Computational Perception

or: An Unofficial Companion Guide to the Georgia Institute of Technology's
CS6476: *Computer Vision*, **CS6475**: *Computational Photography*,
and **CS7638**: *Robotics: AI Techniques*

George Kudrayvtsev
george.k@gatech.edu

Last Updated: October 11, 2019

This work is a culmination of hundreds of hours of effort to create a lasting reference to read along with some of Georgia Tech's graduate courses in the *Computational Perception* specialization. All of the explanations and algorithms are in my own words and many of the diagrams were hand-crafted. The majority of the content is based on lectures created by Dr. Aaron Bobick, Dr. Irfan Essa (for the more photography-oriented material), and Dr. Sebastian Thrun (for the robotics-oriented material) as well as on numerous external academic sources that are linked where appropriate.

Smile for the camera!

The guide is currently actively being enhanced and supplemented with content from both *Computational Photography* and *Robotics: AI Techniques*; check out the first chapter for reading orders if you're here to make dank photos or invent the Terminator rather than just help computers see like humans do.

Many lattés were consumed throughout the making of this companion guide. ☕ If you found it useful and are in a generous mood, feel free to buy me another! Shoot me a donation on Venmo `@george_k_btw` or PayPal `kudrayvtsev@sbcglobal.net` with whatever this guide was worth to you.

Happy studying! 😇

List of Algorithms

Preface

> *I read that Teddy Roosevelt once said, "Do what you can with what you have where you are." Of course, I doubt he was in the tub when he said that.*
>
> — Bill Watterson, *Calvin and Hobbes*

0.1 How to Read This Guide

This companion guide was originally written exclusively with computer vision in mind. However, now that I am taking *Computational Photography*, it has expanded to include concepts from there. There is a lot of overlap between the two.

If you are here for computer vision, it's pretty straightforward: you should just read this guide from front to back *except* for chapter 13, which covers robot planning algorithms that aren't relevant for CV.

If you are here for computational photography, I recommend the following reading order:

- Start with chapters 1–3 as usual, because they provide the fundamentals;
- skip ahead to chapter 7 to learn a little about cameras, but stop once you hit Stereo Geometry since we'll cover that later;
- return to chapter 5 for the Fourier transform; then, chapter 6 covers image pyramids and blending methods;
- read the first two sections of chapter 8 for the mathematical primer on panorama creation, then skip to chapter 9 for the feature matching portion[1];
- the notes are currently lacking content on HDR and PhotoSynth, though the latter is briefly mentioned in Figure 8.11;
- you can return to Stereo Geometry for a discussion of stereo, and camera calibration immediately follows;

[1] This bit needs to be expanded to specifically discuss panorama creation more.

Unfortunately, that about covers it, since I've yet to take notes on the video, lightfield, and "coded photography" (whatever that means?) portions. However, keep an eye on the table of contents because I'll eventually get there!

If you are here for robotics, I recommend the following reading order:

- Start with chapter 12, which covers both Kalman filters and particle filters, then
- follow it up with chapter 13, which covers robot motion planning algorithms,

... and that's it for now—PID controls and SLAM are not yet covered.

0.2 Notation

Before we begin to dive into all things computer vision, here are a few things I do in this notebook to elaborate on concepts:

- An item that is **highlighted like this** is a "term;" this is some vocabulary or identifying word/phrase that will be used and repeated regularly in subsequent sections. I try to cross-reference these any time they come up again to link back to its first defined usage; most mentions are available in the Index.
- An item that is **highlighted like this** is a "mathematical property;" such properties are often used in subsequent sections and their understanding is assumed there.
- The presence of a `TODO` means that I still need to expand that section or possibly to mark something that should link to a future (unwritten) section or chapter.
- An item in a maroon box, like...

> Boxes: **A Rigorous Approach**
>
> ... this often represents fun and interesting asides or examples that pertain to the material being discussed. They are largely optional, but should be interesting to read and have value, even if it's not immediately rewarding.

- An item in a blue box, like...

> Quick Maffs: **Proving That the Box Exists**
>
> ... this is a mathematical aside; I only write these if I need to dive deeper into a concept that's mentioned in lecture. This could be proofs, examples, or just a more thorough explanation of something that might've been "assumed knowledge" in the text.

- An item in a green box, like...

Example 0.1: Examples

... this is an example that explores a theoretical topic with specifics. It's sometimes from lecture, but can also be just something I added in to understand it better myself.

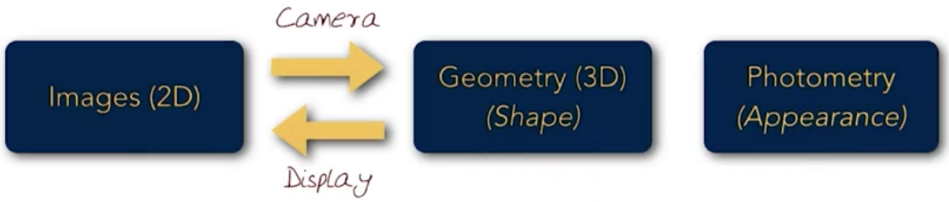
Introduction

> *Every picture tells a story. But sometimes it's hard to know what story is actually being told.*
>
> — Anastasia Hollings, *Beautiful World*

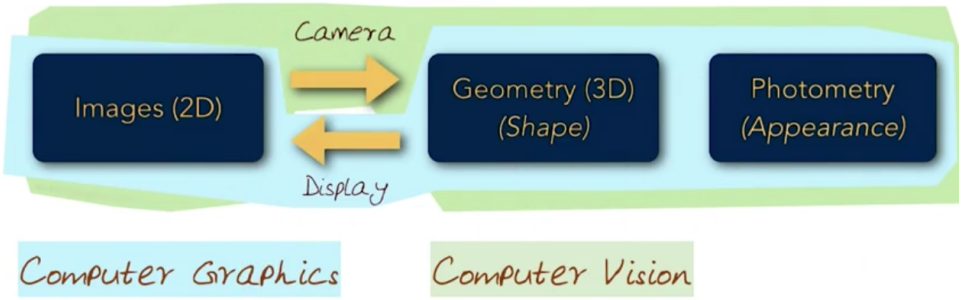
This set of notes is an effort to unify the curriculum across both computer vision and computational photography, cross-pollinating unique ideas from each discipline and consolidating concepts that are shared between the two. Before we begin, it's important to talk about their differences.

1.1 Computational Photography vs. Vision vs. Graphics

So what's the deal with all of these different disciplines? Well, computational photography is unique in the sense that it leverages discoveries and techniques from *all* of the graphics-related disciplines. In general, we can break down the process into three components, facilitated by two types of hardware:

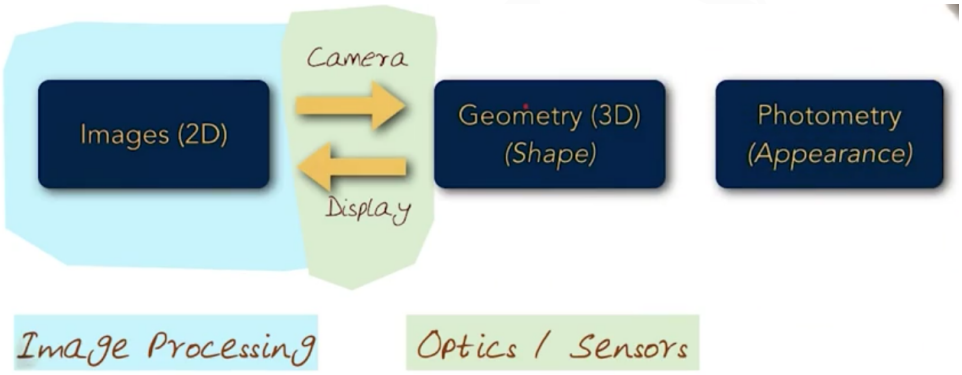


Computer vision is concerned with all of these except *display*. It is the discipline that focuses on looking at 2D images from the real world and inferring information from it, which includes things like geometry as well as a more "high-level" understanding of things like recognizing objects.



On the other hand, computer graphics is the reverse process. It takes information about the scene in 3D (given things like how objects look, or how they behave under various lighting conditions) and creates beautiful 2D images from those descriptions.

Image processing is an underlying discipline for both computer vision and computational photography that is concerned with treating images as functions, doing things like blurring or enhancing them. An understanding of optics and sensors is also critical in all of these disciplines because it provides us with a fundamental understanding of how the 3D world is transformed into 2D (and vice-versa).



1.2 Computer Vision

The goal of computer vision is to create programs that can interpret and analyse images, providing the program with the *meaning* behind the image. This may involve concepts such as *object recognition* as well as *action recognition* for images in motion (colloquially, "videos").

Computer vision is a hard problem because it involves much more complex analysis relative to image processing. For example, observe the following set of images:

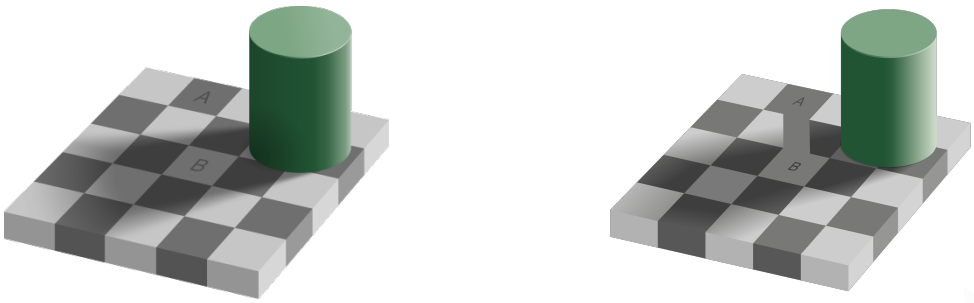


Figure 1.1: The checker shadow illusion.

The two checkered squares A and B have the same color **intensity**, but our brain interprets them differently without the connecting band due to the shadow. Shadows are actually quite important to human vision. Our brains rely on shadows to create depth information and track motion based on shadow movements to resolve ambiguities.

A computer can easily figure out that the intensities of the squares are equal, but it's much harder for it to "see" the illusion like we do. Computer vision involves viewing the image as a whole and gaining a semantic understanding of its content, rather than just processing things pixel-by-pixel.

1.3 Computational Photography

Photography is the science or art of recording light and turning it into an image using sensors (if we're talking about modern digital cameras) or chemistry (if we're talking about the light-sensitive material in film).

$$\underbrace{\text{photo}}_{\text{light}}\underbrace{\text{graphy}}_{\text{drawing}}$$

Computational photography is what arises when you bring the power of modern computers to digital cameras. The complex algorithms that run on your phone when you take a photo (faking depth-of-field, snapping HDR scenes, crafting panoramas in real time, etc.) are all part of the computational photography sphere. In essence, we combine modern principles in computing, digital sensors, optics, actuators, lights, and more to "escape" the limitations of traditional film photography.

The pipeline that takes light from a scene in the real world and outputs something the user can show their friends is quite complex:

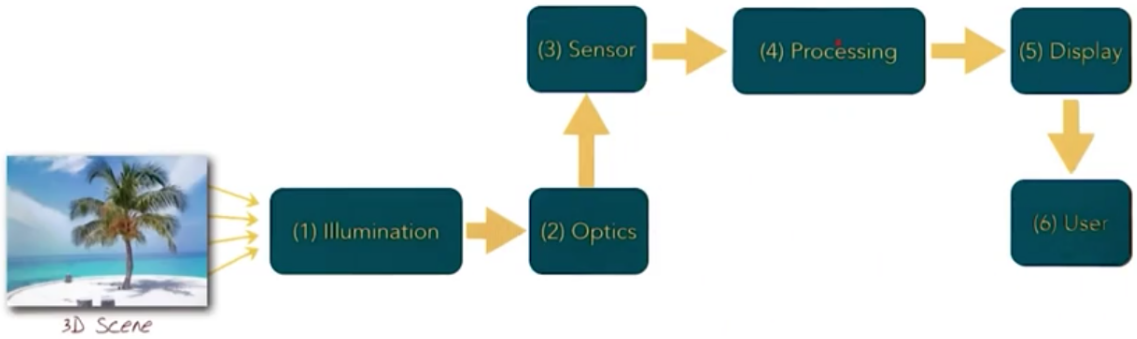
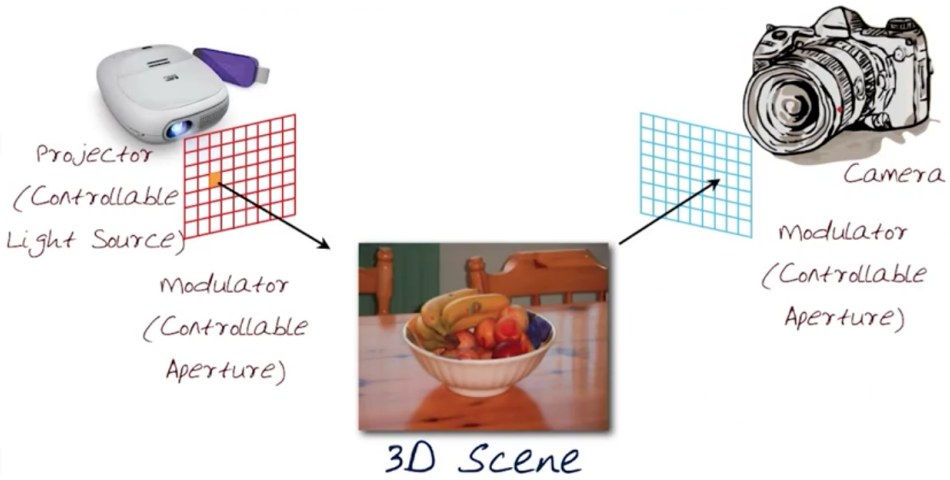


Figure 1.2: The photography pipeline.

The beauty is that computation can be embedded at every level of this pipeline to support photography. We'll be referring to ways to computationally impact each of these elements throughout this guide.

1.3.1 Dual Photography

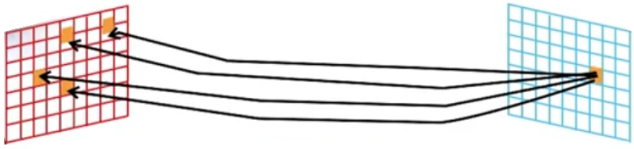
Let's walk through a concrete example of how computing can enable creation of novel images. We'll be discussing **dual photography**, in which a computer controls both the light sources illuminating a scene and the cameras that capture it.[1] We add both a "novel illumination" and "novel camera" to our simple scene capturing setup:



By treating a projector as a controllable light source, we can have it pass through a *modulator* acting as a controllable aperture. We can control which cells in the modulator are open at any given point in time, giving us tight control of the custom lighting we add to the scene. Similarly, we can control the way we capture light that the same way we controlled illumination: we add a similar modulator to the camera in our setup, allowing us to control exactly what the camera captures from the scene.

[1] You can read the original paper from 2007 here.

If we tightly couple the modulation of the lighting and camera, we can see the effect that various changes will have on our resulting photo:



Helmholtz reciprocity essentially states that for a given ray of light, where it comes from and who it's observed by is interchangeable: light behaves the same way "forward" from a light source and "backwards" from an observer.[2]

For our case, this means that by opening up certain areas of the projector's modulator, we can observe and record their effect on the image. We create an association between the "open cell" and the "lit image," and given enough data points we can actually recreate the scene *from the projector's perspective* (see Figure 1.3).

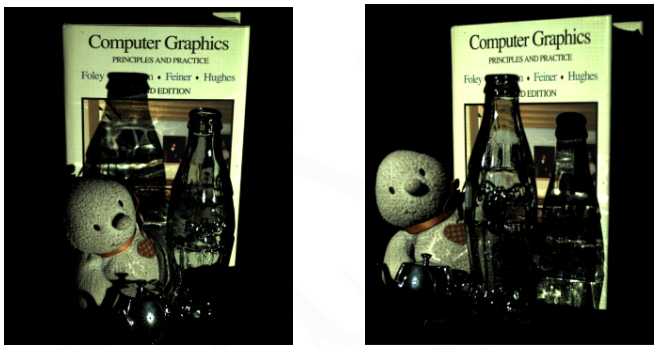


Figure 1.3: On the left is the "primal" image captured by the camera under full illumination of the scene by the projector. On the right is a *recreation* of the scene from a number of controlled illuminations from the perspective of the projector.

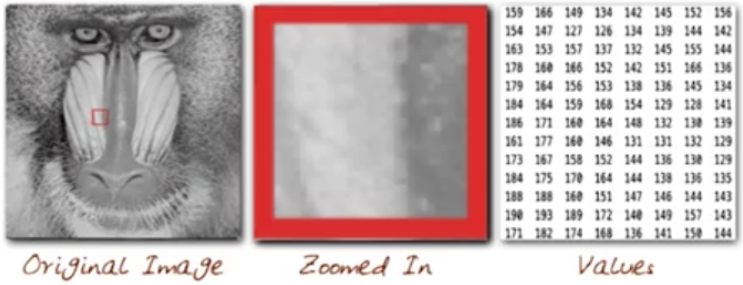
This is an excellent example of how we can use computing to create a novel image from a scenario that would otherwise be impossible to do with traditional film.

[2] Feel free to check out the Wikipedia page for a more thorough explanation, and a later chapter on BRDFs touches on this idea as well.

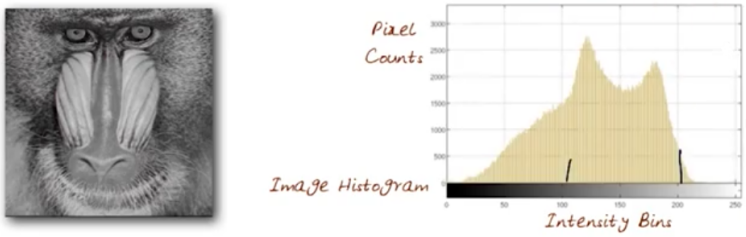
Basic Image Manipulation

> *Beauty is in the eye of the beholder.*
>
> — Margaret Wolfe Hungerford, *Molly Bawn*

DIGITAL images are just arrays of numbers that represent intensities of color arranged in a rectangle. Computationally, we typically just treat them like a matrix.



An important operation we can do with an image (or some *part* of an image) is to analyze its statistical properties using a histogram, which essentially shows the distribution of intensities in an image:



This segues perfectly into the notion of treating images simply as (discrete) functions: mappings of (x, y) coordinates to intensity values.

2.1 Images as Functions

We will begin with black-and-white images in order to keep our representation simple: the intensity represents a range from some minimum (black, often 0) and some maximum (white,

often 1 or 255).

Image processing is then a task of applying operations on this function to transform it into another function: images in, images out. Since it's just a mathematical structure, we can perform traditional mathematical operations on it. For example, when we smooth out peaks and valleys (which we would "visually" perceive as sharp contrasts in image intensity) in the function, the result is a blurred version of that image! We'll explore this in Blurring Images.

More formally, an "image function" is a mapping from R^2 (that is, two real numbers representing a position (x, y) in the image) to R (that is, some intensity or value). In the real world, images have a finite size or dimension, so thus we have: $I : R^2 \mapsto R$. More specifically, $I : x \times y \mapsto R$ where $x \in [a, b], y \in [c, d]$, and $R \in [min, max]$, where min would be some "blackest black" and max would be some "whitest white," and (a, b, c, d) are ranges for the different dimensions of the images, though when actually performing mathematical operations, such interpretations of values become irrelevant.

We can easily expand this to color images, with a vector-valued function mapping each color component:

$$I(x, y) = \begin{bmatrix} r(x, y) \\ g(x, y) \\ b(x, y) \end{bmatrix}$$

2.1.1 Operations on ~~Images~~ Functions

Because images are just functions, we can perform any mathematical operations on them that we can on, well, functions.

Addition Adding two images will result in a blend between the two. As we discussed, though, intensities have a range $[min, max]$; thus, adding is often performed as an average of the two images instead, to not lose intensities when their sum exceeds max:

$$I_{\text{added}} = \frac{I_a}{2} + \frac{I_b}{2}$$

Subtraction In contrast, subtracting two images will give the *difference* between the two. A smaller intensity indicates more similarity between the two source images at that pixel. Note that order of operations matters, though the results are inverses of each other:

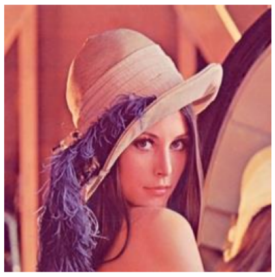
$$I_a - I_b = -(I_b - I_a)$$

Often, we simply care about the *absolute* difference between the images. Because we are often operating in a discrete space that will truncate negative values (for example, when operating on images represented as `uint8`), we can use a special formulation to get this difference:

$$I_{\text{diff}} = (I_a - I_b) + (I_b + I_a)$$

Noise

A common function that is added to a single image is a **noise function**. One of these is called the **Gaussian noise function**: it adds a variation in intensity drawn from a Gaussian normal distribution. We basically add a random intensity value to every pixel on an image. Here we see Gaussian noise added to a classic example image[1] used in computer vision.



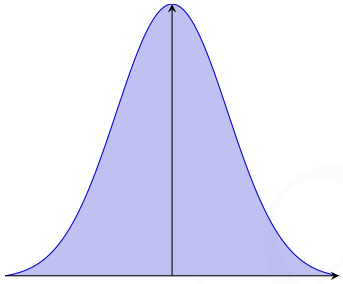


Figure 2.1: A Gaussian (normal) distribution centered at a mean, μ.

Tweaking Sigma On a normal distribution, the mean is 0. If we interpret 0 as an intensity, it would have to be between black (the low end) and white (the high end); thus, the average pixel intensity added to the image should be *gray*. When we tweak σ – the standard deviation – this will affect the amount of noise: a higher σ means a noisier image. Of course, when working with image manipulation libraries, the choice of σ varies depending on the range of intensities in the image: $\sigma = 10$ is much bigger than it sounds if your pixels are $\in [-1.0, +1.0]$.

2.2 Image Filtering

Now that we have discussed *adding* noise to an image, how would we approach *removing* noise from an image? If noise is just a function added to an image, couldn't we just remove that noise by subtracting the noise function again? Unfortunately, this is not possible without knowing the original noise function! This information is *not* stored independently within our image, so we need to somehow extract or derive that information from the merged images instead. Furthermore, because of the common range limitation on pixels, some information may be lost as a result of an "overflowing" (or underflowing) noise addition that results in pixel values outside of that range!

To clarify, consider a single pixel in the range $[0, 255]$ and the intensity 200. If the noise function added 60 intensity, how would you derive the original value of 200 from the new value of 255 even if you *did* know that 60 was added? The original value could be anywhere in the range $[195, 255]$!

[1] of a model named Lena, actually, who is posing for the centerfold of an issue of PlayBoy...

2.2.1 Computing Averages

A common "intuitive" suggestion to remove noise is to replace the value of each pixel with the *average* value of the pixels around it. This is known as a **moving average**. This approach hinges on some key assumptions:

- The "true" value of a pixel is probably similar to the "true" values of the nearby pixels.
- The noise in each pixel is added independently. Thus, the average of the noise around a pixel will be 0.

Consider the first assumption further: shouldn't closer pixels be *more* similar than further pixels, if we consider all pixels that are within some radius? This leads us to trying a **weighted moving average**; such an assumption would result in a smoother representation.

Averages in 2D

Extending a moving average to 2 dimensions is relatively straightforward. You take the average of a range of values in both directions. For example, in a 100×100 image, you may want to take an average over a moving 5×5 square. So, disregarding edge values, the value of the pixel at $(2, 2)$ would be:

$$\begin{aligned} P_{(2,2)} = &P_{(0,0)} + P_{(0,1)} + \ldots + P_{(0,5)} + \\ &P_{(1,0)} + \ldots + P_{(1,5)} + \\ &\ldots \\ &P_{(5,0)} + \ldots + P_{(5,5)} \end{aligned}$$

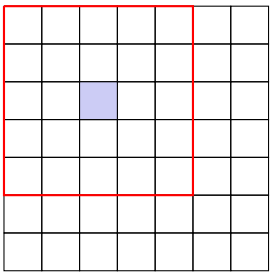


Figure 2.2: A 5×5 averaging window applied to the pixel at $(2, 2)$.

In other words, with our square (typically referred to as a **kernel** or **window**) extending $k = 2$ pixels in both directions, we can derive the formula for **correlation filtering with uniform weights**:

$$G[i, j] = \frac{1}{(2k+1)^2} \cdot \sum_{u=-k}^{k} \sum_{v=-k}^{k} F[i+u, j+v] \tag{2.1}$$

Of course, we decided that non-uniform weights were preferred. This results in a slightly different equation for **correlation filtering with non-uniform weights**, where $H[u, v]$ is the weight function.

$$G[i, j] = \sum_{u=-k}^{k} \sum_{v=-k}^{k} H[u, v] F[i+u, j+v] \tag{2.2}$$

This is also known as **cross-correlation**, denoted $G = H \otimes F$.

Handling Borders Notice that we conveniently placed the red averaging window in Figure 2.2 so that it fell completely within the bounds of the image. What would we do along the top row, for example? We'll discuss this in a little bit, in Boundary Issues.

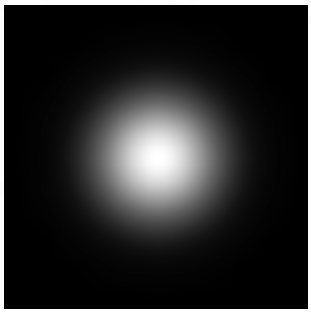
Results We've succeeded in removing *all* noise from an input image. Unfortunately, this throws out the baby with the bathwater if our goal was just to remove *some* extra Gaussian noise (like the speckles in Lena, above), but we've coincidentally discovered a different, interesting effect: *blurring images*.

2.2.2 Blurring Images

> *OK now he was close, tried to domesticate you*
> *But you're an animal, baby, it's in your nature*
> *Just let me liberate you*
>
> — Robin Thicke, *Blurred Lines*

So what went wrong when trying to smooth out the image? Well, a **box filter** like the one in (2.1) (i.e. a filter with uniform weights) is not **smooth** (in the mathematical, not social, sense). We'll define and explore this term later, but for now, suffice to say that a proper blurring (or "smoothing") function should be, well, smooth.

To get a sense of what's wrong, suppose you're viewing a single point of light that is very far away through a camera, and then you made it out of focus. What would such an image look like? Probably something like this:



Now, what kind of filtering function should we apply to an image of such a single bright pixel to produce such a blurry spot? Well, a function that looked like that blurry spot would probably work best: higher values in the middle that fall off (or **attenuate**) to the edges. This is a **Gaussian filter**, which is an application of the 2D **Gaussian function**:

$$h(u,v) = \underbrace{\frac{1}{2\pi\sigma^2}}_{\substack{\text{normalization} \\ \text{coefficient}}} e^{-\frac{u^2+v^2}{\sigma^2}} \tag{2.3}$$

In such a filter, the nearest neighboring pixels have the most influence. This is much like the weighted moving average presented in (2.2), but with weights that better represent

"nearness." Such weights are "circularly symmetric," which mathematically are said to be **isotropic**; thus, this is the isotropic Gaussian filter. Note the normalization coefficient: this value affects the *brightness* of the blur, not the blurring itself.

Gaussian Parameters

The Gaussian filter is a mathematical operation that does not care about pixels. Its only parameter is the **variance** σ, which represents the "amount of smoothing" that the filter performs. Of course, when dealing with images, we need to apply the filter to a particular range of pixels; this is called the kernel.

Now, it's critical to note that modifying the size of the *kernel* is **not** the same thing as modifying the variance. They are related, though. The kernel has to be "big enough" to fairly represent the variance and let it perform a smoother blurring.

2.3 Linearity and Convolution

We are going to continue working with this concept of filtering: applying a filtering function to an image. Naturally, we need to start with some mathematical definitions.

Linearity An operator H is **linear** if the following properties hold (where f_1 and f_2 are some functions, and a is a constant):

- **Additivity**: the operator preserves summation, $H(f_1 + f_2) = H(f_1) + H(f_2)$
- **Multiplicative scaling**, or homogeneity of degree 1: $H(a \cdot f_1) = a \cdot H(f_1)$

With regards to computer vision, linearity allows us to build up an image one piece at a time. We have guarantees that the operator operates identically per-pixel (or per-chunk, or per-frame) as it would on the entire image. In other words, the total is exactly the sum of its parts, and vice-versa.

Shift Invariance The property of **shift invariance** states that an operator behaves the same *everywhere*. In other words, the output depends on the pattern of the image neighborhood, rather than the position of the neighborhood. An operator must give the same result on a pixel regardless of where that pixel (and its neighbors) is located to maintain shift invariance.

2.3.1 Impulses

An **impulse function** in the *discrete* world is a very easy function (or signal) to understand: its value = 1 at a single location. In the *continuous* world, an impulse is an idealized function which is very narrow, very tall, and has a unit area (i.e. an area of 1). In the limit, it has zero width and infinite height; it's integral is 1.

Impulse Responses If we have an unknown system and send an impulse as an input, we get an output (duh?). This output is the **impulse response** that we call $h(t)$. If this "black box" system—which we'll call the unknown operator H—is *linear*, then H can be described by $h(x)$.

Why is that the case? Well, since *any* input to the system is simply a scaled version of the original impulse (for which we know the response), we can describe the output to *any* impulse as following that addition or scaling.

2.3.2 Convolution

Let's revisit the cross-correlation equation from Computing Averages:

$$G[i,j] = \sum_{u=-k}^{k} \sum_{v=-k}^{k} H[u,v]F[i+u, j+v] \tag{2.2}$$

and see what happens when we treat it as a system H and apply impulses. We begin with an impulse signal F (an image), and an arbitrary kernel H:

$$F(x,y) = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix} \qquad H(u,v) = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix}$$

What is the result of filtering the impulse signal with the kernel? In other words, what is $G(x,y) = F(x,y) \otimes H(u,v)$? As we can see in Figure 2.3, the resulting image is a flipped version (in both directions) of the filter H.

We introduce the concept of a **convolution** operator to account for this "flipping." The **cross-*convolution* filter**, or $G = H \circledast F$, is defined as such:

$$G[i,j] = \sum_{u=-k}^{k} \sum_{v=-k}^{k} H[u,v]F[i-u, j-v]$$

This filter flips both dimensions. Convolution filters must be **shift invariant**.

> **Pop Quiz: Convolution of the Gaussian Filter**
>
> What is the difference between applying the Gaussian filter as a convolution vs. a correlation?
>
> Answer: Nothing! Because the Gaussian is an isotropic filter, its symmetry ensures that the order of application is irrelevant. Thus, the distinction only matters for an asymmetric filter.

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix} \longrightarrow \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & i & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

(a) The result (right) of applying the filter H (in red) on F at $(1, 1)$ (in blue).

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix} \longrightarrow \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & i & h & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

(b) The result (right) of subsequently applying the filter H on F at $(2, 1)$ (in blue). The kernel covers a new area in red.

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & i & h & g & 0 \\ 0 & f & e & d & 0 \\ 0 & c & b & a & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

(c) The resulting image (impulse response) after applying the filter H to the entire image.

Figure 2.3: Performing $F \otimes H$.

Properties

Because convolution (and correlation) is both linear- and shift-invariant, it maintains some useful properties:

- **Commutative**: $F \circledast G = G \circledast F$
- **Associative**: $(F \circledast G) \circledast H = F \circledast (G \circledast H)$
- **Identity**: Given the "unit impulse" $e = [\ldots, 0, 1, 0, \ldots]$, then $f \circledast e = f$
- **Differentiation**: $\frac{\partial}{\partial x}(f \circledast g) = \frac{\partial f}{\partial x} \circledast g$. This property will be useful later, in Handling Noise, when we find gradients for edge detection.

Computational Complexity

If an image is $N \times N$ and a filter is $W \times W$, how many multiplications are necessary to compute their convolution $(N \circledast W)$?

Well, a single application of the filter requires W^2 multiplications, and the filter must be applied for every pixel, so N^2 times. Thus, it requires N^2W^2 multiplications, which can grow to be fairly large.

Separability There is room for optimization here for certain filters. If the filter is **separable**, meaning you can get the kernel H by convolving a single column vector by a single

row vector, as in the example:

$$H = \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} \circledast \begin{bmatrix} 1 & 2 & 1 \end{bmatrix}$$

Then we can use the associative property to remove a lot of multiplications. The result, G, can be simplified:

$$\begin{aligned} G &= H \circledast F \\ &= (C \circledast R) \circledast F \\ &= C \circledast (R \circledast F) \end{aligned}$$

So we perform *two* convolutions, but on *smaller* matrices: each one requires WN^2 computations. This is useful if W is large enough such that $2WN^2 \ll W^2N^2$. This optimization used to be *very* valuable and is less important now thanks to Moore's Law, but can still provide a significant benefit: if $W = 31$, for example, it is faster by a factor of ***15***! That's still an order of magnitude.

2.4 Boundary Issues

We have avoided discussing what happens when applying a filter to the edges of an image. We want our resulting image to be the same size as the input, but what do we use as the values to the input when the filter demands pixels beyond its size? There are a few choices:[2]

- **Clipping**: This method simply treats the non-existant pixels as black. Images filtered with this method result in a black border bleeding into the their edges. Such an effect is very noticeable, but may be desireable. It's similar to the artistic "vignette" effect.
- **Wrapping**: This method uses the opposite edge of the image to continue the edge. It was intended for periodic functions and is useful for seamless images, but looks noticeably bad for non-seamless images. The colors from the opposite edge unnaturally bleed into the border.
- **Extending**: This method copies a chunk of the edge to fit the filter kernel. It provides good results that don't have noticeable artifacts like the previous two.
- **Reflecting**: This method copies a chunk of the edge like the previous method, but it mirrors the edge like a reflection. It often results in slightly more natural-looking results, but the differences are largely imperceptible.

[2] Consider, for example the offerings of OpenCV for extending the borders of images.

2	5	6	4	9	6	4
10	7	7	10	6	8	10
1	3	8	5	1	10	7
2	10	8	2	10	5	4
5	4	3	7	4	7	5
6	7	7	3	9	3	9
6	5	7	4	7	1	3

2	2	5	6	4	9	6	4
2	2	5	6	4	9	6	4
10	10	7	7	10	6	8	10
1	1	3	8	5	1	10	7
2	2	10	8	2	10	5	4
5	5	4	3	7	4	7	5
6	6	7	7	3	9	3	9
6	6	5	7	4	7	1	3

Figure 2.4: Padding part of an image to accomodate a 2×2 kernel by extending its border.

Note that the larger your kernel, the more padding necessary, and as a result, the more information is "lost" (or, more accurately, "made up") in the resulting value at that edge.

2.5 More Filter Examples

As we all know from Instagram and Snapchat, there are a lot of techniques out there to change the way an image looks. The following list is obviously not exhaustive, and Figure 2.5 and Figure 2.6 showcase the techniques.

- **Sharpening filter**: This filter accentuates the "differences with the local average," by comparing a more intense version of an image and its box blur.

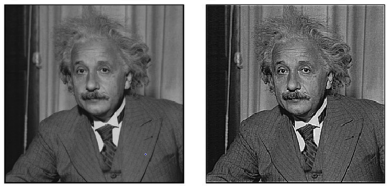


Figure 2.5: A sharpening filter applied to an image of Einstein.

An example sharpening filter could be: $\begin{bmatrix} 0 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 0 \end{bmatrix} - \frac{1}{9}\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$

- **Median filter**: Also called an **edge-preserving filter**, this is actually a ***non*-linear** filter that is useful for other types of noise in an image. For example, a salt-and-pepper noise function would randomly add very-white and very-black dots at random throughout an image. Such dots would significantly throw off an average filter—they are often outliers in the kernel—but can easily be managed by a median filter.

  For example, consider a kernel with intensities as follows:

$$\begin{bmatrix} 10 & 15 & 20 \\ 23 & 90 & 27 \\ 33 & 31 & 30 \end{bmatrix}$$

Figure 2.6: A median filter applied to a salt-and-peppered image of peppers.

where the 90 is clearly an instance of "salt" noise sprinkled into the image. Finding the median:

$$10 \quad 15 \quad 20 \quad 23 \quad \boxed{27} \quad 30 \quad 31 \quad 33 \quad 90$$

results in replacing the center point with intensity = 27, which is much better than a weighted box filter (as in (2.2)) which could have resulted in an intensity of 61.[3]

An interesting benefit of this filter is that any new pixel value *was already present locally*, which means new pixels never have any "weird" values.

Theory in Action: The Unsharp Mask

In Adobe PhotoShop and other editing software, the "unsharp mask" tool would actually sharpen the image. Why?

In the days of actual film, when photos had negatives and were developed in dark rooms, someone came up with a clever technique. If light were shone on a negative that was covered by wax paper, the result was a negative *of the negative* that was blurrier than its original. If you then developed this negative of a negative layered on top of the original negative, you would get a sharper version of the resulting image!

This is a *chemical* replication of the exact same filtering mechanism as the one we described in the sharpening filter, above! We had our original (the negative) and were subtracting (because it was the negative of the negative) a blurrier version of the negative. Hence, again, the result was a sharper developed image.

This blurrier double-negative was called the "unsharp mask," hence the historical name for the editing tool.

2.6 Filters as Templates

Now we will consider filters that aren't simply a representation of intensities in the image, but actually represent some *property* of the pixels, giving us a sort of semantic meaning that

[3] ... if choosing $\frac{1}{9}$ for non-center pixels and $\frac{4}{9}$ for the center.

we can apply to other images.

Filter Normalization Recall how the filters we are working with are linear operators. Well, if we correlated an image (again, see (2.2)), and multiplied that correlation filter by some constant, then the resulting image would be scaled by that same constant. This makes it tricky to compare filters: if we were to compare filtered image 1 against filtered image 2 to see how much the source images "respond" to their filters, we would need to make sure that both filters operate on a similar scale. Otherwise, outputs may differ greatly but not reflect an accurate comparison.

This topic is called **normalized correlation**, and we'll discuss it further later. To summarize things for now, suffice to say that "normalization" means that the standard deviation of our filters will be consistent. For example, we may say that all of our filters will ensure $\sigma = 1$. Not only that, but we also need to normalize the image as we move the kernel across it. Consider two images with the same filter applied, but one image is just a scaled version of the other. We should get the same result, but *only* if we ensure that the standard deviation *within the kernel* is also consistent (or $\sigma = 1$).

Again, we'll discuss implementing this later, but for now assume that all of our future correlations are normalized correlations.

2.6.1 Template Matching

Suppose we make our correlation filter a chunk of our original signal. Then the (normalized) correlation between the signal and the filter would result in an output whose maximum *is where the chunk was from*!

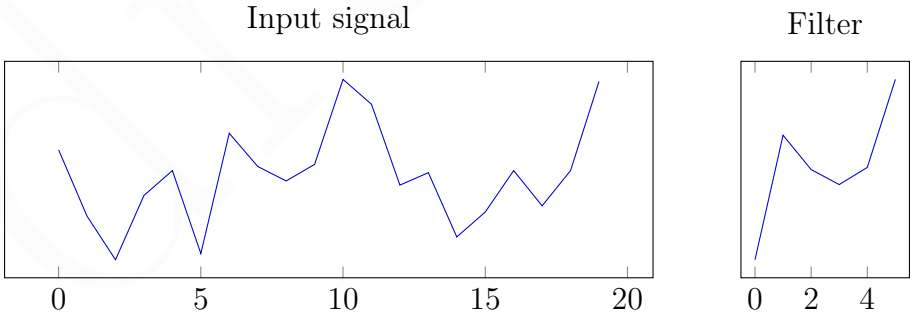


Figure 2.7: A signal and a filter which is part of the signal (specifically, from $x \in [5, 10]$).

Let's discuss this to develop an intuition. A correlation is (see (2.2)) a bunch of multiplications and sums. Consider the signal and a filter as in Figure 2.7. Our signal intensity is centered around 0, so when would the filter and signal result in the largest possible values? We'd naturally expect this to happen when the filter and signal values are the "most similar," or, in other words, equal.

Thus, the maximum of the correlated output represents the location at which the filter matches the original signal! This powerful property, called **template matching** has many useful applications in image processing when we extend the concept to 2 dimensions.

Where's Waldo?

Suppose we have an image from a *Where's Waldo?* book, and we've extracted an image of Waldo himself, like so:

Figure 2.8: An excerpt from a *Where's Waldo?* children's book and a template of Waldo himself.

If we perform a correlation between these two images, using the template of Waldo as our filter, we will get a correlation map whose maximum tells us where Waldo is!

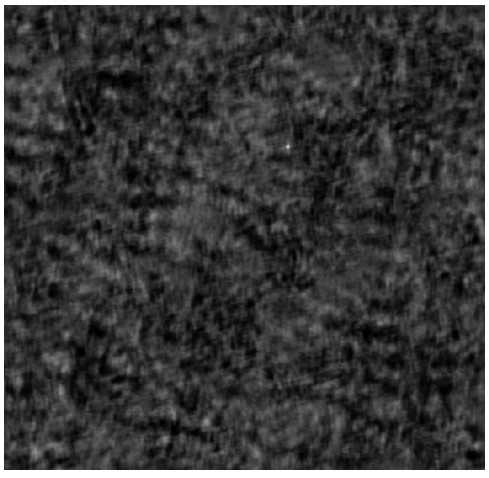


Figure 2.9: The correlation map between the image and the template filter with brightness corresponding to similarity to the template.

See that tiny bright spot around the center of the top half of the correlation map in Figure 2.9? That's Waldo's location in the original image (see Figure 2.8).

Non-Identical Template Matching

What if we don't have a perfect template to start with? For example, what if we want to detect most cars, but only have a single template image of a car. As it turns out, if the stars align – as in, we have similarity in scale, orientation, color, etc. – a template matching filter can still detect similar objects because they result in the "best" match to the template.

How would this relate to finding Waldo? Well, in our example, we had the *exact* template from the source image. The pictures are naturally designed to make it difficult to find Waldo: there are others in red-striped shirts, or in glasses, or surrounded by red-striped textures. Thus, an inexact match may give us an assortment of places where to *start* looking, but is unlikely to pinpoint Waldo perfectly for other images.

Applications

What's template matching useful for? Can we apply it to other problems? What about using it to match shapes, lines, or faces? We must keep in mind the limitations of this rudimentary matching technique. Template matching relies on having a near-perfect representation of the target to use as a filter. Using it to match lines – which vary in size, scale, direction, etc. – is unreliable. Similarly, faces may be rotated, scaled, or have varying features. There are much better options for this kind of matching that we'll discuss later. Something specific, like icons on a computer or words in a specific font, are a viable application of template matching.

Edge Detection

> *"How you can sit there, calmly eating muffins when we are in this horrible trouble, I can't make out. You seem to me to be perfectly heartless."*
>
> *"Well, I can't eat muffins in an agitated manner. The butter would probably get on my cuffs. One should always eat muffins quite calmly. It is the only way to eat them."*
>
> *"I say it's perfectly heartless your eating muffins at all, under the circumstances."*
>
> — Oscar Wilde, *The Importance of Being Earnest*

In this chapter, we will continue to explore the concept of using filters to match certain features we're looking for, like we discussed in Template Matching. Now, though, we're going to be "analysing" *generic* images that we know nothing about in advance. What are some good features that we could try to find that give us a lot to work with in our interpretation of the image?

3.1 The Importance of Edges

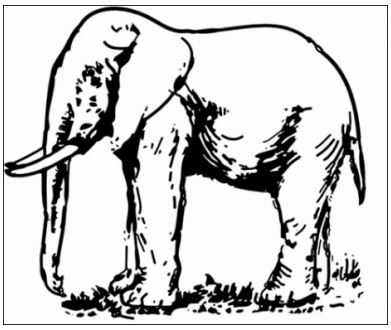


Figure 3.1: An elephant. Can you imagine an archeologist many years from now finding the skeletal remains of an elephant? The ears and tusk would be lost entirely in any rendition, and such a sketch would be a work of fantasy.

Consider a sketch of an elephant, as in Figure 3.1. It's simple and straightforward. There are no colors: it's drawn with a dark pencil. Yet, you can absolutely make out the fact that it's an elephant, not an albino giraffe or an amorphous blob.

All it takes are a few well-placed edges to understand an image. Thus this section is dedicated **edge detection algorithms**.

How can we detect edges? We must return to the idea of Images as Functions. We visually interpret edges as being sharp contrasts between two areas; mathematically, we can say that *edges*

look like steep cliffs in an ~~image~~ function.[1]

Basic Idea: If we can find a neighborhood of an image with strong signs of change, that's an indication that an edge may exist there.

3.2 Gradient Operator

Contrast, or *change*, in a function is modeled by its derivative. Thus, we will be using a differential operator on our images to get a sense of the change that's occurring in it. We'll model these operators as kernels that compute the **image gradient function** over the entire image, then threshold this gradient function to try to select edge pixels.

The gradient operator is the vector of partial derivatives for a particular function. For images, our functions are parameterized by x and y; thus, the gradient operator would be defined as

$$\nabla := \left[\frac{\partial}{\partial x}, \frac{\partial}{\partial y}\right]$$

Direction The gradient points in the direction of most rapid increase in intensity. For an image f, the gradient's direction is given by:

$$\theta = \tan^{-1}\left(\frac{\partial f}{\partial y} / \frac{\partial f}{\partial x}\right) \tag{3.1}$$

Magnitude The "amount of change" in the gradient is given by its magnitude:

$$\|\nabla f\| = \sqrt{\left(\frac{\partial f}{\partial x}\right)^2 + \left(\frac{\partial f}{\partial y}\right)^2}$$

3.2.1 Finite Differences

Well the gradient is all well and good, but we can't apply partial derivatives to images the same way we would in math class. Images are discrete, and so we need to approximate the partial derivative. We could, for instance, consider the following approximation that relies on a **finite difference** between two points:

$$\frac{\partial f(x,y)}{\partial x} \approx f(x+1, y) - f(x, y)$$

This is the **right derivative**, but it's not necessarily the right derivative.[2] If we look at the finite difference stepping in the x direction, we will heavily accentuate *vertical* edges (since we move across them), whereas if we look at finite differences in the y direction, we will heavily accentuate *horizontal* edges.

[1] Steep cliffs? That sounds like a job for slope, doesn't it? Well, derivatives are exactly what's up next.
[2] Get it, because we take a step to the *right* for the partial derivative?

3.2.2 The Discrete Gradient

We want an operator that we can apply to a kernel and use as a correlation (or convolution) filter. Consider the following operator:

$$H := \begin{bmatrix} 0 & 0 & 0 \\ -\frac{1}{2} & 0 & +\frac{1}{2} \\ 0 & 0 & 0 \end{bmatrix}$$

What's up with the $\frac{1}{2}$s? This is the average between the "left derivative" (which would be $\begin{bmatrix} -1 & +1 & 0 \end{bmatrix}$ and the "right derivative" (which would be $\begin{bmatrix} 0 & -1 & +1 \end{bmatrix}$).

Sobel Operator

The **Sobel operator** is a discrete gradient operator that preserves the "neighborliness" of an image that we discussed earlier when talking about the Gaussian blur filter (2.3) in Blurring Images. It looks like this:[3]

$$S_x = \frac{1}{8} \cdot \begin{bmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{bmatrix} \qquad S_y = \frac{1}{8} \cdot \begin{bmatrix} +1 & +2 & +1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} \tag{3.2}$$

We say that the *application* of the Sobel operator is g_x for S_x and g_y for S_y:

$$\nabla I = \begin{bmatrix} g_x & g_y \end{bmatrix}^T$$

The Sobel operator results in edge images that are not *great* but also not too bad. There are other operators that use different constants, such as the **Prewitt operator**:

$$S_x = \begin{bmatrix} -1 & 0 & +1 \\ -1 & 0 & +1 \\ -1 & 0 & +1 \end{bmatrix} \qquad S_y = \begin{bmatrix} +1 & +1 & +1 \\ 0 & 0 & 0 \\ -1 & -1 & -1 \end{bmatrix} \tag{3.3}$$

and the **Roberts operator**:

$$S_x = \begin{bmatrix} 0 & +1 \\ -1 & 0 \end{bmatrix} \qquad S_y = \begin{bmatrix} +1 & 0 \\ 0 & -1 \end{bmatrix} \tag{3.4}$$

but we won't go into detail on these. Instead, let's explore how we can improve our edge detection results.

[3] NOTE: In S_y, positive y is upward; the origin is assumed to be at the bottom-left.

3.2.3 Handling Noise

Our gradient operators perform reasonably-well in idealized scenarios, but images are often very noisy. As we increase the noise in an image, the gradient – and thus its representation of the edges – get much messier.

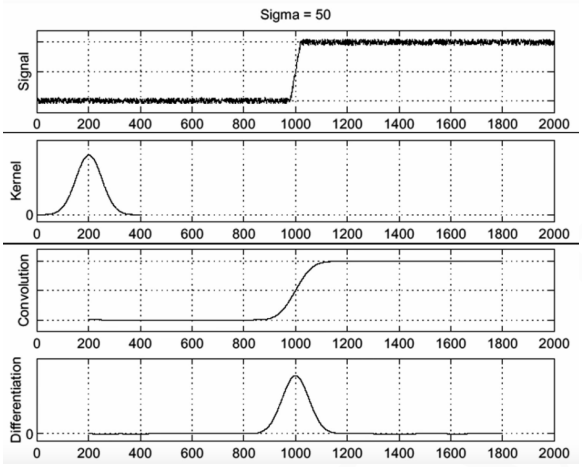


Figure 3.2: Using a smoothing filter (like the Gaussian) to combat image noise affecting the gradient. From top to bottom, we have (a) the original noisy signal, f; (b) the smoothing filter, h; (c) the smoothed signal, $h \circledast f$; and (d) the gradient of the result, $\frac{\partial}{\partial f}(h \circledast f)$.

Of course, we know how to reduce noise: apply a smoothing filter! Then, edges are peaks, as seen in Figure 3.2. We can also perform a minor optimization here; recall that, thanks to the differentiation property (discussed in Properties of Convolution):

$$\frac{\partial}{\partial x}(h \circledast f) = \left(\frac{\partial}{\partial x} h\right) \circledast f \tag{3.5}$$

Meaning we can skip Step (d) in Figure 3.2 and convolve the derivative of the smoothing filter to the signal directly to get the result.

We Have to Go Deeper...

As we saw in Figure 3.2, edges are represented by peaks in the resulting signal. How do we detect peaks? More derivatives! So, consider the **2nd derivative Gaussian operator**:

$$\frac{\partial^2}{\partial x^2} h$$

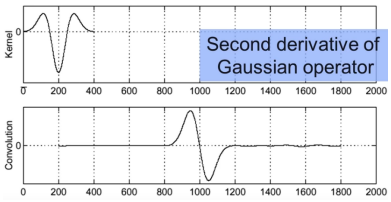


Figure 3.3: Using the 2nd order Gaussian to detect edge peaks.

which we convolve with the signal to detect peaks.
And we do see in Figure 3.3 that we've absolutely detected an edge at $x = 1000$ in the original signal from Figure 3.2.

3.3 Dimension Extension Detection

We've been discussing examples of gradients in one direction. It's time to extend that to two, and thus, to proper images.

