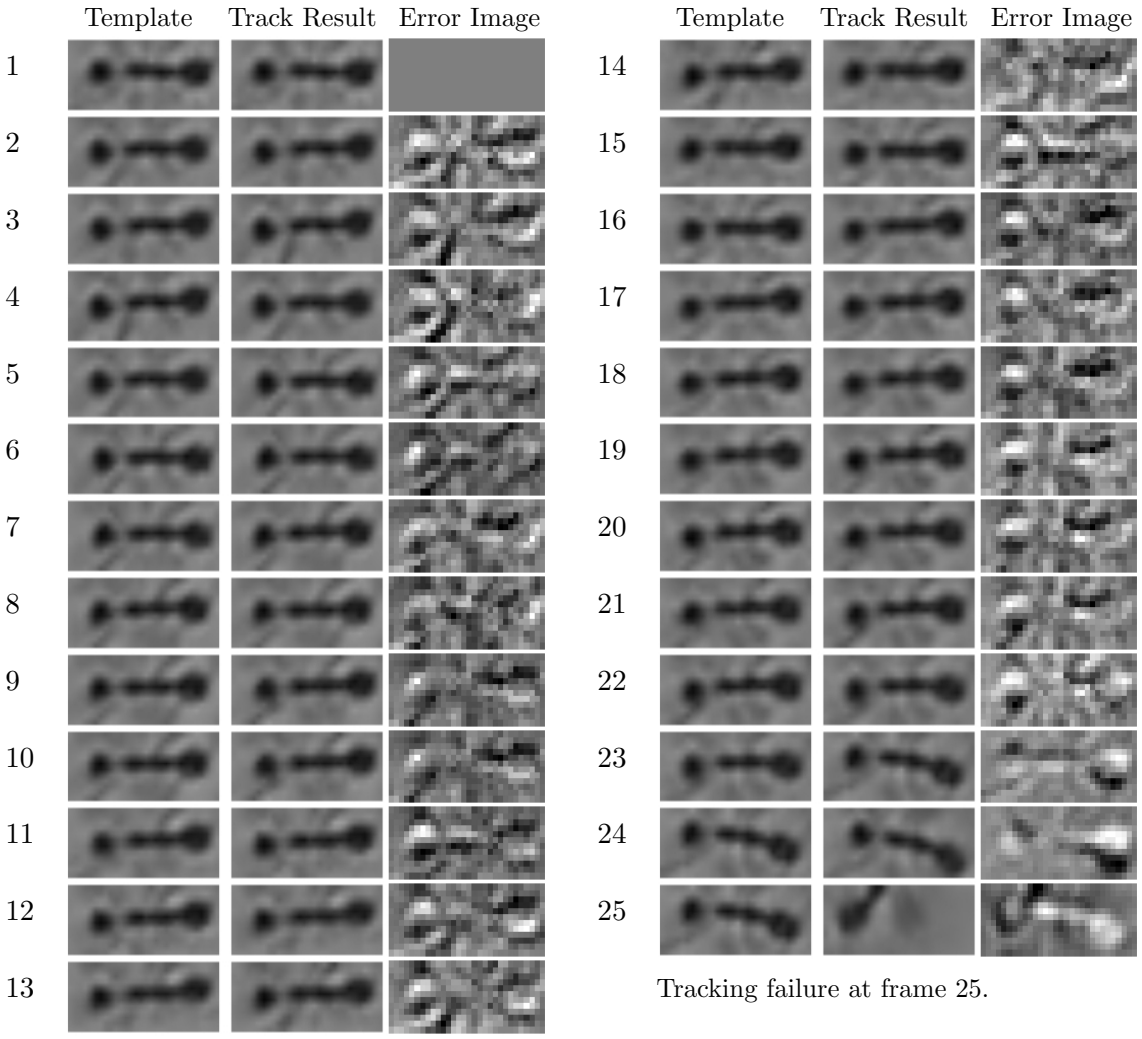
We now consider the ICA algorithm with principle components analysis using as was discussed in Section 3.3.3. In all examples, we chose to use 5 principle components and saw no real improvement using more. The number of iterations to track from frame 2 to frame 3 are shown in Figure 5.8. Notice how the appearance images adjusted template, actually almost disappears and then returns. The full tracking through frames is shown in Figure 5.9. We again begin to have tracking failure at frame 22. We were surprised to find that this actually performed one frame worse than the ICA algorithm from Figure 5.2. Furthermore, we have not even yet left the frame range from which the ant templates were manually sampled in order to construct the PCA basis.

In Figure 5.11, we see (as expected) that the ICA algorithm with principle components analysis using 5 principle components and the additional error image stopping rule achieved some (modest) computational savings over the algorithm without the error image stopping rule.

5.4 Principle Components Analysis within ICA with PCA

Note that in Figure 5.10, the ant starts rotating counterclockwise around frame 12. As in Figure 5.2, the error is compounding since the tracked image becomes the template for the next frame. We added another layer of PCA which we call "PCA within ICA with PCA" where, rather than the template being the last tracked ant, it is the reconstruction of the last tracked ant from the first principle component found using two images. One is the last tracked ant and the other is a "typical" looking ant pulled from the video at a place where it was not particularly contorted. In the case of the upper left ant of Figure 3.3, we used the template that was manually pulled from the first frame. We see in Figure 5.12 that we were able to track the ant for several more frames and more stability before failure.

Figure 5.6: ICA Algorithm Tracking Frame to Frame with Additional Error Image Stopping Criterion: Failure on Frame 25



Tracking failure at frame 25.