When working in 2 dimensions, we need to specify the direction in which we are taking the derivative. Consider, then, the 2D extension of the derivative of the Gaussian filter we started with in (3.5):

$$(I \otimes g) \otimes h_x = I \otimes (g \otimes h_x)$$

Here, g is our Gaussian filter (2.3) and h_x is the x-version of our gradient operator, which could be the Sobel operator (3.2) or one of the others. We prefer the version on the right because it creates reusable function that we can apply to any image, and it operates on a smaller kernel, saving computational power.

Tweaking σ, Revisited Much like in the previous version of Tweaking Sigma, there is an effect of changing σ in our Gaussian smoothing filter g. Here, smaller values of σ detect finer features and edges, because less noise is removed and hence the gradient is more volatile. Similarly, larger values of σ will only leave larger edges detected.

3.4 From Gradients to Edges

So how do we get from these gradient images to actual edges? In general, it's a multi-step process:

Smoothing First, we suppress noise by performing smoothing on the image.

Gradient Then, we compute the gradient to find the areas of the image that have significant localized change (i.e. the "steep cliffs" we described at the start). Recall, though, that this step can be combined with the previous step because of associativity.

Threshold Aside from tweaking σ, we can also clip the gradient to some range of values to limit our edges to the "significant" parts of the gradient we're interested in.

Thinning & Connecting Finally, we perform thinning to get the localized edge pixels that we can work with, and connect these edge pixels to create a "complete" representation of the edge if desired.

There are different algorithms that take different approaches at each of these steps. We will discuss two, but there are many more.

3.4.1 Canny Edge Operator

Designed by John Canny for his Master's thesis, the **Canny edge detector** is an algorithm for creating a proper representation of an image's edges:

1. Filter the image with the derivative of the Gaussian.
2. Find the magnitude and orientation of the gradient.
3. Perform Non-Maximal Suppression, which thins the multi-pixel ridges of the gradient into a single-pixel width.
4. Threshold and link (**hysteresis**) the edges. This is done by choosing two thresholds, low and high, and using the high threshold to start edge curves and the low one to continue them.

Let's discuss the latter two steps in more detail.

Non-Maximal Suppression

This is a fancy term for what amounts to choosing the brightest (maximal) pixel of an edge and discarding the rest (suppression). It works by looking in the gradient direction and keeping the maximal pixel, as shown in here:

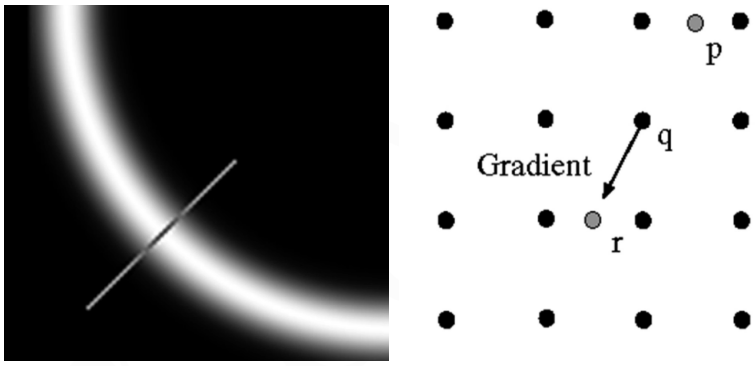


Figure 3.4: Given a series of edge points (left), we can calculate each point's gradient and keep only the ones with high values (right).

Non-maximal suppression is a general-purpose strategy towards filtering out lower-quality results and isn't restricted to edge detection applications. For example, Figure 3.5 demonstrates adaptive NMS applied to feature matching (which we'll discuss *much* later in chapter 9) to only keep high-quality (and evenly-distributed) "points of interest."

Canary Threshold Hysteresis

After non-maximal suppression, some valid edge pixels didn't survive thresholding! We can't just lower the threshold, though, since that would introduce non-edge noise.

The cleverness of the detector comes from using *two* thresholds. First, a high threshold determines "strong edges," which are then linked together. Then, a low threshold is used to determine weak edges that are plausible. Finally, the original strong edges are extended if they can follow weak edges.

The assumption is that all of the edges that we care about have strong edges, but might have some portions that fade out. The weak threshold links strong edges together without adding excess noise from the weaker gradients.

Figure 3.5: Adaptive non-maximal suppression applied to feature detection. Similar clusters of feature points are consolidated into a single point, and the resulting spread of points is distributed well across the image. The same rationale of keeping the "strongest" points is applied to the detected edges. (*These images come from a student project in Berkeley's* Computational Photography *course, here.)*

3.4.2 2nd Order Gaussian in 2D

Recall when we performed the 2nd derivative on the Gaussian filter to create "zero-crossings" at edges in our output signal in Figure 3.3. How do we do this in 2 dimensions? Recall that the Gaussian filter (2.3) is parameterized by (u, v), and so has two possible directions to take the partial derivative. The 2nd derivative has even *more* choices... how do we know which one to use?

The answer? **None of them**. Instead, we apply the **Laplacian** operator, defined as ∇^2:

$$\nabla^2 h = \frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2} \tag{3.6}$$

Once you apply the Laplacian, the zero-crossings are the edges of the image. This operator is an alternative to the Canny Edge Operator and is better under certain circumstances.

Hough Transform

> *"Little pig, little pig won't you let me come in?"*
>
> *"No, no, no, by the hair on my chinny, chin, chin."*
>
> *"Then I'll huff and I'll puff and I'll blow your house in."*
>
> *"No, no, no, Mr. Wolf I will not let you in."*
>
> — *The Three Little Pigs*, a folktale

We can finally begin to discuss concepts that enable "real vision," as opposed to the previous chapters which belonged more in the category of image processing. We gave a function an input image and similarly received an output image. Now, we'll discuss learning "good stuff" about an image, stuff that tells us things about what an image *means*. This includes the presence of lines, circles, particular objects or shapes, and more!

4.1 Line Fitting

We begin with a simple desire: finding the lines in an image. This is a lot more difficult than it sounds, and our models for Edge Detection only get us part of the way. Why isn't edge detection sufficient? Unfortunately, edge detection in the forms we've seen isn't perfect; there are many flaws:

Clutter There are a lot of resulting points and edges that clutter the image and don't necessarily represent lines. Furthermore, having multiple models that we want to find increases the computational complexity of the search.

Partial Matches Even with the clever hysteresis that extends and connects edges in the Canny Edge Operator, the edges that we find won't always represent the entire line present in the original image.

Noise Even without the above imperfections, the resulting edges are noisy. Even if the original image has a perfectly straight line, the detected edge might not be straight: it may deviate a few pixels in some direction(s), be slightly rotated (i.e. of a different slope), or have some extra noise from other extraneous image details like texture.

4.1.1 Voting

Since it is computationally infeasible to simply try every possible line given a set of edge pixels, instead, we need to let the data tell us something. We approach this by introducing **voting**, which is a general technique where we let the features (in this case, edge pixels) vote for all of the models with which they are compatible.

Voting is very straightforward: we cycle through the features, and each casts votes on particular model parameters; then, we look at model parameters with a high number of votes. The idea behind the validity of voting is that all of the outliers – model parameters that are only valid for one or two pixels – are varied: we can rely on valid parameters being elected by the majority. Metaphorically, the "silly votes" for candidates like Donald Duck are evenly distributed over an assortment of irrelevant choices and can thus be uniformally disregarded.

4.1.2 Hough Transform

Pronounced "huff," the **Hough transform** is a voting technique that can be used answer all of our questions about finding lines in an image:

- Given points that belong to a line, what is that line?
- How many lines are there?
- Which points belong to which lines?

The main idea behind the Hough transform is that each edge point (i.e. each pixel found after applying an edge-detecting operator) votes for its compatible line(s). Hopefully, the lines that get many votes are valid.

Hough Space

To achieve this voting mechanism, we need to introduce **Hough space** – also called **parameter space** – which enables a different representation of our desired shapes.

The key is that a *line* $y = mx + b$ in image space represents a *point* in Hough space because we use the *parameters* of the line (m, b). Similarly, a *point* in image space represents a *line* in Hough space through some simple algebraic manipulation. This is shown in Figure 4.1. Given a point (x_0, y_0), we know that all of the lines going through it fit the equation $y_0 = mx_0 + b$. Thus, we can rearrange this to be $b = -x_0m + y_0$, which is a line in Hough space.

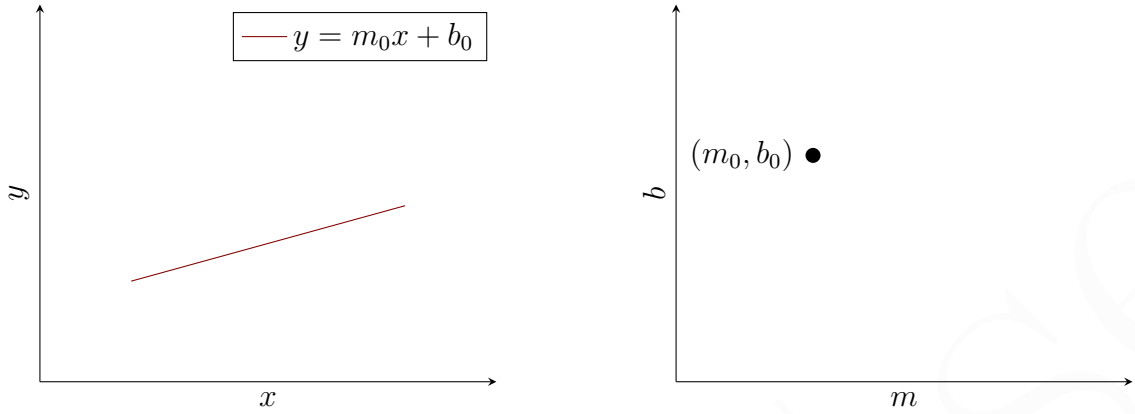
What if we have two points? We can easily determine the line passing through both of those points in Cartesian space by applying the point-slope formula:

$$y - y_1 = m(x - x_1)$$

where, as we all know:

$$m = \frac{y_2 - y_1}{x_2 - x_1}$$

That's simple enough for two points, as there's a distinct line that passes through them by definition. But what if we introduce *more* points, and what if those points can't be modeled



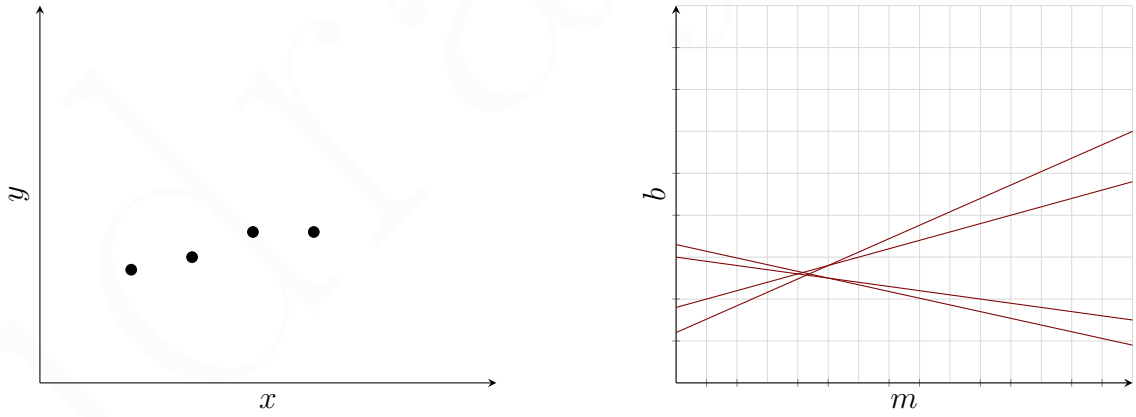
(a) A line on the Cartesian plane, represented in the traditional slope-intercept form.

(b) A parameterized representation of the same line in Hough space.

Figure 4.1: Transforming a line from the Cartesian plane (which we'll call "image space") to a point in Hough space.

by a perfect line that passes through them? We need a line of best fit, and we can use Hough space for that.

A point on an image is a line in Hough space, and two points are two lines. The point at which these lines intersect is the exact (m, b) that we would've found above, had we converted to slope-intercept form. Similarly, a bunch of points produces a bunch of lines in Hough, all of which intersect in (or near) one place. The closer the points are to a particular line of best fit, the closer their points of intersection in Hough space. This is shown in Figure 4.2.



(a) A series of points in image space.

(b) The lines representing the possible parameters for each of those points in Hough space.

Figure 4.2: Finding the line of best fit for a series of points by determining an approximate point of intersection in a discretized Hough space.

If we discretize Hough space into "bins" for voting, we end up with the **Hough algorithm**. In the Hough algorithm, each point in image space votes for every bin along its line in Hough space: the bin with the most votes becomes the line of best fit among the points.

Unfortunately, the (m, b) representation of lines comes with some problems. For example, a vertical line has $m = \infty$, which is difficult to represent and correlate using this algorithm. To avoid this, we'll introduce Polar Representation of Lines.

4.1.3 Polar Representation of Lines

In the polar representation of a line, we can uniquely identify a line by its perpendicular distance from the origin, d, and the angle of that perpendicular with the x-axis, θ. We can see this visually in Figure 4.3.

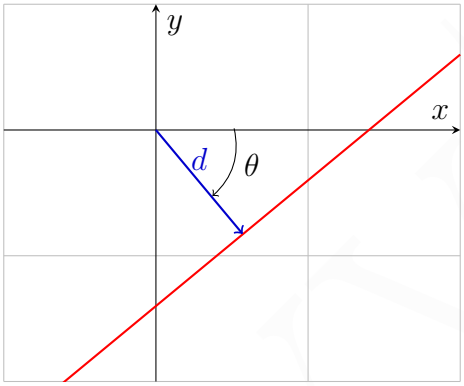


Figure 4.3: Representing a line in polar coordinates.

To get a relationship between the Cartesian and polar spaces, we can derive, via some simple vector algebra, that:

$$x \cos\theta + y \sin\theta = d \tag{4.1}$$

This avoids all of the previous problems we had with certain lines being ill-defined. In this case, a vertical line is represented by $\theta = 0$, which is a perfectly valid input to the trig functions. Unfortunately, though, now our transformation into Hough space is more complicated. If we know x and y, then what we have left in terms of d and θ is a **sinusoid** like in Figure 4.4.

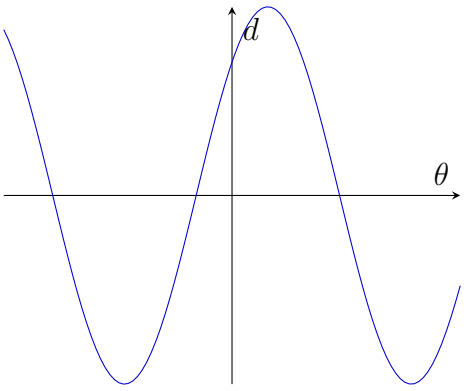


Figure 4.4: A simple sinusoidal function, which is the manifestation of a point in image space when transformed into Hough space.

Note There are multiple ways to represent all possible lines under polar coordinates. Either $d > 0$, and so $\theta \in [0, 2\pi)$, or d can be both positive or negative, and then $\theta \in [0, \pi)$. Furthermore, when working with images, we consider the origin as being in one of the corners, which restricts us to a single quadrant, so $\theta \in [0, \frac{\pi}{2})$. Of course, these are just choices that we make in our representation and don't really have *mathematical* trade-offs.

4.1.4 Hough Algorithm

Let's synthesize all of these concepts and formalize the **Hough transformation algorithm**. We begin with a polar representation for lines, as discussed in, well, Polar Representation of Lines, as well as a **Hough accumulator array** which is a discretization of the coordinate plane used to track votes for particular values of (θ, d). With that, the algorithm is as follows:

Algorithm 4.1: The basic Hough algorithm for line detection.

Input: An image, I
Result: Detected line(s).

Initialize an empty voting array: $H[d, \theta] = 0$
foreach *edge point,* $(x, y) \in I$ **do**
  foreach $\theta \in [0, 180, step)$ **do**
    $d = x \cos\theta + y \sin\theta$
    $H[d, \theta]$ += 1
  end
end
Find the value(s) of (d, θ) where $H[d, \theta]$ is a maximum.
The result is given by $d = x \cos\theta + y \sin\theta$.

Complexity

The **space complexity** of the Hough Algorithm is simply k^n: we are working in n dimensions (2 for lines) and each one gets k bins. Naturally, this means working with more complex objects (like circles as we'll see later) that increase dimension count in Hough space can get expensive fast.

The **time complexity** is linearly proportional to the number of edge points, whereas the voting itself is constant.

4.1.5 Handling Noise

Small amounts of noise end up creating many similar peaks that could be perceived as separate lines. Furthermore, with too-fine of a bin size for the voting array, you could miss the peaks altogether!

Well, what if we smoothed the image in Hough space? It would blend similar peaks together, but would reduce the area in which we need to search. Then, we can run the Hough transform *again* on that smaller area and use a finer grid to find the best peaks.

What about a **lot** of noise? As in, what if the input image is just a random assortment of pixels. We run through the Hough transform expecting to find peaks (and thus lines) when there are none! So, it's useful to have some sort of prior information about our expectations.

4.1.6 Extensions

The most common extension to the Hough transform leverages the gradient (recall, if need be, the Gradient Operator). The following algorithm is nearly identical to the basic version presented in algorithm 4.1, but we eliminate the loop over *all* possible θs.

Algorithm 4.2: The gradient variant of the Hough algorithm for line detection.

Input: An image, I
Result: Detected line(s).

Initialize an empty voting array: $H[d, \theta] = 0$
foreach *edge point,* $E(x, y) \in I$ **do**
  $\theta = \nabla I\big|_{(x,y)}$ `/* or some range influenced by` ∇I `*/`
  $d = x \cos\theta + y \sin\theta$
  $H[d, \theta]$ += 1
end
Find the value(s) of (d, θ) where $H[d, \theta]$ is a maximum.
The result is given by $d = x \cos\theta + y \sin\theta$.

The gradient hints at the direction of the line: it can be used as a starting point to reduce the range of θ from its old range of $[0, 180)$.

Another extension gives stronger edges more votes. Recall that the Canny Edge Operator used two thresholds to determine which edges were stronger. We can leverage this "metadata" information to let stronger edges influence the reuslts of the line detection. strongest Of course, this is far less democratic, but gives more reliable results.

Yet another extension leverages the sampling resolution of the voting array. Changing the discretization of (d, θ) refines the lines that are detected. This could be problematic if there are two similar lines that fall into the same bin given too coarse of a grid. The extension does grid redefinition hierarchically: after determining ranges in which peaks exist with a coarse grid, we can go back to *just* those regions with a finer grid to pick out lines.

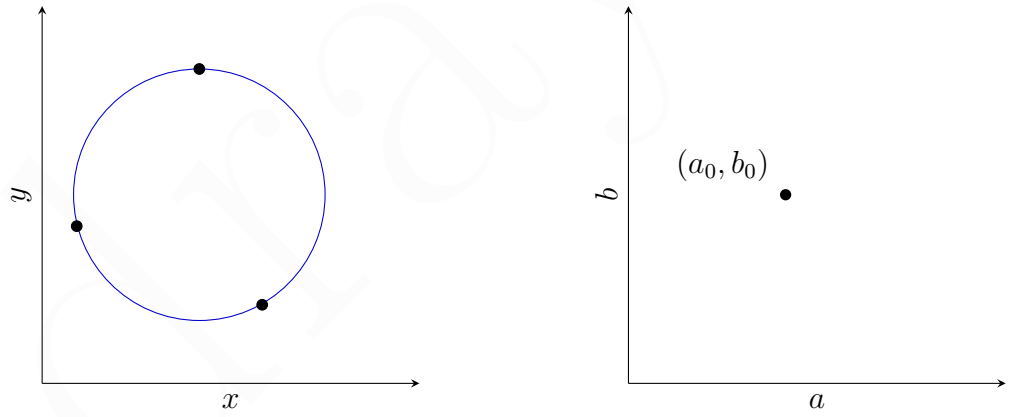
Finally, and most importantly, we can modify the basic Hough algorithm to work on more complex shapes such as circles, squares, or actually *any* shape that can be defined by a template. In fact, that's the subject of the next sections.

4.2 Finding Circles

We begin by extending the parametric model that enabled the Hough algorithm to a slightly more complex shape: **circles**. A circle can be uniquely defined by its **center**, (a, b), and a **radius**, r. Formally, its equation can be stated as:

$$(x_i - a)^2 + (y_i - b)^2 = r^2 \tag{4.2}$$

For simplicitly, we will begin by assuming the radius is known. How does voting work, then? Much like a point on a line in image space was a line in Hough space, a point in image space on a circle is a circle in Hough space, as in Figure 4.5:

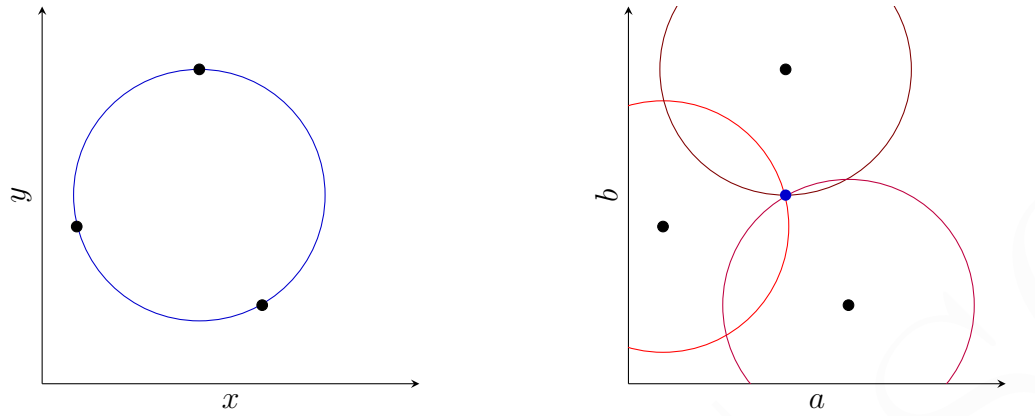


(a) An arbitrary circle in image space defined by a series of points.

(b) A parameterized representation of the same circle in Hough space, assuming a known radius.

Figure 4.5: The transformation from a circle, defined by a handful of points, into a single point in Hough space.

Thus, each point on the not-yet-defined circle votes for a set of points surrounding that same location in Hough space at the known radius, as in Figure 4.6:



(a) The same circle from Figure 4.5a.

(b) The set of points voted on by each corresponding point in Hough space.

Figure 4.6: The voting process for a set of points defining a circle in image space. The overlap of the voting areas in Hough space defines the (a, b) for the circle.

Of course, having a known radius is not a realistic expectation for most real-world scenarios. You might have a range of viable values, or you may know nothing at all. With an **unknown radius**, each point on the circle in image space votes on a set of values in Hough space resembling a *cone*.[1] Again, that's *each point*: growing the dimensionality of our parameterization leads to unsustainable growth of the voting process.

We can overcome this growth problem by taking advantage of the gradient, much like we did in the Extensions for the Hough algorithm to reduce the range of θ. We can visualize the gradient as being a tangent line of the circle: if we knew the radius, a single point and its tangent line would be enough to define the circle. Since we don't, there's a line of possible values for the center. As we see in Figure 4.7, where the blue circle is the circle defined for a specific radius given the green point and its gradient direction, the red line defines the range of possible locations for the center of the circle if we *don't* know the radius.

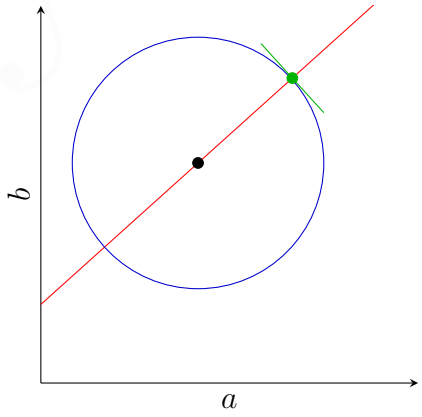


Figure 4.7: Finding the line of possible values for the center of the circle given a point and its gradient. The blue circle represents an example circle if there were a known radius.

With each point in image space now defining a line of votes in Hough space, we've drastically improved the computational complexity

[1] If we imagine the Hough space as being an *abr* plane with r going upward, and we take a known point (a_0, b_0), we can imagine if we *did* know the radius, say $r = 7$, we'd draw a circle there. But if was $r = 4$ we'd draw a circle a little lower (and a little smaller). If we extrapolate for all possible values of r, we get a cone.

of the voting process despite increasing the dimension. This leads us to a basic Hough algorithm for circles:

Algorithm 4.3: The Hough algorithm for circles.

Input: An image, I
Result: A detected circle.

Initialize an empty voting array: $H[a, b, r] = 0$
foreach *edge point,* $E(x, y) \in I$ **do**
  foreach *possible radius* **do**
    foreach *each possible gradient direction* θ **do** `// or an estimate via (3.1)`
      $a = x - r \cos\theta$
      $b = y + r \sin\theta$
      $H[a, b, r] \mathrel{+}= 1$
    end
  end
end
Find the value(s) of (a, b, r) where $H[a, b, r]$ is a maximum.
The result is given by applying the equation of a circle (4.2).

In practice, we want to apply the same tips that we outlined in Extensions and a few others: use edges with significant gradients, choose a good grid discretization, track which points make which votes, and consider smoothing your votes by voting for neighboring bins (perhaps with a different weight).

4.3 Generalization

The advent of the **generalized Hough transform** and its ability to determine the existence of *any* well-defined shape has caused a resurgence in its application in computer vision.

Rather than working with **non-analytic models** that have fixed parameters (like circles with a radius, r), we'll be working with **visual code-words** that describe the object's features rather than using its basic edge pixels. Previously, we knew how to vote given a particular pixel because we had solved the equation for the shape. For an arbitrary shape, we instead determine how to vote by building a **Hough table**.

4.3.1 Hough Tables

A Hough table stores displacement vectors for a particular gradient angle θ. To "train" a Hough table on a particular shape, we can follow these steps, assisted by the arbitrary shape in Figure 4.8:

1. At each boundary point (defined by the shape), compute a displacement vector $\mathbf{r} = c - p_i$, where c can be the center of the object, or just some arbitrary reference point.
2. Measure (or, rather, approximate, since our best guess uses differences with neighboring pixels) the gradient angle θ at the boundary point.
3. Store the displacement vector $\mathbf{r}$ in a table indexed by θ.

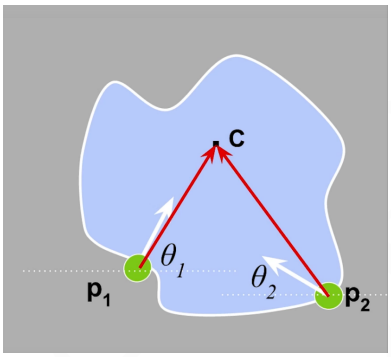
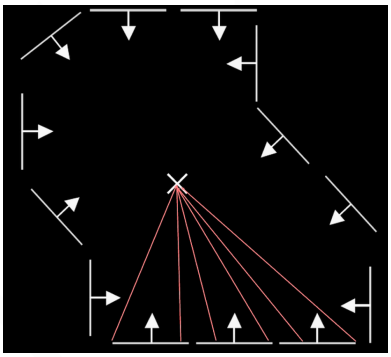


Figure 4.8: An arbitrary shape that outlines building a few indices of its Hough table.

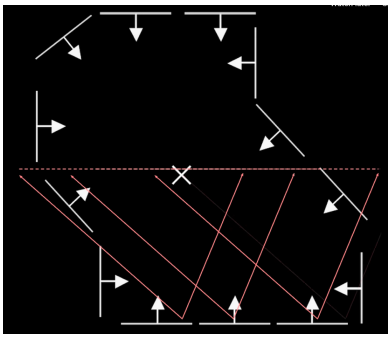
Then, at "recognition" time, we essentially go backwards:

1. At each boundary point, measure the gradient angle θ.
2. Look up all displacements for that θ in the Hough table.
3. Vote for a center at each displacement.

Figure 4.9 demonstrates the voting pattern after accumulating the votes for a feature in the object – in this case, it's the bottom horizontal line.



(a) The training phase for the horizontal line "feature" in the shape, in which the entry for $\theta = 90°$ stores all of these displacement vectors.



(b) The voting phase, in which each of the three points (so far) has voted for the center points after applying the entire set of displacement vectors.

Figure 4.9: Voting and recognition of a single feature (i.e. a horizontal line) in an arbitrary shape. Notice that if we were to extrapolate voting for the entire feature, we'd get strong votes for a horizontal line of possible centers, with the amount of votes growing with more overlap.

After all of the boundary points vote for their "line of possible centers," the strongest point of intersection among the lines will be the initial reference point, c, as seen in Figure 4.10.

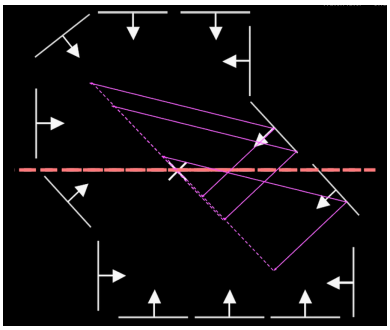


Figure 4.10: A subset of votes cast during the second set of feature points, after completing the voting for the first set that was started in Figure 4.9.

This leads to the following generalized Hough transform algorithm, which (critically!) assumes the *orientation is known*:

Algorithm 4.4: The generalized Hough transform, for *known* orientations.

Input: I, an image.
Input: T, a Hough table trained on the arbitrary object.
Result: The center point of the recognized object.

Initialize an empty voting array: $H[x, y] = 0$
foreach *edge point,* $E(x, y) \in I$ **do**
  Compute the gradient direction, θ
  foreach $\mathbf{v} \in T[\theta]$ **do**
    $H[v_x, v_y]$ += 1
  end
end
The peak in the Hough space is the reference point with the most supported edges.

Of course, variations in orientation are just an additional variable. All we have to do is try all of the possible orientations. Naturally, this is much more expensive since we are adding another dimension to our Hough space. We can do the same thing for the *scale* of our arbitrary shape, and the algorithm is nearly identical to algorithm 4.5, except we vote with

a "master scale" instead of a "master orientation."

ALGORITHM 4.5: The generalized Hough transform, for *unknown* orientations.

Input: I, an image.
Input: T, a Hough table trained on the arbitrary object.
Result: The center point of the recognized object.

Initialize an empty voting array: $H[x, y, t] = 0$
foreach *edge point,* $E(x, y) \in I$ **do**
  foreach *possible* θ^* **do** `// (the "master" orientation)`
    Compute the gradient direction, θ
    $\theta' = \theta - \theta^*$
    foreach $\mathbf{v} \in T[\theta']$ **do**
      $H[v_x, v_y, \theta^*] \mathrel{+}= 1$
    end
  end
end
The peak in the Hough space (which is now (x, y, θ^*)) is the reference point with the most supported edges.

Frequency Analysis

> *The peoples of civilization see their wretchedness increase in direct proportion to the advance of industry.*
>
> — Charles Fourier

The goal of this chapter is to return to image processing and gain an intuition for images from a signal processing perspective. We'll introduce the Fourier transform, then touch on and study an phenomenon called **aliasing**, in which seemingly-straight lines in an image appear jagged. Furthermore, we'll extend this idea into understanding why, to shrink an image in half, we shouldn't just throw out every other pixel.

> Warning: **Here Be Dragons**
>
> Things are boutta get *real* mathematical up in here. Do some review of fundamental linear algebra concepts before going further. You've been warned.

5.1 Basis Sets

A **basis** set B is defined as a **linearly independent** subset of a **vector space** V that spans V. For example, the vectors $\begin{bmatrix}0 & 1\end{bmatrix}$ and $\begin{bmatrix}1 & 0\end{bmatrix}$ make up a basis set for the xy-plane: every vector in the xy-plane can be represented as a linear combination of these two vectors.

Formally: Suppose we have some $B = \{\mathbf{v}_1, \ldots, \mathbf{v}_n\}$, which is a finite subset of a vector space V over a **field** F (such as the real or complex number fields, $\mathbb{R}$ and $\mathbb{C}$). Then, B is a basis for V if it satisfies the following conditions:

- **Linear independence**, which is the property that

$$\begin{aligned} &\forall a_1, \ldots a_n \in F \\ &\text{if } a_1\mathbf{v}_1 + \ldots + a_n\mathbf{v}_n = \mathbf{0} \\ &\text{then } a_1 = \ldots = a_n = 0 \end{aligned}$$

  In other words, for any set of constants such that the linear combination of those constants and the basis vectors is the zero vector, then it *must* be that those constants

are all zero.

- **Spanning property**, which states that

$$\forall \mathbf{x} \in V, \quad \exists a_1, \ldots, a_n \in F$$

such that

$$\mathbf{x} = a_1\mathbf{v}_1 + \ldots + a_n\mathbf{v}_n = \mathbf{0}$$

In English, for any vector in that vector space, we can find *some* constants such that their linear combination with the basis set is the zero vector. But in other words, it means we can make any vector in the vector space with some combination of the (scaled) basis vectors.

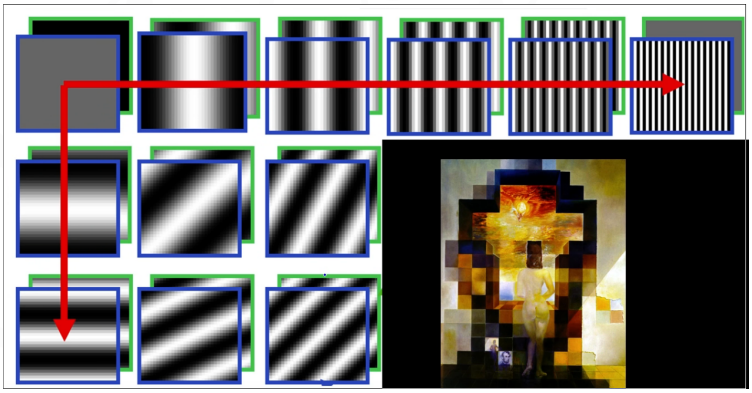
With that in mind, we can consider images as being a *single point* in an $N \times N$ vector space:

$$\begin{bmatrix} x_{00}x_{10}x_{20}\ldots x_{(n-1)0}x_{10}\ldots x_{(n-1)(n-1)} \end{bmatrix}^T$$

We can formulate a simple basis set for this vector space by toggling each pixel as on or off:

$$\left\{ \begin{array}{c} \begin{bmatrix} 000\ldots0100\ldots000 \end{bmatrix}^T \\ \begin{bmatrix} 000\ldots0010\ldots000 \end{bmatrix}^T \\ \vdots \end{array} \right.$$

This is obviously independent and can create any image, but isn't very useful... Instead, we can view an image as a variation in frequency in the horizontal and vertical directions, and the basis set would be vectors that tease away fast vs. slow changes in the image:



This is called the **Fourier basis set**.

5.2 Fourier Transform

Jean Baptiste Joseph Fourier (not the guy in the chapter quote) realised that any periodic (repeating) function can be rewritten as a weighted sum of sines and cosines of different frequencies, and we call this the **Fourier series**. Our building block is a sinusoid:

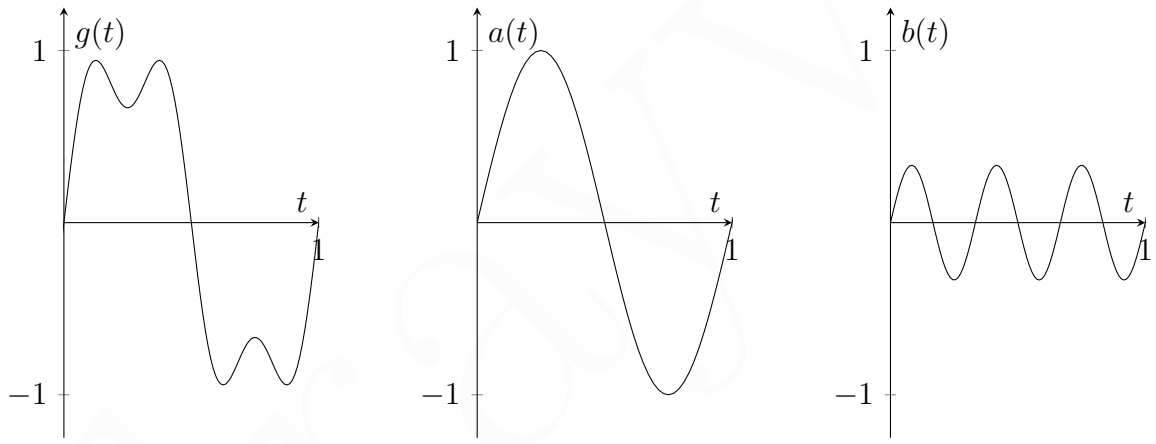
$A\sin(\omega x + \varphi)$. Fourier's conclusion was that if we add enough of these together, we can get *any* signal $f(x)$. Our sinusoid allows three degrees of freedom:

- A is the **amplitude**, which is just a scalar for the sinusoid.
- ω is the **frequency**. This parameter controls the coarseness vs. fine-ness of a signal; as you increase it, the signal "wiggles" more *frequently*.
- φ is the **phase**. We won't be discussing this much because our goal in computer vision isn't often to *reconstruct* images, but rather simply to learn something about them.

Time & Frequency Suppose we have some sample signal:

$$g(t) = \sin(2\pi f t) + \frac{1}{3}\sin(2\pi(3f)t)$$

We can break it down into its "component sinusoids" like so:



If we were to analyse the **frequency spectrum** of this signal, we would see that there is some "influence" (which we'll call **power**) at the frequency f, and $1/3^{\text{rd}}$ of that influence at the frequency $3f$.

Notice that the signal seems to be approximating a square wave? Well, a **square wave** can be written as an infinite sum of odd frequencies:

$$A\sum_{1}^{\infty}\frac{1}{k}\sin(2\pi k t)$$

Now that we've described this notion of the power of a frequency on a signal, we want to transform our signals from being functions of *time* to functions of *frequency*. This algorithm is called the **Fourier transform**.

We want to understand the frequency, ω, of our signal $f(x)$. Let's reparameterize the signal by ω instead of x to get some $F(\omega)$. For every $\omega \in (-\infty, \infty)$, our $F(\omega)$ will **both** hold the

corresponding amplitude A and phase φ. How? By using **complex numbers**:

$$F(\omega) = R(\omega) + iI(\omega) \tag{5.1}$$

$$A = \pm\sqrt{R(\omega)^2 + I(\omega)^2} \tag{5.2}$$

$$\varphi = \tan^{-1}\frac{I(\omega)}{R(\omega)} \tag{5.3}$$

We will see that $R(\omega)$ corresponds to the even part (that is, cosine) and $I(\omega)$ will be the odd part (that is, sine).[1] Furthermore, computing the Fourier transform is just computing a basis set. We'll see why shortly.

First off, the infinite integral of the product of two sinusoids of *differing* frequencies is zero, and the infinite integral of the product of two sinusoids of the *same* frequency is infinite (unless they're perfectly in phase, since sine and cosine cancel out):

$$\int_{-\infty}^{\infty} \sin(ax+\phi)\sin(bx+\varphi)\,dx = \begin{cases} 0, & \text{if } a \neq b \\ \pm\infty, & \text{if } a = b, \phi + \frac{\pi}{2} \neq \varphi \\ 0, & \text{if } a = b, \phi + \frac{\pi}{2} = \varphi \end{cases}$$

With that in mind, suppose $f(x)$ is a simple cosine wave of frequency ω: $f(x) = \cos(2\pi\omega x)$. That means that we can craft a function $C(u)$:

$$C(u) = \int_{-\infty}^{\infty} f(x)\cos(2\pi u x)\,dx$$

that will infinitely spike (or, create an **impulse**... remember Impulses?) wherever $u = \pm\omega$ and be zero everywhere else.

Don't we also need to do this for all *phases*? No! Any phase can be represented as a weighted sum of cosine and sine, so we just need one of each piece. So, we've just created a function that gives us the frequency spectrum of an input signal $f(x)$... or, in other words, the Fourier transform.

To formalize this, we represent the signal as an infinite weighted sum (or **linear combination**) of an infinite number of sinusoids:[2]

$$F(u) = \int_{-\infty}^{\infty} f(x)e^{-i2\pi ux}\,dx \tag{5.4}$$

We also have the **inverse Fourier transform**, which turns a spectrum of frequencies into the original signal in the spatial spectrum:

$$f(x) = \int_{-\infty}^{\infty} F(u)e^{-i2\pi ux}\,du \tag{5.5}$$

[1] An **even** function is symmetric with respect to the y-axis: $\cos(x) = \cos(-x)$. Similarly, an **odd** function is symmetric with respect to the origin: $-\sin(x) = \sin(-x)$.

[2] Recall that $e^{ik} = \cos k + i\sin k$, where $i = \sqrt{-1}$.

5.2.1 Limitations and Discretization

The Fourier transform only exists if the input function f is integrable. That is,

$$\int_{-\infty}^{\infty} |f(x)| \, dx < \infty$$

With that in mind, if there is some range in which f is integrable (but not necessarily $(-\infty, \infty)$), we can just do the Fourier transform in just that range. More formally, if there is some bound of width T outside of which f is zero, then obviously we could integrate from $[-\frac{T}{2}, \frac{T}{2}]$. This notion of a "partial Fourier transform" leads us to the **discrete Fourier transform**, which is the only way we can do this kind of stuff in computers. The discrete Fourier transform looks like this:

$$F(k) = \frac{1}{N} \sum_{x=0}^{N-1} f(x) e^{-i\frac{2\pi kx}{N}} \tag{5.6}$$

Imagine applying this to an N-pixel image. We have x as our discrete "pixel iterator," and k represents the number of "cycles per period of the signal" (or "cycles per image") which is a measurement of how quickly we "wiggle" (changes in intensity) throughout the image.

It's necessarily true that $k \in [-\frac{N}{2}, \frac{N}{2}]$, because the highest possible frequency of an image would be a change from 0 to 255 for every pixel. In other words, every other pixel is black, which is a period of 2 and $\frac{N}{2}$ total cycles in the image.

We can extend the discrete Fourier transform into 2 dimensions fairly simply. The **2D Fourier transform** is:

$$F(u, v) = \frac{1}{2} \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} f(x, y) e^{-i2\pi(ux+vy)} \, dx \, dy \tag{5.7}$$

and the discrete variant is:[3]

$$F(k_x, k_y) = \frac{1}{N} \sum_{x=0}^{N-1} \sum_{y=0}^{N-1} f(x, y) e^{-i\frac{2\pi(k_x x + k_y y)}{N}} \tag{5.8}$$

Now typically when discussing the "presence" of a frequency, we'll be referring to its power as in (5.2) rather than the odd or even parts individually.

5.2.2 Convolution

Now we will discuss the properties of the Fourier transform. When talking about convolution, convolving a function with its Fourier transform is the same thing as multiplying the Fourier transform. In other words, let $g = f \circledast h$. Then, $G(u)$ is

$$G(u) = \int_{-\infty}^{\infty} g(x) e^{-i2\pi ux} \, dx$$

[3] As a tip, the transform works best when origin of k is in the middle of the image.

Of course, we said that $g(x)$ is a convolution of f and h, meaning:

$$G(u) = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} f(\tau)h(x-\tau)e^{-i2\pi ux}\, d\tau\, dx$$

We can rearrange this and perform a change of variables, where $x' = x - \tau$:

$$G(u) = \int_{-\infty}^{\infty} f(\tau)e^{-i2\pi u\tau}\, d\tau \int_{-\infty}^{\infty} h(x')e^{-i2\pi ux'}\, dx'$$

Notice anything? This is a product of Fourier transforms!

$$G(u) = F(u) \cdot H(u)$$

This leads us to the following property:

convolution in the *spatial domain* $\iff$ **multiplication** in the *frequency domain*

This can definitely be useful for performance, but that's less relevant nowadays. Even still, it has applications. Suppose we want to smooth the function $f(x)$. We can convolve it with a Gaussian kernel as in (2.3), *or* we can multiply $F(u)$ by the Fourier transform of the Gaussian kernel, which is actually still a Gaussian! This is demonstrated in Figure 5.1.

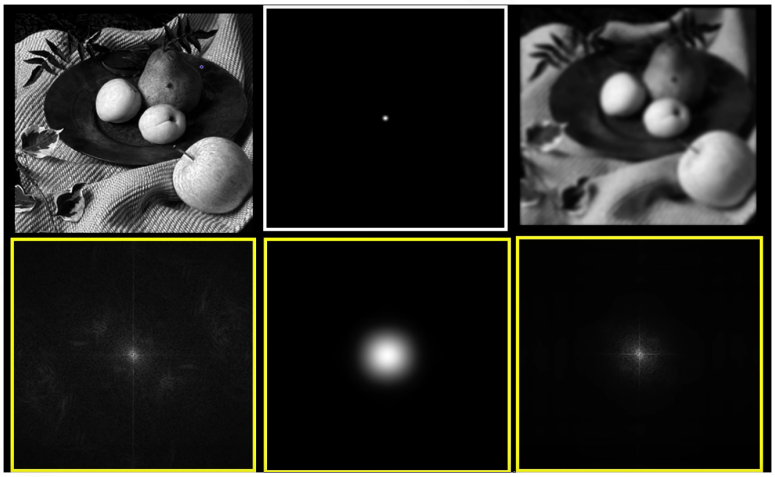


Figure 5.1: Applying a Gaussian filter to an image in both the spatial (top) and frequency (bottom) domains.

This relationship with convolution is just one of the properties of the Fourier transform. Some of the other properties are noted in Table 5.1. An interesting one is the *scaling property*: in the spatial domain, $a > 1$ will shrink the function, whereas in the frequency domain this stretches the inverse property. This is most apparent in the Gaussian filter: a tighter Gaussian (in other words, a smaller σ) in the spatial domain results in a larger Gaussian in the frequency domain (in other words, a $1/\sigma$).

	Spatial Domain (x)	**Frequency Domain** (u)
Linearity	$c_1 f(x) + c_2 g(x)$	$c_1 F(u) + c_2 G(u)$
Convolution	$f(x) \circledast g(x)$	$F(u)G(u)$
Scaling	$f(ax)$	$\frac{1}{\lvert a \rvert} F\left(\frac{u}{a}\right)$
Differentiation	$\frac{d^n f(x)}{dx^n}$	$(i2\pi u)^n F(u)$

Table 5.1: Properties of the Fourier Transform between domains.

5.3 Aliasing

With the mathematical understanding of the Fourier transform and its properties under our belt, we can apply that knowledge to the problem of aliasing.

First, let's talk about the **comb function**, also called an **impulse train**. Mathematically, it's formed like so:

$$\sum_{n=-\infty}^{\infty} \delta(x - nx_0)$$

Where $\delta(x)$ is the magical "unit impulse function," formally known as the **Kronecker delta function**. The train looks like this:

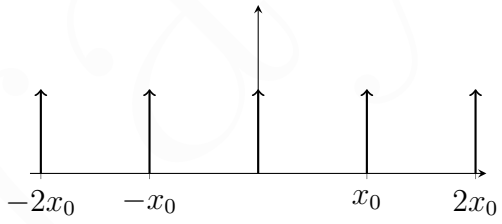


Figure 5.2: An impulse train.

The Fourier transform of an impulse train is a *wider* impulse train, behaving much like the expansion due to the scaling property.

We use impulse trains to sample continuous signals, discretizing them into something understandable by a computer. Given some signal like $\sin(t)$, we can multiply it with an impulse train and get some discrete **samples** that approximate the signal in a discrete way.

Obviously, some information is lost during sampling: our reconstruction might be imperfect if we don't have enough information. This (im)precise notion is exactly what the aliasing phenomenon is, demonstrated by Figure 5.3. The blue signal is the original signal, and the red dots are the samples we took. When trying to reconstruct the signal, the best we can do is the dashed signal, which has a lower frequency. **Aliasing** is the notion that a signal travels "in disguise" as one with a different frequency.

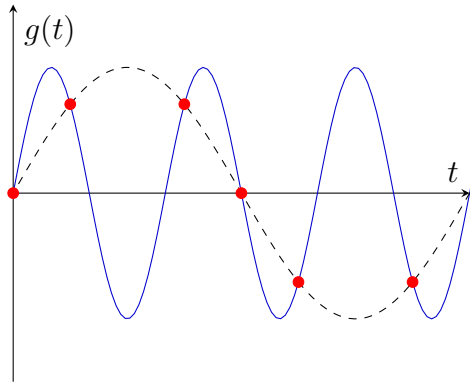


Figure 5.3: The aliasing phenomenon. The high frequency signal in blue is sampled too infrequently (samples in red); during reconstruction, a lower frequency signal (that is incorrect) can be obtained.

THEORY IN ACTION: **The Wheels on the Bus Go Round and... Backwards?**

We've all experienced aliasing in the temporal domain.

In car commercials, we've seen wheels that appear to be spinning backwards while the car moves forwards. This is an example of aliasing, and it occurs because the rotation of the wheel is too fast for a video camera to pick up accurately: it takes a picture every x frames, but the wheel's motion is much faster and the difference from image to image looks more like a small backwards rotation rather than a large forward rotation.

In images, this same thing occurs often. The aliasing problem can be summarized by the idea that there are not enough pixels (samples) to accurately render an intended effect. This begs the question: how can we prevent aliasing?

- An obvious solution comes to mind: take more samples! This is in line with the "megapixel craze" in the photo industry; camera lenses have ever-increasing fidelity and can capture more and more pixels. Unfortunately, this can't go on forever, and there will *always* be uncaptured detail simply due to the nature of going from a continuous to discrete domain.
- Another option would be to get *rid* of the problematic high-frequency information. Turns out, even though this gets rid of some of the information in the image, it's better than aliasing. These are called **low-pass filters**.

5.3.1 Antialiasing

We can introduce low-pass filters to get rid of "unsafe" high-frequency information that we know our sampling algorithm can't capture, while keeping safe, low frequencies. We perform this filtering prior to sampling, and then again after reconstruction. Since we know

that certain frequencies simply did not exist, a reconstruction that results in these high frequencies are incorrect can be safely clipped off.

Let's formalize this idea. First, we define a comb function that can easily represent an impulse train as follows (where M is an integer):

$$\text{comb}_M[x] = \sum_{-\infty}^{\infty} \delta[x - kM]$$

This is an impulse train in which every M, x is a unit impulse. Remember that due to the scaling property, the Fourier transform of the comb function is $\frac{1}{2}\text{comb}_{1/2}(u)$. We can extend this to 2D and define a **bed of nails**, which is just a comb in two directions, which also tightens its Fourier transform if it spreads:

$$\text{comb}_{M,N}(x,y) = \sum_{k=-\infty}^{\infty}\sum_{l=-\infty}^{\infty} \delta(x - kM, y - lN) \iff \frac{1}{MN}\sum_{k=-\infty}^{\infty}\sum_{l=-\infty}^{\infty} \delta\left(u - \frac{k}{M}, v - \frac{l}{N}\right)$$

With this construct in mind, we can multiply a signal by a comb function to get discrete samples of the signal; the M parameter varies the fidelity of the samples. Now, if we consider the Fourier transform of a signal and its resulting convolution with the comb function *after* we do our sampling in the spatial spectrum, we can essentially imagine a repeating FT every $1/M$ steps. *If there is* ***no overlap*** within this repeat, we don't get any distortion in the frequency spectrum.