Figure 5.7: Comparison of the Number of Iterations Used to Track Frames with the ICA Algorithm and the ICA Algorithm with an Error Stopping Rule

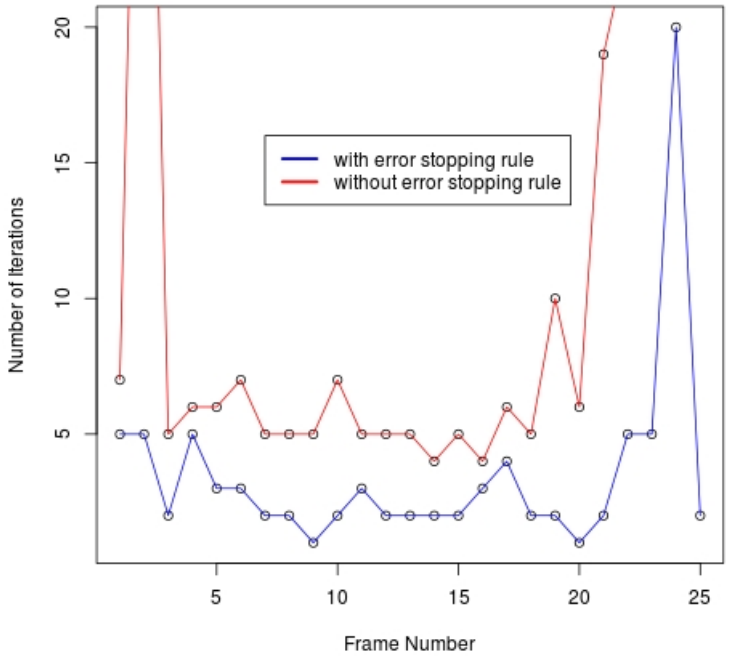


Figure 5.8: ICA with 5 Component PCA: 45 Iterations Between Frames 2 and 3, Templates are Adjusted with Appearance Images