Specifically, if $W < \frac{1}{2M}$, where W is the highest frequency in the signal, we can recover the original signal from the samples. This is why CDs sample at 44 kHz, so that we can recover everything up to 22 kHz, which is the maximal extent of human hearing. If there *is* overlap (which would be the presence of high-frequency content in the signal), it causes aliasing when recovering the signal from the samples: the high-frequency content is masquerading as low-frequency content.

We know how to get rid of high frequencies: use a Gaussian filter! By applying a Gaussian, which now acts as an **anti-aliasing filter**, we get rid of any overlap. Thus, given a signal $f(x)$, we do something like:

$$(f(x) \circledast h(x)) \cdot \text{comb}_M(x)$$

Resizing Images

This anti-aliasing principle is very useful when resizing images. What do you do when you want to make an image half (or a quarter, or an eighth) of its original size? You could just throw out every other pixel, which is called **image subsampling**. This doesn't give very good results: it loses the high-frequency content of the original image because we sample too infrequently.

Instead, we need to do use an antialiasing filter as we've discussed. In other words, first *filter* the image, *then* do subsampling. The stark difference in quality is shown in Figure 5.4.

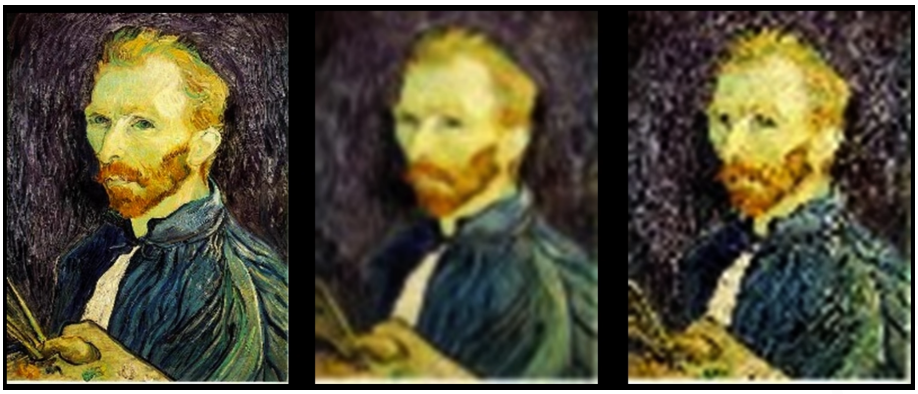


Figure 5.4: The result of down-sizing the original image of Van Gogh (left) using subsampling (right) and filtering followed by subsampling (center). The down-sized image was then blown back up to its original size (by zooming) for comparison.

Image Compression

The discoveries of John Robson and Frederick Campbell of the variation in sensitivity to certain contrasts and frequencies in the human eye[4] can be applied to image compression. The idea is that certain frequencies in images can be represented more coarsely than others. The JPEG image format uses the **discrete cosine transform** to form a basis set. This basis is then applied to 8×8 blocks of the image: and each block's frequency corresponds to a *quantization table* that uses a different number of bits to approximate that frequency with a lower fidelity.

[4] See this page for details; where do *you* stop differentiating the frequencies?

Blending

> *Who in the rainbow can draw the line where the violet tint ends and the orange tint begins? Distinctly we see the difference of the colors, but where exactly does the one first blendingly enter into the other? So with sanity and insanity.*
>
> — Herman Melville, *Billy Budd, Sailor*

Blending two images can result in some beautiful effects. Armed with our (shallow) understanding of the frequency domain, we can leverage it to seamlessly merge parts of independent images together. An example of what we'll learn to create includes the famous "applange":

Our running example will use two leopard print patterns, and our end-goal will be to smoothly blend them right in the middle:

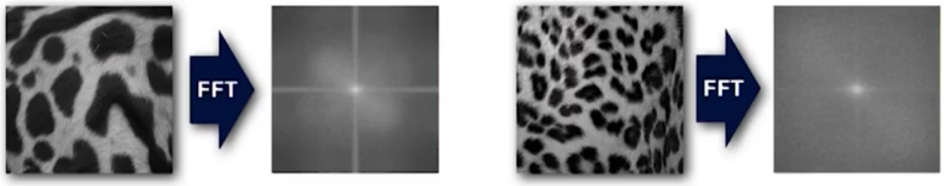
6.1 Simple Approaches: Crossfading and Feathering

Suppose we decide to, at first, do a weighted sum of the images, with the weights being a simple linear gradient from 1 to 0 as we move across the image.

This will result in a crossfaded blend, and we can get better results my lowering the window size (in red below) of what we crossfade:

How do we (automatically) decide on the best window size for a crossfaded blend? We want to avoid any seam artifacts. Ideally, we choose a window size that matches the size of the largest prominent "feature" in the image. Furthermore, we don't want it to be too large; namely, $\leq 2\times$ the size of the *smallest* prominent "feature" to avoid ghosting artifacts (like the semi-transparent patches in the middle and left crossfades, above).

In our discussion of the Fourier transform, we discovered how to break down an image into its component frequencies. A well-represented frequency, then, can be thought of as a prominent feature. Much like how JPEG images drop frequencies that are unimportant to the human eye via the discrete cosine transform, we can use the important frequences to determine good blending windows.



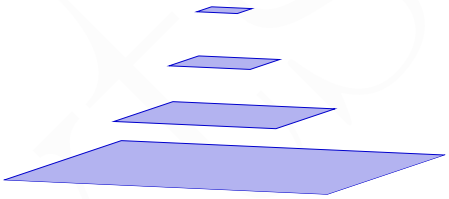
Instead of merging the images, we instead **feather** their component frequencies. In general, the idea of feathering is to blur at the edges just by feathering the hard edges of the masks, we can get a smoother result, around the same as the crossfade with a small window:

We want to extend this principle and feather at every "octave" of our image.

6.2 Image Pyramids

To visualize an "octave" and get a better understanding of this per-octave blending process, we'll first need to introduce the concept of **image pyramids**; this concept is crucial to blending and will be helpful to know when we discuss both Feature Recognition and Motion in later chapters.[1]

The idea of an image pyramid is quite simple: each "level" in the pyramid is half of the width and height of the previous image.



We'll focus on a constructing a specific type of image pyramid: a **Gaussian pyramid**. To halve our image, we will simply pass over it with a Gaussian filter and halve it. This is exactly what we did as part of resizing Van Gogh in Figure 5.4 when we discussed image subsampling; blurring then image *then* subsampling looks far better than subsampling directly. In our case, we just reduce our images *exactly* in half.

However, halving our images is not enough. Ideally, we also want to be able to *reconstruct* a lower pyramid (that is, a larger image) from a higher one. This is the process of **expansion**. Burt & Adelson cleverly crafted what they called a "5-tap filter" that allows us to halve (**reduce**) and double (**expand**) images with a single kernel. A picture is worth a thousand words: this filter is visually demonstrated in Figure 6.1.

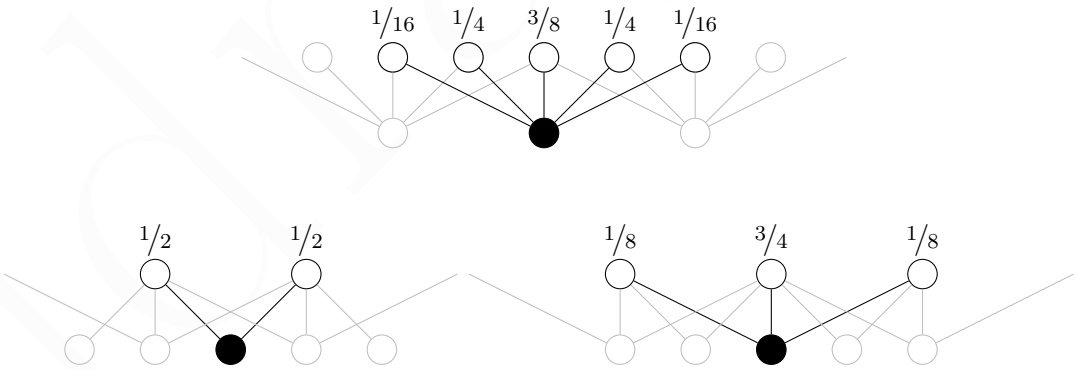


Figure 6.1: The 5-tap filter applied as part of reduction (top) and as part of expansion on even (left) and odd (right) pixels.

Reduction is straightforward as we've already discussed: subsamples combine a weighted blend of their respective 5 "source" pixels.

Expanding an image is a different beast: we must interpret the colors of pixels that lie

[1] Pyramid blending is based on this paper by Burt & Adelson.

"between" ones whose values we know. For "even" pixels in the upscaled image, the values we know are directly on the right and left of the pixel in the smaller image. Thus, we blend them equally. For odd pixels, we have a directly corresponding value, but for a smoother upscale, we also give some weight to its neighbors.[2]

Notice that the expansion scalars in Figure 6.1 can be combined into a single filter since the corresponding pixels will have no value (i.e. multiplication by zero) in the correct places for odd and even pixels. For example, applying the combined filter $\left\{\frac{1}{8}, \frac{1}{2}, \frac{3}{4}, \frac{1}{2}, \frac{1}{8}\right\}$ on an even pixel would multiply the odd filter indices by nothing.

We can now create a pyramid of Gaussians—each level 1/2 the size of the last—for our image as well as reconstruct it back up, albeit with some information lost. What information, specifically? Well recall that the Gaussian is effectively a low-pass filter. That means that at each level in the pyramid, we remove high-frequency information!

In fact, by finding the differences (i.e. subtracting) between a level in a Gaussian pyramid and the expanded version of the higher level, we can actually see the high-frequency information:

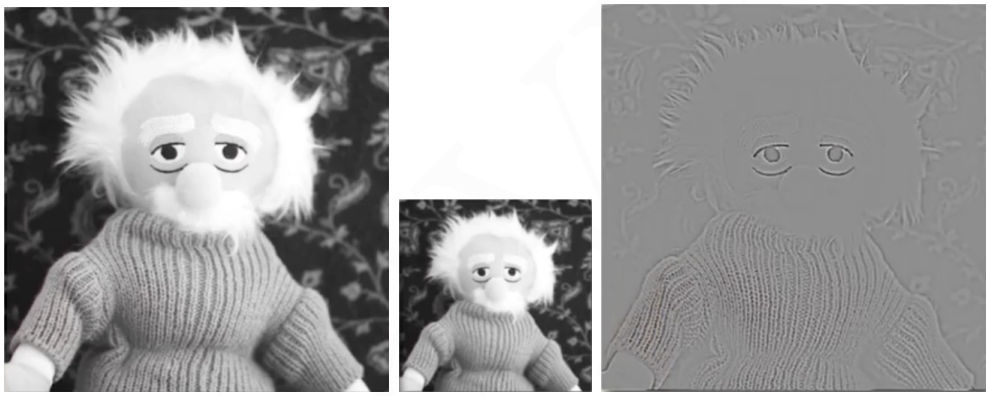


Figure 6.2: The difference between an original and expanded image. On the left is the original base of the Gaussian pyramid, while in the middle is its 1st level—that is, half of the original. If we expand this level back up (i.e. doubling it via the 5-tap filter above) and subtract it from the original, we get the difference image on the right. As you can see, the high-frequency information (e.g. the fine definition on the knitted sweater) is what differentiates them.

The image on the right in Figure 6.2 is called a **Laplacian** image (because it's a good approximation of the 2nd derivative) but more accurately it's simply a **difference of Gaussians**. We can likewise construct a **Laplacian pyramid** alongside its Gaussian pyramid as a representation of high-frequency information lost at each reduction level.

We will use the notation I_0 to refer to the original base image of a pyramid, I_1 as being its next (2x smaller) level, and so on. Similarly, we'll index from the top-down by saying I_{-1} is the *last* (highest, smallest) level in the pyramid.

[2] Whether or not an upscaled pixel has a corresponding value in the parent image depends on how you do the upscaling. If you say that column 1 in the new image is column 1 in the old image, the rules apply (so column 3 comes from column 2, etc.). If you say column *2* in the new image is column 1 in the old image (so column 4 comes from column 2, etc.) then the "odd" and "even" rules are reversed.

Mathematically, then, we can imagine each level of the Laplacian pyramid as being calculated from its corresponding Gaussian and the level above it:

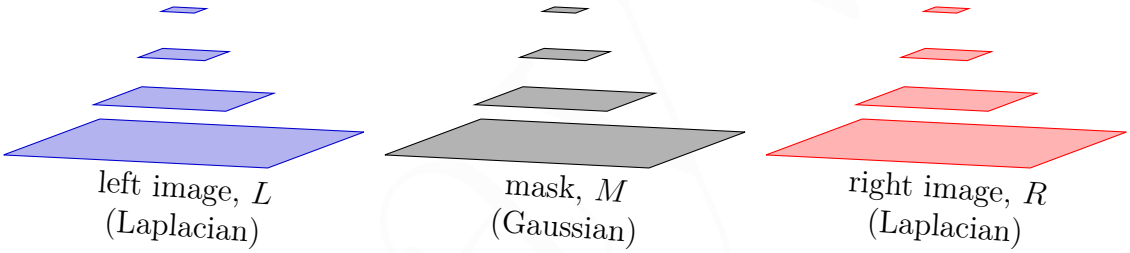
$$L_i = G_i - \text{EXPAND}(G_{i+1})$$

Typically, the last level in a Laplacian pyramid is just the equivalent level in the Gaussian, since G_{i+1} no longer exists. That is, $L_{-1} = G_{-1}$.

6.3 Pyramid Blending

Armed with these two pyramids (Gaussian and Laplacian) for each image in a pair, we can finally implement **pyramid blending**. The Gaussian pyramid is just a means to generating the Laplacian pyramid; that's where the magic happens. Of course, we still need to define a mask or otherwise represent a window in which the blending will occur. This mask image also gets its own Gaussian pyramid.

Going into the blending process, then, we have the following:



At each level, we use the mask to determine which pixels to blend between the images. If R is our **r**esulting image at a *particular pyramid level*, then our blending equation is a simple pixel-wise calculation. At any pixel (x, y), we have:

$$R[x, y] = M[x, y] \cdot L[x, y] + (1 - M[x, y]) \cdot R[x, y]$$

We perform this blend at each pyramid level independently; remember, this gives us a set of blended *Laplacians*. We then need to "collapse" the pyramid to find the ultimate result, F. We do this by building up from the blended Laplacians:

$$\begin{aligned} U &= R_{-2} + \text{EXPAND}(R_{-1}) && \text{base case, then} \\ &= R_{-3} + \text{EXPAND}(U) && \text{iterate on self} \\ &\vdots \\ &= R_0 + \text{EXPAND}(U) \end{aligned}$$

Our final result U is a well-blended version of L and R along M that feathers a blend along each frequency octave!

6.4 Poisson Blending

Pyramid blending involves blending the pixels directly. However, if the images come from drastically different domains (featuring different lighting, colors, etc.), even a perfectly hand-crafted mask will not result in a natural blend. We'll introduce a technique that results in blends that appear natural and far more convincing than the best pyramid blend:

Figure 6.3: A demonstration of how Poisson blending (right) can look more natural in a scene over a naïve cut-and-paste (left) and pyramid blending with a rough mask (middle).

Notice how the tint of the plane in Figure 6.3 is darker to match its surroundings; even with a perfect pyramid blend, the color of the plane body would look unnaturally lit.

> This bonus section is based on the paper that originated the idea, this blog post which provides an intuitive understanding of the algorithm, this Wikipedia article that outlines the discrete Poisson equation, and this work by Evan Wallace that provides great example imagery.
>
> These are all great resources to enhance your understanding of Poisson blending after reading this section.

6.4.1 Flashback: Recovering Functions

First, a brief (and potentially pain-inducing) flashback to calculus. Suppose we have the simple function $f(x) = 4x^2 + 9$. We can take its derivative easily by applying the power rule: $\dot{f}(x) = 2 \cdot (4x^{2-1}) + 0 = 8x$.

Now suppose we only knew $\dot{f}(x) = 8x$. Can we recover $f(x)$? Easy, right? Just integrate:

$$\int 8x = 4x^2 + c$$

Dang, we lost the coefficient! c could be anything... But if we know some **initial condition**,

like that $f(0) = 9$, we can fully recover:

$$\begin{aligned} f(x) &= 4x^2 + c, \quad f(0) = 9 \\ 9 &= 4(0)^2 + c \implies c = 9 \\ \boxed{f(x) &= 4x^2 + 9} \end{aligned}$$

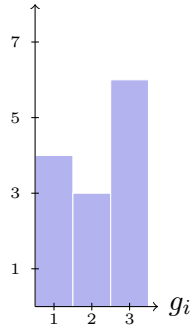
What does this mean? It means that given a derivative (or, in our 2D case, an image gradient) and some initial conditions, we can recover the original function! This is the basic insight behind **Poisson blending**, which blends images in the *gradient* domain.

We start with a *source* image that is our "unnatural" addition and a *target* image into which we will insert it. Typically, we only want to insert part of our source image, so we also have a *binary mask* (that is, only composed of 1 – blend and 0 – don't blend). At a (very) high level, Poisson blending is simple: we insert the masked gradient of the source image into the gradient of the destination image and integrate.

6.4.2 Recovering Complicated Functions

Unfortunately, reality is often disappointing. Above, we recovered a single function with a single initial condition. In an image, we are recovering a bunch of pixels simultaneously with a number of boundary conditions, and the values of all of the pixels depend on each other. Let's work through an example to demonstrate this complication.

For simplicity's sake, let's assume 1D gradients are calculated by a simple neighbor subtraction. That is: $g_i = p_i - p_{i-1}$. Suppose we're given the simple 1D gradient on the right and want to recover the original intensities.



We're also given the essential boundary condition: $p_0 = 5$. We know, then that:

$$\begin{aligned} g_1 &= p_1 - p_0 \\ p_1 &= p_0 + g_1 \\ p_1 &= 5 + 4 = 9 \quad \blacksquare \end{aligned}$$

Now that we know p_1, we can solve for p_2: $g_2 = p_2 - p_1$, and likewise for p_3. Awesome, right? This gives us our **ground truth** of $\{p_0 = 5, p_1 = 9, p_2 = 12, p_3 = 18\}$.[3]

Unfortunately this solving technique does not scale well. The discovery of p_i depends on p_{i-1},—we need $O(n)$ iterations to recover the original. Can we do better? Let's craft a

[3] The pixel intensities always grow because the gradient is always positive. This was done for mathematical (and graphical) convenience; there's nothing inherently limiting this to positive values.

clever way to solve all of these at once:

$$\begin{cases} p_1 = g_1 + p_0 \\ p_2 = g_2 + p_1 \\ p_3 = g_3 + p_2 \end{cases}$$

Suppose we reform this as an optimization problem: we want to minimize the error (difference) between some "guess" for p_1 (call it x_1) and the truth (which we know to be $g_1 + p_0$). Namely, we want:

$$\begin{aligned} x_1 \text{ such that} \min \left[(x_1 - p_0 - g_1)^2\right] \\ x_2 \text{ such that} \min \left[(x_2 - x_1 - g_2)^2\right] \\ x_3 \text{ such that} \min \left[(x_3 - x_2 - g_3)^2\right] \end{aligned}$$

Notice that each x_i depends on the choice for the previous x_{i-1}, so all of these need to be solved at once. To do so, first we craft such an optimization criteria for every pair of points with the goal of minimizing the *total* error in the system:

$$\varepsilon(x_{1..3}) = \min_{x_1,x_2,x_3} \left[\sum_{i=1}^{3} (\underbrace{x_i - x_{i-1}}_{\text{guess at } g_i} - g_i)^2 \right] \text{ where } x_0 = p_0 = 5$$

As we may (or may not) remember from calculus, the minimum value of a function is the one at which the derivative = 0. Thus, we can take the partial derivative of $\varepsilon(\cdot)$ with respect to each variable. For example, w.r.t. x_2:

$$\begin{aligned} 0 &= \frac{\partial \varepsilon}{\partial x_2} \\ 0 &= \frac{\partial}{\partial x_2} \left((x_1 - p_0 - g_1)^2 + (x_2 - x_1 - g_2)^2 + (x_3 - x_2 - g_3)^2 \right) \\ &= \frac{\partial}{\partial x_2} \left((x_2 - x_1 - g_2)^2 + (x_3 - x_2 - g_3)^2 \right) && \text{we know } {}^{\partial}/_{\partial x_2} \text{ will zero out any term w/o } x_2 \\ &= \frac{\partial}{\partial x_2} \left(x_2^2 - 2x_2x_1 - 2x_2g_2 + x_1^2 + 2x_1g_2 + g_2^2 + x_3^2 - 2x_3x_2 - 2x_3g_3 + x_2^2 + 2x_2g_3 + g_3^2 \right) \\ &= \frac{\partial}{\partial x_2} \left(2x_2^2 - 2x_2x_1 - 2x_2g_2 - 2x_3x_2 + 2x_2g_3 \right) && \text{again, most terms will zero out} \\ &= 4x_2 - 2x_3 - 2x_1 - 2g_2 + 2g_3 \\ &= 2x_2 - x_3 - x_1 - g_2 + g_3 && \text{remove common factor} \\ g_2 - g_3 &= 2x_2 - x_3 - x_1 && \text{move knowns to one side} \end{aligned}$$

Notice that we're working with the partial derivative whose inputs are already the gradients; thus this is actually the 2^{nd} derivative (recall that this the Laplacian for images, $\nabla^2 \boldsymbol{I}$).

Anyway, we do this for every $\partial/\partial x_i$ and get the system:

$$\begin{cases} 2x_1 - x_2 & = -g_2 + g_1 + p_0 \\ -x_1 + 2x_2 - x_3 & = -g_3 + g_2 \\ -x_2 + x_3 x & = g_3 \end{cases} \quad \text{where} \quad \begin{cases} p_0 = 5 \\ g_1 = 4 \\ g_2 = 3 \\ g_3 = 6 \end{cases}$$

Plugging in our known g_is and the boundary condition creates a standard $\mathbf{Ax} = \mathbf{b}$ system:

$$\begin{bmatrix} 2 & -1 & 0 \\ -1 & 2 & -1 \\ 0 & -1 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 6 \\ -3 \\ 6 \end{bmatrix} \tag{6.1}$$

Solving this gives us the same pixel values as before: $\{x_1 = 9, x_2 = 12, x_3 = 18\}$! It's **critical** to note that this solution depends *entirely* on our boundary condition, $p_0 = 5$. If we change this to $p_0 = 2$, our solver returns something completely different: $\{x_1 = 6, x_2 = 9, x_3 = 15\}$ **but** our pixel values maintain the same *relative difference* in intensities! In the context of an image, we keep the exact same features but essentially tint them by the boundary pixels.

If we chose some arbitrary set of *different* gradients, we could get the same approximation as above and maintain the same image, except altered to fit the boundary condition. To come full circle back to blending, this means we are blending the source image into the target, but altering it based on the influence of the edge pixels (i.e. boundary conditions) in the *target* image.

6.4.3 Extending to 2D

Things get a little more complex when extending the ideas, equations, and matrices we just discussed to the mathematics of 2 dimensions. However, the main idea is exactly the same: build a matrix that constrains each pixel intensity with its neighbors and associate it with some gradients and pixel intensities.

To form the matrix, we do some simple manipulation. You may have noticed a small pattern in the matrix in (6.1): there's a 2 along the diagonal (except in the first cell) and a -1 around every 2. This is no coincidence. If we'd had 9 pixels, we'd have a matrix like:

$$\begin{bmatrix} 2 & -1 & 0 & \dots & 0 & 0 \\ -1 & 2 & -1 & 0 & \dots & 0 \\ 0 & -1 & 2 & -1 & 0 & \dots \\ 0 & -1 & -1 & 2 & 0 & \dots \\ \vdots & \ddots & \ddots & \ddots & \ddots & \ddots \\ \vdots & \vdots & 0 & -1 & 2 & -1 \\ 0 & 0 & \dots & 0 & -1 & 1 \end{bmatrix}$$

The area around a 2 effectively represents its neighbors. The 2 at $(3, 3)$ in the above matrix (conveniently in green) corresponds to the 3rd pixel in the source image. The four red -1s around it are the neighbors: a 0 at any of these would mean that relative pixel is an edge.

In 2 dimensions, this matrix looks a tiny bit different: instead, there's a **4** along the diagonal (which comes from the kernel for the Laplacian operator). It's just as simple to craft, however.

Our $\mathbf{b}$ vector is easy to form. In two dimensions, we first convolve our source image with the Laplacian operator to easily find the initial values for $\mathbf{b}$. Then, we incorporate boundary conditions (i.e. pixels from the target image that fall on the edge of the masked source image) where appropriate.

We've essentially crafted a linear system representing the **discrete Poisson equation**, a partial differential equation that generalizes the Laplace equation $\nabla^2 \boldsymbol{I} = \mathbf{0}$.

6.4.4 *Je suis une poisson.*

Finally, the absurd amount of math and concepts we just covered is formalized into the ultimate answer to "will it blend?"—algorithm 6.1.

ALGORITHM 6.1: The Poisson blending algorithm.

Input: The source image, $\mathbf{S}_{h\times w}$ matrix
Input: The target image, $\mathbf{T}$
Input: The mask on the source image, $\mathbf{M}_{m\times n}$
Result: The blended result image, $\mathbf{B}$

$\mathbf{L} := \nabla^2 S$
$\mathbf{B} := \text{COPY}(T)$
`/* The pixel values to find and the Poisson coefficient matrix */`
$\mathbf{b}_{mn\times 1} := \mathbf{0}$
$\mathbf{P}_{mn\times mn} := \mathbf{0}$
`/* For convenience, we first craft an array of indices for each pixel.`
`Since each pixel gets its element in the diagonal, we'll just craft it`
`based on the pixel number. */`
$\mathbf{I}_{h\times w} := \mathbf{0}$
$i := 0$
foreach $(x, y) \in \mathbf{S}$ *where* $M[x, y] = 1$ **do** `// every pixel in` S `to blend`
  $\mathbf{I}[x, y] := i$
  $i \mathrel{+}= 1$
end
foreach $(x, y) \in \mathbf{S}$ *where* $M[x, y] = 1$ **do** `// every pixel in` S `to blend`
  $j := \mathbf{I}[x, y]$
  $\mathbf{b}[j] := \mathbf{L}[x, y]$
  foreach *neighbor pixel* $(n_x, n_y) \in \mathbf{S}[x, y]$ **do** `// 4-connected`
    `/* If the neighbor is to be blended, set the coefficient. */`
    if $M[n_x, n_y] = 1$ **then**
      $j_n := \mathbf{I}[n_x, n_y]$
      $\mathbf{P}[j, j_n] := -1$
    `/* Otherwise, integrate the boundary condition (target's pixel). */`
    else
      $\mathbf{b}[j] \mathrel{+}= T[n_x, n_y]$
    end
  end
  $\mathbf{P}[j, j] = 4$
end
$\mathbf{x}_{mn\times 1} := \text{SOLVE}(\mathbf{P}, \mathbf{b})$
`/* Put each solved pixel intensity into the target. */`
$\mathbf{B}[\mathbf{M}] = \mathbf{x}$
return B

Cameras and Images

> *"What's in a name? That which we call a rose*
> *By any other name would smell as sweet."*
>
> — Shakespeare, *Romeo and Juliet [Act II, Scene ii]*

We've been looking at images as 2-dimensional arrays of intensities, and dedicated an entire chapter to treating Images as Functions. But what is an image more realistically? What *is* a photograph that you take (and edit, and filter, and caption) and post on Instagram?

Close one eye, make a square with your fingers, and hold it up. What you see through the square is what your camera captures, and what your screen displays. As in Figure 7.1, this is a called a **projection**. An image, or photograph, is a 2-dimensional projection of a set of 3-dimensional points.

When we go from 3D to 2D, we lose all of the information from the 3rd dimension. We've all seen this in the real world. Fantastic artists create optical illusions that fool us into thinking they have real depth, like in Figure 7.2. Our goal in this chapter is going to be extrapolating this information back.

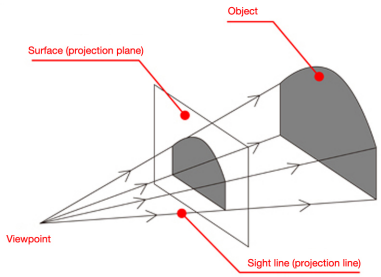


Figure 7.1: A diagram outlining a perspective projection of a 3-dimensional object onto a 2-dimensional surface.

Figure 7.2: An example of anamorphic art and Pavement Art Illusion that, when viewed from the right angle, appears to have depth.

7.1 Cameras

To understand how to computationally work with every element in the photography pipeline, it's essential to understand how a camera works. We'll open with a discussion of camera hardware and its geometry, then abstract away the hardware and work exclusively in the mathematical domain.

> This section comes from the *Computational Photography* lectures, which I feel do a better job explaining camera hardware. Also I skipped this section during CV, so that might also influence things...

When we see objects in the real world, we are typically observing the amount of light they are *reflecting* from a particular light source (like the sun). Figure 7.3 demonstrates how we can control this reflection to capture a photograph.

First, we imagine a "sensor" as just being a large surface that captures all of an object's reflected light. Of course, this captures nothing specific, because all points on the object reflect light in all directions. To solve this, we add a layer between the sensor and the object that has a small hole in the middle to only allow light rays that head in a very specific direction to hit the "sensor." This typically renders a blurry image, since each point on the object is rendered a handful of times on the sensor. In the third and final model, the hole in the layer (known as a **pinhole**, and is generally referred to as an **aperture**) is adjusted so that every light ray hits the sensor exactly once. This renders an in-focus representation of the object we're capturing on the sensor.

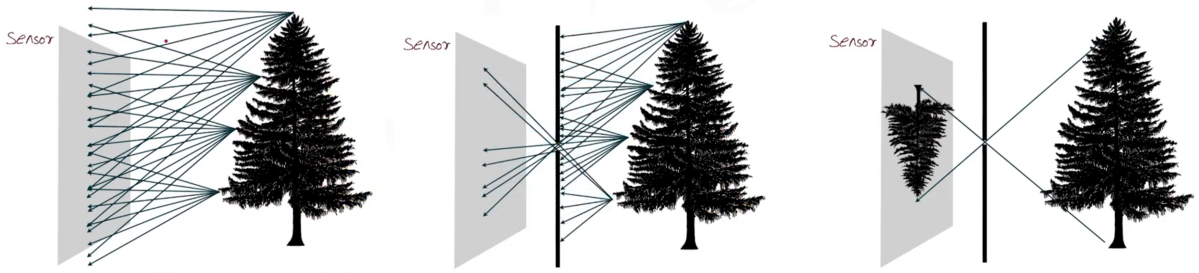


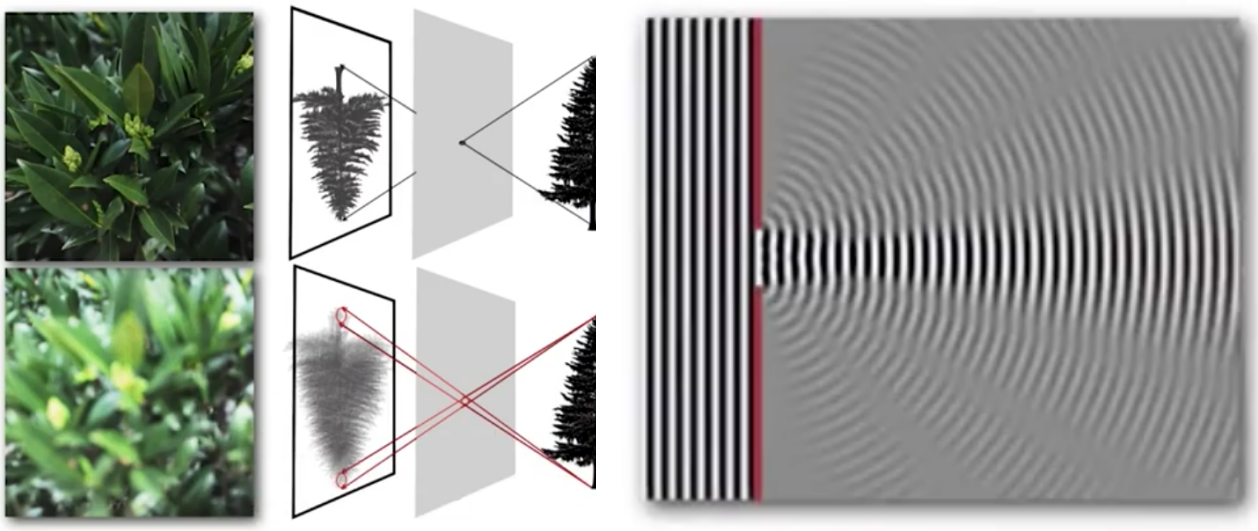
Figure 7.3: The theory behind the pinhole camera module. On the left, no image is captured because all points on the object reflect light in all directions. In the middle, far less rays pass hit the sensor due to a pinhole in a surface separating the sensor and the object. Finally, on the right, the pinhole is configured such that one (or a small number) of focused rays hit the sensor, capturing our image.

7.1.1 Blur

Note that with a larger aperture, more light from each point hits the sensor. This causes a blurring effect. Controlling aperture vs. light quantity is a delicate balance. If your aperture is too low, it is possible that not enough light will hit the sensor (resulting in a dark image);

however, if your aperture is too high, it is possible that light from too many different rays will hit the sensor (resulting in a blurry image).

Furthermore, light behaves strangely when passing through small holes due to the fact that it's a wave. When a light ray hits a tiny hole directly, it will exit the hole in a spherical pattern.



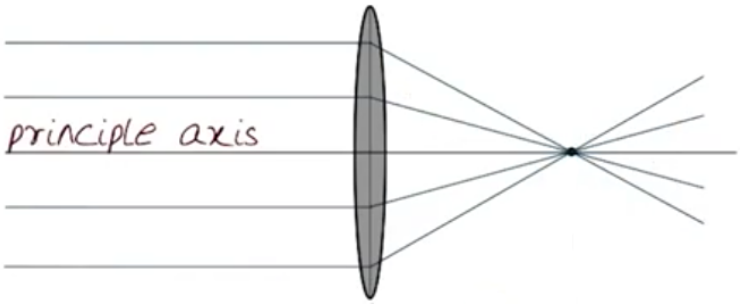
(a) A demonstration of how an increasing aperture causes geometric blur.

(b) How the wave nature of light causes diffraction through an aperture.

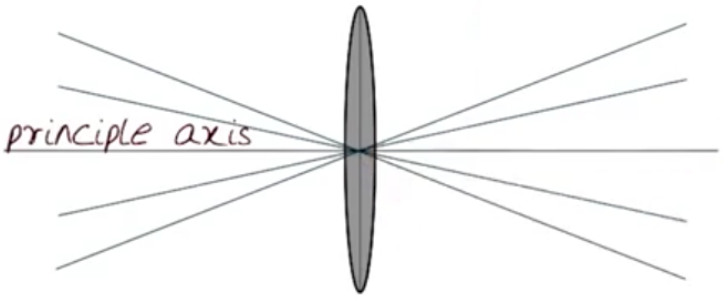
Figure 7.4: Geometry (left) and physics (right) can both cause blur in an image.

7.1.2 Lenses

To avoid making this tradeoff in a pinhole camera model, we introduce lenses. This adds some complexity but isn't unbearable. A lens focuses all of the rays of light that pass through it perpendicularly to a single focal point:

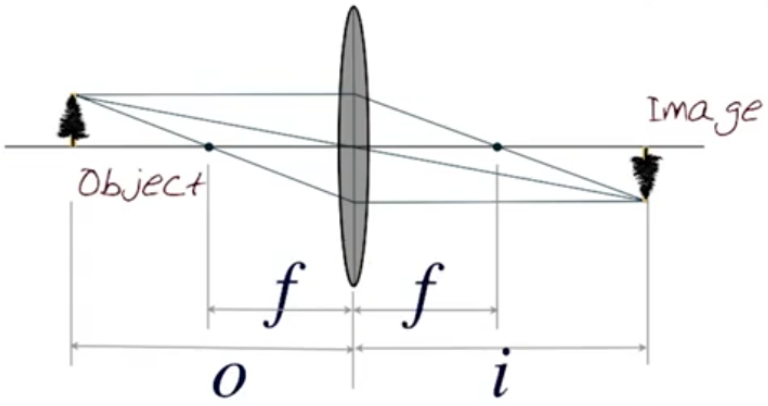


The distance from the lens to the focal point is the **focal length**, f. Rays that pass through the center of a lens are unaffected:



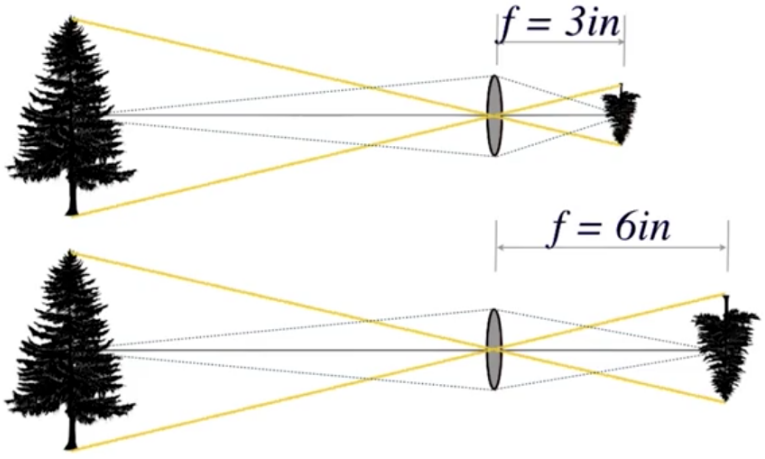
Focal Length

For a given object distance, it will be fully "in focus" at a certain distance past the lens where three rays converge: the ray through the center, the ray that *hits* perpendicular to the lens, and the ray that *comes out* perpendicular to the lens:

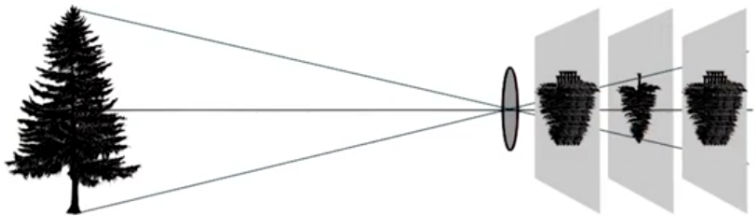


This results in a **lens equation**: $\frac{1}{o} + \frac{1}{i} = \frac{1}{f}$.

Changing the object distance results in a similar change in the scale at which that object is captured (as expected). Changing the focal length, however, changes the optimal point at which we should capture the image:



We typically actually put our sensor *at* the focal plane. If we put it too far from this point, we will get blur. Thus the idea of "focusing" an object in an image is a matter of moving the sensor forward or backwards with respect to the lens:



Field of View

From our basic diagram of a lens-based camera, we can easily calculate the **field of view** that it captures. We will say our sensor height is h, and given this information, calculating the FoV is a matter of simple geometry. We can see from Figure 7.5 on the right that $\theta = 2\tan^{-1}\left(\frac{h}{2f}\right)$.

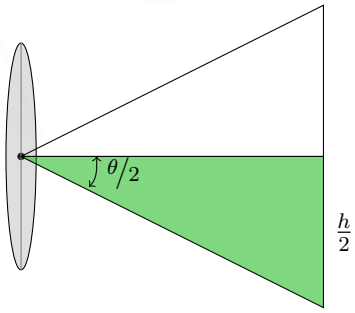
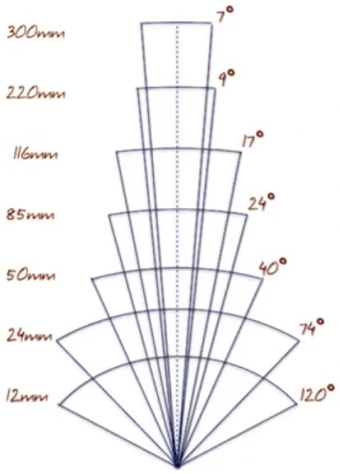


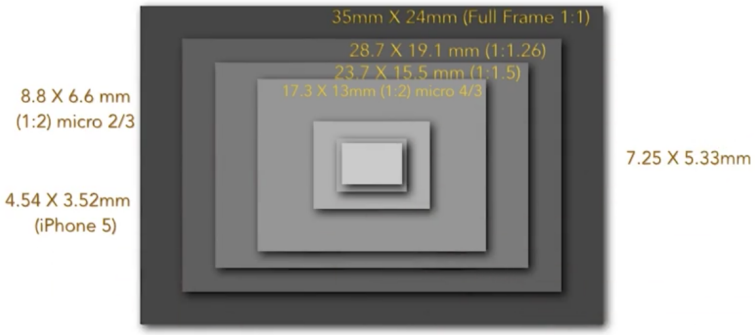
Figure 7.5: The geometry of a lens.

Note that the relationship between focal length and field of view is inversely proportional:



7.1.3 Sensor Sizes

Modern cameras come with a variety of sensor sizes. On the largest end of the spectrum, we have the "full frame" sensor common in traditional film which has a 1 to 1 aspect ratio, and on the smallest end, we have your typical smartphone camera sensor.



7.2 Perspective Imaging

With an understanding of cameras under our belt, we can move on to understanding the notion of "perspective" in a 2D image. To start off, take a look at Figure 7.1: we start with a point in the 3D "real world" at some coordinate, (x, y, z), and this gets *projected* to the **center of projection**, passing through a **projection plane** of some size (intersecting at some location $(x, y, -d)$) that is d units away from the center of projection, centered on the z-axis.

We make the center of projection – which we'll refer to simply as the *camera* – the origin in our coordinate system, and the projection plane (or *image plane*) in front of the camera so that we can treat the image plane as a simple xy-plane. This means the camera looks down the *negative* z-axis, and the image plane has $(0, 0)$ in the *center*, not the top-left corner like we're used to.

We can model our projection scheme by using the similar triangles in Figure 7.6. For some point (x, y, z):

$$(x, y, z) \to \left(-d\frac{x}{z}, -d\frac{y}{z}, -d\right)$$

This models the point of intersection on the projection plane with an arbitrary ray of light (like the blue ray in Figure 7.6). This means that $(x', y') = (-d\frac{X}{Z}, -d\frac{Y}{Z})$.

7.2.1 Homogeneous Coordinates

Unfortunately, the operation we use to convert from 3D space to our 2D projection – division by an ever-changing z to get our new (x', y') – is not a linear transformation. Fortunately, we can use a trick: add one more coordinate (sometimes called the *scaling coordinate*) to make it a linear operation.

Our 2D **homogeneous image coordinates** create a simple mapping from "pixel space" to image space, and a similar one for 3D:

$$(x, y) \Rightarrow \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \qquad (x, y, z) \Rightarrow \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

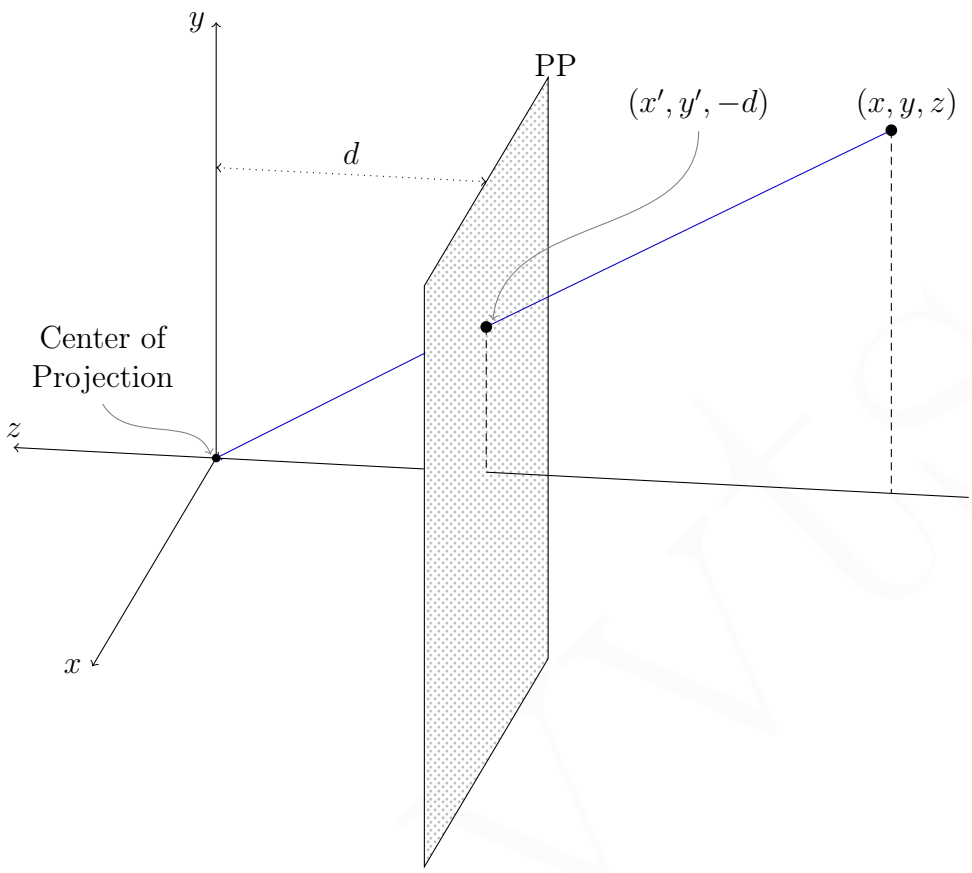


Figure 7.6: A projection coordinate system. PP is the projection (or "image") plane (which is what we see), and the center of projection is the location of the "camera."

To convert *from* some homogeneous coordinate $\begin{bmatrix} x \\ y \\ w \end{bmatrix}$, we use $(x/w, y/w)$, and similarly for 3D, we use $(x/w, y/w, z/w)$. It's interesting to note that homogeneous coordinates are **invariant** under scaling: if you scale the homogeneous coordinate by some a, the coordinate in pixel space will be unaffected because of the division by aw.

Perspective Projection

With the power of homogeneous coordinates, the projection of a point in 3D to a 2D **perspective projection** plane is a simple matrix multiplication in homogeneous coordinates:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1/f & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x \\ y \\ z/f \end{bmatrix} \Rightarrow \left(f\frac{x}{z}, f\frac{y}{z} \right) \Longrightarrow (u, v) \tag{7.1}$$

This multiplication ***is*** a linear transformation! We can do all of our math under homogeneous coordinates until we actually need to treat them as a pixel in an image, which is when we

perform our conversion. Also, here f is the **focal length**, which is the distance from the center of projection to the projection plane (d in Figure 7.6).

How does scaling the projection matrix change the transformation? It doesn't! Recall the invariance property briefly noted previously:

$$\begin{bmatrix} a & 0 & 0 & 0 \\ 0 & a & 0 & 0 \\ 0 & 0 & a/f & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} ax \\ ay \\ az/f \end{bmatrix} \Rightarrow \left(f\frac{x}{z}, f\frac{y}{z} \right)$$

7.2.2 Geometry in Perspective

As we'd hope, points, lines, and polygons in 3-dimensional space correspond to points, lines, and polygons on our projection plane. They don't preserve the same properties, though. For example, parallel lines intersect at what's called the **vanishing point**. We can see this in the mathematics.

A line in 3D can be characterized as parametric vector of equations in t:

$$\mathbf{r}(t) = \begin{cases} x(t) = x_0 + at \\ y(t) = y_0 + bt \\ z(t) = z_0 + ct \end{cases}$$

Let's apply perspective projection to the line:

$$x'(t) = \frac{fx}{z} = \frac{f(x_0 + at)}{z_0 + ct} \qquad y'(t) = \frac{fy}{z} = \frac{f(y_0 + at)}{z_0 + ct}$$

In the limit, as $t \to \pm\infty$, we see (for $c \neq 0$):

$$x'(t) = \frac{fa}{c} \qquad y'(t) = \frac{fb}{c}$$

Notice that the "start of the line," (x_0, y_0, z_0) has disappeared entirely! This means that no matter where a line starts, it will always converge at the vanishing point $(fa/c, fb/c)$. The restriction that $c \neq 0$ means that this property of parallel lines applies to all parallel lines *except* those that exist in the xy-plane, that is, parallel to our projection plane!

All parallel lines in the same *plane* converge at **colinear** vanishing points, which we know as the **horizon**.

Human vision is strongly affected by the notion of parallel lines. See the lecture snippet. **TODO**: Add the illusion and explain why it happens.

7.2.3 Other Projection Models

There are models aside from the perspective projection model that are useful and we'll be studying.

Orthographic Projection

This variant of projection is often used in computer graphics and video games in that are 2 dimensions. The model essentially "smashes" the real world against the projection plane. In 2D games, the assets (textures, sprites, and other images) are already flat and 2-dimensional, and so don't need any perspective applied to them.

Orthographic projection – also called *parallel projection* – can actually be thought of as a special case of perspective projection, with the center of projection (or camera) infinitely far away. Mathematically, $f \to \infty$, and the effect can be seen in Figure 7.7.

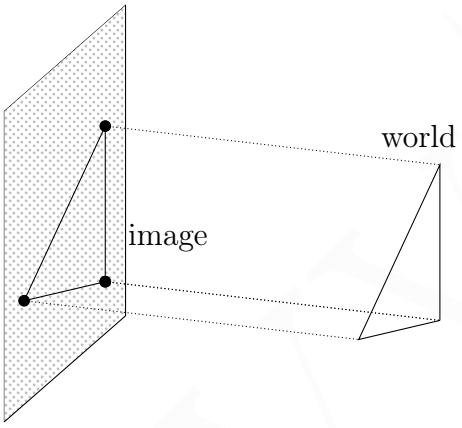


Figure 7.7: An orthographic projection model, in which the z coordinate is dropped entirely with no transformation.

Transforming points onto an orthographic projection just involves dropping the z coordinate. The projection matrix is simple:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \Rightarrow (x, y)$$

Weak Perspective

This perspective model provides a form of 3D perspective, but the scaling happens between *objects* rather than across every point. **Weak perspective** hinges on an important assumption: the change in z (or depth) within a single object is not significant relative to its distance from the camera. For example, a person might be about a foot thick, but they are standing a mile from the camera.

All of the points in an object that is z_0 distance away from the projection plane will be mapped as such:

$$(x, y, z) \to \left(\frac{fx}{z_0}, \frac{fy}{z_0} \right)$$

This z_0 changes from object to object, so each object has its own scale factor. Its projection

matrix is:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1/s \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x \\ y \\ 1/s \end{bmatrix} \Rightarrow (sx, sy)$$

7.3 Stereo Geometry

In general, the topic of this section will be the geometric relationships between the *camera* (2D) and the *scene* (3D). "Stereo" in the context of imaging is just having two views of the same scene; this is much like the human vision system, which uses your two eyeballs to get two slightly-differing views of the world around you. This construct enables us to perceive depth, and we'll be working with that shortly as well.

Structure and depth are inherently ambiguous from single views. There are all kinds of optical illusions like the one in Figure 7.8 that take advantage of this ambiguity. Mathematically, why is this the case? If we refer back to Figure 7.6, we can see that it's because any point along the blue ray (i.e. at any depth) will hit the same point on the projection plane.

Figure 7.8: The largest pumpkin ever grown.

Let's return to the human eye. We see the same scene from two slightly different angles, but this can also be interpreted as seeing the same scene that has *moved slightly*. In other words, we can get a similar understanding of depth from *motion* (which gives us two different views after some step) rather than seeing two angles of the same scene.

In fact, this concept was used to create stereoscopic glasses that, when viewing a special type of image that was two photographs of a scene at different angles, created the *feeling* of depth. A similar principle is applied in the old-school red blue 3D glasses.

Research has shown us that human stereo processing, or **binocular fusion**, is based on a low-level differentiation between changes across our two "cameras," rather than a large-scale recognition of objects and a correlation of those objects across views.

Our simple stereo camera system is shown in Figure 7.9. The cameras are separated by a baseline, B, and their focal length is f. We also have some point P at a distance Z in the camera coordinate system. We can also measure the distances x_l and x_r, which are the points of intersection with the left and right planes, respectively. Since we're working in a coordinate system in which $(0, 0)$ is the center of each camera, $x_l \geq 0$ and $x_r \leq 0$.

What is the expression for Z? To find out, we need to do some simple geometry with similar triangles. We have the first triangle (p_l, P, p_r) and the second triangle (C_L, P, C_R), both highlighted in red in Figure 7.10. This leads to the relationship:

$$\frac{B - x_l + x_r}{Z - f} = \frac{B}{Z}$$

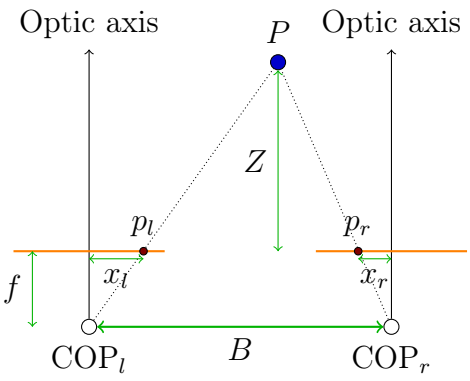


Figure 7.9: A top-down view of a simple stereo camera system, with distances marked for the projection of an arbitrary point P onto both cameras.

Rearranging that gives us:

$$Z = f\frac{B}{x_l - x_r}$$

which is an incredibly useful relationship. We are computing the distance to something in the scene based on the **disparity**, or difference between the two projections. In other words, disparity is inversely proportional to depth.

What if disparity $= 0$? That means a point doesn't change at all in the left or right image. That would be more and more true for objects further away, which is exactly what we saw with Orthographic Projection as our center of projection got infinitely far away.

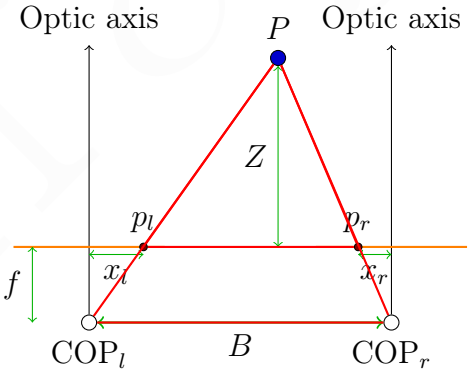


Figure 7.10: The similar triangles (in red) that we use to relate the points projected onto both cameras.

7.3.1 Finding Disparity

Knowing that depth is correlated with disparity, how can we find the disparity between two images? Given a point in an image, we know that a similar point in the other image has to be somewhere within some constaints.

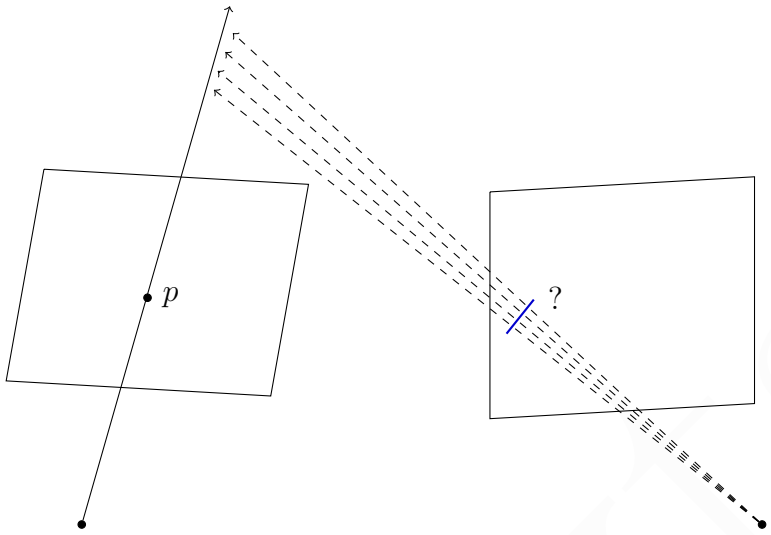


Figure 7.11: The line of potential points (in **blue**) in the right image that could map to the projected point in the left image.

In a project that I did in my undergrad,[1] we used the squared Euclidean distance to find the pixel in one image that was most similar to the pixel in another using a "feature patch":

$$d_E^2 = \sum_{x=0}^{W} \sum_{y=0}^{H} \left(A(x,y) - B(x,y)\right)^2$$

A smaller distance meant a more similar feature patch. It worked pretty well, and we generated some awesome depth maps that we then used to create stereograms. Since we're big smort graduate students now, things are going to get a little more complicated. First, we need to take a foray into epipolar geometry.

7.3.2 Epipolar Geometry

Let's consider the general case in which we do *not* have parallel optical axes as in Figure 7.9. Given a point p in the left image, where can the corresponding point p' be in the right image?

Do you remember the initial problem we had with projection and depth that made optical illusions like Figure 7.8 possible? It was that any point along a ray between the camera its projection plane maps to the same point in the plane. This means that, similarly, mapping that ray to the other camera will create a line of possible points. This notion is one of the principles derived in **epipolar geometry**. Hopefully, Figure 7.11 illustrates this better than my explanation does.

Thus, we have a line of possible values in the right projection plane called the epipolar line. There is a plane that contains this line as well as the camera centers of both images, and that plane forms a corresponding epipolar line on the original (left) image. Any point that

[1] Link: CS61c, Project 1.

is on the epipolar line of one image must be on its corresponding epipolar line on the other image. This is called the **epipolar constraint**.

Let's define some terms used in epipolar geometry before we move forward:

- **baseline**: the line joining the camera centers.
- **epipolar plane**: the plane containing the baseline and some "world point."
- **epipolar line**: to reiterate, this is the intersection of the epipolar plane of a given point with the image planes.
- **epipole**: the point of intersection of the baseline with the image plane. Note that because every epipolar plane also contains the baseline and intersects the image plane, every epipolar line will contain the epipole.

The epipolar constraint reduces the "correspondence problem" to a 1D search across an epipolar line.

7.3.3 Stereo Correspondence

With some epipolar geometry under our belt, we can make progress on finding the disparity (or correspondence) between two stereo images. We're going to start with a lot of assumptions to simplify the scenario, but this allows us to immediately dive into solving the **stereo correspondence** problem. We assume:

- the images planes are parallel (or **co-planar**),
- the same focal length for each camera,
- the epipolar lines are horizontal, and
- the epipolar lines are at the same y location in the image

We'll discuss minimizing these assumptions later in chapter 8 when we introduce Projective Geometry. For now, though, even with the epipolar constraint we don't have enough information to solve the correspondence problem. We need some additional "soft contraints" that will help us identify corresponding points:

- **Similarity**: The corresponding pixel should have a similar intensity.
- **Uniqueness**: There is no more than one matching pixel.
- **Ordering**: Corresponding pixels must maintain their order: pixels ABC in the left image must likewise be matched to $A'B'C'$ in the right image.
- **Limited Disparity Gradient**: The depth of the image shouldn't change "too quickly."

We will focus primarily on the *similarity* soft constraint. For that, we need to expand our set of assumptions a bit to include that:

- *Most* scene points are visible from both views, though they may differ slightly based

on things like reflection angles.

- Image regions for two matching pixels are similar in appearance.

Dense Correspondence Search

Let's describe a simple algorithm for the **dense correspondence search** (dense here indicates that we will try to find matches for every pixel in the image). The process is simple; for each pixel (or window) in the left image:

- Compare with each pixel (or window) in the right image along the epipolar line.
- Choose the position with the minimum "match cost." This can be determined by a number of algorithms such as the sum of square differences (SSD) as we described above[2] or the normalized correlation we discussed in earlier chapters.

Using normalized correlation (as described in Filter Normalization), we can also account for global differences in the images such as one being brighter than the other. Using the SSD may give incorrect results because of such changes affecting the difference calculation.

We may run into issues in which our windows are too small to include a sufficient difference to find a correlation. Imaging a plain white hallway: our "match window" won't vary very much as we move across the epipolar line, making it difficult to find the correct corresponding window. We can increase the window size, of course, or, perhaps, we can simply do matching on high-texture windows only!

All in all, we've ended up with the same algorithm as the one I used in my undergrad (described in Finding Disparity), but now we understand *why* it works and have considered several modifications to increase robustness.

Uniqueness Constraint

Let's briefly touch some of the other soft constraints we described above. The uniqueness constraint states that there is no more than one match in the right image for every point in the left image.

No *more* than one? Yep. It can't be *exactly* one because of **occlusion**: the same scene from different angles will have certain items occluded from view because of closer objects. Certain pixels will only be visible from one side at **occlusion boundaries**.

Ordering Constraint

The ordering constraint specifies that pixels have to be in the same order across both of our stereo images. This is generally true when looking at solid surfaces, but doesn't always hold!

This happens in two cases. When looking at (semi-)*transparent* objects, different viewing angles give different orderings because we can "see through" the surface. This is a rare case,

[2] The Euclidean distance is the same thing as the sum of squared differences when applied to a matrix (also called the **Frobenius norm**).

but another one is much more common: narrow occluding surfaces. For example, consider a 3D scene with a skinny tree in the middle. In your left eye, you may see "left grass, tree, right grass," but in your right eye, you may instead see "tree, more grass" (as in, both the "left" and "right" grass are on the right side of the tree).

Instead of imagining such a contrived scene, you can easily do this experiment yourself. Hold your fingers in front of you, one behind the other. In one eye, the front finger will be on the left, whereas the back finger will be on the left in your other eye.

Unfortunately, the "state-of-the-art" algorithms these days aren't very good at managing violations of the ordering constraint such as the scenarios described here.

7.3.4 Better Stereo Correspondence

The window-matching approach is not very good as a general solution. Here we'll describe a few solutions that are closer to the state-of-the-art in depth detection.

A very basic improvement we can make is treat the image like, well, an image. Instead of assigning the pixels individually to their corresponding pixel, we can treat them as parts of a whole (which they are). In other words, we can optimize correspondence assignments *jointly*. This can be done at different granularities, such as finding disparity a scanline as a time, or even for an entire 2D grid.

Scanlines

For the scanline method, we essentially have a 1D signal from both images and we want the disparity that results in the best *overall* correspondence. This is implemented with a **dynamic programming** formulation. I won't go into detail on the general approach here; instead, I'll defer to this lecture which describes it much more eloquently. This approach results in far better depth maps compared to the naïve window matching method, but still results in streaking artifacts that are the result of the limitation of scanlines. Though scanlines improved upon treating each pixel independently, they are still limited in that they treat every scanline independently without taking into account the vertical direction. Enter 2D grid-based matching.

Grid Matching

We can define a "good" stereo correspondence as being one with a high *match quality*, meaning each pixel finds a good match in the other image, and a good *smoothness* meaning adjacent pixels should (usually) move the same amount. The latter property is similar to the "neighborliness" assumption we described in Computing Averages. Thus, in our search for a good correspondence, we might want to penalize solutions that have these big jumps.

This leads to modeling stereo correspondence as an **energy minimization** problem. Given two images, I_1 and I_2, their matching windows $W_1(i)$ and $W_2(i + D(i))$ (where we say that the second window is the first window plus some disparity $D(i)$), and the resulting disparity image D, we can create this energy minimization model.

We have the **data term**, E_{data}, which is the sum of the difference of squares between our two images given some estimated disparity:

$$E_{\text{data}} = \sum_i \left(W_1(i) - W_2(i + D(i))\right)^2 \tag{7.2}$$

We want to minimize this term, and this is basically all we were looking at when doing Dense Correspondence Search. Now, we also have a **smoothness term**:

$$E_{\text{smooth}} = \sum_{\text{neighbors } i,j} \rho\left(D(i) - D(j)\right) \tag{7.3}$$

Notice that we are only looking at the disparity image in the smoothness term. We are looking at the neighbors of every pixel and determining the size of the "jump" we described above. We define ρ as a **robust norm**: it's small for small amounts of change and it gets expensive for larger changes, but it shouldn't get more expensive for *even* larger changes, which would likely indicate a valid occlusion.

Both of these terms form the total energy, and for energy minimization, we want the D that results in the smallest possible E:

$$E = \alpha E_{\text{data}}(I_1, I_2, D) + \beta E_{\text{smooth}}(D)$$

where α and β are some weights for each energy term. This can be approximated via **graph cut**s,[3] and gives phenomenal results compared to the "ground truth" relative to all previous methods.[4]

7.3.5 Conclusion

Though the scanline and 2D grid algorithms for stereo correspondence have come a long way, there are still many challenges to overcome. Some of these include:

- occlusions
- low-contrast or textureless image regions
- violations of brightness constancy, such as specular reflections
- really large baselines, B (distance between the cameras)
- camera calibration errors (which result in incorrect epipolar lines)

[3] The graph cut algorithm is a well-known algorithm in computer science. It is an algorithm for dividing graphs into two parts in order to minimize a particular value. Using the algorithm for the energy minimization problem we've described was pioneered in this paper.

[4] You can check out the current state-of-the-art at Middlebury.

7.4 Extrinsic Camera Parameters

We began by discussing the properties of a camera and how it maps a 3D scene onto a 2D image via lenses. We extrapolated on this to discuss projection, which described a variety of different ways we can perform this mapping. Then, we discussed stereo geometry, which used the geometric properties of a camera to correlate two images and get an understanding of their depth. Now, we'll return to the properties of a camera and discuss them further.

This time, though, we'll be talking about *extrinisic* camera parameters. These are parameters external to the camera; they can be manipulated by the person holding (or controlling) it. In contrast with intrinsic parameters such as focal length, these include things such as rotating the camera, or simply changing its location with respect to the world.

Recall, first, the perspective projection model we described in Figure 7.6. We introduced Homogeneous Coordinates to handle the non-linearity problem of dividing by Z and turn perspective projection into a simple matrix multiplication. An important assumption of our model was the coordinate system: the center of projection was at the origin, and the camera looked down the z-axis. This model means that the entire world literally revolves around our camera; everything is relative to the center of projection.

This is a limiting model to work with. It'd be nicer if we could relate the coordinate system of the world *to* the coordinate system of the camera; in other words, the camera just becomes another object in the world. This is known as **geometric camera calibration** and is composed of two transformations:

- First, we transform from some (arbitrary) world coordinate system to the camera's 3D coordinate space. These are the **extrinsic parameters**, or the camera pose.
- Then, from the 3D coordinates in the camera's frame to the 2D image plane via projection. This transformation depends on the camera's **intrinsic parameters**, such as the focal length.

We can model the transformation from "world space" to "camera space" as $\mathbf{T}_w^c$. How many degrees of freedom do we have in this transformation? Consider for a moment a cube in space: how many ways can you move it? Well, obviously we can move it along any of the axes in the xyz-plane, but we can also *rotate* it along any of these axes: we can turn it left to right (**yaw**), we can lean it forward and back (**pitch**), and we can twist it back and forth (**roll**). This gives six total degrees of freedom.

Notation Before we dive in, we need to define some notation to indicate which "coordinate frame" our vectors are in. We will say that ${}^A\mathbf{P}$ is the coordinates of $\mathbf{P}$ in frame A. Recall, as well, that a vector can be expressed as the scaled sum of the unit vectors $(\mathbf{i}, \mathbf{j}, \mathbf{k})$. In other words, given some origin ${}^A\mathbf{O}$,

$$ {}^A\mathbf{P} = \begin{bmatrix} {}^Ax \\ {}^Ay \\ {}^Az \end{bmatrix} \iff \overrightarrow{OP} = \left({}^Ax \cdot \mathbf{i}_A\right)\left({}^Ay \cdot \mathbf{j}_A\right)\left({}^Az \cdot \mathbf{k}_A\right) $$

7.4.1 Translation

With that notation in mind, what if we want to find the location of some vector $\mathbf{P}$, whose position we know in coordinate frame A, within some other coordinate frame B. This described in Figure 7.12.

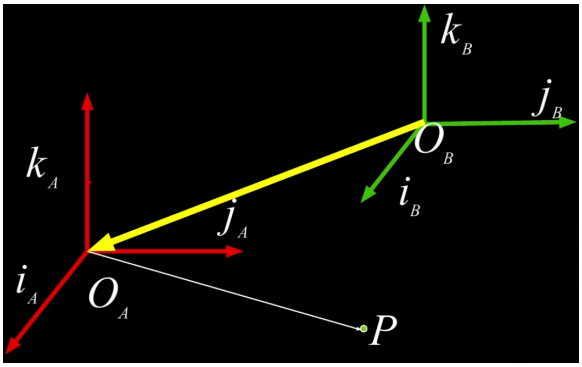


Figure 7.12: Finding P in the coordinate frame B, given ${}^{A}\mathbf{P}$.

This can be expressed as a straightforward translation: it's the sum of $\mathbf{P}$ in our frame and the origin of the other coordinate frame, O_B, expressed in our coordinate frame:

$$ {}^{B}\mathbf{P} = {}^{A}\mathbf{P} + {}^{A}(O_B) $$

The origin of frame B in within frame A is just a simple offset vector. With that in mind, we can model this translation as a matrix multiplication using homogeneous coordinates:

$$ {}^{B}\mathbf{P} = {}^{A}\mathbf{P} + {}^{A}O_B $$

$$ \begin{bmatrix} {}^{B}P_x \\ {}^{B}P_y \\ {}^{B}P_z \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & {}^{A}O_{x,B} \\ 0 & 1 & 0 & {}^{A}O_{y,B} \\ 0 & 0 & 1 & {}^{A}O_{z,B} \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} {}^{A}P_x \\ {}^{A}P_y \\ {}^{A}P_z \\ 1 \end{bmatrix} $$

We can *greatly* simplify this by substituting in vectors for column elements. Specifically, we can use the 3×3 identity matrix, $\mathbf{I}_3$, and the 3-element zero vector as a row, $\mathbf{0}^{\mathrm{T}}$:

$$ \begin{bmatrix} {}^{B}\mathbf{P} \\ 1 \end{bmatrix} = \begin{bmatrix} \mathbf{I}_3 & {}^{A}\mathbf{O_B} \\ \mathbf{0}^{\mathrm{T}} & 1 \end{bmatrix} \begin{bmatrix} {}^{A}\mathbf{P} \\ 1 \end{bmatrix} $$

7.4.2 Rotation

Now things get even uglier. Suppose we have two coordinate frames that share an origin, but they are differentiated by a rotation. We can express this succinctly, from frame A to B, as:

$$ {}^{B}\mathbf{P} = {}^{B}_{A}\mathbf{R}\,{}^{P}\mathbf{A} $$

Where ${}^{B}_{A}\mathbf{R}$ expresses points in frame A in the coordinate system of frame B. Now, what does $\mathbf{R}$ look like? The complicated version is an expression of the basis vectors of frame A in frame B:

$$
\begin{aligned}
{}^{B}_{A}\mathbf{R} &= \begin{bmatrix} \mathbf{i}_A \cdot \mathbf{i}_B & \mathbf{j}_A \cdot \mathbf{i}_B & \mathbf{k}_A \cdot \mathbf{i}_B \\ \mathbf{i}_A \cdot \mathbf{j}_B & \mathbf{j}_A \cdot \mathbf{j}_B & \mathbf{k}_A \cdot \mathbf{j}_B \\ \mathbf{i}_A \cdot \mathbf{k}_B & \mathbf{j}_A \cdot \mathbf{k}_B & \mathbf{k}_A \cdot \mathbf{k}_B \end{bmatrix} && (7.4) \\
&= \begin{bmatrix} {}^{\mathbf{B}}\mathbf{i}_{\mathbf{A}} & {}^{\mathbf{B}}\mathbf{j}_{\mathbf{A}} & {}^{\mathbf{B}}\mathbf{k}_{\mathbf{A}} \end{bmatrix} && (7.5) \\
&= \begin{bmatrix} {}^{\mathbf{A}}\mathbf{i}_{\mathbf{B}}{}^{\mathrm{T}} \\ {}^{\mathbf{A}}\mathbf{j}_{\mathbf{B}}{}^{\mathrm{T}} \\ {}^{\mathbf{A}}\mathbf{k}_{\mathbf{B}}{}^{\mathrm{T}} \end{bmatrix} && (7.6)
\end{aligned}
$$

Each of the components of the point in frame A can be expressed somehow in frame B using all of B's basis vectors. We can also imagine that it's each basis vector in frame B expressed in frame A. (7.6) is an **orthogonal matrix**: all of the columns are unit vectors that are perpendicular.

Example: Rotation about a single axis.

Suppose we just have the xy-plane, and are performing a rotation about the z axis. This scenario, shown in Figure 7.13, is reminiscent of middle school algebra.

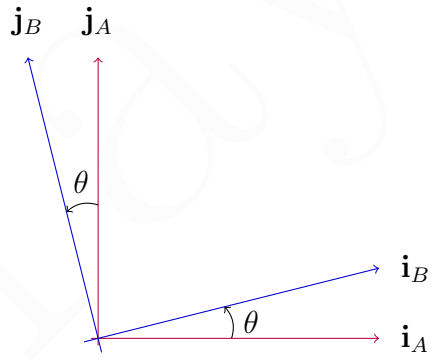


Figure 7.13: A rotation about the z axis from frame A to B by some angle θ.

The rotation matrix for this transformation is:

$$
\mathbf{R}_z(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{7.7}
$$

Of course, there are many ways to combine this rotation in a plane to reach some arbitrary rotation. The most common one is using **Euler angles**, which say: first rotate about the

"world" z, then about the new x, then about the new z (this is also called heading, pitch, and roll). There are other ways to express arbitrary rotations – such as the yaw, pitch, and roll we described earlier – but regardless of which one you use, you have to be careful. The order in which you apply the rotations matters and negative angles matter; these can cause all sorts of complications and incorrect results. The rotation matrices about the three axes are (we exclude z since it's above):

$$\mathbf{R}_y(\kappa) = \begin{bmatrix} \cos\kappa & 0 & -\sin\kappa \\ 0 & 1 & 0 \\ \sin\kappa & 0 & \cos\kappa \end{bmatrix} \tag{7.8}$$

$$\mathbf{R}_x(\varphi) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\varphi & -\sin\varphi \\ 0 & \sin\varphi & \cos\varphi \end{bmatrix} \tag{7.9}$$

Rotation with Homogeneous Coordinates

Thankfully, things are *much* simpler in homogeneous coordinates. It can be expressed as a matrix multiplication, but is **not** commutative, unlike translation. We can say:

$$\begin{bmatrix} {}^B\mathbf{P} \\ 1 \end{bmatrix} = \begin{bmatrix} {}^B_A\mathbf{R} & 0 \\ \mathbf{0}^\mathrm{T} & 1 \end{bmatrix} \begin{bmatrix} {}^A\mathbf{P} \\ 1 \end{bmatrix} \tag{7.10}$$

7.4.3 Total Rigid Transformation

With translation and rotation, we can formulate a **total rigid transformation** between two frames A and B:

$${}^B\mathbf{P} = {}^B_A\mathbf{R}\,{}^B\mathbf{P} + {}^B\mathbf{O_A}$$

First we rotate into the B frame, then we add the origin offset. Using homogeneous coordinates, though, we can do this all in one step:[5]

$$\begin{aligned} \begin{bmatrix} {}^B\mathbf{P} \\ 1 \end{bmatrix} &= \begin{bmatrix} 1 & {}^B\mathbf{O_A} \\ \mathbf{0}^\mathrm{T} & 1 \end{bmatrix} \begin{bmatrix} {}^B_A\mathbf{R} & 0 \\ \mathbf{0}^\mathrm{T} & 1 \end{bmatrix} \begin{bmatrix} {}^A\mathbf{P} \\ 1 \end{bmatrix} \\ &= \begin{bmatrix} {}^B_A\mathbf{R} & {}^B\mathbf{O_A} \\ \mathbf{0}^\mathrm{T} & 1 \end{bmatrix} \begin{bmatrix} {}^A\mathbf{P} \\ 1 \end{bmatrix} \end{aligned}$$

Even more simply, we can just say: $\begin{bmatrix} {}^B\mathbf{P} \\ 1 \end{bmatrix} = \begin{bmatrix} {}^B_A\mathbf{R} & {}^B\mathbf{O_A} \\ \mathbf{0}^\mathrm{T} & 1 \end{bmatrix} \begin{bmatrix} {}^A\mathbf{P} \\ 1 \end{bmatrix} = {}^B_A\mathbf{T} \begin{bmatrix} {}^A\mathbf{P} \\ 1 \end{bmatrix}$

Then, to instead get from frame B to A, we just invert the transformation matrix!

$$\begin{bmatrix} {}^A\mathbf{P} \\ 1 \end{bmatrix} = {}^A_B\mathbf{T} \begin{bmatrix} {}^B\mathbf{P} \\ 1 \end{bmatrix} = \left({}^B_A\mathbf{T}\right)^{-1} \begin{bmatrix} {}^B\mathbf{P} \\ 1 \end{bmatrix}$$

[5] Notice that we reversed the order of the matrix multiplications. From *right to left*, we translate then rotate.

To put this back into perspective, our homogeneous transformation matrix being **invertible** means that we can use the same matrix for going from "world space" to "camera space" to the other way around: it just needs to be inverted.

7.4.4 The Duality of Space

We now have an understanding of how to transform between frames. We can use this for our model of the camera and the world around it. Given a camera frame and world frame, we can use our homogeneous transformation matrix to transform some point $\mathbf{p}$:

$$\begin{bmatrix} {}^{C}\mathbf{p} \\ 1 \end{bmatrix} = \begin{bmatrix} {}^{C}_{W}\mathbf{R} & {}^{C}_{W}\mathbf{t} \\ \mathbf{0}^{\mathrm{T}} & 1 \end{bmatrix} \begin{bmatrix} {}^{W}\mathbf{p} \\ 1 \end{bmatrix}$$

The 4×4 matrix that makes up the first "row" we've shown here (as in, everything aside from the bottom $\begin{bmatrix} 0 & 0 & 0 & 1 \end{bmatrix}$ row) is called the **extrinsic parameter matrix**. The bottom row is what makes the matrix invertible, so it won't always be used unless we need that property; sometimes, in projection, we'll just use the 3×4.

7.4.5 Conclusion

We've derived a way to transform points in the world to points in the camera's coordinate space. That's only half of the battle, though. We also need to convert a point in camera space into a point on an image. We touched on this when we talked about Perspective Imaging, but we're going to need a little more detail than that.

Once we understand these two concepts, we can combine them (as we'll see, "combine" means another beautifully-simple matrix multiplication) and be able to see exactly how any arbitrary point in the world maps onto an image.

7.5 Intrinsic Camera Parameters

Hold the phone, didn't we already *do* intrinsic camera parameters? Well, yes. Our projection matrix from (7.1) encoded the intrinsic properties of our camera, but unfortunately it represented an *ideal* projection. It's used in video games, sure, but everything there is perfectly configured by the developer(s). If we want to apply this to the real world, things aren't so straightforward.

7.5.1 *Real* Intrinsic Parameters

Our notion of "pixels" is convenient and necessary when working with discrete images, but they don't translate so well to the real world. Our camera has a focal length in some real-world units, say, 10mm, that we need to somehow scale to pixels.

In other words, our perfect projection already has another set of factors (2, since pixels might not be square!). Also, we assumed the center of the image aligned with the center

of projection (along the z-axis); of course, our image may have been cropped or otherwise modified. Thus, we need to introduce some offset (u_0, v_0). *Furthermore*, what if there was some skew between the camera axes and they didn't sample perfectly perpendicularly? We'd need to account for this, as well:

$$\begin{aligned} v' \sin(\theta) &= v \\ u' &= u - \cos(\theta) v' \\ &= u - \cot(\theta) v \end{aligned}$$

Combining all of these potential parameters, we get 5 degrees of freedom in this mess:

$$\begin{aligned} u &= \alpha \frac{x}{z} - \alpha \cot(\theta) \frac{y}{z} + u_0 \\ v &= \frac{\beta}{\sin(\theta)} \frac{y}{z} + v_0 \end{aligned}$$

. . . gags.

Can we make this any nicer? Notice the division by z all over the place? We can, again, leverage homogeneous coordinates. Let's shove the whole thing into a matrix:

$$\begin{bmatrix} zu \\ zv \\ z \end{bmatrix} = \begin{bmatrix} \alpha & \alpha \cot(\theta) & u_0 & 0 \\ 0 & \frac{\beta}{\sin(\theta)} & v_0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

More succinctly, we can isolate the **intrinsic parameter matrix** that transforms from a point in camera space to a homogeneous coordinate:

$$\mathbf{p}' = \mathbf{K}\, {}^{c}\mathbf{p}$$

We can get rid of the last column of zeroes, as well as use friendlier parameters. We can say that f is the focal length, as before; s is the skew; a is the aspect ratio; and (c_x, c_y) is the camera offset. This gives us:

$$\mathbf{K} = \begin{bmatrix} f & s & c_x \\ 0 & af & c_y \\ 0 & 0 & 1 \end{bmatrix}$$

Of course, if we assume a perfect universe, f becomes our only degree of freedom and gives us our perspective projection equation from (7.1).

7.6 Total Camera Calibration

We can combine our extrinsic and intrinsic camera parameters to get a direct transformation from an arbitrary 3D coordinate in world space to a 2D pixel on an image. Let's reiterate our

two equations; we begin with our coordinate in world space, ${}^{W}\mathbf{p}$, and get out a homogeneous pixel coordinate, $\mathbf{p}'$.

The intrinsic camera parameters are represented by: $\begin{bmatrix} {}^{C}\mathbf{p} \\ 1 \end{bmatrix} = \begin{bmatrix} {}^{C}_{W}\mathbf{R} & {}^{C}_{W}\mathbf{t} \\ \mathbf{0}^{\mathrm{T}} & 1 \end{bmatrix} \begin{bmatrix} {}^{W}\mathbf{p} \\ 1 \end{bmatrix}$

The extrinsic camera parameters are represented by: $\mathbf{p}' = \mathbf{K}\,{}^{C}\mathbf{p}$

Combining these gives us our **calibration matrix**, $\mathbf{M}$:

$$\mathbf{p}' = \mathbf{K}\left({}^{C}_{W}\mathbf{R}\,{}^{C}_{W}\mathbf{t}\right){}^{W}\mathbf{p}$$
$$\mathbf{p}' = \mathbf{M}\,{}^{W}\mathbf{p}$$

We can also say that the true pixel coordinates are **projectively similar** to their homogeneous counterparts (here, s is our $s$caling value):

$$\begin{bmatrix} u \\ v \\ 1 \end{bmatrix} \simeq \begin{bmatrix} su \\ sv \\ s \end{bmatrix}$$

In summary, we define our full camera calibration equation as being:

$$\mathbf{M}_{3\times4} = \underbrace{\begin{bmatrix} f & s & x'_c \\ 0 & af & y'_c \\ 0 & 0 & 1 \end{bmatrix}}_{\text{intrinsics}} \underbrace{\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}}_{\text{projection}} \underbrace{\begin{bmatrix} \mathbf{R}_{3\times3} & \mathbf{0}_{3\times1} \\ \mathbf{0}_{1\times3} & 1 \end{bmatrix}}_{\text{rotation}} \underbrace{\begin{bmatrix} \mathbf{I}_3 & \mathbf{T}_{3\times1} \\ \mathbf{0}_{1\times3} & 1 \end{bmatrix}}_{\text{translation}} \tag{7.11}$$
$$\underbrace{\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}_{\text{extrinsics}}$$

This equation gives us **11** degrees of freedom! So... what was the point of all of this again? Well, now we'll look into how to *recover* $\mathbf{M}$ given some world coordinates and some pixel in an image. This allows us to reconstruct the entire set of parameters that created that image in the first place!

7.7 Calibrating Cameras

At the heart of what we'll be accomplishing in this section is finding the $\mathbf{M}$ described in (7.11). In general, we have, for some point $\mathbf{x}$ in homogeneous coordinates, a transformation from world to image using $\mathbf{M}$:

$$\mathbf{x}' = \begin{bmatrix} * & * & * & * \\ * & * & * & * \\ * & * & * & * \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \mathbf{M}\mathbf{x}$$

One of the ways this can be accomplished is through placing a known object into the scene; this object will have a set of well-describable points on it. If we know the correspondence between those points in "world space," we can compute a mapping from those points to "image space" and hence derive the camera calibration that created that transformation.

Another method that is mathematically similar is called **resectioning**. Instead of a particular object, we can also determine a series of known 3D points in the entire scene. This results in a scene akin to what's described in Figure 7.14.

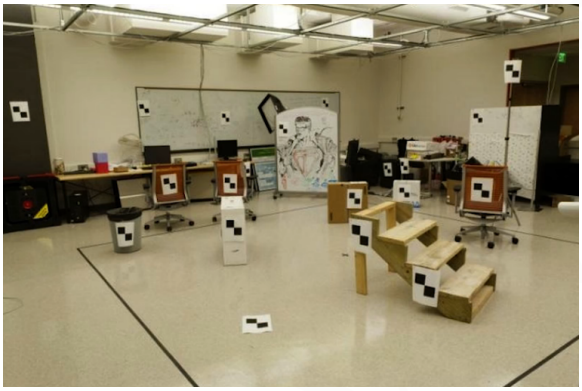


Figure 7.14: A lab outfitted with special markers indicating "known points" for resectioning.

After measuring each point in the real world from some arbitrary origin, we can then correlate those points to an image of the same scene taken with a camera. What does this look like mathematically? Well, for a single known point i, we end up with a pair of equations.

$$\begin{bmatrix} u_i \\ v_i \\ 1 \end{bmatrix} \simeq \begin{bmatrix} wu_i \\ wv_i \\ w \end{bmatrix} = \begin{bmatrix} m_{00} & m_{01} & m_{02} & m_{03} \\ m_{10} & m_{11} & m_{12} & m_{13} \\ m_{20} & m_{21} & m_{22} & m_{23} \end{bmatrix} \begin{bmatrix} X_i \\ Y_i \\ Z_i \\ 1 \end{bmatrix} \tag{7.12}$$

$$u_i = \frac{m_{00}X_i + m_{01}Y_i + m_{02}Z_i + m_{03}}{m_{20}X_i + m_{21}Y_i + m_{22}Z_i + m_{23}} \tag{7.13}$$

$$v_i = \frac{m_{10}X_i + m_{11}Y_i + m_{12}Z_i + m_{13}}{m_{20}X_i + m_{21}Y_i + m_{22}Z_i + m_{23}} \tag{7.14}$$

We can rearrange this, then convert that back to a(n ugly) matrix multiplication:

$$u_i(m_{20}X_i + m_{21}Y_i + m_{22}Z_i + m_{23}) = m_{00}X_i + m_{01}Y_i + m_{02}Z_i + m_{03} \tag{7.15}$$

$$v_i(m_{20}X_i + m_{21}Y_i + m_{22}Z_i + m_{23}) = m_{10}X_i + m_{11}Y_i + m_{12}Z_i + m_{13} \tag{7.16}$$

$$\begin{bmatrix} X_i & Y_i & Z_i & 1 & 0 & 0 & 0 & 0 & -u_iX_i & -u_iY_i & -u_iZ_i & -u_i \\ 0 & 0 & 0 & 0 & X_i & Y_i & Z_i & 1 & -v_iX_i & -v_iY_i & -v_iZ_i & -v_i \end{bmatrix} \begin{bmatrix} m_{00} \\ m_{01} \\ m_{02} \\ m_{03} \\ m_{10} \\ m_{11} \\ m_{12} \\ m_{13} \\ m_{20} \\ m_{21} \\ m_{22} \\ m_{23} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \tag{7.17}$$

This is a set of **homogeneous equations** because they're equal to 0. An obvious solution would be $\mathbf{m} = \mathbf{0}$, but we don't want that: this is a constraint on our solution.

You may recall from linear algebra that we can try to find the **least squares** solution to find an approximation to a system of equations in this form: $\mathbf{Ax} = \mathbf{b}$.[6] This amounts to minimizing $\|\mathbf{Ax}\|$, which is an approximate solution if there isn't one.

We know that $\mathbf{m}$ is only valid up to a particular scale. Consider again our original equation in (7.12): if we scale all of $\mathbf{M}$ by some scale factor, it will go away after division by w. Remember, homogeneous coordinates are invariant under scaling (see Homogeneous Coordinates). Instead, we'll solve for $\mathbf{m}$ as a unit vector, $\hat{\mathbf{m}}$.

Then, the question becomes: which possible unit vector $\hat{\mathbf{m}}$ will minimize $\|\mathbf{A}\hat{\mathbf{m}}\|$? The solution, which we'll show next, is the **eigenvector** of $\mathbf{A}^{\mathrm{T}}\mathbf{A}$ with the smallest **eigenvalue**. We can solve this with 6 or more known points, which correlates to our available degrees of freedom in our calibration matrix $\mathbf{M}$ (see 7.11).

7.7.1 Method 1: Singular Value Decomposition

To reiterate, we want to find the $\mathbf{m}$ such that it minimizes $\|\mathbf{Am}\|$, constrained by $\|\mathbf{m}\| = 1$ (in other words, $\hat{\mathbf{m}}$ is a unit vector).

To derive and describe the process of determining this $\hat{\mathbf{m}}$, we'll be looking at **singular value decomposition**. Any matrix can be decomposed (or "factored") into three matrices, where $\mathbf{D}$ is a **diagonal** matrix and $\mathbf{U}, \mathbf{V}$ are **orthogonal**:[7]

$$\mathbf{A} = \mathbf{UDV}^{\mathrm{T}}$$

[6] Refer to **??** for an explanation of this method.

[7] I won't discuss SVD in detail here, though I may add a section to **??** later; for now, please refer to your favorite textbook on Linear Algebra for details.

Therefore, we're aiming to minimize $\|\mathbf{U}\mathbf{D}\mathbf{V}^T\mathbf{m}\|$. Well, there's a trick here! $\mathbf{U}$ is a matrix made up of unit vectors; multiplying by their magnitude doesn't change the result. Thus,

$$\|\mathbf{U}\mathbf{D}\mathbf{V}^T\mathbf{m}\| = \|\mathbf{D}\mathbf{V}^T\mathbf{m}\| \quad \text{and,}$$
$$\|\mathbf{m}\| = \|\mathbf{V}^T\mathbf{m}\|$$

The latter equivalence is also due to the orthogonality of the matrix. We can conceptually think of $\mathbf{V}$ as a rotation matrix, and rotating a vector doesn't change its magnitude.

Now we're trying to minimize $\|\mathbf{D}\mathbf{V}^T\mathbf{m}\|$ subject to $\|\mathbf{V}^T\mathbf{m}\| = 1$, which is a stricter set of constraints than before. This cool transformation lets us do a substitution: let $\hat{\mathbf{y}} = \mathbf{V}^T\mathbf{m}$. Thus, we're minimizing $\|\mathbf{D}\hat{\mathbf{y}}\|$ (subject to $\|\hat{\mathbf{y}}\| = 1$).

Now, remember that $\mathbf{D}$ is a diagonal matrix, and, by convention, we can sort the diagonal such that it has decreasing values. Thus, $\|\mathbf{D}\hat{\mathbf{y}}\|$ is a minimum when $\hat{\mathbf{y}} = \begin{bmatrix}0 & 0 & \dots & 1\end{bmatrix}^T$; $\hat{\mathbf{y}}$ is putting "all of its weight" in its last element, resulting in the smallest value in $\mathbf{D}$.

Since $\hat{\mathbf{y}} = \mathbf{V}^T\mathbf{m}$, and $\mathbf{V}$ is orthogonal,[8] we know that $\mathbf{m} = \mathbf{V}\hat{\mathbf{y}}$. And since $\hat{\mathbf{y}}$'s only meaningful element is a 1 at the end, then $\mathbf{m}$ is the last column of $\mathbf{V}$!

Thus, if $\hat{\mathbf{m}} = \mathbf{V}\hat{\mathbf{y}}$, then $\hat{\mathbf{m}}$ selects the eigenvector of $\mathbf{A}^T\mathbf{A}$ with the smallest eigenvalue (we say $\hat{\mathbf{m}}$ now because eigenvectors are unit vectors). That leap in logic is explained by the properties of $\mathbf{V}$, which are described in the aside below for those interested.

SVDs and Eigendecomposition

In an SVD, the **singular values** of $\mathbf{A}$ are the square roots of the eigenvalues of $\mathbf{A}^T\mathbf{A}$. Furthermore, the columns of $\mathbf{V}$ are its eigenvectors.

To quickly demonstrate this:

$$\begin{aligned}\mathbf{A}^T\mathbf{A} &= (\mathbf{V}\mathbf{D}^T\mathbf{U}^T)(\mathbf{U}\mathbf{D}\mathbf{V}^T) \\ &= \mathbf{V}\mathbf{D}^T\mathbf{I}\mathbf{D}\mathbf{V} && \text{the transpose of an orthogonal matrix is its inverse} \\ &= \mathbf{V}^T\mathbf{D}^2\mathbf{V} && \text{the transpose of a diagonal matrix is itself}\end{aligned}$$

The last equation is actually the **eigendecomposition** of $\mathbf{A}^T\mathbf{A}$, where $\mathbf{V}$ is composed of the eigenvectors and $\mathbf{D}$ contains the eigenvalues.

7.7.2 Method 2: Inhomogeneous Solution

This method is more straightforward that the previous one, but its results are not as good. We rely again on the fact that homogeneous coordinates are invariant under scaling. Suppose,

[8] A property of an orthogonal matrix is that its transpose is its inverse: $\mathbf{V}^T = \mathbf{V}^{-1}$.

then, we "scale" by $1/m_{23}$... this would result in the following equation:

$$\begin{bmatrix} u_i \\ v_i \\ 1 \end{bmatrix} \simeq \begin{bmatrix} m_{00} & m_{01} & m_{02} & m_{03} \\ m_{10} & m_{11} & m_{12} & m_{13} \\ m_{20} & m_{21} & m_{22} & 1 \end{bmatrix} \begin{bmatrix} X_i \\ Y_i \\ Z_i \\ 1 \end{bmatrix}$$

Contrast this with (7.12) and its subsequent expansions. We would actually have a term that *doesn't* contain an m_{ij} factor, resulting in an **inhomogeneous system of equations** unlike before. Then, we can use least squares to approximate the solution.

Why isn't this method as good? Consider if $m_{23} \approx 0$, or some value close to zero. Setting that value to 1 is dangerous for **numerical stability**.[9]

7.7.3 Advantages and Disadvantages

The technique described in the first method is called the **direct linear calibration transformation**. It has some advantages and disadvantages.

Advantages

- The approach is very simple to formulate and solve. We create a matrix from a set of points and use singular value decomposition, which is a 1-line function in most math libraries.
- We minimize "algebraic error." This means that we are algebraically solving the system of equations. We relied on a specific set of tricks (as we saw), such as the constraint of $\hat{\mathbf{m}}$ as a unit vector to get to a clean, algebraic solution.

Disadvantages

- Because of the algebraic approach, and our restriction of $\hat{\mathbf{m}}$, it doesn't tell us the camera parameters directly. Obviously, it's unlikely that all of the parameters in (7.11) result in a unit vector.
- It's an approximate method that only models exactly what can be represented in the camera calibration matrix (again, see 7.11). If we wanted to include, say, radial distortion – a property we *could* model, just not within the projective transform equation – this method wouldn't be able to pick that up.
- It makes constraints hard to enforce because it assumes everything in $\hat{\mathbf{m}}$ is unknown. Suppose we definitively *knew* the focal length and wanted solutions where that stayed constrained... things get much more complicated.

[9] As per Wikipedia, a lack of numerical stability may significantly magnify approximation errors, giving us highly erroneous results.

- The approach minimizes $\|\mathbf{A}\hat{\mathbf{m}}\|$, which isn't the right error function. It's algebraically convenient to work with this "cute trick," but isn't precisely what we're trying to solve for; after all, we're working in a geometric world.

7.7.4 Geometric Error

Suppose we have our known set of points in the world, $\mathbf{X}$. What we're really trying to do is minimize the difference between the actual points in the image ($\mathbf{x}'$) and our projection of $\mathbf{X}$ given some estimated camera parameters for $\mathbf{M}$ (which results in our set of "predictions," $\mathbf{x}$). Visually, this is shown in Figure 7.15; mathematically, this is:

$$\min E = \sum_i d(x_i', x_i')$$

Since we control $\mathbf{M}$, what we're really trying to find is the $\mathbf{M}$ that minimizes that difference:

$$\min_{\mathbf{M}} = \sum_{i \in \mathbf{X}} d(\mathbf{x}_i', \mathbf{M}\mathbf{x}_i)$$

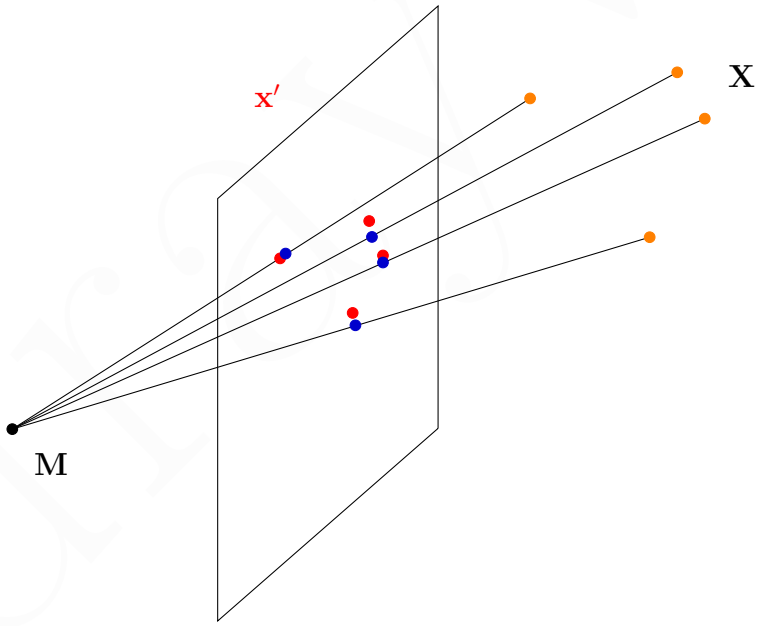


Figure 7.15: The red points are the "true" projection of the real-world points in $\mathbf{X}$ onto the image plane, whereas the blue points are their projection based on the estimated camera parameters $\mathbf{M}$.

This leads us to the "gold standard" algorithm that aims to determine the maximum likelihood estimation[10] of $\mathbf{M}$ described in algorithm 7.1. The minimization process involves

[10] We say it's an estimation because we assume that our guesses for $\mathbf{M}$ are distorted by some level of Gaussian noise; thus, we want to maximize the likelihood of our choice by taking the $\mathbf{M}$ with the least error.

solving a non-linear equation, so use whatever solver flavor you wish for that.

ALGORITHM 7.1: Finding camera calibration by minimizing geometric error.

Input: P: n known mappings from 3D to 2D, $\{\mathbf{X}_i \longleftrightarrow \mathbf{x}'_i\}, i \in [1, n > 6]$.
Result: The best possible camera calibration, $\mathbf{M}$.

if *normalizing* **then** `// (optional)`
  $\tilde{\mathbf{X}} = \mathbf{U}\mathbf{X}$ `// normalization between image and world space`
  $\tilde{\mathbf{x}}' = \mathbf{T}\mathbf{x}'$
end

$\mathbf{M}_0 \leftarrow$ result of the direct linear transformation minimization
Minimize geometric error starting from $\mathbf{M}_0$.
$\min_{\mathbf{M}} = \sum_{i \in \tilde{\mathbf{X}}} d(\tilde{\mathbf{x}}'_i, \tilde{\mathbf{M}}\tilde{\mathbf{X}}'_i)$

if *normalizing* **then** `// (optional)`
  $\mathbf{M} = \mathbf{T}^{-1}\tilde{\mathbf{M}}\mathbf{U}$ `// denormalize the resulting parameter matrix` $\tilde{\mathbf{M}}$
end

return $\mathbf{M}$

7.8 Using the Calibration

Henceforth we will assume that we've found $\mathbf{M}$. Let's not lose sight of what all of this was for... now, we can use $\mathbf{M}$ to find interesting camera parameters, such as the original focal length or the camera center.

Extracting each parameter is a relatively unique process; we'll start with finding the camera center in world coordinate space.

7.8.1 Where's ~~Waldo~~ the Camera?

Before we begin, let's introduce a slight change in notation for $\mathbf{M}$: we will divide it into a 3×3 matrix $\mathbf{Q}$ and its last column, $\mathbf{b}$. In other words, $\mathbf{M} = [\mathbf{Q} \,|\, \mathbf{b}]$.

The camera's location in world space can be found by finding the **null space** of the camera calibration matrix. In other words, if we find some $\mathbf{C}$ such that: $\mathbf{MC} = \mathbf{0}$, then $\mathbf{C}$ is the camera center.

Proof. Suppose we have two points, $\mathbf{p}$ and $\mathbf{c}$, and a ray passing through them also passes through a plane. We can define $\mathbf{x}$ as being a point anywhere along that ray:

$$\mathbf{x} = \lambda\mathbf{p} + (1 - \lambda)\mathbf{c}$$

Then, the resulting projection after applying the camera parameters is:

$$\mathbf{x}_{\text{proj}} = \mathbf{M}\mathbf{x} = \lambda\mathbf{M}\mathbf{p} + (1-\lambda)\mathbf{M}\mathbf{c}$$

Now we must imagine the plane this ray passes through. *All* of the points along $\overrightarrow{pc}$ are projected onto the plane at the exact same point; in other words, regardless of λ, every point on the ray will be projected onto the same point.

Thus, $\mathbf{M}\mathbf{c} = \mathbf{0}$, and the camera center must be in the null space. ■

Simpler. We can actually achieve the same goal by applying a formula. If $\mathbf{M}$ is split into two parts as we described above, then the camera center is found simply:[11]

$$\mathbf{c} = \begin{bmatrix} -\mathbf{Q}^{-1}\mathbf{b} \\ 1 \end{bmatrix}$$

Of course, this is just one property that went into creating the original $\mathbf{M}$, but we can hope that other properties can be derived just as simply.