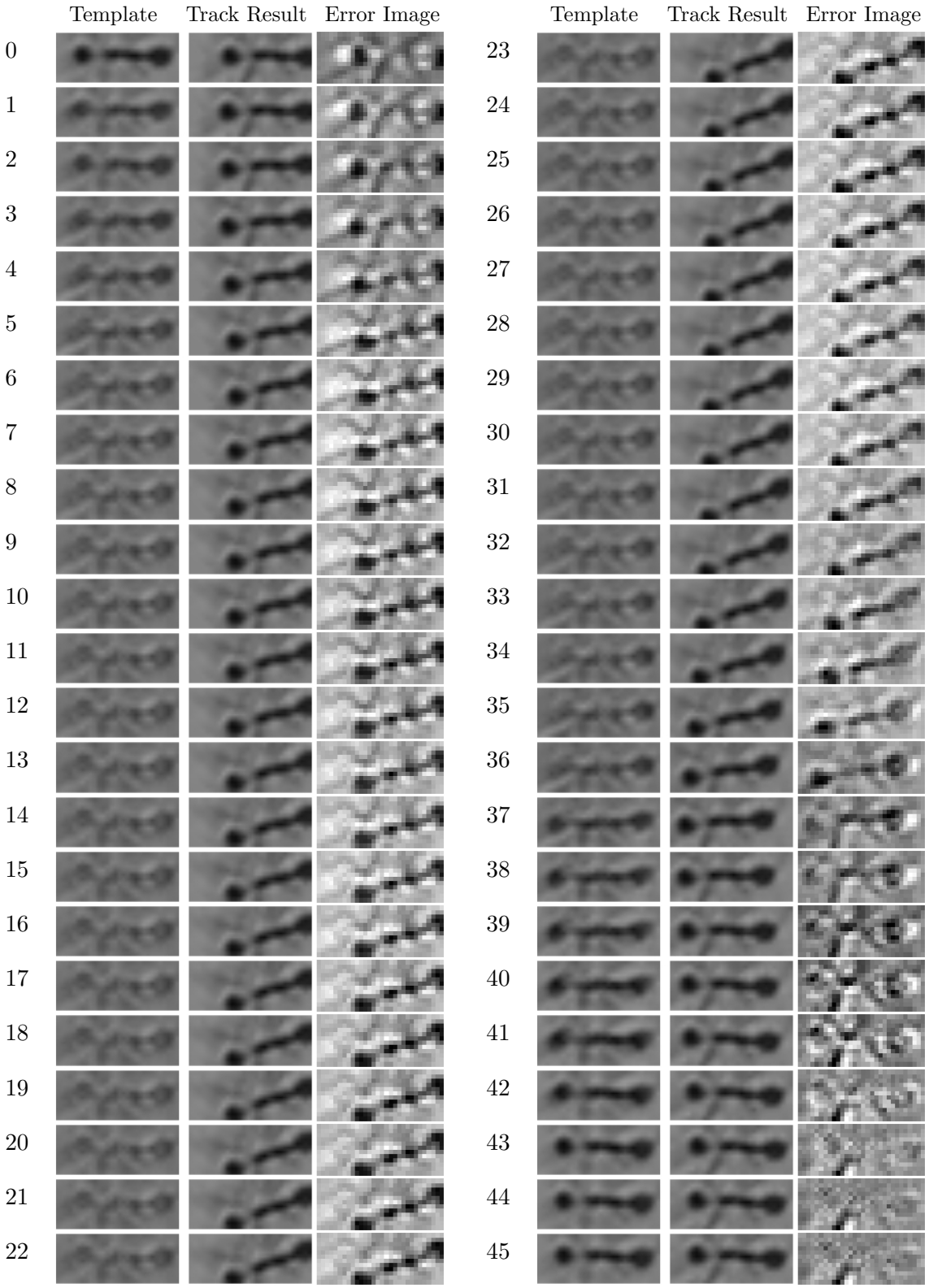
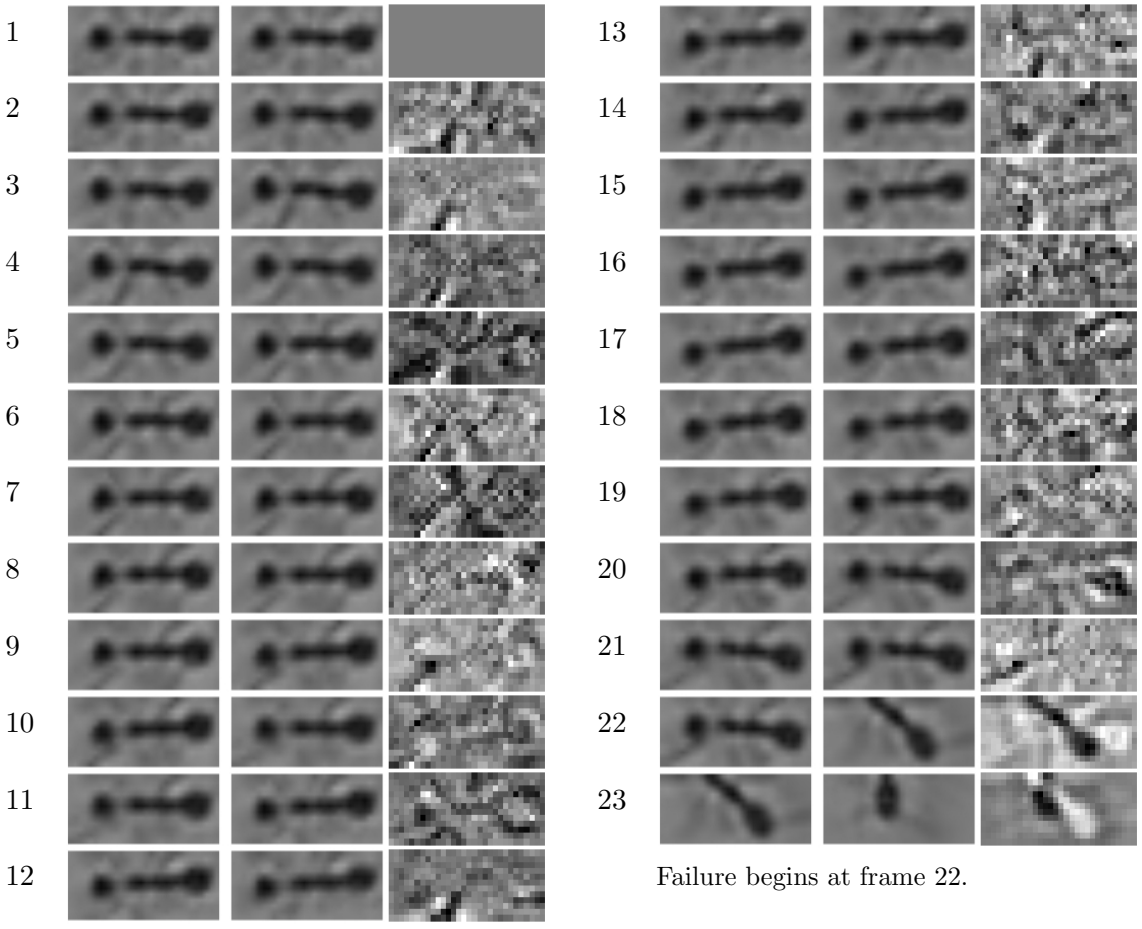
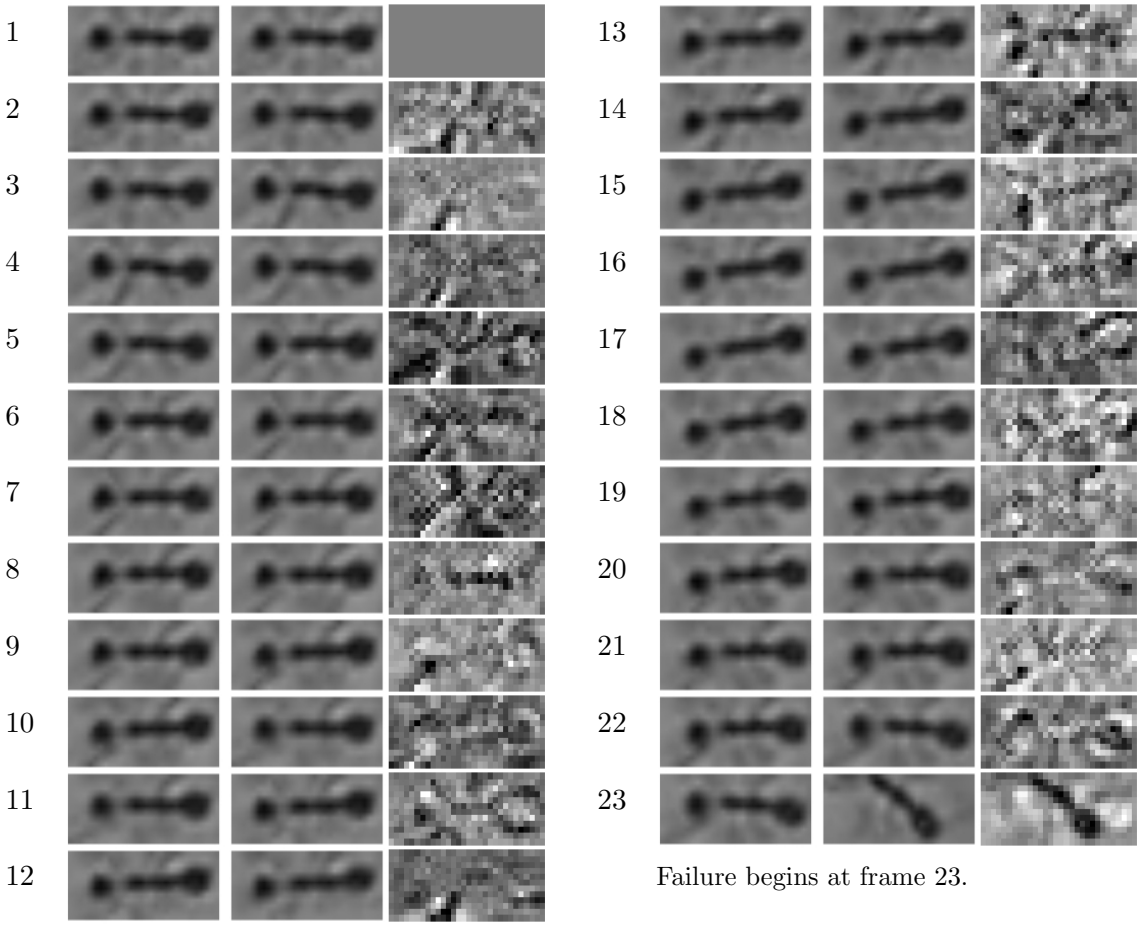


Figure 5.9: ICA with PCA Algorithm Tracking Frame to Frame with Failure on Frame 22



Failure begins at frame 22.