7.9 Calibrating Cameras: Redux

The modern method for calibrating cameras is much simpler. Instead of a series of known points in a 3D scene, we can use a checkerboard on a firm surface. The checkerboard is then moved around the scene in various locations and orientations. The analysis of the checkerboards gives us the intrinsics, and the extrinsics can be obtained if we noted the locations of our checkerboards in world space.

The method's simplicity and availability makes it the dominant method of camera calibration today; it's supported directly by OpenCV and code is freely available.

> **Theory in Action: Calibration in Robotics**
>
> This method is useful in robotics in calibration a camera with respect to a robot's local coordinate system.
>
> Suppose we mount such a checkerboard to a robot's arm. The robot knows the precise position of its arm (with respect to itself). Thus, every time the camera takes a picture, we can correlate the checkerboard in the image to the robot's measurements and calibrate the camera with respect to the robot's coordinate system!

[11] The proof is left as an exercise to the reader. ☺

Multiple Views

> *There are things known and there are things unknown, and in between are the doors of perception.*
>
> — Aldous Huxley, *The Doors of Perception*

In this chapter we'll be discussing the idea of ***n*-views**: what can we learn when given multiple different images of the same scene? Mostly, we'll discuss $n = 2$ and create mappings from one image to another. We discussed this topic briefly in chapter 7 when talking about Stereo Geometry; we'll return to that later in this chapter as well, but we'll also be discussing other scenarios in which we create relationships from one image to another.

8.1 Image-to-Image Projections

There are many types of transformations; we open by discussing their mathematical differences.

Recall our discussion of perspective projection in 3D (see Equation 7.1): we had a 4×4 matrix and could model all of our various transformations (translation, rotation, etc.) under this framework by using Homogeneous Coordinates. This framework is convenient for a lot of reasons, and it enables us to chain transformations by continually multiplying. We can similarly adapt this model here with a 3×3 projective transformation matrix.

Translation This is our simplest transformation. We've seen this before: $\mathbf{x}' = \mathbf{x} + \mathbf{t}$. In our projective transformation model,

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

This transformation preserves all of the original properties: lengths, angles, and orientations all remain unchanged. Importantly, as well, *lines remain lines* under a translation transformation.

Euclidean Also called a **rigid body transformation**, this is the 2D version of the rotation

matrix we saw in (7.7) combined with a translation:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta & t_x \\ \sin\theta & \cos\theta & t_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

This transformation preserves the same properties as translation aside from orientation: lengths, angles, and lines are all unchanged.

Similarity Under a **similarity transformation**, we add scaling to our rigid body transformation:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} a\cos\theta & -a\sin\theta & t_x \\ a\sin\theta & a\cos\theta & t_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

Affine An **affine transformation** is one we've yet to examine but is important in computer vision (and graphics). It allows 6 degrees of freedom (adding **shearing** to the aforementioned translation, rotation, and scaling), and it enables us to map any 3 points to any other 3 points while the others follow the transformation.

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} a & b & c \\ d & e & f \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

It preserves a very different set of properties relative to the others we've seen: parallel lines, ratios of areas, and lines (as in, lines stay lines) are preserved.

Combining all of these transformations gives us a **general projective transformation**, also called a **homography**. It allows 8 degrees of freedom:[1]

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} \simeq \begin{bmatrix} wx' \\ wy' \\ w \end{bmatrix} = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

It's important to be aware of how many points we need to determine the existence of all of these transformations. Determining a translation only requires a single point correspondence between two images: there are two unknowns (t_x, t_y), and one correspondence gives us this relationship. Similarly, for a homography, we need (at least) 4 correspondence points to determine the transformation.

8.2 The Power of Homographies

Why do homogeneous coordinates make sense? Let's return to our understanding of the 3D to 2D projection through an image plane to some camera center. In Figure 8.1, we can see

[1] The last element is a 1 for the same reason as it was in Method 2: Inhomogeneous Solution: because the homogeneous projective matrix is invariant under scale, we can scale it all by this last element and it will have no effect. This would give us a different w than before, but that doesn't matter once we convert back to our new (x', y').

such a projection. The point on the image plane at $(x, y, 1)$ is just the intersection of the ray with the image plane (in blue); that means *any* point on the ray projects onto the image plane at that point.

The ray is just the scaled intersection point, (sx, sy, s), and that aligns with our understanding of how homogeneous represent this model.

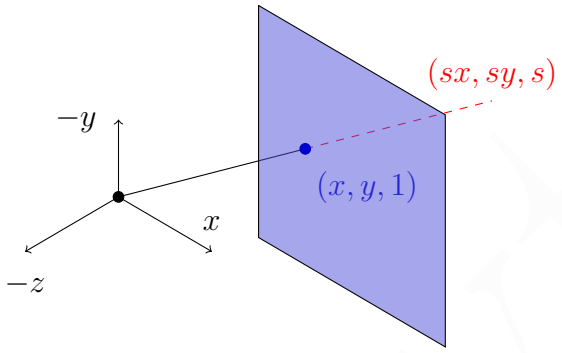


Figure 8.1: Each point on the image plane at $(x, y, 1)$ is represented by a ray in space, (sx, sy, s). All of the points on the ray are equivalent: $(x, y, 1) \simeq (sx, sy, s)$.

So how can we determine how a particular point in one projective plane maps onto another projective plane? This is demonstrated in Figure 8.2: we cast a ray through each pixel in the first projective plane and draw where that ray intersects the other projective plane.

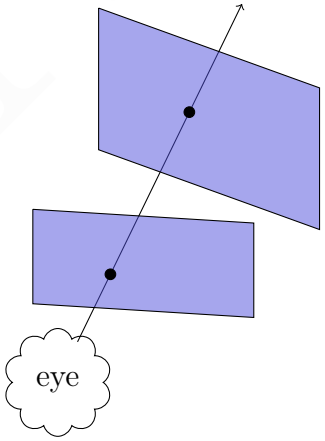


Figure 8.2: We can project a ray from the eye through each pixel in one projection plane and see the corresponding pixel in the other projection plane.

Notice that where this ray hits the world is actually irrelevant, now! We are no longer working with a 3D reprojection onto a 2D plane; instead, we can think about this as a 2D **image warp** (or just a transformation) from one plane to another. This basic principle is how we create **image mosaics** (colloquially, **panoramas**). To reiterate, homographies allow us to map projected points from one plane to another.

8.2.1 Creating Panoramas

Panoramas work because the camera center doesn't change as we rotate the camera; this means we don't need any knowledge about the 3D scene we're recording. A panorama can be stitched from multiple images via the following steps:

- Take a sequence of images from the same position, rotating the camera about its optical center.
- Compute a transformation between the first image and the second. This gives us a mapping between the images, showing how a particular pixel moved as we rotated, and which new pixels were introduced.
- Transform the second image to overlap with the first according to that transformation.
- Blend the images together, stitching them into one.

We can repeat this process as needed for every image in the sequence and get our panorama. We can interpret our mosaic naturally in 3D: each image is reprojection onto a common plane, and the mosaic is formed on that plane. This is demonstrated in Figure 8.3.[2]

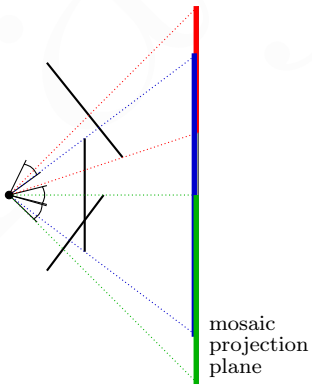


Figure 8.3: Reprojection of a series of images from the same camera center onto a "mosaic plane."

That's all well and good, but what about the math? That's what we're really interested in, right? ☺ Well, the transformation of a pixel from one projection plane to another (along a

[2] Because we're reprojecting onto a plane, we're limited by a 180° field of view; if we want a panorama of our entire surroundings, we'd need to map our mosaic of images on a different surface (say, a cylinder). The 180° limitation can be thought of as having a single camera with a *really* wide lens; obviously, it still can't see what's "behind" it.

ray, as we already showed in Figure 8.2) is just a homography:

$$\begin{bmatrix} wx' \\ wy' \\ w \end{bmatrix} = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \tag{8.1}$$

$$\mathbf{p}' = \mathbf{H}\mathbf{p} \tag{8.2}$$

How do we solve it? Well, there's the boring way and the cool way.

The Boring Way: Inhomogeneous Solution We can set this up as a system of linear equations with a vector $\mathbf{h}$ of 8 unknowns: $\mathbf{A}\mathbf{h} = \mathbf{b}$. Given at least 4 corresponding points between the images (the more the better),[3] we can solve for $\mathbf{h}$ using the least squares method as described in **??**: $\min \|\mathbf{A}\mathbf{h} - \mathbf{b}\|^2$.

The Cool Way: Déjà Vu We've actually already seen this sort of problem before: recall the trick we used in Method 1: Singular Value Decomposition for solving the camera calibration matrix! We can apply the same principles here, finding the eigenvector with the smallest eigenvalue in the singular value decomposition.

More specifically, we create a similar system of equations as in Equation 7.12, except in 2D. Then, we can rearrange that into an ugly matrix multiplication like in Equation 7.17 and pull the smallest eigenvector from the SVD.

8.2.2 Homographies and 3D Planes

In this section we'll demonstrate how the 8 degrees of freedom in a homography allow us to create mappings between planes. Recall, first, the camera calibration matrix and its effect on world-space coordinates:

$$\begin{bmatrix} u \\ v \\ 1 \end{bmatrix} \simeq \begin{bmatrix} m_{00} & m_{01} & m_{02} & m_{03} \\ m_{10} & m_{11} & m_{12} & m_{13} \\ m_{20} & m_{21} & m_{22} & m_{23} \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

Suppose, though, that *all* of the points in the world were lying on a plane. A plane is represented by a normal vector and a point on the plane, combining into the equation: $d = ax + by + cz$. We can, of course, rearrange things to solve for z: $z = \frac{d+ax+by}{-c}$, and then plug that into our transformation!

$$\begin{bmatrix} u \\ v \\ 1 \end{bmatrix} \simeq \begin{bmatrix} m_{00} & m_{01} & m_{02} & m_{03} \\ m_{10} & m_{11} & m_{12} & m_{13} \\ m_{20} & m_{21} & m_{22} & m_{23} \end{bmatrix} \begin{bmatrix} x \\ y \\ {}^{d-ax-by}/_{c} \\ 1 \end{bmatrix}$$

[3] For now, we assume an existing set of correctly-mapped pixels between the images; these could be mapped by a human under forced labor, like a graduate student. Later, in the chapter on Feature Recognition, we'll see ways of identifying correspondence points automatically.

Of course, this effects the overall camera matrix. The effect of the 3rd column (which is always multiplied by x, y, and a constant) can be spread to the other columns. This gives us:

$$\begin{bmatrix} u \\ v \\ 1 \end{bmatrix} \simeq \begin{bmatrix} m'_{00} & m'_{01} & 0 & m'_{03} \\ m'_{10} & m'_{11} & 0 & m'_{13} \\ m'_{20} & m'_{21} & 0 & m'_{23} \end{bmatrix} \begin{bmatrix} x \\ y \\ {}^{d-ax-by}/_{c} \\ 1 \end{bmatrix}$$

But now the camera matrix is a 3×3 homography! This demonstrates how homographies allow us to transform (or warp) between arbitrary planes.

8.2.3 Image Rectification

Since we can transform between arbitrary planes using homographies, we can actually apply that to images to see what they would look like from a different perspective!

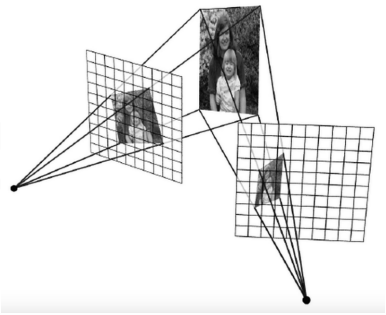
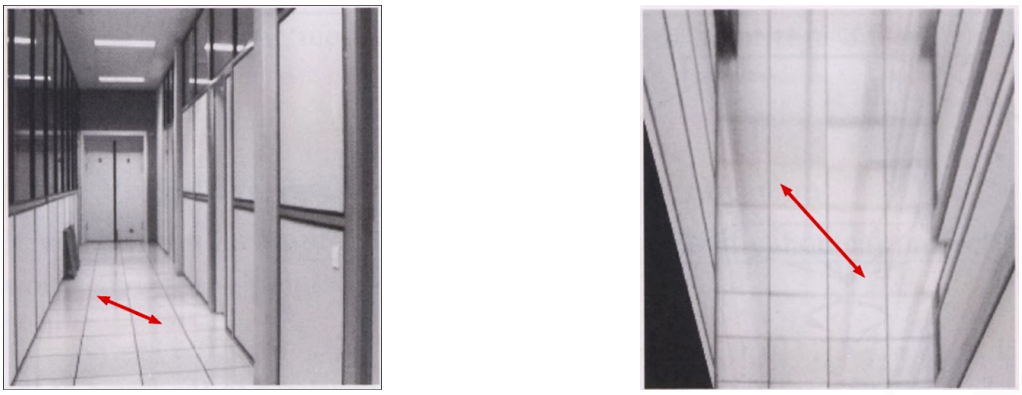


Figure 8.4: Mapping a plane (in this case, containing an image) onto two other planes (which, in this case, are the projection planes from two different cameras).

This process is called **image rectification** and enables some really cool applications. We can do things like rectify slanted views by restoring parallel lines or measure grids that are warped by perspective like in Figure 8.5.



(a) The original image. **(b)** After unwarping.

Figure 8.5: Applying unwarping to an image in order to measure the floor length.

How do we do this image warping and unwarping process? Suppose we're given a source image, $f(x, y)$ and a transformation function $T(x, y)$ that relates each point in the source image plane to another plane. How do we build the transformed image, $g(x', y')$? There are two approaches, one of which is incorrect.

Forward Warping

The naïve approach is to just pump every pixel through the transformation function and copy that intensity into the resulting (x', y'). That is, for each pixel $(x, y) \in f), (x', y') = T(x, y)$.

Unfortunately, though our images are discretized into pixels, our transformation function may not be. This means that a particular $(x', y') = T(x, y)$ may not correspond to an individual pixel! Thus, we'd need to distribute the color from the original pixel around the pixels *near* the transformed coordinate. This is known as **splatting**, and, as it sounds, doesn't result in very clean transformed images.

Inverse Warping

Instead of mapping from the source to the destination, we instead look at every pixel in the destination image and figure out where it came from in the source. Thus, we *start* with (x', y'), then find $(x, y) = T^{-1}(x', y')$.

We still have the problem of non-discrete locations, but can come up with a better solution. Now we know the neighboring pixels of our source location, and can perform **interpolation** on their intensities to get a better approximation of the intensity that belongs at the destination location.

The simplest approach would be **nearest neighbor interpolation**, which just takes the intensity of the closest pixel.

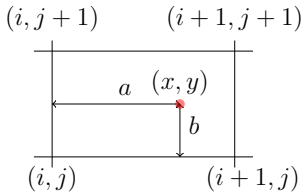


Figure 8.6: Finding the values for bilinear interpolation of (x, y).

This doesn't give great results. A better approach that is known as **bilinear interpolation**, which weighs the neighboring intensities based on the distance of the location (x, y) to each of its neighbors. Given a location, we can find its distances from a discrete pixel at (i, j): $a = x - i, b = y - j$. Then we calculate the final intensity in the destination image at (x', y'):

$$\begin{aligned} g(x', y') = &(1-a)(1-b) && f[i, j] \\ &+ a(1-b) && f[i+1, j] \\ &+ ab && f[i+1, j+1] \\ &+ (1-a)b && f[i, j+1] \end{aligned}$$

This calculation is demonstrated visually in Figure 8.6. There are more clever ways of interpolating, such as **bicubic interpolation** which uses **cubic splines**, but bilinear interpolation is mathematically simpler and still gives great results.

8.3 Projective Geometry

In this section we'll be getting much deeper into the mathematics behind **projective geometry**; specifically, we'll explore the duality of points and lines and the power of homogeneous coordinates (again!). We'll use this to build a mathematical basis *(no pun intended)* for approaching n-views in the subsequent section.

Recall, first, the geometric significance of homogeneous coordinates. In Figure 8.1 we saw that a point on the image is a ray in space, and every point along the ray is projected onto that same point in the image; we defined this as being *projectively similar*.

8.3.1 Alternative Interpretations of Lines

Homogeneous coordinates are also useful as representations of lines. Recall the standard form of the equation of a line in 2D:

$$ax + by + c = 0$$

Here, a, b, and c are all integers, and $a > 0$. From this, we can imagine a "compact" representation of the line that only uses its constants: $\begin{bmatrix} 2 & 3 & -12 \end{bmatrix}$. If we want the original equation back, we can dot this vector with $\begin{bmatrix} x & y & 1 \end{bmatrix}$:

$$\begin{bmatrix} a & b & c \end{bmatrix} \cdot \begin{bmatrix} x & y & 1 \end{bmatrix} = 0 \tag{8.3}$$

Now recall also an alternative definition of the dot product: the angle between the two vectors, θ, scaled by their magnitudes:

$$\mathbf{a} \cdot \mathbf{b} = \|\mathbf{a}\| \, \|\mathbf{b}\| \cos\theta$$

This indicates that the vector $\begin{bmatrix} a & b & c \end{bmatrix}$ is perpendicular to every possible point (x, y) in the $z = 1$ plane... or the ray passing through the origin and the point $(x, y, 1)$.

There's another interpretation, as well. Suppose we find a vector perpendicular to the line, as in Figure 8.7. The slope of the normal line is the reciprocal of the initial slope: $m' = -1/m$, and it passes through the origin $(0, 0)$.

In addition to this normal, we can define the line by also including a minimum distance from the origin, d. A line can be uniquely identified by this distance and normal vector; we can represent it as $\begin{bmatrix} n_x, n_y, d \end{bmatrix}$, where $\hat{\mathbf{n}} = \begin{bmatrix} n_x, n_y \end{bmatrix}$, and $d = c/\sqrt{a^2+b^2}$.[4]

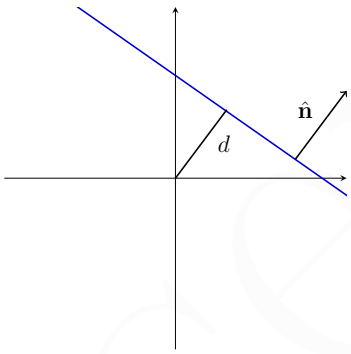


Figure 8.7: An alternative interpretation of a line.

8.3.2 Interpreting 2D Lines as 3D Points

First, we will demonstrate the relationship between lines in a 2D plane and points in 3D space under projective geometry.

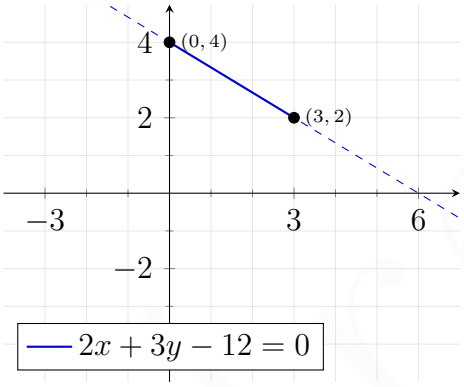


Figure 8.8: A line plotted on the standard xy-plane. Two points on the line and the segment connecting them are highlighted.

Suppose we plot some line in 2D, for example $2x + 3y - 12 = 0$ as in Figure 8.8. We can imagine a "compact" representation of the line by just using its constants: $\mathbf{m} = \begin{bmatrix} 2 & 3 & -12 \end{bmatrix}$, as we described above.

Now what if we drew this same line on a projection plane in 3D, like on the image plane at $z = 1$ in Figure 8.1? Then, suppose we visualize the rays that begin at the origin and pass through the two points on the line, $(0, 4, \mathbf{1})$ and $(3, 2, \mathbf{1})$ (notice the additional coordinate now that we're in 3-space).

The area between these rays is an infinite triangle into space, right? Overall, it's part of an entire plane! We have 3 points in space: the 2 points on the line and the origin, and three points define a plane. The intersection of that plane and the image plane *is the line* $2x + 3y - 12 = 0$, except now it's at $z = 1$. What we get is something that looks like Figure 8.9.

The most important and exciting part of this visualization is the orange normal vector. We can determine the normal by calculating the cross product between our two rays:

$$\begin{aligned} \begin{bmatrix} 0 & 4 & 1 \end{bmatrix} \times \begin{bmatrix} 3 & 2 & 1 \end{bmatrix} &= \begin{vmatrix} \hat{\mathbf{i}} & \hat{\mathbf{j}} & \hat{\mathbf{k}} \\ 0 & 4 & 1 \\ 3 & 2 & 1 \end{vmatrix} \\ &= (4 \cdot 1 - 2 \cdot 1)\hat{\mathbf{i}} - (0 \cdot 1 - 3 \cdot 1)\hat{\mathbf{j}} + (0 \cdot 2 - 3 \cdot 4)\hat{\mathbf{k}} \\ &= \begin{bmatrix} 2 & 3 & -12 \end{bmatrix} \end{aligned}$$

[4] Wikipedia defines the minimum distance between a point and a line: see this link.

Look familiar...? That's **m**, the constants from the original line! **This means that we can represent any line on our image plane by a point in the world using its constants.** The point defines a normal vector for a plane passing through the origin that creates the line where it intersects the image plane.

Similarly, from this understanding that a line in 2-space can be represented by a vector in 3-space, we can relate points in 2-space to lines in 3-space. We've already shown this when we said that a point on the projective plane lies on a ray (or line) in world space that passes through the origin and that point.

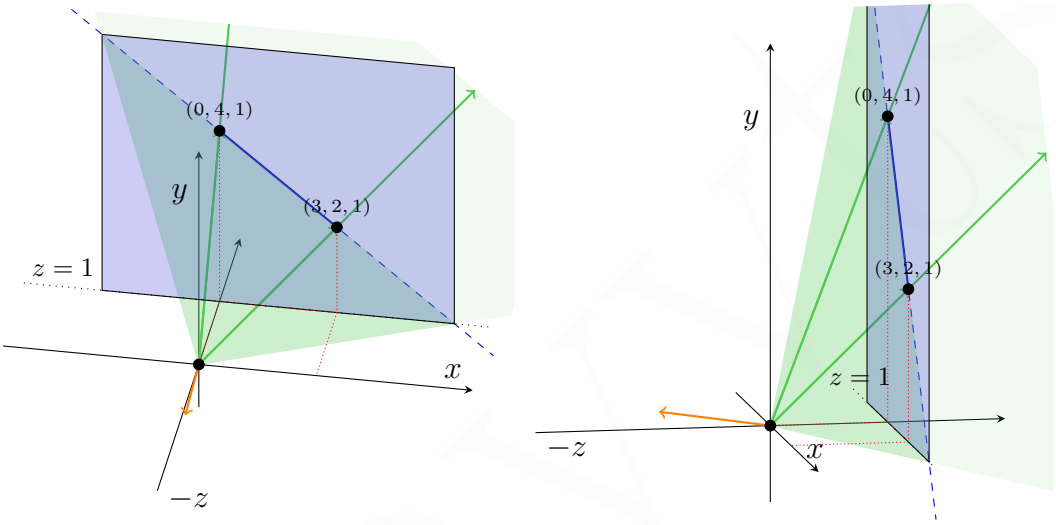


Figure 8.9: Two views of the same line from Figure 8.8 plotted on a projection plane (in blue) at $z = 1$. The green rays from the origin pass through the same points in the line, forming the green plane. As you can see, the intersection of the green plane with the blue plane create the line in question. Finally, the orange vector is the normal vector perpendicular to the green plane.

8.3.3 Interpreting 2D Points as 3D Lines

Suppose we want to find the intersection between two lines in 2-space. Turns out, it's simply their cross product! It (possibly) makes sense intuitively. If we have two lines in 2D (that are defined by a normal vector in 3D), their intersection is a point in the 2D plane, which is a ray in 3D.

Let's consider an example. We'll use $2x+3y-12 = 0$ as before, and another line, $2x-y+4 = 0$. Their intersection lies at $(0, 4, 1)$; again, we're in the $z = 1$ plane.[5] What's the cross

[5] Finding this intersection is trivial via subtraction: $4y - 16 = 0 \longrightarrow y = 4, x = 0$.

product between $\begin{bmatrix}2 & 3 & -12\end{bmatrix}$ and $\begin{bmatrix}2 & -1 & 4\end{bmatrix}$ (let's call it $\mathbf{v}$)?

$$\begin{aligned}\mathbf{v} = \begin{bmatrix}2 & 3 & -12\end{bmatrix} \times \begin{bmatrix}2 & -1 & 4\end{bmatrix} &= \begin{vmatrix}\hat{\mathbf{i}} & \hat{\mathbf{j}} & \hat{\mathbf{k}} \\ 2 & 3 & -12 \\ 2 & -1 & 4\end{vmatrix} \\ &= (3 \cdot 4 - (-1 \cdot -12))\hat{\mathbf{i}} - (2 \cdot 4 - (2 \cdot -12))\hat{\mathbf{j}} + (2 \cdot -1 - (2 \cdot 3))\hat{\mathbf{k}} \\ &= \begin{bmatrix}0 & -32 & -8\end{bmatrix}\end{aligned}$$

Of course, we need to put this vector along the ray into our plane at $z = 1$, which would be where $\mathbf{v}_z = 1$:

$$\begin{aligned}\hat{\mathbf{v}} = \frac{\mathbf{v}}{\|\mathbf{v}\|} &= \frac{\begin{bmatrix}0 & -32 & -8\end{bmatrix}}{\sqrt{(-32)^2 + (-8)^2}} \\ &= \begin{bmatrix}0 & -4/\sqrt{17} & -1/\sqrt{17}\end{bmatrix} \\ -\sqrt{17}\hat{\mathbf{v}} &= \begin{bmatrix}0 & 4 & 1\end{bmatrix}\end{aligned}$$

Convenient, isn't it? It's our point of intersection!

This leads us to **point-line duality**: given any formula, we can switch the meanings of points and lines to get another formula.

What if we wanted to know if, for example, a point $\mathbf{p}$ existed on a line $\mathbf{l}$? Well, the line is a normal vector in 3D defining a plane, and if the point in question existed on the line, it would be *on* that plane, and thus its ray would be perpendicular to the normal. So if $\mathbf{p} \cdot \mathbf{l} = \mathbf{p}^{\mathrm{T}}\mathbf{l} = 0$, then it's on the line!

Quick Maffs: **Proving the Cross Product and Intersection Equivalence**

I wasn't satisfied with my "proof by example" in Interpreting 2D Points as 3D Lines, so I ventured to make it more mathematically rigorous by keeping things generic.

We begin with two lines in 2-space and their vectors:

$$\begin{aligned} ax + by + c &= 0 & \mathbf{m} &= \begin{bmatrix}a & b & c\end{bmatrix} \\ dx + ey + f &= 0 & \mathbf{n} &= \begin{bmatrix}d & e & f\end{bmatrix}\end{aligned}$$

We can find their cross product as before:

$$\begin{aligned}\mathbf{v} = \begin{bmatrix}a & b & c\end{bmatrix} \times \begin{bmatrix}d & e & f\end{bmatrix} &= \begin{vmatrix}\hat{\mathbf{i}} & \hat{\mathbf{j}} & \hat{\mathbf{k}} \\ a & b & c \\ d & e & f\end{vmatrix} \\ &= (bf - ec)\hat{\mathbf{i}} - (af - dc)\hat{\mathbf{j}} + (ae - db)\hat{\mathbf{k}}\end{aligned}$$

This is a line in 3-space, but we need it to be at $z = 1$ on the projection plane, so

we divide by the z term, giving us:

$$\left(\frac{bf - ec}{ae - db}, \frac{dc + af}{ae - db}, 1\right)$$

Our *claim* was that this is the intersection of the lines. Let's prove that by solving the system of equations directly. We start with the system in matrix form:

$$\begin{bmatrix} a & b \\ d & e \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} -c \\ -f \end{bmatrix}$$

This is the general form $\mathbf{Mx} = \mathbf{y}$, and we solve by multiplying both sides by $\mathbf{M}^{-1}$. Thankfully we can find the inverse of a 2×2 matrix fairly easily:[a]

$$\mathbf{M}^{-1} = \frac{1}{ad - bc} \begin{bmatrix} e & -b \\ -d & a \end{bmatrix}$$

What does this expand to in our system? Well,

$$\begin{aligned} \mathbf{M}^{-1}\mathbf{Mx} &= \mathbf{M}^{-1}\mathbf{y} \\ \begin{bmatrix} x \\ y \end{bmatrix} &= \frac{1}{ad - bc} \begin{bmatrix} e & -b \\ -d & a \end{bmatrix} \begin{bmatrix} -c \\ -f \end{bmatrix} \\ &= \frac{1}{ad - bc} \begin{bmatrix} -ec + bf \\ cd - af \end{bmatrix} \\ (x, y) &= \left(\frac{bf - ec}{ad - bc}, \frac{cd - af}{ad - bc}\right) \end{aligned}$$

Look familiar? Boom, done. ■

[a] Refer to this page.

8.3.4 Ideal Points and Lines

Our understanding of a point on the image plane as being the projection of every point on a ray in space (again, as shown in Figure 8.1 and other figures) relies on the fact that the ray does in fact intersect the image plane. When isn't this true? For rays parallel to the image plane! This would be any ray confined to an xy-plane outside of $z = 1$.

We call the points that form these rays **ideal points**. If you squint hard enough, you may be able to convince yourself that these rays intersection with the image plane at infinity, so we represent them as: $\mathbf{p} \simeq (x, y, 0)$; when we go back to non-homogeneous coordinates, we divide by zero, which makes x and y blow up infinitely large.

We can extend this notion to **ideal lines** as well: consider a normal vector $\mathbf{l} \simeq (a, b, 0)$. This is actually mathematically coherent: it defines a plane perpendicular to the image plane, and the resulting line goes through the origin of the *image plane*, which we call the **principle point**.

8.3.5 Duality in 3D

We can extend this notion of point-line duality into 3D as well. Recall the equation of a plane:

$$ax + by + cz + d = 0$$

where (a, b, c) is the normal of the plane, and $d = ax_0 + by_0 + cz_0$ for some point on the plane (x_0, y_0, z_0). Well, projective points in 3D have 4 coordinates, as we've already seen when we discussed rotation and translation for Extrinsic Camera Parameters: $\begin{bmatrix} wx & wy & wz & w \end{bmatrix}$. We use this to define planes as homogeneous coordinates in 3D! A plane N is defined by a 4-vector $\begin{bmatrix} a & b & c & d \end{bmatrix}$, and so $\mathbf{N} \cdot \mathbf{p} = 0$.

8.4 Applying Projective Geometry

With the vocabulary and understanding under our belt, we can now aim to apply this geometric understanding to multiple views of a scene. We'll be returning back to stereo correspondence, so before we begin, re-familiarize yourself with the terms and concepts we established previously when discussing Epipolar Geometry.

To motivate us again, this is our scenario:

> *Given two views of a scene, from two cameras that don't necessarily have parallel optical axes (à la Figure 7.9), what is the relationship between the location of a scene point in one image and its location in the other?*

We established the epipolar constraint, which stated that a point in one image lies on the epipolar line in the other image, and vice-versa. These two points, along with the baseline vector between the cameras, created an epipolar plane.

We need to turn these geometric concepts into algebra so that we can calculate them with our computers. To do that, we need to define our terms. We have a world point, $\mathbf{X}$, and two camera origins, $\mathbf{c}_1$ and $\mathbf{c_2}$. The rays from the world point to the cameras are $\mathbf{p}_1$ and $\mathbf{p}_2$, respectively, intersecting with their image planes at $\mathbf{x}_1$ and $\mathbf{x}_2$.

8.4.1 Essential Matrix

We will assume that we have *calibrated* cameras, so we know the transformation from one origin to the other: some translation $\mathbf{t}$ and rotation $\mathbf{R}$. This means that:

$$\mathbf{x}_2 = \mathbf{R}\mathbf{x}_1 + \mathbf{t}$$

Let's cross both sides of this equation by $\mathbf{t}$, which would give us a vector normal to the pixel ray and translation vector (a.k.a. the baseline):

$$\begin{aligned} \mathbf{t} \times \mathbf{x}_2 &= \mathbf{t} \times \mathbf{R}\mathbf{x}_1 + \mathbf{t} \times \mathbf{t} \\ &= \mathbf{t} \times \mathbf{R}\mathbf{x}_1 \end{aligned}$$

Now we dot both sides with $\mathbf{x}_2$, noticing that the left side is $\mathbf{0}$ because the angle between a vector and a vector intentionally made perpendicular to that vector will always be zero!

$$\begin{aligned}\mathbf{x}_2 \cdot (\mathbf{t} \times \mathbf{x}_2) &= \mathbf{x}_2 \cdot (\mathbf{t} \times \mathbf{R}\mathbf{x}_1) \\ \mathbf{0} &= \mathbf{x}_2 \cdot (\mathbf{t} \times \mathbf{R}\mathbf{x}_1)\end{aligned}$$

Continuing, suppose we say that $\mathbf{E} = [\mathbf{t}_\times]\mathbf{R}$.[6] This means:

$$\mathbf{x}_2^{\mathrm{T}}\mathbf{E}\mathbf{x}_1 = \mathbf{0}$$

which is a really compact expression relating the two points on the two image planes. $\mathbf{E}$ is called the **essential matrix**.

Notice that $\mathbf{E}\mathbf{x}_1$ evaluates to some vector, and that vector multiplied by $\mathbf{x}_2^{\mathrm{T}}$ (or dotted with $\mathbf{x}_2$) is zero. We established earlier that this relationship is part of the point-line duality in projective geometry, so this single *point*, $\mathbf{E}\mathbf{x}_2$, defines a *line* in the other $\mathbf{c}_1$ camera frame! Sound familiar? That's the epipolar line.

We've converted the epipolar constraint into an algebraic expression! Well what if our cameras are not calibrated? Theoretically, we should be able to determine the epipolar lines if we're given enough points correlated between the two images. This will lead us to the...

8.4.2 Fundamental Matrix

Let's talk about **weak calibration**. This is the idea that we know we have two cameras that properly behave as projective cameras should, but we don't know any details about their calibration parameters (such as focal length). What we're going to do is *estimate* the epipolar geometry from a (redundant) set of point correspondences across the uncalibrated cameras.

Recall the full calibration matrix from (7.11), compactly describing something like this:

$$\begin{bmatrix} wx_{im} \\ wy_{im} \\ w \end{bmatrix} = \mathbf{K}_{\text{int}}\mathbf{\Phi}_{\text{ext}} \begin{bmatrix} x_w \\ y_w \\ z_w \\ 1 \end{bmatrix}$$

For convenience sake, we'll assume that there is no skew, s. This makes the intrinsic parameter matrix $\mathbf{K}$ invertible:

$$\mathbf{K} = \begin{bmatrix} -f/s & 0 & o_x \\ 0 & -f/s_y & o_y \\ 0 & 0 & 1 \end{bmatrix}$$

[6] This introduces a different notation for the cross product, expressing it as a matrix multiplication. This is explained in further detail in **??**, in **??**.

Recall, though, that the extrinsic parameter matrix is what maps points from world space to points in the camera's coordinate frame (see The Duality of Space), meaning:

$$\mathbf{p}_{im} = \mathbf{K}_{\text{int}} \underbrace{\mathbf{\Phi}_{\text{ext}} \mathbf{p}_w}_{\mathbf{p}_c}$$
$$\mathbf{p}_{im} = \mathbf{K}_{\text{int}} \mathbf{p}_c$$

Since we said that the intrinsic matrix is invertible, that also means that:

$$\mathbf{p}_c = \mathbf{K}_{\text{int}}^{-1} \mathbf{p}_{im}$$

Which tells us that we can find a *ray* through the camera and the world (since it's a homogeneous point in 2-space, and recall point-line duality) corresponding to this point. Furthermore, for two cameras, we can say:

$$\mathbf{p}_{c,\text{left}} = \mathbf{K}_{\text{int,left}}^{-1} \mathbf{p}_{im,\text{left}}$$
$$\mathbf{p}_{c,\text{right}} = \mathbf{K}_{\text{int,right}}^{-1} \mathbf{p}_{im,\text{right}}$$

Now note that we don't *know* the values of $\mathbf{K}_{\text{int}}$ for either camera since we're working in the uncalibrated case, but we *do* know that there are *some* parameters that would calibrate them. Furthermore, there is a well-defined relationship between the left and right points in the calibrated case that we defined previously using the essential matrix. Namely,

$$\mathbf{p}_{c,\text{right}}^{\text{T}} \mathbf{E} \mathbf{p}_{c,\text{left}} = \mathbf{0}$$

Thus, via substitution, we can say that

$$\left(\mathbf{K}_{\text{int,right}}^{-1} \mathbf{p}_{im,\text{right}}\right)^{\text{T}} \mathbf{E} \left(\mathbf{K}_{\text{int,left}}^{-1} \mathbf{p}_{im,\text{left}}\right)^{\text{T}} = \mathbf{0}$$

and then use the properties of matrix multiplication[7] to rearrange this and get the **fundamental matrix**, $\mathbf{F}$:

$$\mathbf{p}_{im,\text{right}}^{\text{T}} \underbrace{\left(\left(\mathbf{K}_{\text{int,right}}^{-1}\right)^{\text{T}} \mathbf{E} \mathbf{K}_{\text{int,left}}^{-1}\right)}_{\mathbf{F}} \mathbf{p}_{im,\text{left}} = \mathbf{0}$$

This gives us a beautiful, simple expression relating the image points on the planes from both cameras:

$$\mathbf{p}_{im,\text{right}}^{\text{T}} \mathbf{F} \mathbf{p}_{im,\text{left}} = \mathbf{0}$$

Or, even more simply: $\mathbf{p}^{\text{T}} \mathbf{F} \mathbf{p}' = \mathbf{0}$. This is the **fundamental matrix constraint**, and given enough correspondences between $\mathbf{p} \to \mathbf{p}'$, we will be able to solve for $\mathbf{F}$. This matrix is very powerful in describing how the epipolar geometry works.

[7] Specifically, we use the fact that $(\mathbf{AB})^{\text{T}} = \mathbf{B}^{\text{T}} \mathbf{A}^{\text{T}}$, as shown here.

Properties of the Fundamental Matrix

Recall from Projective Geometry that when we have an $\mathbf{l}$ such that $\mathbf{p}^T\mathbf{l} = \mathbf{0}$, that $\mathbf{l}$ describes a line in the image plane. Well, the epipolar line in the p image associated with p' is defined by: $\mathbf{l} = \mathbf{F}\mathbf{p}'$; similarly, the epipolar line in the prime image is defined by: $\mathbf{l}' = \mathbf{F}^T\mathbf{p}$. This means that the fundamental matrix gives us the epipolar constraint between two images with some correspondence.

What if $\mathbf{p}'$ was on the epipolar line in the prime image for *every* point $\mathbf{p}$ in the original image? That occurs at the epipole! We can solve for that by setting $\mathbf{l} = \mathbf{0}$, meaning we can find the two epipoles via:

$$\mathbf{F}\mathbf{p}' = \mathbf{0}$$
$$\mathbf{F}^T\mathbf{p} = \mathbf{0}$$

Finally, the fundamental matrix is a 3×3 **singular** matrix. It's *singular* because it maps from homogeneous 2D points to a 1D family (which are points *or* lines under point-line duality), meaning it has a rank of 2. We will prove this in this aside and use it shortly.

The power of the fundamental matrix is that it relates the *pixel* coordinates between two views. We no longer need to know the intrinsic parameter matrix. With enough correspondence points, we can reconstruct the epipolar geometry with an *estimation* of the fundamental matrix, without knowing anything about the true intrinsic or extrinsic parameter matrix. In Figure 8.10, we can see this in action. The green lines are the estimated epipolar lines in both images derived from the green correspondence points, and we can see that points along a line in one image are also exactly along the corresponding epipolar line in the other image.

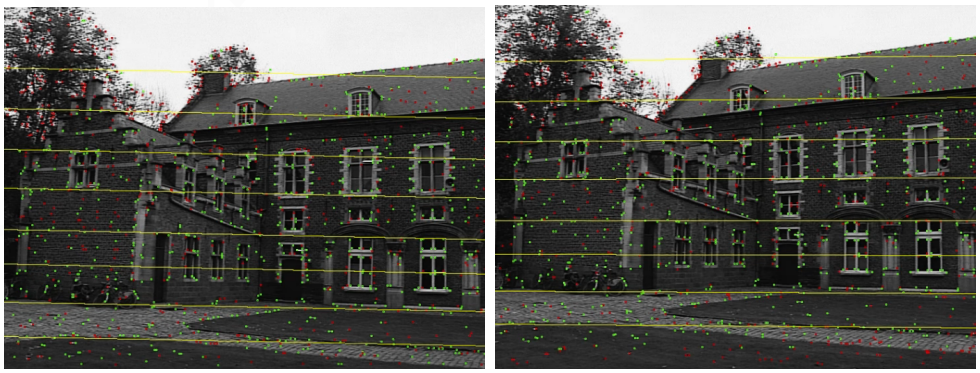


Figure 8.10: Estimation of epipolar lines given a set of correspondence points (in green).

Computing the Fundamental Matrix From Correspondences

Each point correspondence generates *one* constraint on $\mathbf{F}$. Specifically, we can say that:[8]

$$\underbrace{\begin{bmatrix} u' & v' & 1 \end{bmatrix} \begin{bmatrix} f_{11} & f_{12} & f_{13} \\ f_{21} & f_{22} & f_{23} \\ f_{31} & f_{32} & f_{33} \end{bmatrix}}_{\mathbf{p}^{\mathrm{T}}\mathbf{F}} \underbrace{\begin{bmatrix} u \\ v \\ 1 \end{bmatrix}}_{\mathbf{p}'} = \mathbf{0}$$

Multiplying this out, and generalizing it to n correspondences, gives us this massive system:

$$\begin{bmatrix} u'_1u_1 & u'_1v_1 & u'_1 & v'_1u_1 & v'_1v_1 & v'_1 & u_1 & v_1 & 1 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ u'_nu_n & u'_nv_n & u'_n & v'_nu_n & v'_nv_n & v'_n & u_n & v_n & n \end{bmatrix} \begin{bmatrix} f_{11} \\ f_{12} \\ f_{13} \\ f_{21} \\ f_{22} \\ f_{23} \\ f_{31} \\ f_{32} \\ f_{33} \end{bmatrix} = \mathbf{0}$$

We can solve this via the same two methods we've seen before: using the trick with singular value decomposition, or scaling to make $f_{33} = 1$ and using the least squares approximation. We explained these in full in Method 1: Singular Value Decomposition and Method 2: Inhomogeneous Solution, respectively.

Unfortunately, due to the fact that our point correspondences are estimations, this actually doesn't give amazing results. Why? Because we didn't pull rank on our matrix $\mathbf{F}$! It's a rank-2 matrix, as we demonstrate in this aside, but we didn't enforce that when solving/approximating our 3×3 matrix with correspondence points and assumed it was full rank.[9]

How can we enforce that? Well, first we solve for $\mathbf{F}$ as before via one of the two methods we described. Then, we take the SVD of *that* result, giving us: $\mathbf{F} = \mathbf{U}\mathbf{D}\mathbf{V}^{\mathrm{T}}$.

The diagonal matrix is the singular values of $\mathbf{F}$, and we can enforce having only rank 2 by setting the last value (which is the smallest value, since we sort the diagonal in decreasing order by convention) to zero:

$$\mathbf{D} = \begin{bmatrix} r & 0 & 0 \\ 0 & s & 0 \\ 0 & 0 & t \end{bmatrix} \implies \hat{\mathbf{D}} = \begin{bmatrix} r & 0 & 0 \\ 0 & s & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

Then we recreate a better $\hat{\mathbf{F}} = \mathbf{U}\hat{\mathbf{D}}\mathbf{V}^{\mathrm{T}}$, which gives much more accurate results for our epipolar geometry.

[8] You may notice that this looks awfully similar to previous "solve given correspondences" problems we've done, such as in (7.17) when Calibrating Cameras.

[9] Get it? Because pulling rank means taking advantage of seniority to enforcing some rule or get a task done, and we didn't enforce the rank-2 constraint on $\mathbf{F}$? Heh, nice.

Proof: F is a Singular Matrix of Rank 2

We begin with two planes (the *left plane* and the *right plane*) that have some points $\mathbf{p}$ and $\mathbf{p}'$, respectively, mapping some world point $\mathbf{x}_\pi$. Suppose we know the exact homography that can make this translation: $\mathbf{p}' = \mathbf{H}_\pi \mathbf{p}$.

Then, let $\mathbf{l}'$ be the epipolar line in the right plane corresponding to $\mathbf{p}$. It passes through the epipole, $\mathbf{e}'$, as well as the correspondence point, $\mathbf{p}'$.

We know, thanks to point-line duality in projective geometry, that this line $\mathbf{l}'$ is cross product of $\mathbf{p}'$ and $\mathbf{e}'$, meaning:[a]

$$\begin{aligned}\mathbf{l}' &= \mathbf{e}' \times \mathbf{p}' \\ &= \mathbf{e}' \times \mathbf{H}_\pi \mathbf{p} \\ &= \left[\mathbf{e}'_\times\right] \mathbf{H}_\pi \mathbf{p}\end{aligned}$$

But since $\mathbf{l}'$ is the epipolar line for $\mathbf{p}$, we saw that this can be represented by the fundamental matrix: $\mathbf{l}' = \mathbf{F}\mathbf{p}$. Meaning:

$$\left[\mathbf{e}'_\times\right] \mathbf{H}_\pi = \mathbf{F}$$

That means the rank of $\mathbf{F}$ is the same as the rank of $[\mathbf{e}'_\times]$, which is **2**! *(just trust me on that part...)*

[a] As we showed in Interpreting 2D Points as 3D Lines, a line passing through two points can be defined by the cross product of those two points.

Fundamental Matrix Applications

There are many useful applications of the fundamental matrix:

- **Stereo Image Rectification**: Given two radically different views of the same object, we can use the fundamental matrix to rectify the views into a single plane! This makes epipolar lines the same across both images and reduces the dense correspondence search to a 1D scanline.
- **Photo Synth**: From a series of pictures from different views of a particular scene, we can recreate not only the various objects in the scene, but also map the locations from which those views were taken in a composite image, as demonstrated below in Figure 8.11.
- **3D Reconstruction**: Extending on the PhotoSynth concept, given enough different angles of a scene or structure, we can go so far as to recreate a *3D model* of that structure.

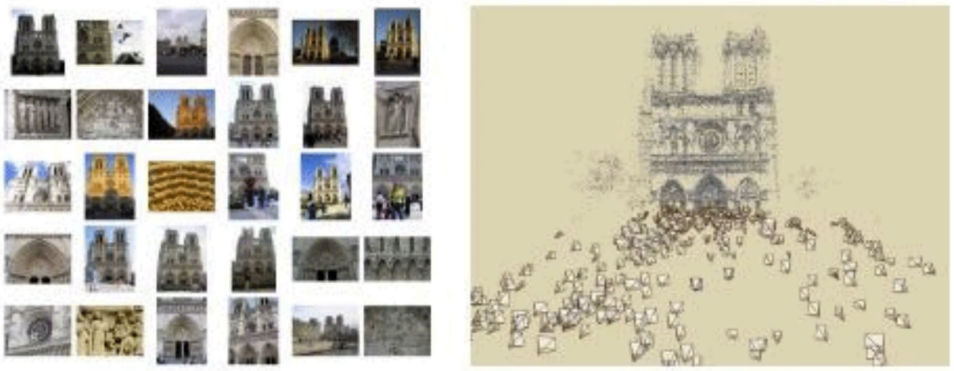


Figure 8.11: The Photosynth project (link). An assortment of photos of the Notre Dame (left) can be mapped to a bunch of locations relative to the object (right).

8.5 Summary

For 2 views into a scene (sorry, I guess we never did get into n-views ☺), there is a geometric relationship that defines the relationship between rays in one view and another view. This is Epipolar Geometry.

These relationships can be captured algebraically (and hence computed), with the essential matrix for calibrated cameras and the fundamental matrix for *un*calibrated cameras. We can find these relationships with enough point correspondences.

This is proper computer vision! We're no longer just processing images. Instead, we're getting "good stuff" about the scene: determining the actual 3D geometry from our 2D images.

Feature Recognition

> *Since we live in a world of appearances, people are judged by what they seem to be. If the mind can't read the predictable features, it reacts with alarm or aversion. Faces which don't fit in the picture are socially banned. An ugly countenance, a hideous outlook can be considered as a crime and criminals must be inexorably discarded from society.*
>
> — Erik Pevernagie, *"Ugly mug offense"*

Until this chapter, we've been creating relationships between images and the world with the convenient assumption that we have sets of well-defined correspondence points. As noted in footnote 3, these could have been the product of manual labor and painstaking matching, but we can do better. Now, we will discover ways to automatically creating correspondence relationships, which will lead the way to a more complete, automated approach to computer vision.

Our goal, to put it generally, is to find points in an image that can be *precisely* and *reliably* found in other images.

We will detect features (specifically, **feature points**) in both images and match them together to find the corresponding pairs. Of course, this poses a number of problems:

- **How can we detect the same points *independently* in both images?** We need whatever algorithm we use to be consistent across images, returning the same feature points. In other words, we need a **repeatable detector**.
- **How do we correlate feature points?** We need to quantify the same interesting feature points a similar way. In other words, we need a *reliable* and *distinctive* **descriptor**. For a rough analogy, our descriptor could "name" a feature point in one image *Eddard*, and name the same one *Ned* in the other, but it definitely shouldn't name it *Robert* in the other.

Feature points are used in *many* applications. To name a few, they are useful in 3D reconstruction, motion tracking, object recognition, robot navigation, and much much more...

So what makes a good feature?

- **Repeatability / Precision**: The same feature can be found in several images pre-

cisely despite geometric and photometric transformations. We can find the same point with the same precision and metric regardless of where it is in the scene.

- **Saliency / Matchability**: Each feature should have a distinctive description: when we find a point in one image, there shouldn't be a lot of candidate matches in the other image.
- **Compactness / Efficiency**: There should be *far* fewer features than there are pixels in the image, but there should be "enough."
- **Locality**: A feature occupies a relatively small area of the image, and it's robust to clutter and occlusion. A neighborhood that is too large may change drastically from a different view due to occlusion boundaries.

9.1 Finding Interest Points

The title of this section may give the answer away, but what's something that fits all of the criteria we mentioned in our introduction? Consider, for a moment, a black square on a white background, as such:

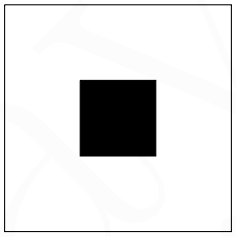


Figure 9.1: A simple image. What sections would make good features?

What areas of the image would make good features? Would the center of the black square be good? Probably not! There are a lot of areas that would match an all-black area: everywhere else on the square! What about some region of the left (or any) edge? No again! We can move along the edge and things look identical.

What about the *corners*? The top-right corner is incredibly distinct for the image; no other region is quite like it.

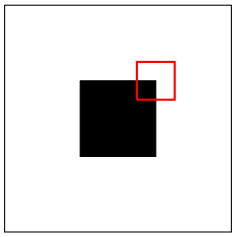


Figure 9.2: A corner feature, highlighted from Figure 9.1.

That begs the question: how do we detect corners? We can describe a corner as being an area with significant change in a variety of directions. In other words, the *gradients* have more than one direction.

9.1.1 Harris Corners

Developed in 1988 by Harris & Stephens (though Harris gets all the recognition), the technique we'll describe in this section finds these aforementioned "feature" areas in which the gradient varies in more than one direction.

Harris corners are based on an approximation model and an error model. We start with an image, I. We can say that the change in appearance after shifting that image some (small) amount (u, v), over some window function w, is represented by an error function that's the sum of the changes:

$$E(u,v) = \sum_{x,y} w(x,y)\left[I(x+u, y+v) - I(x,y)\right]^2$$

The window function can be a simple piecewise function that's 1 within the window and 0 elsewhere, or a Gaussian filter that will weigh pixels near the center of the window appropriately.

We can view the error function visually as an image, as well. The error function with no shift (i.e. $E(0,0)$) will be 0, since there is no change in pixels; it would be a black pixel. As we shifted, the error would increase towards white. Now suppose we had an image that was the same intensity everywhere (like, for example, the center region of a black square, maybe?). Its error function would *always* be zero regardless of shift. This gives us an intuition for the use of the error function: we want regions that have error in all shift directions.

We are working with small values of (u, v): a large error for a small shift might indicate a corner-like region. How do we model functions for small changes? We use *Taylor expansions*. A second-order Taylor expansion of $E(u, v)$ about $(0, 0)$ gives us a local quadratic approximation for small values of u and v.

Recall, from calculus oh-so-long-ago, the **Taylor expansion**. We approximate a function $F(x)$ for some small δ value:

$$F(\delta x) \approx F(0) + \delta x \cdot \frac{dF(0)}{dx} + \frac{1}{2}\delta x^2 \cdot \frac{d^2F(0)}{dx^2}$$

Things get a little uglier in two dimensions; we need to use matrices. To approximate our error function $E(u, v)$ for small values of u and v, we say:

$$E(u,v) \approx E(0,0) + \begin{bmatrix} u & v \end{bmatrix} \begin{bmatrix} E_u(0,0) \\ E_v(0,0) \end{bmatrix} + \frac{1}{2}\begin{bmatrix} u & v \end{bmatrix} \begin{bmatrix} E_{uu}(0,0) & E_{uv}(0,0) \\ E_{uv}(0,0) & E_{vv}(0,0) \end{bmatrix} \begin{bmatrix} u \\ v \end{bmatrix}$$

where E_n indicates the gradient of E in the n direction, and similarly E_{nm} indicates the 2^{nd}-order gradient in the n then m direction.

When we actually find, expand, and simplify all of these derivatives (which we do in this aside), we get the following:

$$E(u,v) \approx \frac{1}{2}\begin{bmatrix} u & v \end{bmatrix}\begin{bmatrix} \sum_{x,y} 2w(x,y)I_x(x,y)^2 & \sum_{x,y} 2w(x,y)I_x(x,y)I_y(x,y) \\ \sum_{x,y} 2w(x,y)I_x(x,y)I_y(x,y) & \sum_{x,y} 2w(x,y)I_y(x,y)^2 \end{bmatrix}\begin{bmatrix} u \\ v \end{bmatrix} \quad (9.1)$$

We can simplify this expression further with a substitution. Let $\mathbf{M}$ be the **second moment matrix** computed from the image derivatives:

$$\mathbf{M} = \sum_{xy} w(x,y)\begin{bmatrix} I_x^2 & I_xI_y \\ I_xI_y & I_y^2 \end{bmatrix}$$

Then, simply (for values of u and v near 0):

$$E(u,v) \approx \begin{bmatrix} u & v \end{bmatrix}\mathbf{M}\begin{bmatrix} u \\ v \end{bmatrix} \quad (9.2)$$

Let's examine the properties of this magical moment matrix, $\mathbf{M}$, and see if we can get some insights about how a "corner-like" area would look.

(Not So) Quick Maffs: Slogging Through the Taylor Expansion

So what do the derivatives of the error function look like? We'll get through them one step at a time. Our error function is:

$$E(u,v) = \sum_{x,y} w(x,y)\left[I(x+u,y+v) - I(x,y)\right]^2$$

We'll start with the first order derivatives E_u and E_v. For this, we just need the "power rule": $\frac{d}{dx}(f(x))^n = nf(x)^{n-1}\cdot\frac{d}{dx}f(x)$. Remember that u is the shift in the x direction, and thus we need the image gradient in the x direction, and similarly for v and y.

$$E_u(u,v) = \sum_{x,y} 2w(x,y)\left[I(x+u,y+v) - I(x,y)\right]I_x(x+u,y+v)$$
$$E_v(u,v) = \sum_{x,y} 2w(x,y)\left[I(x+u,y+v) - I(x,y)\right]I_y(x+u,y+v)$$

Now we will take the 2nd derivative with respect to u, giving us E_{uu} (and likewise for E_{vv}). Recall, briefly, the "product rule":

$$\frac{d}{dx}f(x)g(x) = f(x)\frac{d}{dx}g(x) + g(x)\frac{d}{dx}f(x)$$

This gives us:

$$\begin{aligned}
E_{uu}(u,v) &= \sum_{x,y} 2w(x,y)I_x(x+u,y+v)I_x(x+u,y+v) \\
&\quad + \sum_{x,y} 2w(x,y)\left[I(x+u,y+v) - I(x,y)\right] I_{xx}(x+u,y+v) \\
E_{vv}(u,v) &= \sum_{x,y} 2w(x,y)I_y(x+u,y+v)I_y(x+u,y+v) \\
&\quad + \sum_{x,y} 2w(x,y)\left[I(x+u,y+v) - I(x,y)\right] I_{yy}(x+u,y+v)
\end{aligned}$$

Now the only thing that remains is the cross-derivative, E_{uv}, which now requires gradients in both x and y of the image function as well as the cross-derivative in x-then-y of the image.

$$\begin{aligned}
E_{uv}(u,v) &= \sum_{x,y} 2w(x,y)I_x(x+u,y+v)I_y(x+u,y+v) \\
&\quad + \sum_{x,y} 2w(x,y)\left[I(x+u,y+v) - I(x,y)\right] I_{xy}(x+u,y+v)
\end{aligned}$$

These are all absolutely disgusting, but, thankfully, we're about to make a bunch of the terms disappear entirely since we are evaluating them at $(0,0)$.

Onward, brave reader.

Plugging in $(u=0, v=0)$ into each of these expressions gives us the following set of newer, simpler expressions:

$$\begin{aligned}
E_u(0,0) &= E_v(0,0) = 0 \\
E_{uu}(0,0) &= \sum_{x,y} 2w(x,y)I_x(x,y)^2 \\
E_{vv}(0,0) &= \sum_{x,y} 2w(x,y)I_y(x,y)^2 \\
E_{uv}(0,0) &= \sum_{x,y} 2w(x,y)I_x(x,y)I_y(x,y)
\end{aligned}$$

Notice that all we need now is the first-order gradient of the image in each direction, x and y. What does this mean with regards to the Taylor expansion? It expanded to:

$$E(u,v) \approx E(0,0) + \begin{bmatrix} u & v \end{bmatrix} \begin{bmatrix} E_u(0,0) \\ E_v(0,0) \end{bmatrix} + \frac{1}{2} \begin{bmatrix} u & v \end{bmatrix} \begin{bmatrix} E_{uu}(0,0) & E_{uv}(0,0) \\ E_{uv}(0,0) & E_{vv}(0,0) \end{bmatrix} \begin{bmatrix} u \\ v \end{bmatrix}$$

But the error at $(0,0)$ of the image *is just 0*, since there is no shift! And we've already seen that its first-order derivatives are 0 as well. Meaning the above expansion

simplifies greatly:

$$E(u,v) \approx \frac{1}{2}\begin{bmatrix} u & v \end{bmatrix}\begin{bmatrix} E_{uu}(0,0) & E_{uv}(0,0) \\ E_{uv}(0,0) & E_{vv}(0,0) \end{bmatrix}\begin{bmatrix} u \\ v \end{bmatrix}$$

and we can expand each term fully to get the final Taylor expansion approximation of E near $(0,0)$:[a]

$$E(u,v) \approx \frac{1}{2}\begin{bmatrix} u & v \end{bmatrix}\begin{bmatrix} \sum_{x,y} w' I_x(x,y)^2 & \sum_{x,y} w' I_x(x,y) I_y(x,y) \\ \sum_{x,y} w' I_x(x,y) I_y(x,y) & \sum_{x,y} w' I_y(x,y)^2 \end{bmatrix}\begin{bmatrix} u \\ v \end{bmatrix}$$

[a] I don't have enough horizontal space for the entire expression, actually, so I substitute $w' = 2w(x,y)$ for compactness-sake.

Properties of the 2nd Moment Matrix

With our Taylor expansion, the function $E(u,v)$ is being locally approximated by a quadratic. The function $E(u,v)$ is a surface, and one of its "slices" would look like this:

$$\begin{bmatrix} u & v \end{bmatrix}\mathbf{M}\begin{bmatrix} u \\ v \end{bmatrix} = \text{some constant}$$

$$\sum I_x^2 u^2 + 2\sum I_x I_y uv + \sum I_y^2 v^2 = k$$

You may (or may not... I sure didn't) notice that this looks like the equation of an ellipse in uv-space:

$$au^2 + buv + cv^2 = d$$

Thus our approximation of E is a series of these ellipses stack on top of each other, with varying values of k.

Consider a case for $\mathbf{M}$ in which the gradients are horizontal xor vertical, so I_x and I_y are never non-zero at the same time. That would mean $\mathbf{M}$ looks like:

$$\mathbf{M} = \sum_{xy} w(x,y)\begin{bmatrix} I_x^2 & 0 \\ 0 & I_y^2 \end{bmatrix} = \begin{bmatrix} \lambda_1 & 0 \\ 0 & \lambda_2 \end{bmatrix}$$

This wouldn't be a very good corner, though, if $\lambda_2 = 0$, since it means all of the change was happening in the horizontal direction, and similarly for $\lambda_1 = 0$. These would be *edges* more-so than *corners*. This might trigger a lightbulb of intuition:

> *If either λ is close to 0, then this is **not** a good corner, so look for areas in which both λs are large!*

With some magical voodoo involving linear algebra, we can get the diagonalization of $\mathbf{M}$ that hints at this information:

$$\mathbf{M} = \mathbf{R}^{-1}\begin{bmatrix} \lambda_1 & 0 \\ 0 & \lambda_2 \end{bmatrix}\mathbf{R}$$

Where now the λs are the eigenvalues. Looking back at our interpretation of each slice as an ellipse, $\mathbf{R}$ gives us the orientation of the ellipse and the λs give us the lengths of its major and minor axes.

A slice's "cornerness" is given by a having large and proportional λs. More specifically,

$$\begin{cases} \lambda_1 \gg \lambda_2 \text{ or } \lambda_2 \gg \lambda_1, & \text{edge} \\ \lambda_1, \lambda_2 \approx 0, & \text{flat region} \\ \lambda_1, \lambda_2 \gg 0 \text{ and } \lambda_1 \sim \lambda_2, & \text{corner} \end{cases}$$

Rather than finding these eigenvalues directly (which is expensive on 80s computers because of the need for `sqrt`), we can calculate the "cornerness" of a slice indirectly by using the determinant and the trace of $\mathbf{M}$. This is the **Harris response function**:

$$R = \det \mathbf{M} - \alpha \operatorname{trace}(\mathbf{M})^2 = \lambda_1 \lambda_2 - \alpha(\lambda_1 + \lambda_2)^2$$

Empirically, $\alpha \in [0.04, 0.06]$. The classification breakdown is now:

$$\begin{cases} |R| \approx 0, & \text{flat region} \\ R \ll 0, & \text{edge} \\ R \gg 0, & \text{corner} \end{cases}$$

9.1.2 Harris Detector Algorithm

The culmination of the understanding of the 2nd moment matrix in our approximation of the error function at $(0, 0)$, specifically the response function R, leads to the **Harris detector algorithm**:

Algorithm 9.1: The basic Harris detector algorithm.

Input: An image, I
Result: A set of "interesting" corner-like locations in the image.

Compute Gaussian derivatives at each pixel.
Compute $\mathbf{M}$ in a Gaussian window around each pixel.
Compute R.
Threshold R.
Find the local maxima of R via Non-Maximal Suppression.

9.1.3 Improving the Harris Detector

Studies have shown us that the Harris detector is consistent (invariant) under changes involving *rotation* and *intensity* (as long as they're additive or multiplicative changes and you

handle thresholding issues), but they are **not** invariant under scaling. Empirically, we've seen that scaling by a factor of 2 reduces repeatability (i.e. the measurement of finding the same features) by ~80% (see the Harris line in Figure 9.4).

Intuitively, we can imaging "zooming in" on a corner and applying the detector to parts of it: each region would be treated like an edge! Thus, we want a **scale invariant detector**.

If we scaled our region by the same amount that the image was scaled, we wouldn't have any problems. Of course, we don't know how much the image was scaled (or even if it *was* scaled). How can we choose corresponding regions *independently* in each image? We need to design a region function that is scale invariant.

A scale invariant function is one that is consistent across images given changes in region size. A naïve example of a scale invariant function is average intensity. Given two images, one of which is a scaled version of the other, there is *some* region size in each in which the average intensity is the same for both areas. In other words, the average intensity "peaks" in some place(s) and those peaks are correlated based on their independent scales.

A *good* scale invariant function has just one stable peak. For most images, a good function is one that responds well to contrast (i.e. a sharp local intensity change... remember The Importance of Edges?) We can apply the Laplacian (see Equation 3.6) of the Gaussian filter (the "Mexican hat operator," as Prof. Bobick puts it). But to avoid the 2nd derivative nonsense, we can revisit an old friend from Blending: the difference of Gaussians:

$$\text{DoG} = G(x, y, k\sigma) - G(x, y, \sigma)$$

which gives an incredibly similar result to the "true" 2nd derivative. Conveniently, both of these kernel operators are entirely invariant to both rotation and scale.

SIFT Detector

This leads to a technique called **SIFT**: **s**cale **i**nvariant **f**eature **t**ransform.

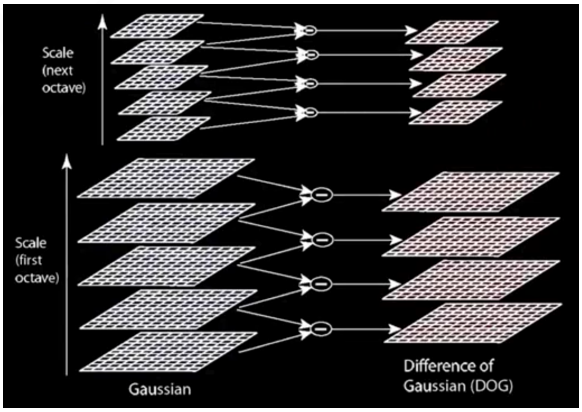


Figure 9.3: Creating a difference of Gaussian pyramid. The images in each octave are 2x blurrier than their previous one in the set, and each octave is 1/2 the scale of the previous octave.

The general idea is that we want to find robust extrema in both space *and* scale. We can imagine "scale space" as being different scaled versions of the image, obtained via interpolation of the original.

For each point in the original, we can say it has neighbors to the left and right (our traditional understanding) as well as neighbors "up and down" in scale space. We can use that scale space to create a pyramid of the differences of Gaussians (or, as we previously called it, a Laplacian pyramid), then eliminate the maxima corresponding to edges; this just leaves the corners. Note that this is a completely different method of corner detection; we aren't hanging out with Harris anymore!

Recall that a Laplacian pyramid is a "stack" of images; each item in the stack is difference of Gaussians at various scales, as in Figure 9.3. Each point in an image is compared to its 8 local neighbors as well as its 9 neighbors in the images "above" and "below" it; if you're a maxima of all of those points, you're an extremum! Once we've found these extrema, we threshold the contrast and remove extrema on edges; that results in detected feature points that are robust to scaling.

Harris-Laplace Detector

There is a scale-invariant detector that uses *both* of the detectors we've described so far. The **Harris-Laplace detector** combines the difference of Gaussians with the Harris detector: we find the local maximum of the Harris corner in space and the Laplacian in scale.

Comparison

The robustness of all three of these methods under variance in scale is compared in Figure 9.4. As you can see, the Harris-Laplace detector works the "best," but take these metrics with a grain of salt: they were published by the inventors of Harris-Laplace.

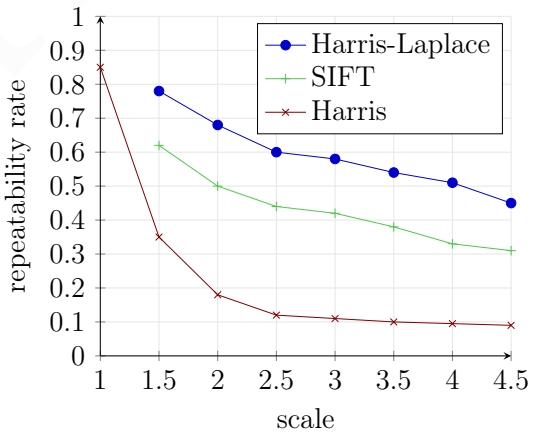


Figure 9.4: A comparison between different feature-point detectors under variations in scale.

9.2 Matching Interest Points

We now have a variety of tools to *detect* interest points, but how do we *match* them? To do that, we need a **descriptor**: we need to "describe" the point in one image (perhaps based on its neighboring points) and find a matching description in the right image.

We described our ideal descriptor in the introduction (you can jump back if you want): they should be *almost* the same in both image, i.e. invariant, as well as *distinctive*.

Being distinctive is challenging: since a feature point's neighborhood will change from one point to the next, we need some level of flexibility in how we match them up, but we don't want to be *too* flexible and match other feature points that are also similar but lie elsewhere. In a sense, we have a tradeoff here between how invariant we want to be and how distinctive we can be.

Simple Solution? Is there something simple we can do here? We have the feature points from our Harris Detector Algorithm, can't we just use correlation on each feature point window on the other image and choose the peak (i.e. something much like Template Matching)?

Unfortunately, correlation is not rotation-invariant, and it's fairly sensitive to photometric changes. Even normalized correlation is sensitive to non-linear photometric changes and slight geometric distortions as well.

Furthermore, it's slow: comparing one feature to *all other* feature points is not an ideal solution from an algorithmic perspective ($O(n^2)$).

We'll instead introduce the **SIFT descriptor** to solve these problems.

9.2.1 SIFT Descriptor

We've already discussed the SIFT Detector, but another – and more popular – part of Lowe's work in SIFT was the SIFT descriptor. The idea behind the descriptor was to represent the image content as a "constellation" of local features that are invariant to translation, scale, rotation, and other imaging parameters. For example, in Figure 9.5 we can see the same features being found in both images, and the SIFT features themselves are different than each of the individual features; they've been normalized to match.

We can imagine that if we found enough of these matching features, we'd be able to get some insight into the object they're a part of as a whole. We'd be able to say something like, "Oh, these 4 features are part of some object that's been rotated *[this much]* in the other image."

The basic approach we'll run through is, given our set of keypoints (or features or interest points or ...), we assign an orientation to each one (or some group). Then, we can build a description for the point based on the assigned orientation.

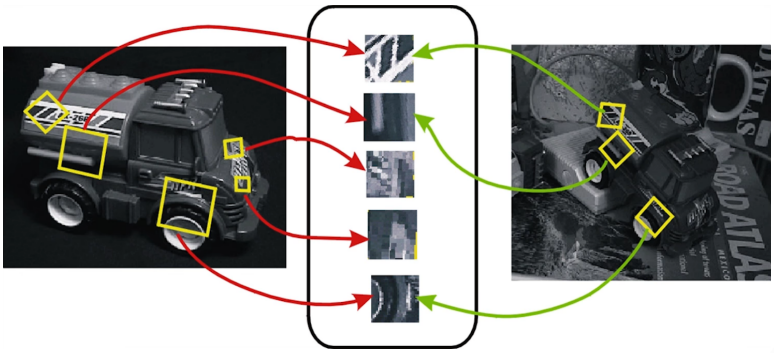


Figure 9.5: An example of the SIFT descriptor finding transformation-invariant features in two images.

Orientation Assignment

We want to compute the "best" orientation for a feature. Handily enough, the base orientation is just the dominant direction of the gradient. Intuitively, we know that *some* rotation of the feature patch will result in a maximum gradient magnitude. So if we rotate all features towards the direction of their maximum gradient, we can compare their similarity directly.

More concretely, to localize orientation to a feature we create a histogram of local gradient directions and their resulting magnitudes at a selected scale – 36 bins (for 36 possible rotations, an arbitrary choice). The canonical orientation is assigned to the peak of the smoothed histogram. Thus, each feature point has some properties: its (x, y) coordinates, and an *invariant* scale and orientation.

Keypoint Description

We want a descriptor that, again, is highly distinctive and as invariant as possible to photometric changes. First, we normalize: rotate a keypoint's window based on the standard orientation, then scale the window size based on the the keypoint's scale.

Now, we create a **feature vector** based upon:

- a histogram of gradients, which we determined previous when finding the orientation
- weighed by a centered Gaussian filter, to appropriately value the center gradients more

We take these values and create a 4×4 grid of bins for each. In other words, the first bin would contain the weighted histogram for the top-left corner of the window, the second for the next sector to the right, and so on.

Minor Details There are a lot of minor tweaks here and there that are necessary to make SIFT work. One of these is to ensure smoothness across the entire grid: pixels can affect multiple bins if their gradients are large enough. This prevents abrupt changes across bin boundaries. Furthermore, to lower the impact of highly-illumined areas (whose gradients would dominate a bin), they encourage clamping the gradient to be ≤ 0.2 after the rotation normalization. Finally, we normalize the entire feature vector (which is 16× the window

size) to be magnitude 1.

Evaluating the Results

I recommend watching this lecture and the following one to see the graphs and tables that evaluate how well the SIFT descriptor works under a variety of transformations to an image.

TODO: *Replicate the graphs, I guess.*

9.2.2 Matching Feature Points

We *still* haven't address the problem of actually matching feature points together, but we're making progress. We now have our feature vector from the SIFT Descriptor for our images, and can finally tackle correlating features together.

Nearest Neighbor

We could, of course, use a naïve nearest-neighbor algorithm and compare a feature in one image to all of the features in the other. Even better, we can use the *kd*-tree algorithm[1] to find the *approximate* nearest neighbor. SIFT modifies the algorithm slightly: it implements the best-bin-first modification by using a heap to order bins by their distance from the query point. This gives a 100-1000x speedup and gives the correct result 95% of the time.[2]

Wavelet-Based Hashing

An alternative technique computes a short 3-vector descriptor from the neighborhood using a **Haar wavelet**.[3] Then, you quantize each value into 10 overlapping bins, giving 10^3 possible entries. This greatly reduces the amount of features we need to search through.

Locality-Sensitive Hashing

The idea behind this technique, and locality-sensitive hashing in general, is to construct a hash function that has similar outputs for similar inputs. More rigorously, we say we craft

[1] Unlike in lecture, understanding of this algorithm isn't assumed ☺. The *kd*-tree algorithm partitions a k-dimensional space into a tree by splitting each dimension down its median. For example, given a set of (x, y) coordinates, we would first find the median of the x coordinates (or the ys… the dimension is supposed to be chosen at random), and split the set into two piles. Then, we'd find the median of the y coordinates *in each pile* and split them down further. When searching for the nearest neighbor, we can use each median to quickly divide the search space in half. It's an *approximate* method, though: sometimes, the nearest neighbor will lie in a pile across the divide. For more, check out this video which explains things succinctly, or the Wikipedia article for a more rigorous explanation.

[2] For reference, here is a link to the paper: Indexing w/o Invariants in 3D Object Recognition. See Figure 6 for a diagram "explaining" their modified *kd*-tree algorithm.

[3] Per Wikipedia, Haar wavelets are a series of rescaled square-wave-like functions that form a basis set. A wavelet (again, per Wikipedia) is an oscillation that begins at zero, increases, then decreases to zero (think of $\sin(\theta) \in [0, \pi]$). We'll briefly allude to these again descriptors when we discuss features in the Viola-Jones Face Detector much later in chapter 14.

a hash function $g : R^d \to U$ such that for any two points p and q (where D is some distance function):

$$D(p, q) \leq r \Rightarrow \Pr[g(p) = g(q)] \gg 0$$
$$D(p, q) > cr \Rightarrow \Pr[g(p) = g(q)] \sim 0$$

In English, we say that if the distance between p and q is high, the probability of their hashes being the same is "small"; if the distance between them is low, the probability of their hashes being the same is "not so small."

If we can construct such a hash function, we can jump to a particular bin and find feature points that are similar to a given input feature point and, again, reduce our search space down significantly.

9.2.3 Feature Points for Object Recognition

We can use our feature descriptors to find objects in images. First, we have the training phase: given a training image,[4] we extract its outline, then compute its keypoints.

During recognition, we search for these keypoints to identify possible matches. Out of these matches, we identify *consistent* solutions, such as those that hold under an affine transformation. To identify an affine transformation, we need 3 corresponding points.[5]

Because we only need these 3 points, we only need 3 features to match across images! This makes the recognition resilient to occlusion of the objects.

9.3 Coming Full Circle: Feature-Based Alignment

Let's have a cursory review of the things we've been discussing since chapter 8.

We discussed associating images together given sets of *known* points, essentially finding the transformations between them. We did this using The Power of Homographies and aligned images together when Creating Panoramas and doing Image Rectification; we also used the Fundamental Matrix for simplifying our epipolar geometry and doing things like stereo image rectification.

This (often) amounted to solving a system of equations in the form $\mathbf{Ax} = \mathbf{y}$ via the methods like the SVD trick.

In this chapter, we've developed some techniques to identify "useful" points (features) and approximately match them together. These now become our "known points," but the problem is that they aren't *truly* known. They're estimations and have both minor inaccuracies and completely incorrect outliers. This makes our techniques from before much less reliable.

[4] A good training image would be the object, alone, on a black background.

[5] Recall that an affine transformation has 6 unknowns: 2 for translation, 2 for rotations and scaling, and 2 for shearing. This is covered more in Image-to-Image Projections.

Instead, we're now approaching finding transformations between images as a form of **model fitting**. Essentially, what we have is a series of *approximately* corresponding points, and we want to find the transformation that fits them the best. We've seen this before in chapter 4 when discussing line and circle fitting by voting for parameters in Hough Space.

At its heart, model fitting is a procedure that takes a few concrete steps:

- We have a function that returns a predicted data set. This is our **model**.
- We have a function that computes the difference between our data set and the model's predicted data set. This is our **error function**.
- We tune the model parameters until we minimize this difference.

We already have our model: a transformation applied to a group of feature points. We already have our data set: the "best matches" computed by our descriptor (we call these **putative matches**). Thus, all that remains is a robust error function.

9.3.1 Outlier Rejection

Our descriptor gives the qualitatively "best" match for a particular feature, but how can we tell if it's actually a *good* match?

Remember the basic principle we started with when Finding Disparity for stereo correspondence? We used the sum of square differences between a patch in one image and all of the patches in the other image, choosing the one with the smallest "distance." We could do the same thing here, comparing the patch around two matching interest points and thresholding them, discarding matches with a large difference.

This begs the question: how do we choose a threshold? Thankfully, someone has done a thorough empirical investigation to determine the probability of a match being correct relative to its "corresponding patch" error.

Nearest Neighbor Error

Just checking the best matching feature patch has a low error for correct matches, but it's *also* low for a decent number of incorrect matches. Instead, we can do something a little more clever.

Intuitively, we can imagine that in the incorrect case, the matching patch is essentially a random patch in the image that happened to have the best matching value. What about the *next* best match for that feature? Probably, again, a different random patch in the image! Their errors probably don't differ by much. In the correct case, though, the best match is probably *really* strong, and its next best match is probably a random patch in the image. Thus, their errors would differ by a **lot**.

This is backed up by empirical evidence: comparing the ratio of error between the first-nearest-neighbor and the second-nearest-neighbor (i.e. e_{NN_1}/e_{NN_2}) across correct and incorrect matches shows a stark difference between the cases. Specifically, setting a threshold of 0.4

on the error ratio (meaning there's a 60% difference between the best and next-best match) would eliminate nearly all of the incorrect matches.

Of course, we'll be throwing out the baby with the bathwater a bit, since a lot of correct matches will also get discarded, but we still have enough correct matches to work with.

Unfortunately, we still do have lots of outliers. How can we minimize their impact on our model?

9.3.2 Error Functions

Recall, again, the Hough voting process for line fitting. We can imagine the voting process as a way of minimizing the distance between a set of points and the line that fits them. In essence, what we were doing was very similar to the least squares approximation to a linear system; the figure below shows how we were minimizing the vertical error between points and our chosen line.

Our error function in this case is the vertical distance between point (x_i, y_i) and the line, for all points. $mx_i + b$ is the "real y" at that x value via the true equation of the line:

$$E = \sum_{i=1}^{n} (y_i - (mx_i - b))^2$$

Solving this minimization again leads us to the standard least squares approximation for a linear system, as discussed in detail in **??**.

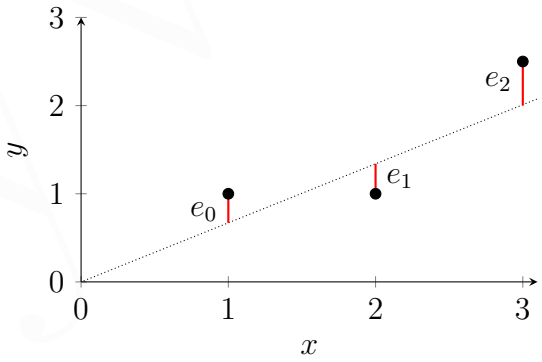


Figure 9.6: Fitting a line to a set of points, with the errors in red.

The problem with vertical least-squares error approximation is that it causes huge discrepancies as lines get more vertical (and fails entirely for completely vertical lines). Consider the error in Figure 9.7: the point is very close to the line, but the error is very high.

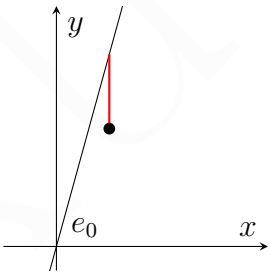


Figure 9.7: Measuring vertical error over-emphasizes errors in steep lines.

What would be a better error metric is the *perpendicular* distance between the point and the line. In Projective Geometry, we talked about how a line in the form $ax + by = d$ has a normal vector $\hat{\mathbf{n}} = \begin{bmatrix} a & b \end{bmatrix}$ and it d away from the origin at its closest. With that, we can instead define a different error function:

$$E = \sum_{i=1}^{n} (ax_i + by_i - d)^2$$

To minimize this, which we do via vector calculus and algebraic ma-

nipulation, we come up with the homogeneous equation:[6]

$$\frac{dE}{d\mathbf{h}} = 2(\mathbf{U}^T\mathbf{U})\mathbf{h} = \mathbf{0}$$

Where $\mathbf{h} = \begin{bmatrix} a & b \end{bmatrix}$ and $\mathbf{U}$ is a matrix of the differences of averages for each point (i.e. $\forall i\ x_i - \overline{x}$). Amazingly, we can solve this with the SVD trick we've seen time and time again.

This seems like a strong error function, but it has a fundamental assumption. We assume that the noise to our line is corrupted by a Gaussian noise function perpendicular to the line.

Scrrrrr. Hold up, **what**?

We're just assuming that the errors in our data set (i.e. the noise) happens to follow a standard bell curve, with most of the noisy data being close to the "true" model and few correct points being far away.

Hm, well when you put it that way, it sounds more reasonable.

Onward!

To make our assumption mathematical, we are saying that our noisy point (x, y) is the true point on the line perturbed along the normal by some noise sampled from a zero-mean Gaussian with some σ:

$$\begin{bmatrix} x \\ y \end{bmatrix} = \underbrace{\begin{bmatrix} u \\ v \end{bmatrix}}_{\text{true point}} + \underbrace{\epsilon}_{\text{noise}} \underbrace{\begin{bmatrix} a \\ b \end{bmatrix}}_{\hat{\mathbf{n}}}$$

We say that this is our underlying **generative model**:[7] a Gaussian perturbation.

Of course, this comes with a natural pitfall: the model is extremely non-robust to non-Gaussian noise. Outliers to the generative model have a *huge* impact on the final model because *squared* error heavily penalizes outliers.

How does this tie back to our original discussion? Well, even though we tried to minimize outliers in the previous sections, there still will be some. Those outliers will have a massive impact on our approximation of the transformation between images if we used the sum of squared errors.

Thus, we need something better. We need a **robust estimator**. The general approach of a robust estimator is to minimize:

$$\sum_i \rho(r_i(x_i, \theta); \sigma)$$

[6] I'm trying not to bore you with the derivation, but you can refer to the original lecture for the details.
[7] i.e. a model that generates the noise

Here, r finds the "residual" of the i^{th} point with respect to the model parameters, θ. This is our "distance" or error function that measures the difference between the model's output and our test set. ρ is a robust function with a scale parameter, σ; the idea of ρ is to not let outliers have a large impact on the overall error. An example of a robust error function is: $\rho(u;\sigma) = \frac{u^2}{\sigma^2+u^2}$

For a small residual u (i.e. small error) relative to σ, this behaves much like the squared distance: we get $\approx {u^2}/{\sigma^2}$. As u grows, though, the effect "flattens out": we get ≈ 1.

Of course, again, this begs the question of how to choose this parameter. What makes a good scale, σ? Turns out, it's fairly scenario-specific. The error function we defined above is very sensitive to scale.

It seems like we keep shifting the goalposts of what parameter we need to tune, but in the next section we'll define an error function that still *has* a scale, but is much less sensitive to it and enables a much more robust, general-purpose solution.

9.3.3 RANSAC

The **RANSAC error function**, or **ran**dom **sa**mple **c**onsensus error function, relies on the (obvious) notion that it would be easy to fit a model if we knew which points belonged to it and which ones didn't. We'll return to the concept of "consistent" matches: which sets of matches are consistent with a particular model?

Like in the Hough Transform, we can count on the fact that wrong matches will be relatively random, whereas correct matches will be consistent with each other. The basic main idea behind RANSAC is to loop over a randomly-proposed model, find which points belong to it (**inliers**) and which don't (**outliers**), eventually choosing the model with the most inliers.

For any model, there is a minimal set of s points that can define it. We saw this in Image-to-Image Projections: translations had 2, homographies had 4, the fundamental matrix has 8, etc. The general RANSAC algorithm is defined in algorithm 9.2 below.

How do we choose our distance threshold t, which defines whether or not a point counts as an inlier or outlier of the instantiated model? That will depend on the way we believe the noise behaves (our generative model). If we assume a Gaussian noise function like we did previously, then the *distance* d to the noise is modeled by a **Chi distribution**[8] with k degrees of freedom (where k is the dimension of the Gaussian). This is defined by:

$$f(d) = \frac{\sqrt{2}e^{-\frac{d^2}{2\sigma^2}}}{\sqrt{\pi}\sigma},\ d \geq 0$$

We can then define our threshold based on what percentage of inliers we want. For example, choosing $t^2 = 3.84\sigma^2$ means there's a 95% probability that when $d < t$, the point is an inlier.

[8] The Chi distribution is of the "square roots of the sum of squares" of a set of independent random variables, each following a standard normal distribution (i.e. a Gaussian) (per Wikipedia). "That" is what we were looking to minimize previously with the perpendicular least squares error, so it makes sense!

ALGORITHM 9.2: General RANSAC algorithm.

Input: F, the set of matching feature points.
Input: T, a "big enough" consensus set, ***or*** N, the number of trials.
Result: The best-fitting model instance.

$C_i \leftarrow \{\}$

```
/* Two versions of the algorithm:  threshold or iteration count */
```
while $|C_i| < T$ *or* $N > 0$ **do**

```
    /* Sample s random points from the set of feature points.  */
```
    $p = \text{SAMPLE}(F; s)$

```
    /* Instantiate the model from the sample.  */
```
    $M = \text{MODEL}(p)$

    $C_i = \{p_i \in F \mid M(p_i) > t\}$

    $N = N - 1$ `// if doing the iteration version...`

end

return M

Now, how many iterations N should we perform? We want to choose N such that, with some probability p, at least one of the random sample sets (i.e. one of the C_is) is completely free from outliers. We base this off of an "outlier ratio" e, which defines how many of our feature points we expect to be bad.

Let's solve for N:

- s – the number of points to compute a solution to the model
- p – the probability of success
- e – the proportion of outliers, so the % of inliers is $(1 - e)$.
- $\Pr[\text{sample set with } \textit{all} \text{ inliers}] = (1 - e)^s$
- $\Pr[\text{sample set with at least one outlier}] = (1 - (1 - e)^s)$
- $\Pr[\text{all } N \text{ samples have outliers}] = (1 - (1 - e)^s)^N$
- But we want the chance of all N having outliers to be really small, i.e. $< (1-p)$. Thus, we want $(1 - (1 - e)^s)^N < (1 - p)$.

Solving for N gives us... *drumroll*...

$$N > \frac{\log(1 - p)}{\log\left(1 - (1 - e)^s\right)}$$

The beauty of this probability relationship for N is that it scales incredibly well with e. For

example, for a *99% chance* of finding a set of $s = 2$ points with no outliers, with an $e = 0.5$ outlier ratio (meaning *half* of the matches are outliers!), we only need **17 iterations**.

Some more RANSAC values are in Table 9.1. As you can see, N scales relatively well as e increases, but less-so as we increase s, the number of points we need to instantiate a model.

	Proportion of outliers, e						
s	5%	10%	20%	25%	30%	40%	50%
2	2	3	5	6	7	11	17
3	3	4	7	9	11	19	35
4	3	5	9	13	17	34	72
5	4	6	12	17	26	57	146
6	4	7	16	24	37	97	293
7	4	8	20	33	54	163	588
8	5	9	26	44	78	272	1177

Table 9.1: Values for N in the RANSAC algorithm, given the number of model parameters, s, and the proportion of outliers.

Finally, when we've found our best-fitting model, we can recompute the model instance using all of the inliers (as opposed to just the 4 we started with) to average out the overall noise and get a better estimation.

Adapting Sample Count The other beautiful thing about RANSAC is that the number of features we found is completely irrelevant! All we care about is the number of model parameters and our expected outlier ratio. Of course, we don't know that ratio *a priori*[9], but we can adapt it as we loop. We can assume a worst case (e.g. $e = 0.5$), then adjust it based on the *actual* amount of inliers that we find in our loop(s). For example, finding 80% inliers then means $e = 0.2$ for the next iteration. More formally, we get algorithm 9.3.

Example 9.1: **Estimating a Homography**

Just to make things a little more concrete, let's look at what estimating a homography might look like. Homographies need 4 points, and so the RANSAC loop might look something like this:

1. Select 4 feature points at random.
2. Compute their exact homography, $\mathbf{H}$.
3. Compute the inliers in the entire set of features where $SSD(\mathbf{p}'_i, \mathbf{H}\mathbf{p}_i) < \epsilon$.
4. Keep the largest set of inliers.
5. *Re*compute the least squares $\mathbf{H}$ estimate on all of the inliers.

[9] fancy Latin for "from before," or, in this case, "in advance"

Algorithm 9.3: Adaptive RANSAC algorithm.

Input: F, the set of matching feature points.
Input: (s, p, t), the # of model parameters, probability of success, and inlier threshold.
Result: The best-fitting model instance.

$N = \infty, c = 0, e = 1.0$ `// c is the number of samples`

while $N > c$ **do**
  $p = \text{Sample}(F; s)$
  $M = \text{Model}(p)$
  $C_i = \{\forall x_i \in F \mid M(x_i) > t\}$ `// find the inliers`
  $e_0 = 1 - \frac{\|C_i\|}{\|F\|}$
  if $e_0 < e$ **then**
    $e = e_0$
    $N = \frac{\log(1-p)}{\log(1-(1-e)^s)}$
  end
  $c = c + 1$
end

return C_i, $\text{Model}(C_i)$

Benefits and Downsides

RANSAC is used a lot in the wild because of its many benefits. It's a simple and general solution that can be applied to many different models; we can have a much larger number of parameters than, for example, the generalized Hough transform, and they're easier to choose. Finally, and probably most-importantly, it's robust to a large number of outliers; this takes the load off of feature point detection.

On the other hand, it does have some pitfalls. As we saw, computational time increases rapidly as the number of parameters increases; as we approach $s > 10$, things get a little hairy. Furthermore, it's not ideal at fitting multiple models simultaneously; thresholding becomes much more finicky. Finally, its biggest downside is that it's *really* not good for approximate models. If, for example, you have something that's "kind of a plane," RANSAC struggles; it needs a precise, exact model to work with.

9.4 Conclusion

We've made some incredible strides in automating the concepts we described in the chapter on Multiple Views. Given some images, we can make good guesses at corresponding points then fit those to a particular model (such as a fundamental matrix) with RANSAC and determine the geometric relationship between images. These techniques are pervasive in computer vision and have plenty of real-world applications like robotics.

Potpourri: Photometry & Color Theory

> *"Drawing makes you look at the world more closely. It helps you to see what you're looking at more clearly. Did you know that?"*
>
> *I said nothing.*
>
> *"What colour's a blackbird?" she said.*
>
> *"Black."*
>
> *"Typical!"*
>
> — David Almond, *Skellig*

There are a number of topics in computer vision that that have a certain level of "assumed knowledge" that shouldn't be assumed so lightly. This chapter covers some of these ideas, bringing the world of light and color into a picture *(no pun intended)* that is otherwise dominated by nitty gritty mathematics that borderline belongs in a signal processing or machine learning course.

Though there is still some math involved, insights about the psychology of human vision, the phenomenon of "color," and the interactions between light and objects will provide a foundational perspective *(okay, that time it was intended)* on computer vision and enable us to think about images as more than a set of pixels.

10.1 Photometry

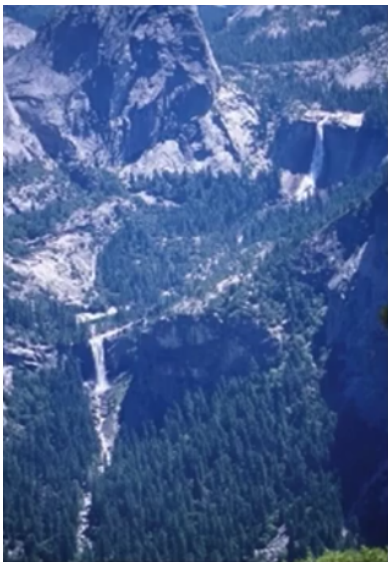
Photometry is concerned with measuring light in terms of its *perceived* brightness by the human eye. This is in contrast with *radio*metry, which is more concerned with the physical, absolute properties of light, such as its wavelength. We'll begin by discussing these properties and give a brief overview of how light interacts with objects to get an understanding of the physics, but computer vision is more concerned with *extracting* this model from images, and gaining an understanding of the scene based on how humans perceive it.

The physical models (and their virtual emulations) we'll be discussing describe various visual phenomena demonstrated in Figure 10.1.

(a) Shadows: how can we perceive (or infer) the fact that this isn't just a textured sidewalk, but rather the shadow of a tree?

(b) Reflections: it's entirely possible that the window isn't reflective at all, but rather has a nature scene painted on top of it.

(c) Refractions: reminiscent of Image Rectification, can we determine the true geometry that lies under the water?



(d) Scattering: is it excess light bouncing around the scene, or are the trees really that pale?

Figure 10.1: Our visual world contains a number of fascinating phenomena, some of which are demonstrated here. All of these follow a set of physical rules and can be emulated, but how can we determine when such phenomena are occurring, and not an inherent part of the scene?

10.1.1 BRDF

The appearance of a surface is largely dependent on three factors: the viewing angle, the surface's material properties, and its illumination. Before we get into these in detail, we need to discuss some terms from radiometry.

Radiance The energy carried by a ray of light is **radiance**, L. It's a measure of power per unit area, perpendicular to the direction of travel, per unit solid angle. That's super confusing, so let's dissect. We consider the power of the light falling into an area, then consider its angle, and finally consider the size of the "cone" of light that hits us. The units are watts per square meter per steradian: $Wm^{-2}sr^{-1}$.

Irradiance On the other hand, we have the energy *arriving* at a surface, E. In this case, we just have the power in a given direction per unit area (units: Wm^{-2}). Intuitively, light coming at a steep angle is less powerful than light straight above us, and so for an area receiving radiance $L(\theta, \varphi)$ coming in from some light source size $d\omega$, the corresponding irradiance is $E(\theta, \varphi) = L(\theta, \varphi) \cos\theta d\omega$.

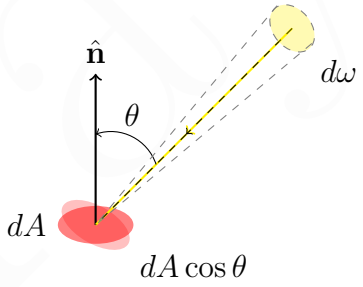


Figure 10.2: The radiance model.

If you don't understand this stuff super well, that's fine. I don't, hence the explanation is lacking. We'll be culminating this information into the **BRDF** – the **b**idirectional **r**eflectance **d**istribution **f**unction – and not worrying about radiance anymore. The BRDF is the ratio between the irradiance at the surface from the incident direction and the radiance from the surface in the viewing direction, as seen below in Figure 10.3. In other words, it's the percentage of the light reflected from the light coming in:

$$\text{BRDF: } f(\theta_i, \varphi_i; \theta_r, \varphi_r) = \frac{L_{\text{surface}}(\theta_r, \varphi_r)}{E_{\text{surface}}(\theta_i, \varphi_i)} \tag{10.1}$$

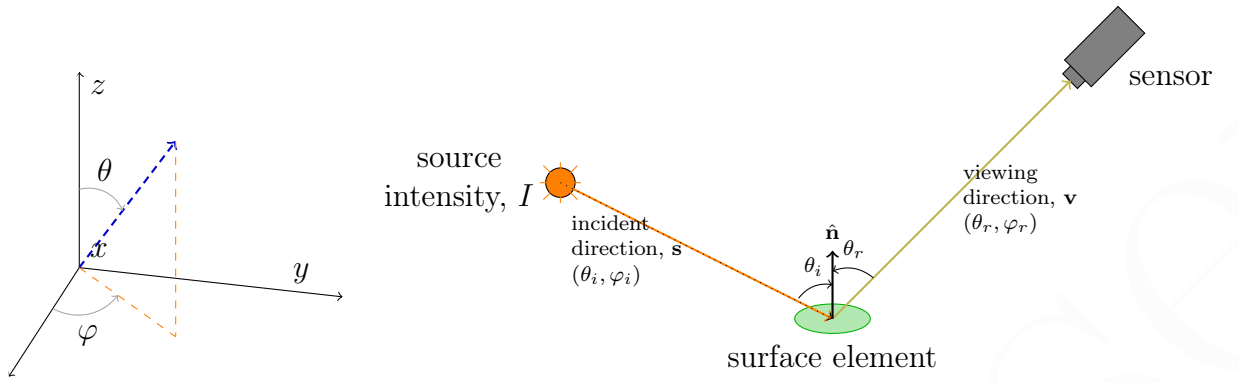


Figure 10.3: The model captured by the BRDF.

BRDFs can be incredibly complicated for real-world objects, but we'll be discussing two reflection models that can represent most objects to a reasonably accurate degree: **diffuse reflection** and specular reflection. With diffuse reflection, the light penetrates the surface, scattering around the inside before finally radiates out. Surfaces with a lot of diffuse reflection, like clay and paper, have a soft, matte appearance. The alternative is specular reflection, in which most of the light bounces off the surface. Things like metals have a high specular reflection, appearing glossy and have a lot of highlights.

With these two components, we can get a reasonable estimation of light intensity off of most objects. We can say that image intensity = body reflection + surface reflection. Let's discuss each of these individually, then combine them into a unified model.

Diffuse Reflection

First, we'll discuss a specific mathematical model for body (diffuse) reflection: the **Lambertian BRDF**. This guy Lambert realized a clever simplification: a patch on a surface with high diffusion (as before, things like clay and paper) looks equally bright from every direction. That means that the incident angle is the only one that matters; this is **Lambert's law**.

As expected, the more perpendicular the light source, the brighter the patch looks, but equally bright regardless of where you look at it from. Mathematically, we are independent of the $\mathbf{v}$ in Figure 10.3. Thus, the Lambertian BRDF is just a constant, known as the **albedo**:

$$f(\theta_i, \varphi_i; \theta_r, \varphi_r) = \rho_d$$

Then, the surface radiance is:

$$L = I \cos\theta_i = \rho_d I(\hat{\mathbf{n}} \cdot \hat{\mathbf{s}}) \tag{10.2}$$

Specular Reflection

Consider a perfect mirror. The only time you see the reflection of a particular object is when your viewing angle matches the incident angle of that object with the mirror. In other

words, it's visible when $\theta_i = \theta_v$ *and* when $\varphi_i = \pi + \varphi_v$, meaning the viewing direction is in the same plane as the incident direction. In other words, it's not sufficient to match up your "tilt," you also need to make sure you're in line with the point as well.

Thus, the BRDF for mirror reflections is a "double delta function:"

$$f(\theta_i, \varphi_i; \theta_r, \varphi_r) = \rho_s \delta(\theta_i - \theta_v)\delta(\varphi_i + \pi - \varphi_v)$$

where $\delta(x)$ is a simple toggle: 1 if $x = 0$ and 0 otherwise. Then, the surface radiance simply adds the intensity:

$$L = I\rho_s \delta(\theta_i - \theta_v)\delta(\varphi_i + \pi - \varphi_v)$$

We can simplify this with vector notation, as well. We can say that $\mathbf{m}$ is the "mirror direction," which is that perfect reflection vector for an incoming $\mathbf{s}$, and that *vech* is the "half-angle," i.e. the vector halfway between $\mathbf{s}$ and $\mathbf{v}$. Then:

$$L = I\rho_s \delta(\mathbf{m} - \mathbf{v}) = I\rho_s \delta(\mathbf{n} - \mathbf{h})$$

Most things aren't perfect mirrors, but they can be "shiny." We can think of a shiny, glossy object as being a blurred mirror: the light from a point now spreads over a particular area, as shown in Figure 10.4. We simulate this by raising the angle between the mirror and the viewing directions to an exponent, which blurs the source intensity more with a larger angle:

$$L = I\rho_s (\hat{\mathbf{m}} \cdot \hat{\mathbf{v}})^k$$

A larger k makes the specular component fall off faster, getting more and more like a mirror.