Figure 5.10: ICA with PCA Algorithm Tracking Frame to Frame with Error Image Exit Criterion: Failure on Frame 23



Failure begins at frame 23.

Figure 5.11: Comparison of the Number of Iterations Used to Track Frames with the ICA with PCA Algorithm and the ICA Algorithm with an Error Stopping Rule

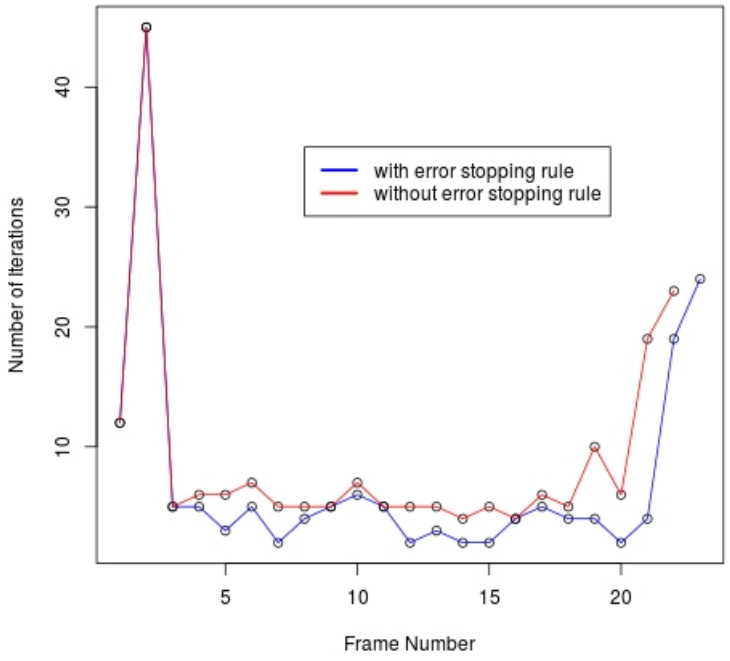
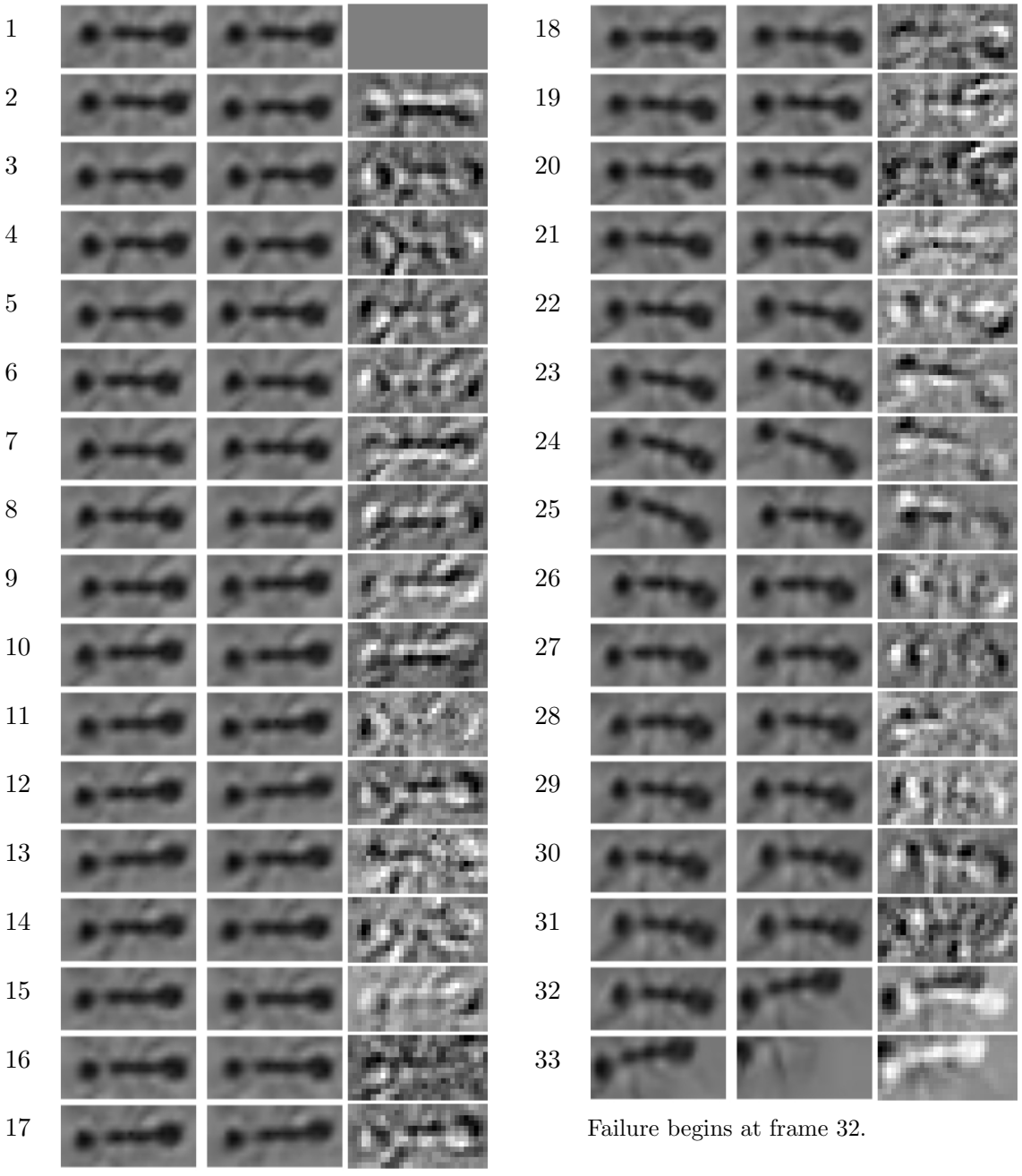


Figure 5.12: PCA within ICA with PCA Algorithm Tracking Frame to Frame with Error Image Exit Criterion: Failure on Frame 33



Failure begins at frame 32.