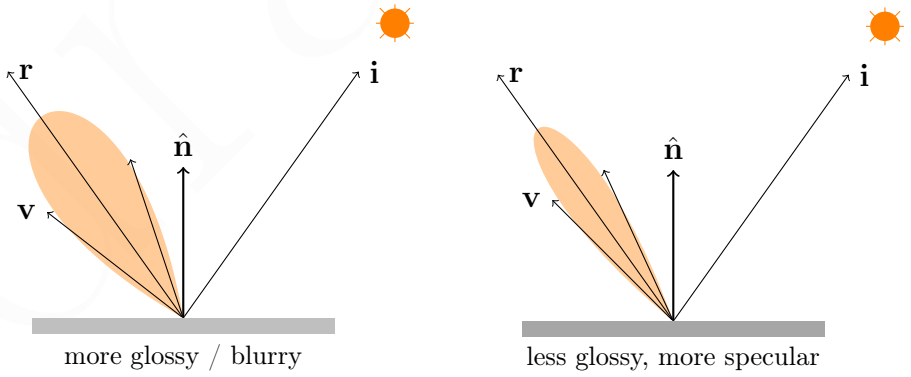


Figure 10.4: The spread of light as the specularity of a material changes.

10.1.2 Phong Reflection Model

We can approximate the BRDF of many surfaces by combining the Lambertian and specular BRDFs. This is called the **Phong reflection model**; it combines the "matte-ness" and "shininess" of an object.

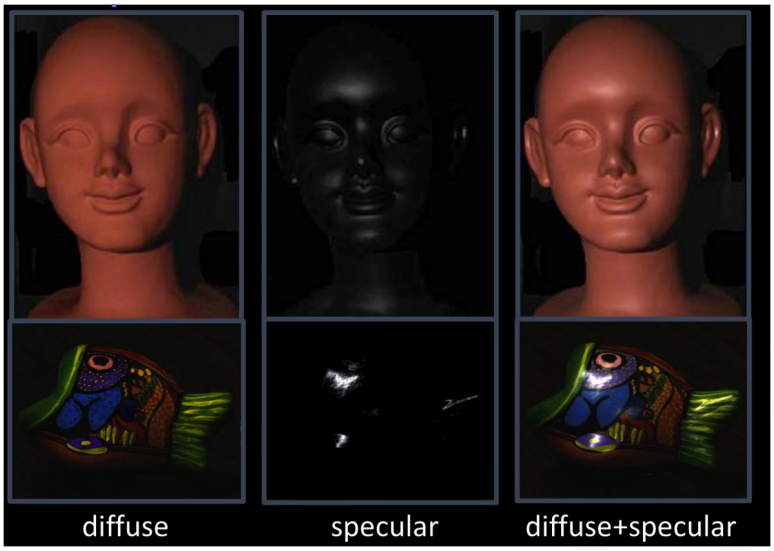


Figure 10.5: Combining the diffuse and specular components of an object using the Phong model.

10.1.3 Recovering Light

Remember the illusion in the Introduction, with the squares being the same intensity yet appearing different due to the shadow? In this section, we'll be fighting the battle of extracting lighting information and understanding its effect on a scene and differentiating it from the surface itself.

We will assume a Lambertian surface; recall, such a surface's "lightness" can be modeled as:

$$L = I\rho\cos\theta$$

If we combine the incoming light intensity and the angle at which it hits the surface into an "energy function", and represent the albedo as a more complicated "reflectance function," we get:

$$L(x, y) = R(x, y) \cdot E(x, y)$$

Of course, if we want to recover R from L (which is what "we see"), we can't without knowing the lighting configuration. The question is, then, how can we do this?

The astute reader may notice some similarities between this and our discussion of noise removal from chapter 2. In Image Filtering, we realized that we couldn't remove noise that was added to an image without knowing the exact noise function that generated it, especially if values were clipped by the pixel intensity range. Instead, we resorted to various filtering methods that tried to make a best-effort guess at the "true" intensity of an image based on some assumptions.

Likewise here, we will have to make some assumptions about our scene in order to make our "best effort" attempt at extracting the true reflectance function of an object:

1. **Light is slowly varying.**

   This is reasonable for our normal, planar world. Shadows are often much softer than the contrast between surfaces, and as we move through our world, the affect of the lighting around us does not change drastically within the same scene.

2. **Within an object, reflectance is constant.**

   This is a *huge* simplification of the real world, but is reasonable given a small enough "patch" that we treat as an object. For example, your shirt's texture (or possibly your skin's, if you're a comfortable at-home reader) is relatively the same in most places. This leads directly into the next assumption, which is that...

3. **Between objects, reflectance varies suddenly.**

   We've already taken advantage of this before when discussing edges: the biggest jump in variations of color and texture come *between* objects.

To simplify things even further, we'll be working exclusively with intensity; color is out of the picture for now. The model of the world we've created that is composed of these assumptions is often called *the Mondrian world*, after the Dutch painter, despite the fact that it doesn't do his paintings justice.

Retinex Theory

Dreamt up by Edwin Land (founder of Polaroid), **retinex** is a model for removing slow variations from an image. These slow variations would be the result of lighting under our Mondrian world; thus, removing them would give an approximation of the pure reflectance function.

There are many approaches to this. Consider the following:

- Take the logarithm of the entire thing:[1]

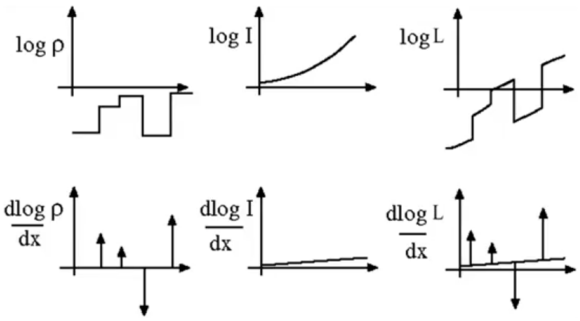
$$\log L(x, y) = \log R(x, y) + \log E(x, y)$$

- Run the result through a high-pass filter to keep high-frequency content, perhaps with the derivative.
- Threshold the result to remove small low frequencies.
- Finally, invert that to get back the original result (integrate, exponentiate).

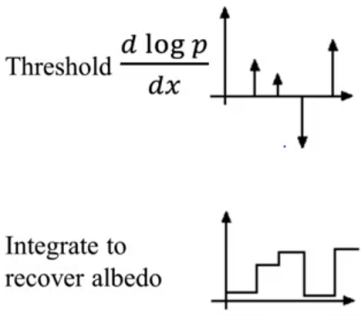
Let's walk through the steps for a 1D signal shown in Figure 10.6. Our "known" is the image L, and we want to recover ρ. We've established that we're assuming L is made up of ρ and some slowly-varying light intensity, I. Notice that $\log \rho < 0$ because recall that the albedo is a percentage of the light, and $\log n < 0$ where $n \in [0, 1]$.

The product of the logarithms (e.g. the image L) is the same stair pattern as in $\log \rho$ but

[1] Recall, of course, that the logarithm of a product is the sum of their logarithms.



(a) The breakdown of an image L into its component parts (ρ and I) under retinex assumptions.



(b) Recovering the albedo, ρ, from $\frac{d \log L}{dx}$.

Figure 10.6: An example of retinex applied to a 1D signal.

likewise slowly varying. Similarly, their derivatives combine the same way. In the bottom-right, we see the derivative of the logarithm of the image L.

If we threshold this $\frac{d \log L}{dx}$ based on its absolute value, we get a "flattened" version of it, which can be easily integrated to recover the initial albedo, ρ. Of course, we still have the unknown constant of integration meaning we can't know the "absolute" brightness of the objects, but this makes sense considering that even our human eye couldn't tell you the exact color of an object under, for an extreme example, a very bright light.

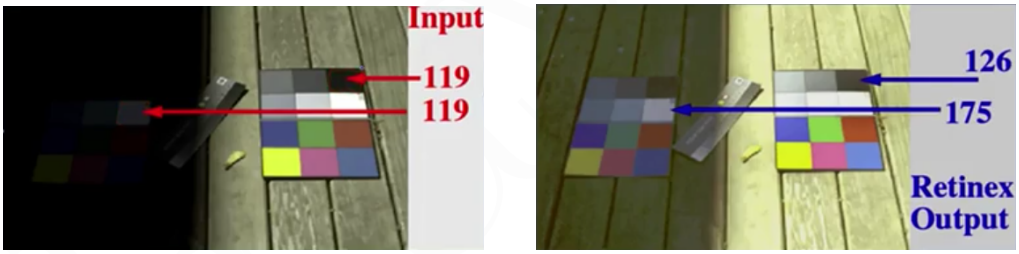


Figure 10.7: Using retinex to extract the influence of light on an image and recover the underlying colors. On the left, we see the initial illusion: the white square on the paper lying in the dark shadow has the same pixel intensity as the black square lying in the sun, but we obviously understand that they are different colors based on lighting. After retinex, the computer sees that as well.

Retinex processing is a little more complicated in 2 dimensions, but we won't get into the nitty-gritty. More interestingly, retinex falls apart quickly on 3D objects because the illumination of the object changes quickly due to edges. For the color changes in Figure 10.8, retinex would assume the changes are due to object variation rather than a change in illumination.

The human eye has no problem differentiating color under most lighting conditions: coal looks black regardless of whether or not you're holding it indoors, in a mine shaft, or outside on a sunny day. These are the principles of **color constancy** and **lightness constancy**: we can determine the hue and saturation of objects under different colors and intensities of lighting. Weirdly enough, though we can often determine the color of an object under a variety of lighting conditions and colors, we can't determine (as well) the color of the light

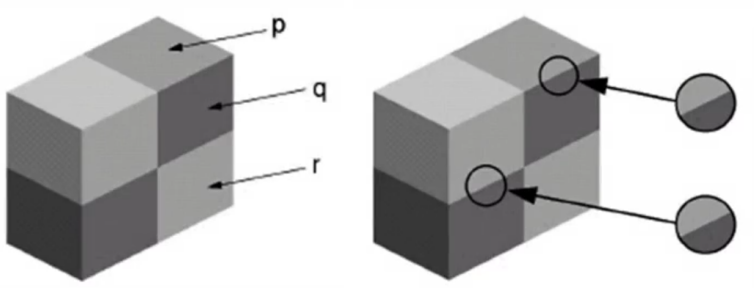


Figure 10.8: The human eye likely perceives the square pointed to by p as being darker than the square pointed to by r, yet they are the same intensity. Retinex falls apart when these sharp shape variations exist.

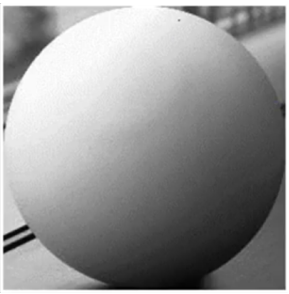
itself that's hitting the object.

There are a variety of ways to simulate this constancy computationally that photographers might be familiar with. You can assume that the average color in any given scene is gray, determining the lighting conditions based on violations of this assumptions. This is automating **white balancing**. A similar method is assuming that the brightest thing in the image is white: color variation on said thing is attributed to lighting conditions. For example, if the brightest thing was a light-blue spot, the lighting is assumed to be causing a light blue tint to the entire image. Another assumption is that an image should span the entire color gamut; again, any variations are attributed to lighting.

These methods work okay for photography, mostly because our brain will undo any inconsistencies and mistakes since it's much better at this than the computational method was. None of these methods work well for computer vision, though, with respect to actually understanding the scene and being able to analyze it effectively. It's a relatively unsolved problem.

10.1.4 Shape from Shading

This general area of work is focused on determining shape based on shading or illumination. For example, we can determine that the following object is a sphere based on the shading that occurs along its outer surface.



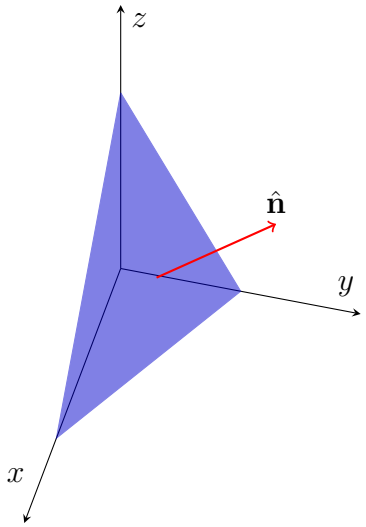
You can tell that the light source comes from the top-left corner. This is exactly the thing that artists take into account when trying to draw realistic objects.

Can we determine the relationship between intensity (which is a product of shading and lighting) and the underlying shape? Well as we've briefly discussed, there is a relationship between intensity and lighting in the reflectance function; we'll thus be building a reflectance map as part of our computational model for this process.

First, though, we need to begin with some notation. We assume that we start with a surface $z(x, y)$. As we move in the x and y direction, we can determine how z changes; this is the partial derivative. We'll define p and q to be these derivatives:

$$-\frac{\partial z}{\partial x} = p \qquad -\frac{\partial z}{\partial y} = q$$

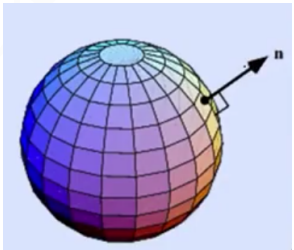
Now suppose we have a point on the surface. From it, we can define two tangent vectors in the direction of each partial derivative: $\mathbf{t}_x = \begin{bmatrix}1, 0, -p\end{bmatrix}^{\mathrm{T}}$ and $\mathbf{t}_y = \begin{bmatrix}0, 1, -q\end{bmatrix}^{\mathrm{T}}$. This is just a 1-unit step into the x and y directions off of the surface and measure how much z changes for that step (e.g. $\mathbf{t}_x$ steps in the x direction, so only x and z change).



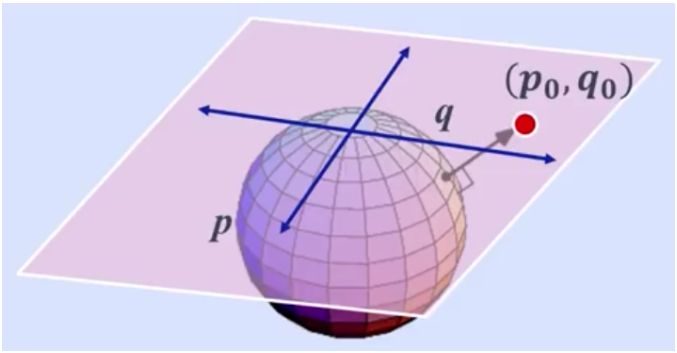
The surface normal, then, is just the cross product of these two vectors:

$$\hat{\mathbf{n}} = \frac{\mathbf{N}}{\|\mathbf{N}\|} = \frac{\mathbf{t}_x \times \mathbf{t}_y}{\|\mathbf{t}_x \times \mathbf{t}_y\|} = \frac{1}{\sqrt{p^2 + q^2 + 1}} \cdot \begin{bmatrix}p, q, 1\end{bmatrix}^{\mathrm{T}}$$

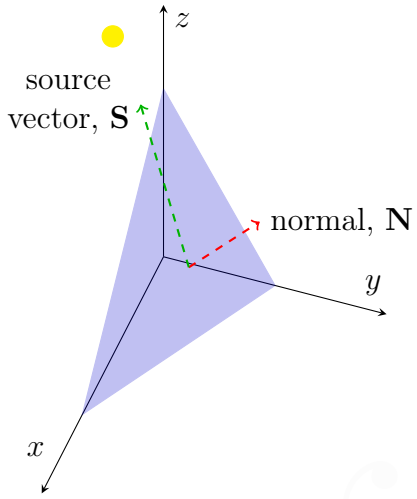
This is the algebraic relationship between the surface normal and p and q, but a geometric interpretation would also be useful. For that, we need to introduce the **Gaussian sphere**. The Gaussian sphere is a representation of *all possible surface normals*:



There's a mapping between the normal vectors on a surface and locations on the Gaussian sphere. Specifically, p and q are a projection into the Gaussian sphere. We can imagine a normal vector on the Gaussian sphere extending from the center to infinity in a particular direction. If we map the pq-plane as being at the top of the sphere, said vector will intersect the plane at a particular location, (p_0, q_0):



This pq-plane at $z = 1$ is also called **gradient space**. Thus we can define every normal vector by its (p, q) value. This notion will lead us to the reflectance map.



Suppose we have a vector to the incoming light source on a particular point on the surface. Much like the normal vector before, we have a projection of said vector into the pq-plane, so we have a (p, q) for $\mathbf{N}$ and likewise a (p_S, q_S) for $\mathbf{S}$. We can find the unit vectors of both just like we did before:

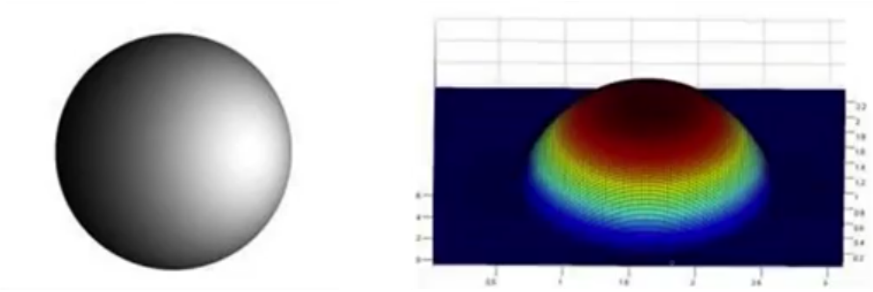
$$\hat{\mathbf{n}} = \frac{N}{\|N\|} = \frac{[p, q, 1]}{\sqrt{p^2 + q^2 + 1}}$$
$$\hat{\mathbf{s}} = \frac{S}{\|S\|} = \frac{[p_S, q_S, 1]}{\sqrt{p_S^2 + q_S^2 + 1}}$$

Let's suppose we have a Lambertian surface (same albedo everywhere), meaning the only thing that matters is the incident angle between $\hat{\mathbf{s}}$ and $\hat{\mathbf{n}}$ (recall Lambert's law), call that θ_i. We need $\cos\theta_i$ to determine the surface radiance as per (10.2): $L = I\cos\theta_i$.

We can represent that value via the dot product of our normals:

$$\cos\theta_i = \hat{\mathbf{n}} \cdot \hat{\mathbf{s}} = \frac{pp_S + qq_S + 1}{\sqrt{p^2 + q^2 + 1}\sqrt{p_S^2 + q_S^2 + 1}}$$

With all of that in mind, we can *finally* talk about shape from shading. Given an input image, can we recover the 3D shape of the object?



The **reflectance map** relates the image brightness $I(x, y)$ to the surface orientation (p, q) for a *given* source direction and a *given* surface reflectance.

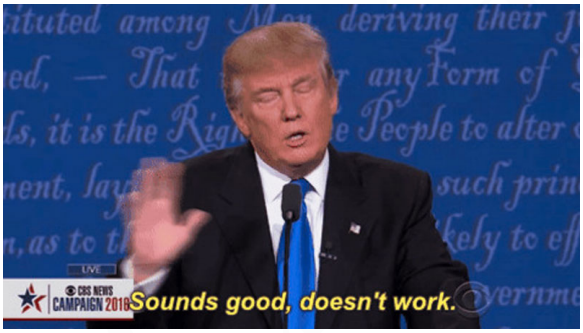
Let's begin again with the Lambertian case. Given a source brightness k and reflectance ρ, the brightness is $I = \rho k \cos\theta_i$ (this is straight from Equation 10.2). Let $\rho k = 1$ as a simplification, so we just have $I = \cos\theta_i = \hat{\mathbf{n}} \cdot \hat{\mathbf{s}}$.

Our reflectance map then becomes:

$$I = \cos\theta_i = \hat{\mathbf{n}} \cdot \hat{\mathbf{s}} = \frac{pp_S + qq_S + 1}{\sqrt{p^2 + q^2 + 1}\sqrt{p_S^2 + q_S^2 + 1}} = R(p, q)$$

Unfortunately all of this derivation was pointless because images are rarely as perfect as we need them to be, and the absurd amount of assumptions necessary to make the math work out is too restrictive for real-world vision.

In other words...



(**TODO**: Finish this set of lectures lol)

10.2 Color Theory

We begin our discussion of **color theory** in the physical realm with the rods and cones of the human eye, shown to the right in Figure 10.9.

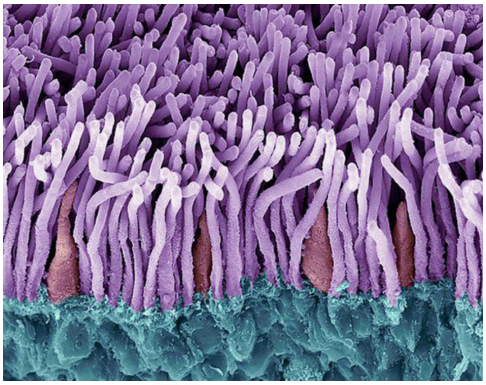


Figure 10.9: A psuedo-colorized cross section of the human retina, highlighting the rods (in the darker pink) and the cones (in the brighter purple).

Cones are responsible for color discrimination; there are 6-7 million cones in the retina and they're concentrated in the center of your visual field enabling high-resolution vision. There are three types of cones: though commonly referred to as red (composing 64% of cones), green (composing 32%), and blue (composing 2%) cones due to their sensitivity to different wavelengths of light (see Figure 10.10), they are more accurately short, medium, and long cones.

This notion of three types of cones corresponding to three color values is the foundation behind **tristimulus color theory**, which weighs the sensitivity spectrum by the amount of cones present and attempts to match any spectral color in the grade-school rainbow spectrum via some combination of these "primary" colors.

Can any single wavelength of light be emulated by a *positive* linear combination of red, green, and blue light? Actually, **no**. Certain spectral colors needed some additional red light to be emulatable because we couldn't add *negative* red light to our mixture. Figure 10.11 shows the necessary RGB combination to emulate the entire spectrum of visible light.

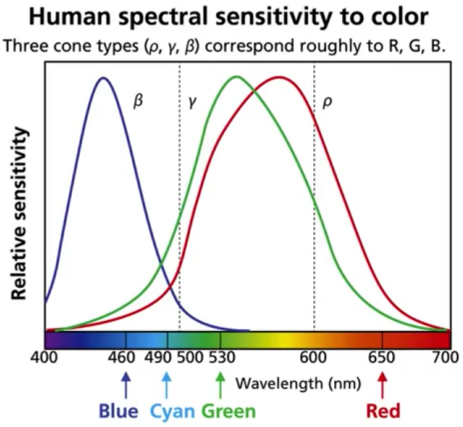


Figure 10.10: The sensitivity spectrum of the three types of cones in the human retina.

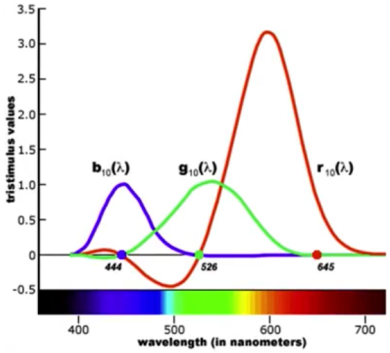


Figure 10.11: The necessary weights of red, green, and blue light necessary to emulate a particular wavelength in the visible light spectrum. Notice the necessity for "negative" red light between blue and green.

Motion

> *Mathematics is unable to specify whether motion is continuous, for it deals merely with hypothetical relations and can make its variable continuous or discontinuous at will. The paradoxes of Zeno are consequences of the failure to appreciate this fact and of the resulting lack of a precise specification of the problem. The former is a matter of scientific description a posteriori, whereas the latter is a matter solely of mathematical definition a priori. The former may consequently suggest that motion be defined mathematically in terms of continuous variable, but cannot, because of the limitations of sensory perception, prove that it must be so defined.*
>
> — Carl B. Boyer, *The History of the Calculus and Its Conceptual Development*

Motion will add another dimension to our images: *time*. Thus, we'll be working with sequences of images: $I(x, y, t)$. Unlike the real world, in which we perceive continuous motion,[1] digital video is merely a sequence of images with changes between them. By changing the images rapidly, we can imitate fluid motion. In fact, studies have been done to determine what amount of change humans will still classify as "motion;" if an object moves 10 feet between images, is it really moving?

There are many applications for motion; let's touch on a few of them to introduce our motivation for the math we'll be slugging through in the remainder of this chapter.

Background Subtraction Given a video with a relatively static scene and some moving objects, we can extract just the moving objects (or just the background!). We may want to then overlay the "mover" onto a different scene, model its dynamics, or apply other processing to the extracted object.

Shot Detection Given a video, we can detect when it cuts to a different scene, or shot.

Motion Segmentation Suppose we have many moving objects in a video. We can segment out each of those objects individually and do some sort of independent analyses. Even in scenarios in which the objects are hard to tell apart spatially, motion gives us another level of insight

[1] *Arguably* continuous, as the chapter quote points out. How can we prove that motion is continuous if our senses operate on some discrete intervals? At the very least, we have discretization by the rate at which a neuron can fire. Or, perhaps, there's never a moment in which a neuron isn't firing, so we can claim that's effectively continuous sampling? But if Planck time is the smallest possible unit of time, does that mean we don't live in a continuous world in the first place, so the argument is moot? Philosophers argue on...

for separating things out.

There are plenty of other applications of motion: video stabilization, learning dynamic models, recognizing events, estimating 3D structure, and more!

Our brain does a ***lot*** of "filling in the gaps" and other interpretations based on even the most impoverished motion; obviously this is hard to demonstrate in written form, but even a handful of dots arranged in a particular way, moving along some simple paths, can look like an actual walking person to our human perception. The challenge of computer vision is interpreting and creating these same associations.

11.1 Motion Estimation

The motion estimation techniques we'll be covering fall into two categories. The first is **feature-based methods**, in which we extract visual features and track them over multiple frames. We've already seen much of what goes on in the former method when we discussed Feature Recognition. These result in sparse motion fields, but achieve more robust tracking; this works well when there is significant motion. The second is **dense methods**, in which we directly recover the motion at *each pixel*, based on spatio-temporal image brightness variations. We get denser motion fields, but are highly sensitive to appearance changes. This method is well-suited for video, since there are many samples and motion between any two frames is small (aside from shot cuts).

We'll be focusing more on the dense flow methods, though we will see how some of the more cutting edge motion estimation models rely heavily on a combination of both approahces.

We begin by defining, and subsequently recovering, **optic flow**. Optic flow is the *apparent* motion of surfaces or objects. For example, in Figure 11.1, the Rubix cube has rotated counter-clockwise a bit, resulting in the optic flow diagram on the right.

Figure 11.1: The optic flow diagram (right) from a rotated Rubix cube.

Our goal is to recover the arrows in Figure 11.2. How do we estimate that motion? In a way, we are solving a pixel correspondence problem much like we did in Stereo Correspondence, though the solution is much different. Given some pixel in $I(x, y, t)$, look for nearby pixels of the same color in $I(x, y, t + 1)$. This is the **optic flow problem**.

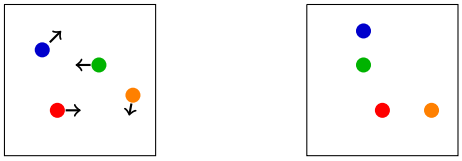


Figure 11.2: The problem definition of optical flow; how do we recover the motion vectors in the left image?

Like we've seen time and time again, we need to establish some assumptions (of varying validity) to simplify the problem:

- **Color constancy**: assume a corresponding point in one image looks the same in the next image. For grayscale images (which we'll be working with for simplicity), this is called **brightness constancy**. We can formulate this into a constraint by saying that there must be some (u, v) change in location for the corresponding pixel, then:

$$I(x, y, t) = I(x + u', y + v', t + 1)$$

- **Small motion**: assume points do not move very far from one image to the next. We can formulate this into a constraint, as well. Given that same (u, v), we assume that they are *very* small, say, 1 pixel or less. The points are changing smoothly.

  Yep, you know what that means! Another Taylor expansion. I at some location $(x+u, y+v)$ can be expressed exactly as:

$$I(x + u, y + v) = I(x, y) + \frac{\partial I}{\partial x}u + \frac{\partial I}{\partial y}v + \ldots \textit{[higher order terms]}$$

  We can disregard the higher order terms and make this an approximation that holds for small values of u and v.

We can combine these two constraints into the following equation (the full derivation comes in this aside), called the **brightness constancy constraint** equation:

$$I_x u + I_y v + I_t = 0 \tag{11.1}$$

We have two unknowns describing the direction of motion, u and v, but only have one equation! This is the **aperture problem**: we can only see changes that are perpendicular to the edge. We can determine the component of $\begin{bmatrix} u & v \end{bmatrix}$ in the gradient's direction, but not in the perpendicular direction (which would be along an edge). This is reminiscent of the edge problem in Finding Interest Points: patches along an edge all looked the same. Visually, the aperture problem is explained in Figure 11.3 below.

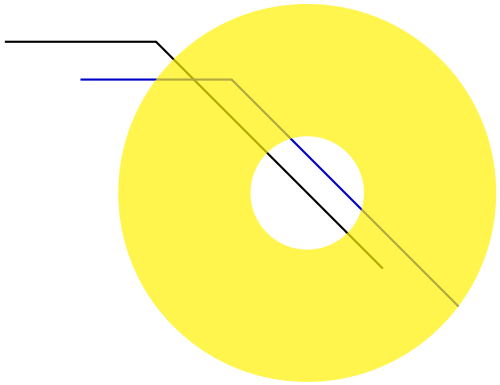


Figure 11.3: The aperture problem. Clearly, motion from the **black** line to the **blue** line moves it down and right, but through the view of the aperture, it appears to have moved *up* and right.

Quick Maffs: **Combining Constraints**

Let's combine the two constraint functions for optic flow. Don't worry, this will be much shorter than the last aside involving Taylor expansions. We begin with our two constraints, rearranged and simplified for convenience:

$$0 = I(x+u', y+v', t+1) - I(x,y,t)$$
$$I(x+u, y+v) \approx I(x,y) + \frac{\partial I}{\partial x}u + \frac{\partial I}{\partial y}v$$

Then, we can perform a substitution of the second into the first and simplify:

$$\begin{aligned} 0 &\approx I(x,y,t+1) + I_x u + I_y v - I(x,y,t) \\ &\approx [I(x,y,t+1) - I(x,y,t)] + I_x u + I_y v \\ &\approx I_t + I_x u + I_y v \\ &\approx I_t + \nabla I \cdot \begin{bmatrix} u & v \end{bmatrix} \end{aligned}$$

In the limit, as u and v approach zero (meaning the Δt between our images gets smaller and smaller), this equation becomes exact.

Notice the weird simplification of the image gradient: is I_x the gradient at t or $t+1$? Turns out, it doesn't matter! We assume that the image moves *so* slowly that the derivative actually doesn't change. This is dicey, but works out "in the limit," as Prof. Bobick says.

Furthermore, notice that I_t is the **temporal derivative**: it's the change in the image over time.

To solve the aperture problem, suppose we formulate Equation 11.1 as an error function:

$$e_c = \iint_{image} (I_x u + I_y v + I_t)^2 \, dx \, dy$$

Of course, we still need another constraint. Remember when we did Better Stereo Correspondence? We split our error into two parts: the raw data error (7.2) and then the *smoothness* error (7.3), which punished solutions that didn't behave smoothly. We apply the same logic here, introducing a **smoothness constraint**:

$$e_s = \iint_{image} \left(u_x^2 + u_y^2\right) + \left(v_x^2 + v_y^2\right) \, dx \, dy$$

This punishes large changes to u or v over the image. Now given both of these constraints, we want to find the (u, v) at each image point that minimizes:

$$e = e_s + \lambda e_c$$

Where λ is a weighing factor we can modify based on how much we "believe" in our data (noise, lighting, artifacts, etc.) to change the effect of the brightness constancy constraint.

This is a **global** constraint on the motion flow field; it comes with the disadvantages of such global assumptions. Though it allows you to bias your solution based on prior knowledge, local constraints perform much better. Conveniently enough, that's what we'll be discussing next.

11.1.1 Lucas-Kanade Flow

As we mentioned, a better approach to solving the aperture problem is imposing local constraints on the pixel we're analysing. One method, known as the **Lucas-Kanade method**, imposes the same (u, v) constraint on an entire patch of pixels.

For example, if we took the 5×5 block around some pixel, we'd get an entire system of equations:

$$\underbrace{\begin{bmatrix} I_x(p_1) & I_y(p_1) \\ I_x(p_2) & I_y(p_2) \\ \vdots & \vdots \\ I_x(p_{25}) & I_y(p_{25}) \end{bmatrix}}_{\mathbf{A},\, 25\times 2} \underbrace{\begin{bmatrix} u \\ v \end{bmatrix}}_{\mathbf{d},\, 2\times 1} = - \underbrace{\begin{bmatrix} I_t(p_1) \\ I_t(p_2) \\ \vdots \\ I_t(p_{25}) \end{bmatrix}}_{\mathbf{b},\, 25\times 1}$$

This time, we have *more* equations than unknowns. Thus, we can use the standard least squares method on an over-constrained system (as covered in **??**) to find the best approximate solution: $(\mathbf{A}^{\mathrm{T}}\mathbf{A})\,\mathbf{d} = \mathbf{b}$. Or, in matrix form:

$$\begin{bmatrix} \sum I_x I_x & \sum I_x I_y \\ \sum I_x I_y & \sum I_y I_y \end{bmatrix} \begin{bmatrix} u \\ v \end{bmatrix} = - \begin{bmatrix} \sum I_x I_t \\ \sum I_y I_t \end{bmatrix}$$

Each summation occurs over some $k \times k$ window. This equation is *solvable* when the pseudoinverse

does, in fact, exist; that is, when $\mathbf{A}^T\mathbf{A}$ is invertible. For that to be the case, it must be **well-conditioned**: the ratio of its eigenvalues, λ_1/λ_2, should not be "too large."[2]

Wait... haven't we already done some sort of eigenvalue analysis on a matrix with image gradients? Yep! Recall our discussion of the Properties of the 2nd Moment Matrix when finding Harris Corners? Our $\mathbf{A}^T\mathbf{A}$ *is* a moment matrix, and we've already considered what it means to have a good eigenvalue ratio: it's a corner!

We can see a tight relationship between the two ideas: use corners as good points to compute the motion flow field. This is explored briefly when we discuss Sparse Flow later; for now, though, we continue with dense flow and approximate the moment matrix for every pixel.

Improving Lucas-Kanade

Our small motion assumption for optic flow rarely holds in the real world. Often, motion from one image to another greatly exceeds 1 pixel. That means our first-order Taylor approximation doesn't hold: we don't necessarily have linear motion.

To deal with this, we can introduce iterative refinement. We find a flow field between two images using the standard Lucas-Kanade approach, then perform image warping using that flow field. This is I'_{t+1}, our *estimated* next time step. Then we can find the flow field between that estimation and the truth, getting a new flow field. We can repeat this until the estimation converges on the truth! This is **iterative Lucas-Kanade**, formalized in algorithm 11.1.

Algorithm 11.1: Iterative Lucas-Kanade algorithm.

Input: A video (or sequence of images), $I(x, y, t)$.
Input: Some window for Lucas-Kanade, W.
Result: A motion flow field.

$V \leftarrow \mathbf{0}$
$I'_{t+1} \leftarrow I_t$
while $I'_{t+1} \neq I_{t+1}$ **do**
  `// Find the Lucas-Kanade flow field.`
  $dV = \texttt{LK}(I'_{t+1}, I_{t+1}, W)$
  `// Warp our estimate towards the real image, then try again.`
  $I'_{t+1} = \texttt{Warp}(I'_{t+1}, I_{t+1}, dV)$
  $V \mathrel{+}= dV$
end
return V

---

[2] From numerical analysis, the **condition number** relates to how "sensitive" a function is: given a small change in input, what's the change in output? Per Wikipedia, a high condition number means our least squares solution may be extremely inaccurate, even if it's the "best" approximation. In some ways, this is related to our discussion of Error Functions for feature recognition: recall that vertical least squares gave poor results for steep lines, so we used perpendicular least squares instead.

Hierarchical Lucas-Kanade

Iterative Lucas-Kanade improves our resulting motion flow field in scenarios that have a little more change between images, but we can do even better. The big idea behind **hierarchical Lucas-Kanade** is that we can make large changes seem smaller by just making the image smaller. To do that, we reintroduce the idea of Gaussian pyramids that we discussed as part of **??** and used when Improving the Harris Detector.

Essentially, we create a Gaussian pyramid for each of our images in time, then build up our motion field from the lowest level. The following disgusting flow chart (**TODO**: recreate this in a much nicer way) diagrams the process of hierarchical Lucas-Kanade. To better understand the subsequent explanation in words, read the chart from the top down:

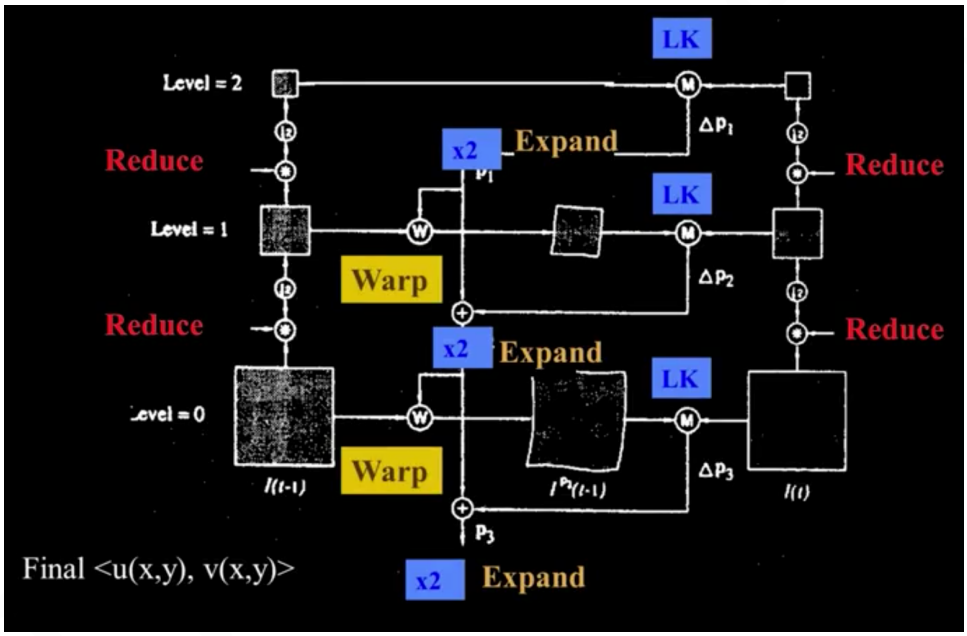


Figure 11.4: The flowchart outlining the hierarchical Lucas-Kanade algorithm. On the left and right are the image pyramids for I_{t-1} and I_t, respectively, and the process moves from the top down.

We're finding the flow from $I_{t-1} \to I_t$. We'll use the notation P_{t-1}^{-1} to denote the top (smallest) layer of the image pyramid of I_{t-1}, P_t^{-2} to denote the second-smallest layer of I_t, and so on.

We start at the highest level in the image pyramids which contain the smallest images. We calculate their flow field using standard (or iterative) Lucas-Kanade. This flow field is small, just like the images we compared. However, if we expand (recall the 5-tap filter of **??**) and double the flow field, we can get a rough approximation of the *next* pyramid level's flow field.

With that approximate flow field, we can warp (again, recall Image Rectification) the *actual* P_{t-1}^{-2} towards P_t^{-2}, which will give us something *close*, but not quite there. If you're following along in the flowchart, this is represented by the small box in the middle of the second horizontal line. Let's just call this "intermediate" image I.

Now, since I is closer to P_t^{-2} than P_{t-1}^{-2}, finding that flow would give us a more accurate approxi-

mation, since less distance = more accurate flow. This gives us a new flow field, but we can't just expand and double it again and repeat the process. Remember, it's far smaller than the actual flow field between P_{t-1}^{-3} and P_t^{-3} since it's far closer to P_t^{-2} than P_{t-1}^{-2}... Thus, we need to add the *previous* flow field in to accumulate the *total* flow over time.

Now we can expand, double, and warp P_{t-1}^{-3} again as before. We can continue this process throughout the entire pyramid (the flowchart only has two levels) and eventually get an accumulated flow field that takes us from $I_{t-1} \to I_t$ far better than a naïve Lucas-Kanade calculation would.

Algorithm 11.2: The hierarchical Lucas-Kanade algorithm.

Input: Two images from a sequence, I_t and I_{t+1}.
Input: The number of Gaussians to create in each pyramid, L.
Result: An estimated motion flow field, V.

```
// Generate the pyramids for each image.
```
$P_t \leftarrow \texttt{GaussianPyramid}(I_t, L)$
$P_{t+1} \leftarrow \texttt{GaussianPyramid}(I_{t+1}, L)$
$L \mathrel{-}= 1$

```
// Our current "intermediate" image between I_t[i] and I_{t+1}[i].
```
$I' \leftarrow P_t[L]$
$V \leftarrow \mathbf{0}$

while $L \geq 1$ **do**
```
    // Estimate flow from our current guess of t+1 to the "true" t+1.
```
    $dV = \texttt{Iterative-LK}(I', P_{t+1}[L])$
```
    // Grow our accumulated flow field approximation.
```
    $V = 2 \cdot \texttt{Expand}(V + dV)$
```
    // Use it to warp the next level's t image towards t+1.
```
    $I' = \texttt{Warp}(P_t[L-1], V)$
    $L \mathrel{-}= 1$
end

```
// We need to do this one more time for the largest level, without
   expansion or warping since we're out of "next" levels.
```
return $V + \texttt{Iterative-LK}(I', P_{t+1}[0])$

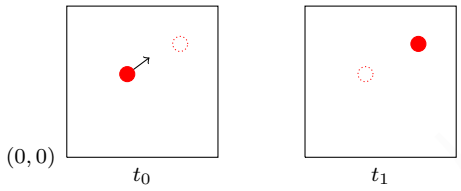
Sparse Flow

We realized earlier that the ideal points for motion detection would be corner-like, to ensure the accuracy of our gradients, but our discussion ended there. We still went on to estimate the motion flow field for every pixel in the image. **Sparse Lucas-Kanade** is a variant of Hierarchical Lucas-Kanade that is only applied to interest points.

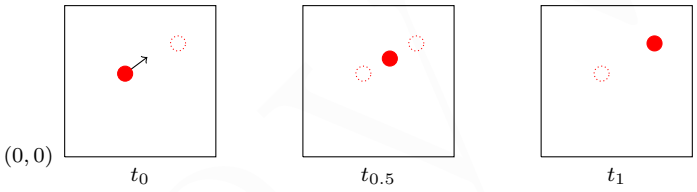
11.1.2 Applying Lucas-Kanade: Frame Interpolation

A useful application of hierarchical Lucas-Kanade is frame interpolation. Given two frames with a large delta, we can approximate the frames that occurred in-between by interpolating the motion flow fields.

Consider the following contrived example for a single point:

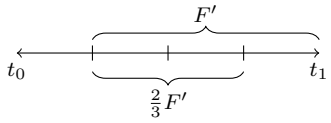


Suppose hierarchical Lucas-Kanade told us that the point moved $(+3.5, +2)$ from t_0 to t_1 (specifically, from $(1, 1)$ to $(4.5, 3)$). We can then easily estimate that the point must have existed at $(2.75, 2)$ at $t_{0.5}$ by just halving the flow vector and warping, as before. As shorthand, let's say Ft is the warping of t with the flow field F, so then $t_{0.5} = \frac{F}{2}t_0$.



Of course, hierarchical Lucas-Kanade rarely gives such accurate results for the flow vectors. Each interpolated frame would continually propogate the error between the "truth" at t_1 and our "expected truth" based on the flow field. Much like when we discussed iterative Lucas-Kanade, we can improve our results by iteratively applying HLK as we interpolate!

Suppose we wanted $t_{0.25}$ and $t_{0.75}$ from our contrived example above, and we had the flow field F. We start as before, with $t_{0.25} = \frac{F}{4}t_0$. Instead of making $t_{0.75}$ use $\frac{3F}{4}t$, though, we *recalculate* the flow field from our estimated frame, giving us $F' = t_{0.25} \to t_1$. Then, we apply $t_{0.75} = \frac{2}{3}F't_{0.25}$, since $t_{0.75}$ is $^2/_3{}^{\text{rds}}$ of the way along F'.



11.2 Motion Models

Until now, we've been treating our corresponding points relatively independently. We don't really have a concept of objects (though they are an emergent property of groups of similar flow vectors). Of course, that isn't generally how motion works in the real world.

Suppose we *knew* that our motion followed a certain pattern, or that certain objects in the scene moved in a certain way. This would allow us to further constrain our motion beyond the simple constraints that we imposed earlier (smoothness, brightness constancy, etc.). For example, we know that objects closer to the camera move more on the camera image plane than objects further away

from the camera, for the same amount of motion. Thus, if we know (or determine/approximate) the depth of a region of points, we can enforce another constraint on their motion and get better flow fields.

To discuss this further, we need to return to some concepts from basic Newtonian physics. Namely, the fact that a point rotating about some origin with a rotational velocity $\boldsymbol{\omega}$ that also has a translational velocity t has a total velocity of: $\mathbf{v} = \boldsymbol{\omega} \times \mathbf{r} + \mathbf{t}$.

In order to figure out how the point is moving in the image, we need to convert from these values in world space (X, Y, Z) to image space (x, y). We've seen this before many times. The perspective projection of these values in world space equates to $\left(f\frac{X}{Z}, f\frac{Y}{Z}\right)$ in the image, where f is the focal length. How does this relate to the velocity, though? Well the velocity is just the derivative of the position. Thus, we can take the derivative of the image space coordinates to see, for a world-space velocity V:[3]

$$\begin{aligned} u = v_x &= f\frac{ZV_x - ZV_z}{z^2} = f\frac{V_x}{Z} - \left(f\frac{X}{Z}\right)\frac{V_z}{Z} \\ &= f\frac{V_x}{X} - f\frac{V_z}{z} \\ v = v_y &= f\frac{ZV_y - ZV_z}{Z^2} = f\frac{V_y}{Z} - \left(f\frac{Y}{Z}\right)\frac{V_z}{Z} \\ &= f\frac{V_y}{Y} - f\frac{V_z}{Z} \end{aligned}$$

This is still kind of ugly, but we can "matrixify" it into a much cleaner equation that isolates terms into things we know and things we don't:

$$\begin{bmatrix} u(x,y) \\ v(x,y) \end{bmatrix} = \frac{1}{Z(x,y)}\mathbf{A}(x,y)\mathbf{t} + \mathbf{B}(x,y)\boldsymbol{\omega}$$

Where $\mathbf{t}$ is the (unknown) translation vector and $\boldsymbol{\omega}$ is the unknown rotation, and $\mathbf{A}$ and $\mathbf{B}$ are defined as such:

$$\begin{aligned} \mathbf{A}(x,y) &= \begin{bmatrix} -f & 0 & x \\ 0 & -f & y \end{bmatrix} \\ \mathbf{B}(x,y) &= \begin{bmatrix} {}^{(xy)}/_{f} & -{}^{(f+x^2)}/_{f} & y \\ {}^{(f+y^2)}/_{f} & -{}^{(xy)}/_{f} & -x \end{bmatrix} \end{aligned}$$

The beauty of this arrangement is that $\mathbf{A}$ and $\mathbf{B}$ are functions of things we know, and they relate our world-space vectors $\mathbf{t}$ and $\boldsymbol{\omega}$ to image space. This is the **general motion model**. We can see that the depth in world space, $Z(x, y)$, only impacts the translational term. This corresponds to our understanding of **parallax motion**, in that the further away from a camera an object is, the less we perceive it to move for the same amount of motion.

11.2.1 Known Motion Geometry

Suppose we know that there's a plane in our image, and so the motion model has to follow that of a plane. If we can form equations to represent the points on the plane in space, we can use them to

[3] Recall the quotient rule for derivatives: $\frac{d}{dx}\left(\frac{f(x)}{g(x)}\right) = \frac{f'(x)g(x) - g'(x)f(x)}{g(x)^2}$.

enforce constraints on our motion model and get a better approximation of flow since we are more confident in how things move.

Perspective Under world-space perspective projection, we have the standard equation of a plane: $aX + bY + cZ + d = 0$.

We saw some time ago, when discussing homographies, that we can find the image-space coordinates from any plane with our homogeneous system in (8.1). We didn't expand it at the time, but it would look like this:

$$u(x, y) = a_1 + a_2x + a_3y + a_7x^2 + a_8xy$$
$$v(x, y) = a_4 + a_5x + a_6y + a_7xy + a_8y^2$$

Recall that 4 correspondence points are needed to solve this equation.

Orthographic On the other hand, if our plane lies at a sufficient distance from the camera, the distance between points on the plane is miniscule relative to that distance. As we learned when discussing Orthographic Projection, we can effectively disregard depth in this case. The simplified pair of equations, needing 3 correspondence points to solve, is then:

$$u(x, y) = a_1 + a_2x + a_3y \tag{11.2}$$
$$v(x, y) = a_4 + a_5x + a_6y \tag{11.3}$$

This is an affine transformation!

11.2.2 Geometric Motion Constraints

Suppose we now use our aforementioned pair of equations for an affine transformation as a constraint on the motion model. Previously, we could only enforce the brightness constancy constraint equation over a small window; if the window was too large, there's no way we could guarantee that the colors would stay constant.

Since our constraint now incorporates a transformation model rather than an assumption, we can reliably work with much larger windows. Recall the constraint from before, (11.1):

$$I_xu + I_yv + I_t = 0$$

By substituting in the affine transformation equations from (11.2), we get:

$$I_x(a_1 + a_2x + a_3y) + I_y(a_4 + a_5x + a_6y) + I_t \approx 0$$

This is actually a relaxation on our constraint, even though the math gets more complicated. Now we can do the same least squares minimization we did before for the Lucas-Kanade method but allow an affine deformation for our points:

$$\text{Err}(\mathbf{a}) = \sum \left[I_x(a_1 + a_2x + a_3y) + I_y(a_4 + a_5x + a_6y) + I_t\right]^2$$

Much like before, we get a nasty system of equations and minimize its result:

$$\begin{bmatrix} I_x & I_x x_1 & I_x y_1 & I_y & I_y x_1 & I_y y_1 \\ I_x & I_x x_2 & I_x y_2 & I_y & I_y x_2 & I_y y_2 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ I_x & I_x x_n & I_x y_n & I_y & I_y x_n & I_y y_n \end{bmatrix} \cdot \begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \end{bmatrix} = - \begin{bmatrix} I_t^1 \\ I_t^2 \\ \vdots \\ I_t^n \end{bmatrix}$$

11.2.3 Layered Motion

Obviously, all of the motion in an image is unlikely to fit into a single motion model, whether that be an affine transformation or homography or some other model. If you have multiple motion models in the same image, you want to identify those individually. The basic idea behind layered motion is to break the image sequence into "layers" which all follow some coherent motion model.[4]

Figure 11.5 demonstrates this visually. Under layered motion, each layer is defined by an alpha mask (which identifies the relevant image region) and an affine motion model like the one we just discussed.