Bibliography

[1] C. Andrieu, A. Doucet, and R. Holenstein. Particle Markov chain Monte Carlo methods. Journal of the Royal Statistical Society, Ser. B, 72(2):1–33, 2010.

[2] S. Baker, R. Gross, and I. Matthews. Lucas-Kanade 20 years on: A unifying framework: Part 3. Technical Report CMU-RI-TR-03-35, Robotics Institute, Pittsburgh, PA, November, 2003.

[3] S. Baker and I. Matthews. Equivalence and efficiency of image alignment algorithms. Proceedings of the International Conference on Computer Vision and Pattern Recognition, Kauai Island, HI, USA, pages 1090–1097, 2001.

[4] S. Baker and I. Matthews. Lucas-Kanade 20 years on: A unifying framework. International Journal of Computer Vision, 56:221–255, 2004.

[5] N. Diehl. Methoden zur allgemeinen Bewegungsschätzung in Bildfolgen. PhD dissertation, Technische Universitt Hamburg-Harburg, Ersch. als Fortschrittbericht (Reihe 10, Nr.92) der VDI-Zeitschriften, VDI-Verlag, March 1988.

[6] M. Dimitrijevic, S. Ilic, and P. Fua. Accurate face models from uncalibrated and ill-lit video sequences. Computer Vision and Pattern Recognition, 2:1034–1041, 2004.

[7] G.D. Hager and P.N. Belhumeur. Efficient region tracking with parametric models of geometry and illumination. IEEE, PAMI, 20, 1998.

[8] Z. Khan, T. Balch, and F. Dellaert. An MCMC-based particle filter for tracking multiple interacting targets. In in Proc. ECCV, pages 279–290, 2003.

[9] B.D. Lucas and T. Kanade. An iterative image restoration technique with an appliation to stereo vision. Proceedings of the 7th International Joint Conference on Artificial Intelligence (IJCAI '81), pages 674–679, 1981.

Appendix A

Computer Vision Tracking with R

"OpenCV" (Open Source Computer Vision) is an open source library of computer vision and tracking programming functions that is easily acquired and ideal for use in the tracking problems addressed in this thesis. However, as we wished to "reinvent the wheel" in order to fully understand the minutiae involved in coding tracking algorithms, we started from scratch in R. We present some basic algorithms here in case they may others in the future.

A.1 The Lucas-Kanade Algorithm

A.1.1 Einstein with a Simple Translation Warp

```
### LK algorithm for Einstein using jpegs
### Pure translation warp only


## install jpeg package
#install.packages("jpeg")


# load jpeg library
library("jpeg")


# read in images
img<-readJPEG("Figures/einstein.jpg")
template<-readJPEG("Figures/einstein_face.jpg") #warning, t and T are reserved
```

```
# even though images are greyscale, the greys are specified in RGB
# triples with equal R, G, and B. Re-specify as intensities in [0,1]
img<-img[,,1]
template<-template[,,1]


# Currently img is stored as a matrix so that img[1,1] is in
# the upper left corner and img[1,2] is one pixel to the right.
# However, to plot with "image", must turn things around. This
# also happens to be Cartesian space.
img<-t(img[nrow(img):1,])
template<-t(template[nrow(template):1,])

# height and width of image and template
width<-dim(img)[1]
height<-dim(img)[2]
w<-dim(template)[1]
h<-dim(template)[2]

# display image
par(mar=c(1,1,1,1))
image(1:width,1:height,img, axes = FALSE, col = grey(seq(0, 1, length = 256)),
asp=1.05,main="",xlab="",ylab="")


# Compute the image gradient; need dummy border pixels.
hold<-matrix(0,(width+2),(height+2))
hold[2:(width+1),2:(height+1)]<-img
hold[,1]<-hold[,2]
```

```
hold[,(height+2)]<-hold[,(height+1)]

hold[1,]<-hold[2,]

hold[(width+2),]<-hold[(width+1),]


Ix<-matrix(0,width,height)

for(i in 2:(width+1)){

Ix[(i-1),]<-(hold[(i+1),2:(height+1)]-hold[(i-1),2:(height+1)])/2

}



Iy<-matrix(0,width,height)

for(i in 2:(height+1)){

Iy[,(i-1)]<-(hold[2:(width+1),(i+1)]-hold[2:(width+1),(i-1)])/2

} #Cartesian




# initial position for template

p<-c(180,400)       #initial guess for p

ll<-p+c(1,1)        #lower left

ur<-ll+c(w-1,h-1)   #upper right


# plot template on top (don't greyscale so can see)

image(ll[1]:ur[1],ll[2]:ur[2],template, add=T,asp=1,col = heat.colors(20,

alpha = 0.4))



readline(prompt = "Pause. Press <Enter> to continue...")


epsilon = 10^(-10)
```

```
deltapnorm<-1 #arbitrary but larger than epsilon

while(deltapnorm>epsilon)
{
# Find values of pixels for image in the current template rectangle
# position NOTE 1: Will have an error if upper right pixel + (1,1) goes
# off image-- not coding this for the sake of clarity. NOTE 2: Currently
# img is in "matrix space". Switch to Cartesian space to find pixels in
# terms of xy-coordinates. NOTE 3: For a matrix A in R, A[3.9,5.7] will
# truncate indices and return A[3,5].
fracx <- ll[1]-floor(ll[1])
fracy<-ll[2]-floor(ll[2])
imgatwarp<-(1-fracx)*(1-fracy)*img[ll[1]:ur[1],ll[2]:ur[2]]+
fracx*(1-fracy)*img[(ll[1]+1):(ur[1]+1),ll[2]:ur[2]]+
(1-fracx)*fracy*img[ll[1]:ur[1],(ll[2]+1):(ur[2]+1)]+
fracx*fracy*img[(ll[1]+1):(ur[1]+1),(ll[2]+1):(ur[2]+1)]

# find the error image
errorimage<-template-imgatwarp

# Find values of pixels for gradient in the current template rectangle
# position.
gradatwarpx<-(1-fracx)*(1-fracy)*Ix[ll[1]:ur[1],ll[2]:ur[2]]+
fracx*(1-fracy)*Ix[(ll[1]+1):(ur[1]+1),ll[2]:ur[2]]+
(1-fracx)*fracy*Ix[ll[1]:ur[1],(ll[2]+1):(ur[2]+1)]+
fracx*fracy*Ix[(ll[1]+1):(ur[1]+1),(ll[2]+1):(ur[2]+1)]

gradatwarpy<-(1-fracx)*(1-fracy)*Iy[ll[1]:ur[1],ll[2]:ur[2]]+
fracx*(1-fracy)*Iy[(ll[1]+1):(ur[1]+1),ll[2]:ur[2]]+
(1-fracx)*fracy*Iy[ll[1]:ur[1],(ll[2]+1):(ur[2]+1)]+
```

```
fracx*fracy*Iy[(ll[1]+1):(ur[1]+1),(ll[2]+1):(ur[2]+1)]

# Jacobian is identity matrix so omit.

# Find "bigsum" from Step 6 of LK algorithm.
bigsum<-matrix(0,2,1)
bigsum[1,1]<-sum(gradatwarpx*errorimage)
bigsum[2,1]<-sum(gradatwarpy*errorimage)

# find Hessian
H<-matrix(0,2,2)
H[1,1]<-sum(gradatwarpx^2)
H[1,2]<-sum(gradatwarpx*gradatwarpy)
H[2,1]<-H[1,2]
H[2,2]<-sum(gradatwarpy^2)

# find change in parameters
deltap<-solve(H)%*%bigsum

# new p
p<-p+t(deltap)

# norm of deltap
deltapnorm<-sqrt(deltap[1]^2+deltap[2]^2)
print(noquote(paste("deltapnorm =",round(deltapnorm,5))))

# replot underlying image
image(ll[1]:ur[1],ll[2]:ur[2],imgatwarp,add=T,col = grey(seq(0, 1, length = 256)))

# plot template in new position
```

```
ll<-p+c(1,1)       #lower left
ur<-ll+c(w-1,h-1)   #upper right

image(ll[1]:ur[1],ll[2]:ur[2],template, add=T,asp=1,col = heat.colors(20,alpha = 0.4))

} #end while
```

A.2 The Compositional Alignment Algorithm

A.2.1 Einstein with a Simple Translation Warp

```
### CA algorithm for Einstein using jpegs
### Pure translation warp only

## install jpeg package
#install.packages("jpeg")

# load jpeg library
library("jpeg")

# read in images
img<-readJPEG("Figures/einstein.jpg")
template<-readJPEG("Figures/einstein_face.jpg") #warning, t and T are reserved

# even though images are greyscale, the greys are specified in RGB
# triples with equal R, G, and B. Re-specify as intensities in [0,1]
img<-img[,,1]
template<-template[,,1]


# Currently img is stored as a matrix so that img[1,1] is in
```

```
# the upper left corner and img[1,2] is one pixel to the right.
# However, to plot with "image", must turn things around. This
# also happens to be Cartesian space.
img<-t(img[nrow(img):1,])
template<-t(template[nrow(template):1,])

# height and width of image and template
width<-dim(img)[1]
height<-dim(img)[2]
w<-dim(template)[1]
h<-dim(template)[2]

# display image
par(mar=c(1,1,1,1))
image(1:width,1:height,img, axes = FALSE, col = grey(seq(0, 1, length = 256)),asp=1.05,
main="",xlab="",ylab="")


# initial position for template
p<-c(180,400)
ll<-p+c(1,1)       #lower left
ur<-ll+c(w-1,h-1)   #upper right

# Plot template on top. (Don't greyscale so can see.)
image(ll[1]:ur[1],ll[2]:ur[2],template, add=T,asp=1,col = heat.colors(20,
   alpha = 0.4))

# Jacobian is identity so omit.

readline(prompt = "Pause. Press <Enter> to continue...")
```

```
epsilon = 10^(-10)
deltapnorm<-1 #arbitrary but larger than epsilon


while(deltapnorm>epsilon)
{
# Find values of pixels for image in the current template rectangle
# position NOTE 1: Will have an error if upper right pixel + (1,1) goes
# off image-- not coding this for the sake of clarity. NOTE 2: Currently
# img is in "matrix space". Switch to Cartesian space to find pixels in
# terms of xy-coordinates. NOTE 3: For a matrix A in R, A[3.9,5.7] will
# truncate indices and return A[3,5].
fracx <- ll[1]-floor(ll[1])
fracy<-ll[2]-floor(ll[2])
imgatwarp<-(1-fracx)*(1-fracy)*img[ll[1]:ur[1],ll[2]:ur[2]]+
fracx*(1-fracy)*img[(ll[1]+1):(ur[1]+1),ll[2]:ur[2]]+
(1-fracx)*fracy*img[ll[1]:ur[1],(ll[2]+1):(ur[2]+1)]+
fracx*fracy*img[(ll[1]+1):(ur[1]+1),(ll[2]+1):(ur[2]+1)]


# find the error image
errorimage<-template-imgatwarp


# Compute the gradient of I(W), need border pixels. Hold is
# imgatwarp with border pixels. Should really take into account
# possibility that warp function is mapping to edge in which case
# need dummy border pixels. For example,  will have an error if
# upper right pixel + (1,1) goes off image-- not coding this for
```

```
# the sake of clarity.
hold<-(1-fracx)*(1-fracy)*img[(ll[1]-1):(ur[1]+1),(ll[2]-1):(ur[2]+1)]+
fracx*(1-fracy)*img[(ll[1]+1-1):(ur[1]+1+1),(ll[2]-1):(ur[2]+1)]+
(1-fracx)*fracy*img[(ll[1]-1):(ur[1]+1),(ll[2]+1-1):(ur[2]+1+1)]+
fracx*fracy*img[(ll[1]+1-1):(ur[1]+1+1),(ll[2]+1-1):(ur[2]+1+1)]


IWx<-matrix(0,w,h)
for(i in 2:(w+1)){
IWx[(i-1),]<-(hold[(i+1),2:(h+1)]-hold[(i-1),2:(h+1)])/2
}

IWy<-matrix(0,w,h)
for(i in 2:(h+1)){
IWy[,(i-1)]<-(hold[2:(w+1),(i+1)]-hold[2:(w+1),(i-1)])/2
}

# Find "bigsum" from Step 6 of CA algorithm.
bigsum<-matrix(0,2,1)
bigsum[1,1]<-sum(IWx*errorimage)
bigsum[2,1]<-sum(IWy*errorimage)

# Find Hessian.
H<-matrix(0,2,2)
H[1,1]<-sum(IWx^2)
H[1,2]<-sum(IWx*IWy)
H[2,1]<-H[1,2]
H[2,2]<-sum(IWy^2)

# Find change in parameters.
```

```
deltap<-solve(H)%*%bigsum


# new p
p<-p+t(deltap)


# norm of deltap
deltapnorm<-sqrt(deltap[1]^2+deltap[2]^2)
print(noquote(paste("deltapnorm =",round(deltapnorm,5))))


# Replot underlying image.
image(ll[1]:ur[1],ll[2]:ur[2],imgatwarp,add=T,col = grey(seq(0, 1, length = 256)))


# Plot template in new position.
ll<-p+c(1,1)       #lower left
ur<-ll+c(w-1,h-1)   #upper right


image(ll[1]:ur[1],ll[2]:ur[2],template, add=T,asp=1,col = heat.colors(20,alpha = 0.4))


} #end while
```

A.3 The Inverse Compositional Alignment Algorithm

A.3.1 Einstein with a Simple Translation Warp

```
### ICA algorithm for Einstein using jpegs
### Pure translation warp only


## install jpeg package
#install.packages("jpeg")


# load jpeg library
```

```
library("jpeg")

# read in images
img<-readJPEG("Figures/einstein.jpg")
template<-readJPEG("Figures/einstein_face.jpg") #warning, t and T are reserved

# Even though images are greyscale, the greys are specified in RGB
# triples with equal R, G, and B. Re-specify as intensities in [0,1].
img<-img[,,1]
template<-template[,,1]


# Currently img is stored as a matrix so that img[1,1] is in
# the upper left corner and img[1,2] is one pixel to the right.
# However, to plot with "image", must turn things around. This
# also happens to be Cartesian space.
img<-t(img[nrow(img):1,])
template<-t(template[nrow(template):1,])

# height and width of image and template
width<-dim(img)[1]
height<-dim(img)[2]
w<-dim(template)[1]
h<-dim(template)[2]

# display image
par(mar=c(1,1,1,1))
image(1:width,1:height,img, axes = FALSE, col = grey(seq(0, 1, length = 256)),asp=1.05,
main="",xlab="",ylab="")
```

```
# initial position for template
p<-c(180,400)
ll<-p+c(1,1)       #lower left
ur<-ll+c(w-1,h-1)   #upper right

# Plot template on top. (Don't greyscale so can see.)
image(ll[1]:ur[1],ll[2]:ur[2],template, add=T,asp=1,col = heat.colors(20,
   alpha = 0.4))

# Jacobian is identity so omit.

readline(prompt = "Pause. Press <Enter> to continue...")


# Compute the template gradient, need dummy pixels.
hold<-matrix(0,(w+2),(h+2))
hold[2:(w+1),2:(h+1)]<-template
hold[,1]<-hold[,2]
hold[,(h+2)]<-hold[,(h+1)]
hold[1,]<-hold[2,]
hold[(w+2),]<-hold[(w+1),]

Tx<-matrix(0,w,h)
for(i in 2:(w+1)){
Tx[(i-1),]<-(hold[(i+1),2:(h+1)]-hold[(i-1),2:(h+1)])/2
}

Ty<-matrix(0,w,h)
for(i in 2:(h+1)){
```

```
Ty[,(i-1)]<-(hold[2:(w+1),(i+1)]-hold[2:(w+1),(i-1)])/2
}



# find Hessian
H<-matrix(0,2,2)
H[1,1]<-sum(Tx^2)
H[1,2]<-sum(Tx*Ty)
H[2,1]<-H[1,2]
H[2,2]<-sum(Ty^2)




epsilon = 10^(-10)
deltapnorm<-1 #arbitrary but larger than epsilon



while(deltapnorm>epsilon)
{
# Find values of pixels for image in the current template rectangle
# position NOTE 1: Will have an error if upper right pixel + (1,1) goes
# off image-- not coding this for the sake of clarity. NOTE 2: Currently
# img is in "matrix space". Switch to Cartesian space to find pixels in
# terms of xy-coordinates. NOTE 3: For a matrix A in R, A[3.9,5.7] will
# truncate indices and return A[3,5].
fracx <- ll[1]-floor(ll[1])
fracy<-ll[2]-floor(ll[2])
imgatwarp<-(1-fracx)*(1-fracy)*img[ll[1]:ur[1],ll[2]:ur[2]]+
fracx*(1-fracy)*img[(ll[1]+1):(ur[1]+1),ll[2]:ur[2]]+
(1-fracx)*fracy*img[ll[1]:ur[1],(ll[2]+1):(ur[2]+1)]+
```

```
fracx*fracy*img[(ll[1]+1):(ur[1]+1),(ll[2]+1):(ur[2]+1)]

# find the error image
errorimage<-template-imgatwarp

# Find "bigsum" from Step 4 of ICA algorithm.
bigsum<-matrix(0,2,1)
bigsum[1,1]<-sum(Tx*errorimage)
bigsum[2,1]<-sum(Ty*errorimage)

# Find change in parameters.
deltap<-solve(H)%*%bigsum

# new p
p<-p+t(deltap)

# norm of deltap
deltapnorm<-sqrt(deltap[1]^2+deltap[2]^2)
print(noquote(paste("deltapnorm =",round(deltapnorm,5))))

# Replot underlying image.
image(ll[1]:ur[1],ll[2]:ur[2],imgatwarp,add=T,col = grey(seq(0, 1, length = 256)))

# Plot template in new position.
ll<-p+c(1,1)       #lower left
ur<-ll+c(w-1,h-1)   #upper right

image(ll[1]:ur[1],ll[2]:ur[2],template, add=T,asp=1,col = heat.colors(20,alpha = 0.4))

} #end while
```

Appendix B

Principle Components Analysis of Image Sequence in R

```
# load the png library
library(png)

# read in faces
face1<-readPNG("face1.png")
face2<-readPNG("face2.png")
face3<-readPNG("face3.png")
face4<-readPNG("face4.png")
face5<-readPNG("face5.png")
face6<-readPNG("face6.png")

# remove redundantly store greyscale values
face1<-face1[,,1]
face2<-face2[,,1]
face3<-face3[,,1]
face4<-face4[,,1]
face5<-face5[,,1]
face6<-face6[,,1]

# image dimenstions
w<-dim(face1)[2]
h<-dim(face1)[1]
```

```
par(mfrow=c(3,1))
plot(c(0,6*w+50),c(0,h),type="n",axes=F,xlab="",ylab="",main="Original",asp=1)
rasterImage(face1,0,0,w,h)
rasterImage(face2,w+10,0,2*w+10,h)
rasterImage(face3,2*(w+10),0,(2*w+10)+(w+10),h)
rasterImage(face4,3*(w+10),0,(2*w+10)+2*(w+10),h)
rasterImage(face5,4*(w+10),0,(2*w+10)+3*(w+10),h)
rasterImage(face6,5*(w+10),0,(2*w+10)+4*(w+10),h)

X<-rbind(c(face1),c(face2),c(face3),c(face4),c(face5),c(face6))

for(i in 1:6){
X[i,]<-X[i,]-mean(X[i,1])
}

A<-(1/(length(c(face1))-1))*X%*%t(X)

hold<-eigen(A,TRUE)

D<-hold$values
E<-hold$vectors

numfaces <- readline("How many faces should I use?")
numfaces<-as.numeric(numfaces)
{
if(numfaces %in% 1:6){print("Ok",quote=FALSE)}
else{print("Enter an integer value between 1 and 6",quote=FALSE)}
}
```

```
P<-t(E)
Pprime<-P[1:numfaces,] #or PPrime<-P{1:2,], etc...

# Pprime will be matrix only if take more than 1 row.
# Otherwise must force to matrix.
if(!is.matrix(Pprime)){Pprime<-t(as.matrix(Pprime))}

newdata<-Pprime%*%X

plot(c(0,6*w+50),c(0,h),type="n",axes=F,xlab="",ylab="",main=paste("Eigenfaces
     (Using ",numfaces,")",sep=""),asp=1)
for(i in 1:dim(newdata)[1]){
Y<-matrix(newdata[i,],ncol=dim(face1)[2])
Y<-t(Y[nrow(Y):1,])
width<-dim(Y)[1]
height<-dim(Y)[2]
image(((i-1)*(w+10)):(i*w+(i-1)*10),1:h,Y, axes = FALSE, col = grey(seq(0,
     1, length = 256)),asp=1.05,main="Eigenface",xlab="",ylab="",add=T)
}

reconstruct<-t(Pprime)%*%newdata
plot(c(0,6*w+50),c(0,h),type="n",axes=F,xlab="",ylab="",main="Reconstruction from
     Eigenfaces",asp=1)
face1r<-matrix(reconstruct[1,],ncol=dim(face1)[2])
face1r<-t(face1r[nrow(face1r):1,])
image(1:w,1:h,face1r, axes = FALSE, col = grey(seq(0, 1, length = 256)),asp=1.05,
     main="Eigenface",xlab="",ylab="",add=T)
face2r<-matrix(reconstruct[2,],ncol=dim(face1)[2])
face2r<-t(face2r[nrow(face2r):1,])
image((w+10):(2*w+10),1:h,face2r, axes = FALSE, col = grey(seq(0, 1, length = 256)),
```

```
     asp=1.05,main="Eigenface",xlab="",ylab="",add=T)
face3r<-matrix(reconstruct[3,],ncol=dim(face1)[2])
face3r<-t(face3r[nrow(face3r):1,])
image((2*w+2*10):((3*w+2*10)),1:h,face3r, axes = FALSE, col = grey(seq(0, 1,
    length = 256)), asp=1.05,main="Eigenface",xlab="",ylab="",add=T)
face4r<-matrix(reconstruct[4,],ncol=dim(face1)[2])
face4r<-t(face4r[nrow(face4r):1,])
image((3*w+3*10):((4*w+3*10)),1:h,face4r, axes = FALSE, col = grey(seq(0, 1,
    length = 256)), asp=1.05,main="Eigenface",xlab="",ylab="",add=T)
face5r<-matrix(reconstruct[5,],ncol=dim(face1)[2])
face5r<-t(face5r[nrow(face5r):1,])
image((4*w+4*10):((5*w+4*10)),1:h,face5r, axes = FALSE, col = grey(seq(0, 1,
    length = 256)), asp=1.05,main="Eigenface",xlab="",ylab="",add=T)
face6r<-matrix(reconstruct[6,],ncol=dim(face1)[2])
face6r<-t(face6r[nrow(face6r):1,])
image((5*w+5*10):((6*w+5*10)),1:h,face6r, axes = FALSE, col = grey(seq(0, 1,
    length = 256)), asp=1.05,main="Eigenface",xlab="",ylab="",add=T)
```