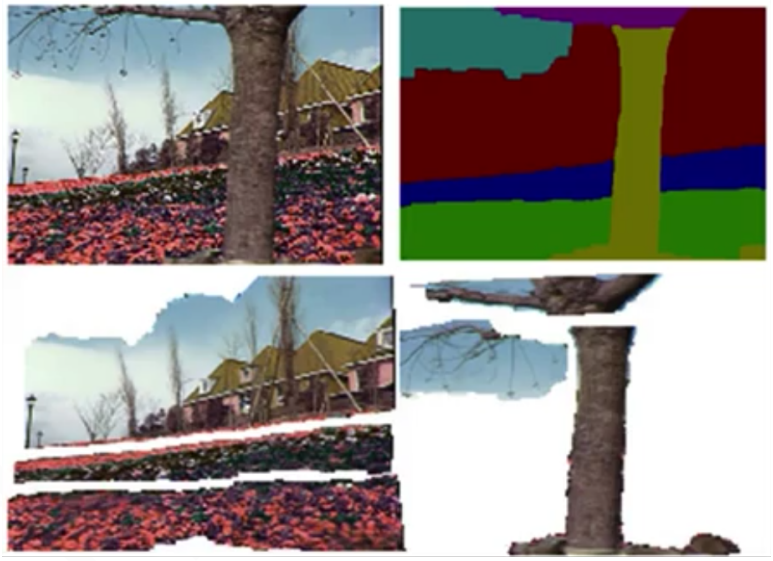
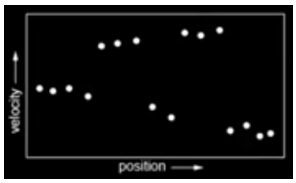


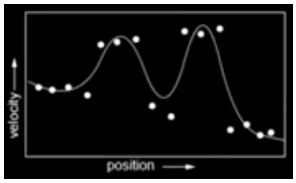
Figure 11.5: Breaking an image sequence into coherent layers of motion.

How does the math work? First, get an approximation of the "local flow" via Lucas-Kanade or other methods. Then, segment that estimation by identifying large leaps in the flow field. Finally, use the motion model to fit each segment and identify coherent segments. Visually, this is demonstrated below:

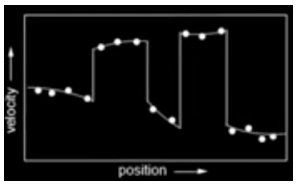
[4] First studied in 1993 in the paper "Layered Representation for Motion Analysis" (link).



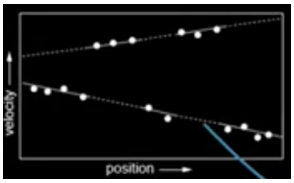
(a) The true motion flow, $u(x, y)$.



(b) The local flow estimation.



(c) Segmenting the local flow estimation.



(d) Fitting the motion model to the segmentations, identifying the foreground and background "lines." The blue line points to an identified occlusion in the scene.

Figure 11.6: Applying an affine motion model to an image sequence to segment it into the foreground and background layers.

The implementation difficulties lie in identifying segments and clustering them appropriately to fit the intended motion models.

Tracking

> *"I followed your footsteps," he said, in answer to the unspoken question. "Snow makes it easy."*
>
> *I had been tracked, like a bear. "Sorry to make you go to all that trouble," I said.*
>
> *"I didn't have to go that far, really. You're about three streets over. You just kept going in loops."*
>
> *A really inept bear.*
>
> — Maureen Johnson, *Let It Snow: Three Holiday Romances*

What's the point of dedicating a separate chapter to tracking? We've already discussed finding the flow between two images; can't we replicate this process for an entire sequence and track objects as they move? Well, as we saw, there are a lot of limitations of Lucas-Kanade and other image-to-image motion approximations.

- It's not always possible to compute optic flow. The Lucas-Kanade method needs a lot of stars to align to properly determine the motion field.
- There could be large displacements of objects across images if they are moving rapidly. Lucas-Kanade falls apart given a sufficiently large displacement (even when we apply improvements like hierarchical LK). Hence, we probably need to take dynamics into account; this is where we'll spend a lot of our focus initially.
- Errors are compounded over time. When we discussed frame interpolation, we tried to minimize the error drift by recalculating the motion field at each step, but that isn't always possible. If we only rely on the optic flow, eventually the compounded errors would track things that no longer exist.
- Objects getting occluded cause optic flow models to freak out. Similarly, when those objects appear again (called **disocclusion**s), it's hard to reconcile and reassociate things. Is this an object we lost previously, or a new object to track?

We somewhat incorporated dynamics when we discussed using Motion Models with Lucas-Kanade: we expected points to move along an affine transformation. We could improve this further by combining this with Feature Recognition, identifying good features to track and likewise fitting

motion models to them.[1] This is good, but only gets us so far.

Instead, we will focus on **tracking with dynamics**, which approaches the problem differently: given a model of expected motion, we should be able to predict the next frame without actually seeing it. We can then use that frame to adjust the dynamics model accordingly. This integration of dynamics is the differentiator between feature *detection* and *tracking*: in detection, we detect objects *independently* in each frame, whereas with tracking we *predict* where objects will be in the next frame using an estimated dynamic model.

The benefit of this approach is that the trajectory model restricts the necessary search space for the object, and it also improves estimates due to reduced measurement noise due to the smoothness of the expected trajectory model.

As usual, we need to make some fundamental assumptions to simplify our model and construct a mathematical framework for continuous motion. In essence, we'll be expecting small, gradual change in pose between the camera and the scene. Specifically:

- Unlike small children, who have no concept of object permeance, we assume that objects do not spontaneously appear or disappear in different places in the scene.
- Similarly, we assume that the camera does not move instantaneously to a new viewpoint, which would cause a massive perceived shift in scene dynamics.

Feature tracking is a multidisciplinary problem that isn't exclusive to computer vision. There are elements of engineering, physics, and robotics at play. Thus, we need to take a detour into state dynamics and estimation in order to model the dynamics of an image sequence.

12.1 Modeling Dynamics

We can view dynamics from two perspectives: *inference* or *induction*, but both result in the same statistical model.

12.1.1 Tracking as Inference

Let's begin our detour by establishing the terms in our inference system. We have our *hidden state*, X, which is made up of the true parameters that we care about. We have our *measurement*, Z, which is a noisy observation of the underlying state. At each time step t, the real state changes from $X_{t-1} \to X_t$, resulting in a new noisy observation Z_t.

> If you are following along in the lectures, you may notice a discrepancy in notation: Z_t for measurements instead of Y_t. This is to align with the later discussion on Particle Filters in which the lectures *also* switch from Y_t to Z_t to stay consistent with the literature these concepts come from (and to steal PowerPoint slides).
>
> This guide uses Z immediately, instead, to maintain consistency throughout.

Our mission, should we choose to accept it, is to recover the ~~most likely~~ estimated distribution of the state X_t given all of the observations we've seen thus far and our knowledge about the dynamics of the state transitions.

[1] Refer to the original paper, "Good Features to Track," for more (link).

More formally, we can view tracking as an adjustment of a probability distribution. We have some distribution representing our prediction or current belief for t, and the actual resulting measurement at t. We need to adjust our prediction based on the observation to form a new prediction for $t+1$.

Our prediction can be expressed as the likelihood of a state given all of the previous observations:

$$\Pr[X_t \,|\, Z_0 = z_0, Z_1 = z_1, \ldots, Z_{t-1} = z_{t-1}]$$

Our correction, then, is an updated estimate of the state after introducing a new observation $Z_t = z_t$:

$$\Pr[X_t \,|\, Z_0 = z_0, Z_1 = z_1, \ldots, Z_{t-1} = z_{t-1}, Z_t = z_t]$$

We can say that tracking is the process of propogating the posterior distribution of state given measurements across time. We will again make some assumptions to simplify the probability distributions:

- We will assume that we live in a Markovian world in that only the immediate past matters with regards to the actual hidden state:

  $$\Pr[X_t \,|\, X_0, X_1, \ldots, X_{t-1}] = \Pr[X_t \,|\, X_{t-1}]$$

  This latter probability, $\Pr[X_t \,|\, X_{t-1}]$, is the **dynamics model**.

- We will also assume that our noisy measurements (more specifically, the probability distribution of the possible measurements) only depends on the current state, rather than everything we've observed thus far:

  $$\Pr[Z_t \,|\, X_0, Z_0, \ldots, X_{t-1}, Z_{t-1}, X_t] = \Pr[Z_t \,|\, X_t]$$

  This is called the **observation model**, and much like the small motion constraint in Lucas-Kanade, this is the most suspect assumption. Thankfully, we wont be exploring relaxations to this assumption, but one example of such a model is conditional random fields, if you'd like to explore further.

These assumptions are represented graphically in Figure 12.1. Readers with experience in statistical modeling or machine learning will notice that this is a hidden Markov model.

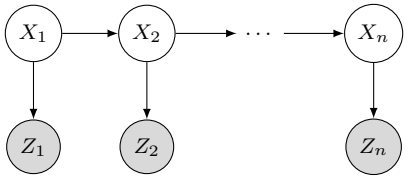


Figure 12.1: A graphical model for our assumptions.

12.1.2 Tracking as Induction

Another way to view tracking is as an inductive process: if we know X_t, we can apply induction to get X_{t+1}.

As with any induction, we begin with our **base case**: this is our initial prior knowledge that predicts a state in the absence of any evidence: $\Pr[X_0]$. At the very first frame, we correct this given $Z_0 = z_0$. After that, we can just keep iterating: given a corrected estimate for frame t, predict then correct frame $t+1$.

12.1.3 Making Predictions

Alright, we can finally get into the math.

Given: $\Pr[X_{t-1} \mid z_0, \dots, z_{t-1}]$
Guess: $\Pr[X_t \mid z_0, \dots, z_{t-1}]$

To solve that, we can apply the **law of total probability** and **marginalization** if we imagine we're working with the joint set $X_t \cap X_{t-1}$.[2] Then:

$$
\begin{aligned}
&= \int \Pr[X_t, X_{t-1} \mid z_0, \dots, z_{t-1}] \; dX_{t-1} \\
&= \int \Pr[X_t \mid X_{t-1}, z_0, \dots, z_{t-1}] \Pr[X_{t-1} \mid z_0, \dots, z_{t-1}] \; dX_{t-1} \\
&= \int \Pr[X_t \mid X_{t-1}] \Pr[X_{t-1} \mid z_0, \dots, z_{t-1}] \; dX_{t-1} && \text{independence assumption from the } \textbf{dynamics} \text{ model}
\end{aligned}
$$

To explain this equation in English, what we're saying is that the likelihood of being at a particular spot (this is X_t) depends on the probability of being at that spot given that we were at some *previous* spot weighed by the probability of that previous spot actually happening (our corrected estimate for X_{t-1}). Summing over all of the possible "previous spots" (that is, the integral over X_{t-1}) gives us the marginalized distribution of X_t.

12.1.4 Making Corrections

Now, given a predicted value $\Pr[X_t \mid z_0, \dots, z_{t-1}]$ and the current observation z_t, we want to compute $\Pr[X_t \mid z_0, \dots, z_{t-1}, z_t]$, essentially folding in the new measurement:[3]

$$
\begin{aligned}
\Pr[X_t \mid z_0, \dots, z_{t-1}, z_t] &= \frac{\Pr[z_t \mid X_t, z_0, \dots, z_{t-1}] \cdot \Pr[X_t \mid z_0, \dots, z_{t-1}]}{\Pr[z_t \mid z_0, \dots, z_{t-1}]} && (12.1) \\
&= \frac{\Pr[z_t \mid X_t] \cdot \Pr[X_t \mid z_0, \dots, z_{t-1}]}{\Pr[z_t \mid z_0, \dots, z_{t-1}]} && \text{independence assumption from the } \textbf{observation} \text{ model} \\
&= \frac{\Pr[z_t \mid X_t] \cdot \Pr[X_t \mid z_0, \dots, z_{t-1}]}{\int \Pr[z_t \mid X_t] \Pr[X_t \mid z_0, \dots, z_{t-1}] \; dX_t} && \text{conditioning on } X_t
\end{aligned}
$$

As we'll see, the scary-looking denominator is just a normalization factor that ensures the probabilities sum to 1, so we'll never really need to worry about it explicitly.

[2] Specifically, the law of total probability states that if we have a joint set $A \cap B$ and we know all of the probabilities in B, we can get $\Pr[A]$ if we sum over all of the probabilities in B. Formally, $\Pr[A] = \sum_n (\Pr[A, B_n]) = \sum_n (\Pr[A \mid B_n] \Pr[B_n])$. For the latter equivalence, recall that $\Pr[U, V] = \Pr[U \mid V] \Pr[V]$; this is the **conditioning** property.

In our working example, X_t is part of the same probability space as X_{t-1} (and all of the X_is that came before it), so we can apply the law, using the integral instead of the sum.

[3] We are applying Bayes' rule here, which states that $\Pr[A \mid B] = \frac{\Pr[B \mid A] \cdot \Pr[A]}{\Pr[B]}$.

12.1.5 Summary

We've developed the probabilistic model for predicting and subsequently correcting our state based on some observations. Now we can dive into actual analytical models that apply these mathematics and do tracking.

12.2 Kalman Filter

Let's introduce an analytical model that leverages the prediction and correction techniques we mathematically defined previously: the **Kalman filter**. We'll be working with a specific case of **linear dynamics**, where the predictions are based on a linear function applied to our previous beliefs perturbed by some Gaussian noise function.[4]

12.2.1 Linear Models

Linear dynamics include very familiar processes from physics. Recall the basic equation for motion: given an initial position $\mathbf{p}_0$, the next position is based on its velocity $\mathbf{v}$: $\mathbf{p}_1 = \mathbf{p}_0 + \mathbf{v}$. In general, then, the **linear** (hey look at that) **equation** that represents position at any time t is: $\mathbf{p}_t = t\mathbf{v} + \mathbf{p}_0$.

Of course, this does not account for changes in velocity or the specific changes in time, but this simple model is still a valid Kalman filter.

Dynamics Model

More formally, a linear *dynamics* model says that the state at some time, $\mathbf{x}_t$, depends on the previous state $\mathbf{x}_{t-1}$ undergoing some linear transformation $\mathbf{D}_t$, plus some level of Gaussian process noise Σ_{d_t} which represents uncertainty in how "linear" our model truly is.[5] Mathematically, we have our normal distribution function N,[6] and so

$$\mathbf{x}_t \sim N(\underbrace{\mathbf{D}_t\mathbf{x}_{t-1}}_{\text{mean}}, \underbrace{\Sigma_{d_t}}_{\text{variance}})$$

Notice the subscript on $\mathbf{D}_t$ and d_t: with this, we indicate that the *transformation itself* may change over time. Perhaps, for example, the object is moving with some velocity, and then starts rotating. In our examples, though, these terms will stay constant.

For example, suppose we're tracking position and velocity as part of our state: $\mathbf{x}_t = \begin{bmatrix} p_x & p_y \\ v_x & v_y \end{bmatrix}_t$

And our "linear transformation" is the simple fact that the new position is the old position plus the velocity; so, given the transformation matrix: $\mathbf{D} = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}$ our dynamics model indicates:

$$\mathbf{D}\mathbf{x}_t = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} p_x & p_y \\ v_x & v_y \end{bmatrix}_t = \begin{bmatrix} p_x + v_x & p_y + v_y \\ v_x & v_y \end{bmatrix}$$

[4] It's easy to get lost in the math. Remember, "Gaussian noise" is just a standard bell curve, so this essentially means "really boring motion that we're a little unsure about."

[5] This is "capital sigma," not a summation. It symbolizes the variance (std. dev squared): $\Sigma = \sigma^2$.

[6] $N(\mu, \sigma^2)$ is a compact way to specify a multivariate Gaussian (since our state can have n variables). The notation $x \sim N(\mu, \sigma^2)$ means x is "distributed as" a Gaussian centered at μ with variance σ^2.

Meaning the new state after a prediction based on the dynamics model would just be a noisy version of that:

$$\mathbf{x}_{t+1} = \mathbf{D}\mathbf{x}_t + noise$$

Essentially, $\mathbf{D}$ is crafted to fit the *specific Kalman filter for your scenario*. Formulating the matrix is just a matter of reading off the coefficients in the linear equation(s); in this case:

$$\mathbf{D} = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} \mapsto \begin{cases} \mathbf{p}' = 1 \cdot \mathbf{p} + 1 \cdot \mathbf{v} \\ \mathbf{v}' = 0 \cdot \mathbf{p} + 1 \cdot \mathbf{v} \end{cases}$$

Measurement Model

We also have a linear *measurement* model describing our observations of the world. Specifically (and unsurprisingly), the model says that the measurement $\mathbf{z}_t$ is linearly transformed by $\mathbf{M}_t$, plus some level of Gaussian measurement noise:

$$\mathbf{z}_t \sim N(\mathbf{M}_t\mathbf{x}_t, \Sigma_{m_t})$$

$\mathbf{M}$ is also sometimes called the **extraction matrix** or also the **measurement function** because its purpose is to extract the measurable component from the state vector.

For example, if we're storing position *and velocity* in our state $\mathbf{x}_t$, but our measurement can only provide us with position, then our extraction matrix would be $\begin{bmatrix} 1 & 0 \end{bmatrix}$, since:

$$\mathbf{M}_t\mathbf{x}_t = \begin{bmatrix} 1 & 0 \end{bmatrix} \begin{bmatrix} p_x & p_y \\ v_x & v_y \end{bmatrix}_t = \begin{bmatrix} p_x & p_y \end{bmatrix}_t$$

So our measurement is the current position perturbed by some uncertainty:

$$\mathbf{z}_t = \mathbf{p}_t + noise$$

Now note that this probably seems a little strange. Our measurement is based on the existing state? Well, there is going to be a difference between what we "actually observe" (the *real* measurement $\mathbf{z}_t$ from our sensors) versus how our model thinks things behave. **In a perfect world**, the sensor measurement matches our predicted position exactly, but the *error* between these ($\mathbf{z}_t - \mathbf{M}\mathbf{x}_t$) will dictate how we adjust our state.

12.2.2 Notation

Before we continue, we need to introduce some standard notation to track things. In our predicted state, $\Pr[X_t \,|\, z_0, \ldots, z_{t-1}]$, we say that the mean and standard deviation of the resulting Gaussian distribution is μ_t^- and σ_t^-. For our corrected state, $\Pr[X_t \,|\, z_0, \ldots, z_{t-1}, z_t]$ (notice the introduction of the latest measurement, z_t), we similarly say that the mean and standard deviation are μ_t^+ and σ_t^+.

Again, at time t,

- μ_t^-, σ_t^- — state mean and variance after *only* prediction.
- μ_t^+, σ_t^+ — state mean and variance after prediction *and* measurement correction.

12.2.3 Kalman in Action

In order to further develop our intuition, let's continue to work through an even simpler 1D dynamics model and do a full tracking step (prediction followed by correction) given our new notation.

Prediction

Let's suppose our dynamics model defines a state X as being just a scaled version of the previous state plus some uncertainty:

$$X_t \sim N\left(dX_{t-1}, \sigma_d^2\right) \tag{12.2}$$

Clearly, X_t is a Gaussian distribution that is a function of the previous Gaussian distribution X_{t-1}. We can update the state by simply update its mean and variance accordingly.

The mean of a scaled Gaussian is likewise scaled by that constant. The variance, though, is both multiplied by that constant squared *and* we need to introduce some additional noise to account for prediction uncertainty. Meaning:

Update the means: $\mu_t^- = d\mu_{t-1}^-$
Update the variance: $(\sigma_t^-)^2 = \sigma_d^2 \left(d\sigma_{t-1}^+\right)^2$

Correction

Suppose our mapping of states to measurements similarly relies on a constant, m:

$$z_t \sim N\left(mX_t, \sigma_m^2\right)$$

The Kalman filter defines our new Gaussian (the simplified Equation 12.1) as another adjustment:

Update the mean:

$$\mu_t^+ = \frac{\mu_t^- \sigma_m^2 + m z_t (\sigma_t^-)^2}{\sigma_m^2 + m^2 (\sigma_t^-)^2}$$

Update the variance:

$$(\sigma_t^+)^2 = \frac{\sigma_m^2 (\sigma_t^-)^2}{\sigma_m^2 + m^2 (\sigma_t^-)^2}$$

The proof of this is left as an exercise to the reader... ☺

Intuition

Let's unpack that monstrosity of an update to get an intuitive understanding of what this new mean, μ_t^+, really is. Let's first divide the entire thing by m^2 to "unsimplify." We get this mess:

$$\mu_t^+ = \frac{\frac{\mu_t^- \sigma_m^2}{m^2} + \frac{z_t}{m}(\sigma_t^-)^2}{\frac{\sigma_m^2}{m^2} + (\sigma_t^-)^2} \tag{12.3}$$

In blue we have our prediction of X_t; it's weighted by the variance of X_t computed from the measurement (in red). Then, in orange, we have our measurement guess of X_t, weighted by the variance of the prediction (in green).

Notice that all of this is divided by the sum of the weights (in red and green): **this is just a weighted average of our prediction and our measurement guess based on variances**!

This gives us an important insight that applies to the Kalman filter regardless of the dimensionality we're working with. Specifically, our corrected distribution for X_t is a weighted average of the prediction (i.e. based on all prior measurements except z_t) and the measurement guess (i.e. with z_t incorporated).

Let's take the equation from (12.3) and substitute a for the measurement variance and b for the prediction variance. We get:

$$\mu_t^+ = \frac{a\mu_t^- + b\dfrac{z_t}{m}}{a+b}$$

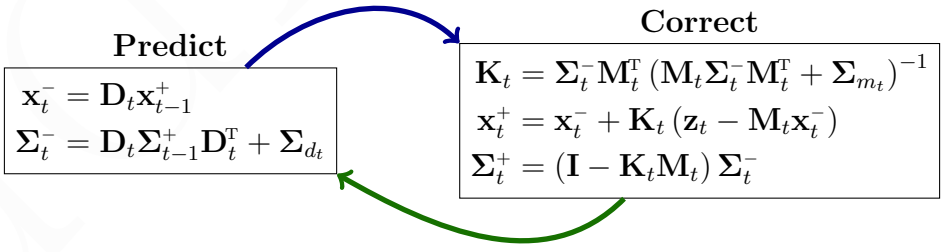
We can do some manipulation (add $b\mu_t^- - b\mu_t^-$ to the top and factor) to get:

$$\begin{aligned} &= \frac{(a+b)\mu_t^- + b\left(\dfrac{z_t}{m} - \mu_t^-\right)}{a+b} \\ &= \mu_t^- + \frac{b}{a+b}\left(\frac{z_t}{m} - \mu_t^-\right) \\ \mu_t^+ &= \mu_t^- + k\left(\frac{z_t}{m} - \mu_t^-\right) \end{aligned}$$

Where k is known as the **Kalman gain**. What does this expression tell us? Well, the new mean μ_t^+ is the old predicted mean plus a weighted "residual": the difference between the measurement and the prediction. In other words, it's adjusted based on how wrong the prediction was!

12.2.4 N-dimensional Kalman Filter

Even under our first running example, we didn't have a single dimension: our state vector had both position and velocity. Under an N-dimensional model, Kalman filters give us some wonderful matrices:



We now have a Kalman gain *matrix*, $\mathbf{K}_t$. As our estimate covariance approaches zero (i.e. confidence in our prediction grows), the residual gets less weight from the gain matrix. Similarly, if our measurement covariance approaches zero (i.e. confidence in our measurement grows), the residual gets more weight.

Let's look at this math a little more thoroughly because it's absolutey a lot to take in.

The prediction step should look somewhat familiar: $\mathbf{x}_t^-$ is the result of applying the dynamics model to the state, and its covariance matrix (the uncertainty) is likewise adjusted by the process noise.

The correction step is hairier, but is just an adjustment of our simple equations to N dimensions. First, recall from above (12.3) the Kalman gain:

$$k = \frac{b}{a+b} = \frac{(\sigma_t^-)^2}{\frac{\sigma_m^2}{m^2} + (\sigma_t^-)^2}$$

Similarly here, recall that the closest equivalent to division in linear algebra is multiplication by the inverse. Also, note that given a n-dimensional state vector:

- $\mathbf{\Sigma}_t^-$ is its $n \times n$ covariance matrix, and
- $\mathbf{M}_t$ is the $m \times n$ extraction matrix, where m is the number of elements returned by our sensors.

Then we can (sort of) see this as a division of the prediction uncertainty by that very uncertainty combined with the measurement uncertainty ($\mathbf{M}_t$ is there for magical purposes):

$$\mathbf{K}_t = \underbrace{\underbrace{\mathbf{\Sigma}_t^- \mathbf{M}_t^{\mathrm{T}}}_{n \times m} \left(\underbrace{\mathbf{M}_t \mathbf{\Sigma}_t^- \mathbf{M}_t^{\mathrm{T}}}_{m \times m} + \mathbf{\Sigma}_{m_t} \right)^{-1}}_{n \times m}$$

Now we can see the state update as an adjustment based on the gain and the residual error between the prediction and the measurement:

$$\mathbf{x}_t^+ = \mathbf{x}_t^- + \mathbf{K}_t \underbrace{(\mathbf{z}_t - \mathbf{M}_t \mathbf{x}_t^-)}_{\text{residual}}$$

And the newly-corrected uncertainty is scaled based on the gain:

$$\mathbf{\Sigma}_t^+ = (\mathbf{I} - \mathbf{K}_t \mathbf{M}_t)\, \mathbf{\Sigma}_t^-$$

All of this feels hand-wavy for a very good reason: I barely understand it. Feel free to read the seminal paper on Kalman filters for more confusing math.

12.2.5 Summary

The Kalman filter is an effective tracking method due to its simplicity, efficiency, and compactness. Of course, it does impose some fairly strict requirements and has significant pitfalls for that same reason. The fact that the tracking state is always represented by a Gaussian creates some huge limitations: such a unimodal distribution means we only really have one true hypothesis for where the object is. If the object does not strictly adhere to our linear model, things fall apart rather quickly.

We know that a fundamental concept in probability is that as we get more information, certainty increases. This is why the Kalman filter works: with each new measured observation, we can derive a more confident estimate for the new state. Unfortunately, though, "always being more certain" doesn't hold the same way in the real world as it does in the Kalman filter. We've seen that the variance decreases with each correction step, narrowing the Gaussian. Does that always hold, intuitively? We may be more sure about the distribution, but not necessarily the variance within that distribution. Consider the following extreme case that demonstrates the pitfalls of the Kalman filter.

In Figure 12.2, we have our prior distribution and our measurement. Intuitively, where should the corrected distribution go? When the measurement and prediction are far apart, we would think that we can't trust either of them very much. We can count on the truth being between them, sure, and that it's probably closer to the measurement. Beyond that, though, we can't be sure. We wouldn't have a very high peak and our variance may not change much. In contrast, as we see in Figure 12.2, Kalman is *very* confident about its corrected prediction.

This is one of its pitfalls.

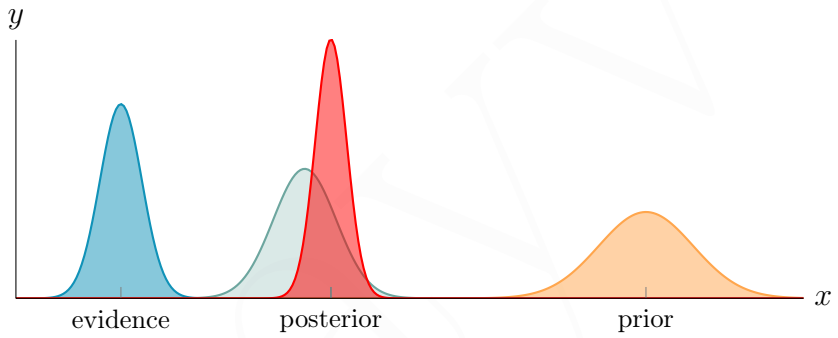


Figure 12.2: One of the flaws of the Kalman model is that it is always more confident in its distribution, resulting in a tighter Gaussian. In this figure, the red Gaussian is what the Kalman filter calculates, whereas the blue-green Gaussian may be a more accurate representation of our intuitive confidence about the truth. As you can see, Kalman is way more confident than we are; its certainty can **only grow** after a measurement.

Another downside of the Kalman filter is this restriction to linear models for dynamics. There are extensions that alleviate this problem called extended Kalman filters (EKFs), but it's still a limitation worth noting.

More importantly, though, is this Gaussian model of noise. If the real world doesn't match with a Gaussian noise model, Kalman struggles. What can we do to alleviate this? Perhaps we can actually determine (or at least approximate) the noise distribution as we track?

> **Note**
>
> This section was based on multiple additional sources, including the course *Robotics: AI Techniques*, its respective notes on the topic (which dive a little deeper into

updating μ^+ and σ^+), and this fantastic post that breaks down each variable in the Kalman filter; these might be good avenues for further exploration.

12.3 Particle Filters

The basic idea behind **particle filtering** and other sampling-based methods is that we can approximate the probability distribution with a set of n weighted particles, x_t. Then, the density is represented by both where the particles are and their weight; we can think of weight as being multiple particles in the same location.

Now we view $\Pr[x = x_0]$ as being the probability of drawing an x with a value (really close to) x_0. Our goal, then, is to make drawing a particle a very close approximation (with equality as $n \to \infty$) of the underlying distribution. Specifically, that

$$\Pr[x_t \in x_t] \approx \Pr[x_t \,|\, z_{1\ldots t}]$$

We'll also be introducing the notion of **perturbation** into our dynamics model. Previously, we had a linear dynamics model that only consisted of our predictions based on previous state. Perturbation – also called *control* – allows us to modify the dynamics by some known model. By convention, perturbation is an input to our model using the parameter u.

Example 12.1: **Perturbation**

Consider, for example, a security camera tracking a person. We don't know how the person will move but can approximate their trajectory based on previous predictions and measurements: this is what we did before.

The *camera* can also move, and this is a *known* part of our dynamic model. We can add the camera's panning or tilting to the model as an input, adjusting the predictions accordingly.

Adding control to a Kalman filter is simple: given a movement vector $\mathbf{u}$, the prediction (see (12.2)) just incorporates it accordingly:

$$\mathbf{x}_{t+1} \sim N(\mathbf{D}\mathbf{x}_t, \mathbf{\Sigma}_t) + \mathbf{u}$$

If the movement has its own uncertainty, it's a sum of Gaussians:

$$\mathbf{x}_{t+1} \sim N(\mathbf{D}\mathbf{x}_t, \mathbf{\Sigma}_t) + N(\mathbf{u}, \mathbf{\Sigma}_u)$$

12.3.1 Bayes Filters

The framework for our first particle filtering approach relies on some given quantities:

- As before, we need somewhere to start. This is our **prior** distribution, $\Pr[X_0]$. We may be very unsure about it, but must exist.

- Since we've added perturbation to our dynamics model, we now refer to it as an **action model**. We need that, too:

$$\Pr[x_t \mid u_t, x_{t-1}]$$

  Note: We (consistently, unlike lecture) use u_t for inputs occuring *between* the state x_{t-1} and x_t.

- We additionally need the **sensor model**. This gives us the likelihood of our *measurements* given some object location: $\Pr[z \mid X]$. In other words, how likely are our measurements *given* that we're at a location X. It is **not** a distribution of possible object locations based on a sensor reading.

- Finally, we need our stream of observations, $\mathbf{z}$, and our known action data, $\mathbf{u}$:

$$\text{data} = \{u_1, z_2, \ldots, u_t, z_t\}$$

Given these quantities, what we *want* is the estimate of X at time t, just like before; this is the **posterior** of the state, or **belief**:

$$Bel(x_t) = \Pr[x_t \mid u_1, z_2, \ldots, u_t, z_t]$$

The assumptions in our probabilistic model is represented graphically in Figure 12.3, and result in the following simplifications:[7]

$$\Pr[z_t \mid X_{0:t}, z_{1:t-1}, u_{1:t}] = \Pr[z_t \mid x_t]$$
$$\Pr[x_t \mid X_{1:t-1}, z_{1:t-1}, u_{1:t}] = \Pr[x_t \mid x_{t-1}, u_t]$$

In English, the probability of the current measurement, given all of the past states, measurements, and inputs *only* actually depends on the current state. This is sometimes called **sensor independence**. Second: the probability of the current state – again given all of the goodies from the past – actually only depends on the previous state and the current input. This Markovian assumption is akin to the independence assumption in the dynamics model from before.

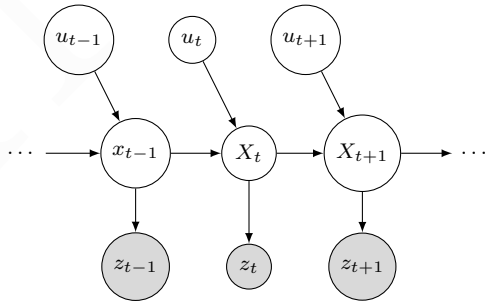


Figure 12.3: A graphical model for Bayes filters.

As a reminder, Bayes' Rule (described more in footnote 3) can also be viewed as a proportionality (η is the normalization factor that ensures the probabilities sum to one):

$$\Pr[x \mid z] = \eta \Pr[z \mid x] \Pr[x] \tag{12.4}$$
$$\propto \Pr[z \mid x] \Pr[x] \tag{12.5}$$

[7] The notation $n_{a:b}$ represents a range; it's shorthand for $n_a, n_{a+1}, \ldots, n_b$.

With that, we can apply our given values and manipulation our belief function to get something more useful. Graphically, what we're doing is shown in Figure 12.4 (and again, more visually, in Figure 12.5), but mathematically:

$$
\begin{aligned}
Bel(x_t) &= \Pr[x_t \,|\, u_1, z_2, \ldots, u_t, z_t] \\
&= \eta \Pr[z_t \,|\, x_t, u_1, z_2, \ldots, u_t] \Pr[x_t \,|\, u_1, z_2, \ldots, u_t] && \text{Bayes' Rule} \\
&= \eta \Pr[z_t \,|\, x_t] \Pr[x_t \,|\, u_1, z_2, \ldots, u_t] && \text{sensor independence} \\
&= \eta \Pr[z_t \,|\, x_t] \int \Pr[x_t \,|\, x_{t-1}, u_1, z_2, \ldots, u_t] \cdot \Pr[x_{t-1} \,|\, u_1, z_2, \ldots, u_t] \; dx_{t-1} \\
& && \text{total probability} \\
&= \eta \Pr[z_t \,|\, x_t] \int \Pr[x_t \,|\, x_{t-1}, u_t] \cdot \Pr[x_{t-1} \,|\, u_1, z_2, \ldots, u_t] \; dx_{t-1} && \text{Markovian assumption} \\
&= \eta \Pr[z_t \,|\, x_t] \underbrace{\int \Pr[x_t \,|\, x_{t-1}, u_t] \cdot Bel(x_{t-1}) \; dx_{t-1}}_{\text{predictions } \textit{before} \text{ measurement}} && \text{substitution}
\end{aligned}
$$

This results in our final, beautiful recursive relationship between the previous belief and the next belief based on the sensor likelihood:

$$
Bel(x_t) = \eta \Pr[z_t \,|\, x_t] \int \Pr[x_t \,|\, x_{t-1}, u_t] \cdot Bel(x_{t-1}) \; dx_{t-1} \tag{12.6}
$$

We can see that there is an inductive relationship between beliefs. The green and blue sections correspond to the calculations we did with the Kalman filter: we need to first find the prediction distribution *before* our latest measurement. Then, we fold in the actual measurement, which is described by the sensor likelihood model from before.

With the mathematics out of the way, we can focus on the basic particle filtering algorithm. It's formalized in algorithm 12.1 and demonstrated graphically in Figure 12.5, but let's walk through the process informally.

We want to generate a certain number of samples (n new particles) from an existing distribution, given an additional input and measurement (these are S_{t-1}, u_t, and z_t respectively). To do that, we need to first choose a particle from our old distribution, which has some position and weight. Thus, we can say $p_j = \langle x_{t-1,j}, w_{t-1,j} \rangle$. From that particle, we can incorporate the control and create a new distribution using our action model: $\Pr[x_t \,|\, u_t, x_{t-1,j}]$. Then, we sample from *that* distribution, getting our new particle state x_i. We need to calculate the significance of this sampled particle, so we run it through our sensor model to reweigh it: $w_i = \Pr[z_i \,|\, x_i]$. Finally, we update our normalization factor to keep our probabilities consistent and add it to the set of new particles, S_t.

12.3.2 Practical Considerations

Unfortunately, math is one thing but reality is another. We need to take some careful considerations when applying particle filtering algorithms to real-world problems.

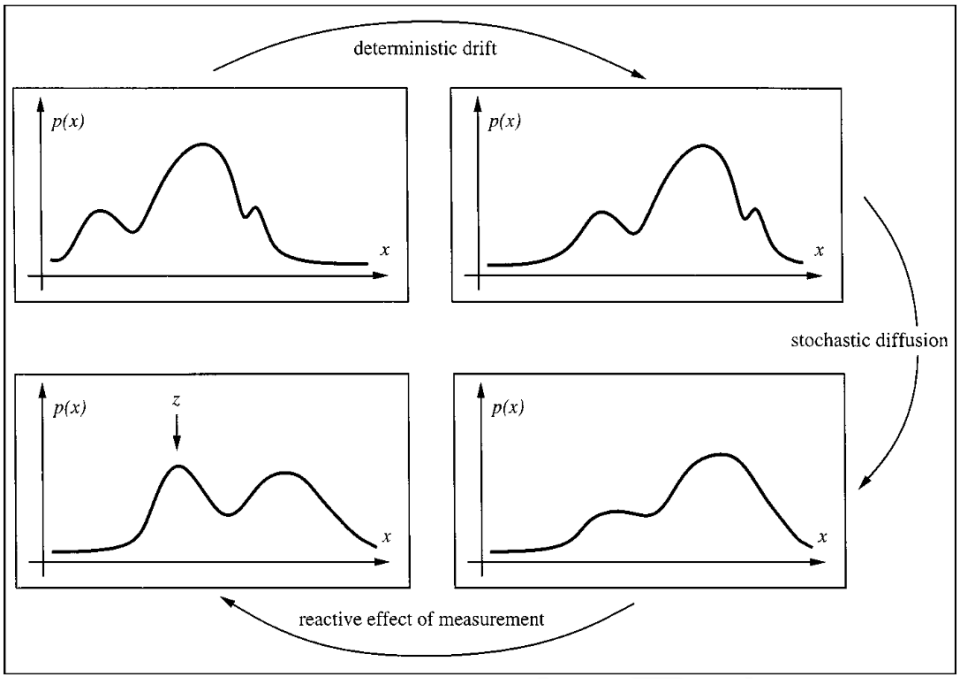


Figure 12.4: Propogation of a probability density in three phases: drift due to the deterministic object dynamics, diffusion due to noise, and reactive reinforcement due to observations.

Sampling Method

We need a lot of particles to sample the underlying distribution with relative accuracy. Every timestep, we need to generate a completely new set of samples after working all of our new information into our estimated distribution. As such, the efficiency or algorithmic complexity of our sampling method is very important.

We can view the most straightforward sampling method as a direction on a roulette wheel, as in Figure 12.6a. Our list of weights covers a particular range, and we choose a value in that range. To figure out which weight that value refers to, we'd need to perform a binary search. This gives a total $O(n \log n)$ runtime. Ideally, though, sampling runtime should grow linearly with the number

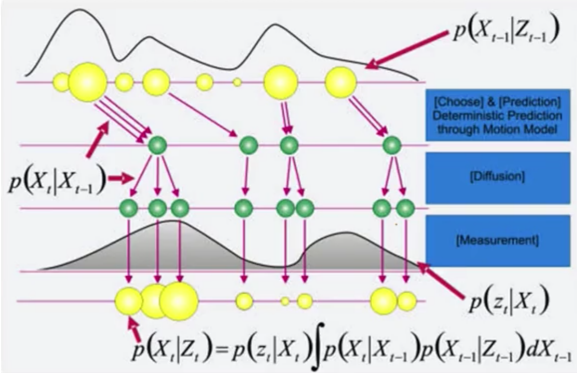


Figure 12.5: A graphic representation of particle filtering.

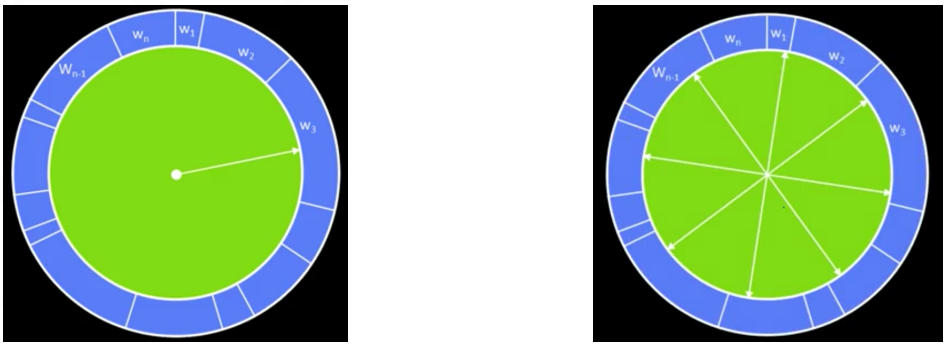
ALGORITHM 12.1: Basic particle filtering algorithm.

Input: A set of weighted particles,
$S_{t-1} = \{\langle x_{1,t-1}, w_{1,t-1}\rangle, \langle x_{2,t-1}, w_{2,t-1}\rangle, \ldots \langle x_{n,t-1}, w_{n,t-1}\rangle\}$.
Input: The current input and measurement: u_t and z_t.
Result: A sampled distribution.

$S_t = \varnothing$
$\eta = 0$
for $i \in [1, n]$ **do** `/* resample n samples */`
  $x_{j,t-1} = \texttt{Sample}(S_{t-1})$ `/* sample a particle state */`
  $x_{i,t} = \texttt{Sample}\left(\Pr\left[x_t \,|\, u_t, x_{j,t-1}\right]\right)$ `/* sample after incorporating input */`
  $w_{i,t} = \Pr\left[z_t \,|\, x_{i,t}\right]$ `/* reweigh particle */`
  $\eta \mathrel{+}= w_{i,t}$ `/* adjust normalization */`
  $S_t = S_t \cup \langle x_{i,t}, w_{i,t}\rangle$ `/* add to current particle set */`
end
$\mathbf{w}_t = {}^{\mathbf{w}_t}/_{\eta}$ `/* normalize weights */`
return S_t

of samples!

As a clever optimization, we can use the systematic resampling algorithm (also called **stochastic universal sampling**), described formally in algorithm 12.2. Instead of viewing the weights as a roulette wheel, we view it as a wagon wheel. We plop down our "spokes" at a random orientation, as in Figure 12.6b. The spokes are $1/n$ distance apart and determining their weights is just a matter of traversing the distance between each spoke, achieving $O(n)$ linear time for sampling!



(a) as a roulette wheel **(b)** as a wagon wheel

Figure 12.6: Two methods of sampling from our set of weighted particles, with 12.6b being the more efficient method.

Sampling Frequency

We can add another optimization to lower the frequency of sampling. Intuitively, when would we even *want* to resample? Probably when the estimated distribution has changed significantly from our initial set of samples. Specifically, we can only resample when there is a significant variance in the particle weights; otherwise, we can just *reuse* the samples.

Highly peaked observations

What happens to our particle distribution if we have incredibly high confidence in our observation? It'll nullify a large number of particles by giving them zero weight. That's not very good, since it wipes out all possibility of other predictions. To avoid this, we want to intentionally add noise to both our action and sensor models.

In fact, we can even smooth out the distribution of our samples by applying a Kalman filter to them individually: we can imagine each sample as being a tiny little Gaussian rather than a discrete point in the state space. In general, overestimating noise reduces the number of required samples and avoids overconfidence; let the measurements focus on increasing certainty.

Recovery from failure

Remember our assumption regarding object permeance, in which we stated that objects wouldn't spontaneously (dis)appear? If that *were* to happen, our distributions have no way of handling that case because there are unlikely to be *any* particles corresponding to the new object. To correct for

this, we can apply some randomly distributed particles every step in order to catch any outliers.

ALGORITHM 12.2: The stochastic universal sampling algorithm.

Input: A particle distribution, S, and the expected number of output samples, n.
Result: A set of n samples.

$S' = \varnothing$
$c_1 = w_1$
`/* We use` $S[i]_x$ `and` $S[i]_w$ `for the state or weight of the` i`th particle. */`
for $i \in [2, n]$ **do**
  $c_i = c_{i-1} + S[i]_w$ `/* generate the CDF (outer ring) from the weights */`
end
$u_1 = U[0, 1/n]$ `/* initialize the first CDF bin */`
$i = 1$
for $j \in [1, n]$ **do**
  while $u_j > c_i$ **do**
    $i \mathrel{+}= 1$ `/* skip until the next CDF ring boundary */`
  end
  $S' = S' \cup \{\langle S[i]_x, 1/n \rangle\}$ `/* insert sample from CDF ring */`
  $u_{j+1} = u_j + 1/n$
end
return S'

12.4 Real Tracking

We've been defining things very loosely in our mathematics. When regarding an object's state, we just called it X. But what *is* X? What representation of features describing the object do we put it to reliably track it? How do those features interact with our measurements. Similarly, what *are* our measurements, Z? What do they measure, and does measuring those things really make us more certain about X?

A lot of the content in this section comes from this paper (in addition to the lectures, obviously), which was the paper that brought these probabilistic tracking models to computer vision. Feel free to give it a read for historical context and deeper technical explanations.

12.4.1 Tracking Contours

Suppose we wanted to track a hand, which is a fairly complex object. The hand is moving (2 degrees of freedom: x and y) and rotating (1 degree of freedom: θ), but it also can change shape. Using principal component analysis, a topic covered soon in chapter 14, we can encode its shape and get a total of 12 degrees of freedom in our state space; that requires a *looot* of particles.

Figure 12.7: Tracking the movement of a hand using an edge detector.

What about measurement? Well, we'd expect there to be a relationship between edges in the picture and our state. Suppose X is a contour of the hand. We can measure the edges and say Z is the sum of the distances from the nearest high-contrast features (i.e. edges) to the contour, as in Figure 12.8. Specifically,

$$\Pr[\mathbf{z} \mid X] \propto \exp\left(-\frac{(d \text{ to edge})^2}{2\sigma^2}\right)$$

...which looks an awful lot like a Gaussian; it's proportional to the distance to the nearest strong edge. We can then use this Gaussian as our sensor model and track hands reliably.

Figure 12.8: A contour and its normals. High-contrast features (i.e. edges) are sought out along these normals.

Figure 12.9: Using edge detection and contours to track hand movement.

12.4.2 Other Models

In general, you can use any model as long as you can compose all of the aforementioned requirements for particle filters: we need an object state, a way to make predictions, and a sensor model.

As another example, whose effectiveness is demonstrated visually in Figure 12.10, we could track hands and head movement using color models and optical flow. Our state is the location of a colored blog (just a simple (x, y)), our prediction is based upon the calculated optic flow, and our sensor model describes how well the predicted models match the color.

A *Very* Simple Model

Let's make this as simple as we possibly can.

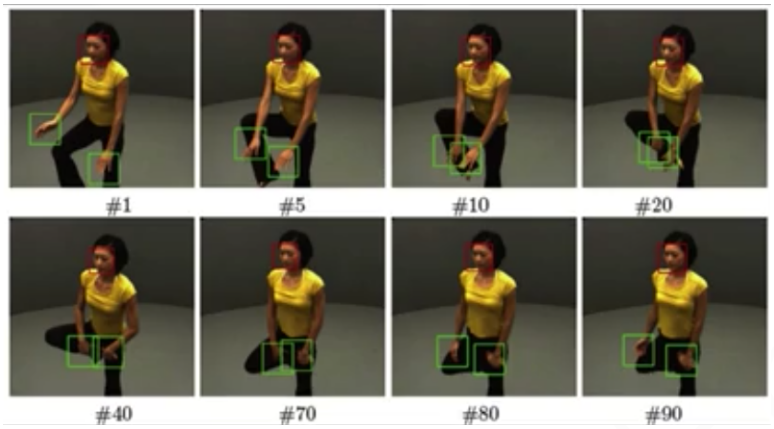


Figure 12.10: Using colored blobs to track a head and hands.

Suppose our state the location of an arbitrary image patch, (x, y)? Then, suppose we don't know anything about, well, anything, so we model the dynamics as literally just being random noise. The sensor model, like we had with our barebones Dense Correspondence Search for stereo, will just be the mean square error of pixel intensities.

Welcome to the model for the next Problem Set in **CS6476**.

12.5 Mean-Shift

The **mean-shift algorithm** tries to find the *modes* of a probability distribution; this distribution is often represented discretely by a number of samples as we've seen. Visually, the algorithm looks something like Figure 12.11 below.

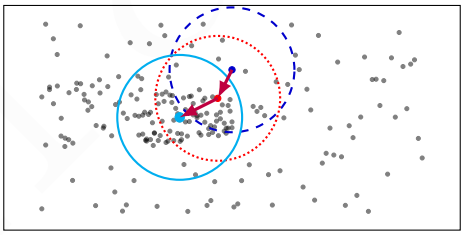


Figure 12.11: Performing mean-shift 2 times to find the area in the distribution with the most density (an approximation of its mode).

This visual example hand-waves away a few things, such as what shape defines the **region of interest** (here it's a circle) and how big it should be, but it gets the point across. At each step (from blue to red to finally cyan), we calculate the mean, or **center of mass** of the region of interest. This results in a **mean-shift vector** from the region's center to the center of mass, which we follow to draw a new region of interest, repeating this process until the mean-shift vector gets arbitrarily small.

So how does this relate to tracking?

Well, our methodology is pretty similar to before. We start with a pre-defined model in the first

frame. As before, this can be expressed in a variety of ways, but it may be easiest to imagine it as an image patch and a location. In the following frame, we search for a region that most closely matches that model within some neighborhood based on some similarity function. Then, the new maximum becomes the starting point for the next frame.

What truly makes this mean-shift tracking is the model and similarity functions that we use. In mean-shift, we use a feature space which is the **quantized color space**. This means we create a histogram of the RGB values based on some discretization of each channel (for example, 4 bits for each channel results in a 64-bin histogram). Our model is then this histogram interpreted as a probability distribution function; this is the region we are going to track.

Let's work through the math. We start with a target model with some histogram centered at 0. It's represented by $\mathbf{q}$ and contains m bins; since we are interpreting it as a probability distribution, it also needs to be normalized (sum to 1):

$$\mathbf{q} = \{q_u\}_{u\in[1..m]} \qquad \sum_{u=1}^{m} q_u = 1$$

We also have some target candidate centered at the point y with its own color distribution, and p_u is now a function of y

$$\mathbf{p}(y) = \{p_u(y)\}_{u\in[1..m]} \qquad \sum_{u=1}^{m} p_u = 1$$

We need a similarity function $f(y)$ to compute the difference between these two distributions now; maximizing this function will render the "best" candidate location: $f(y) = f[\mathbf{q}, \mathbf{p}(y)]$.

12.5.1 Similarity Functions

There are a large variety of similarity functions such as min-value or Chi squared, but the one used in mean-shift tracking is called the **Bhattacharyya coefficient**. First, we change the distributions by taking their element-wise square roots:

$$\mathbf{q}' = (\sqrt{q_1}, \sqrt{q_2}, \dots, \sqrt{q_m})$$
$$\mathbf{p}'(y) = \left(\sqrt{p_1(y)}, \sqrt{p_2(y)}, \dots, \sqrt{p_m(y)}\right)$$

Then, the Bhattacharyya relationship is defined as the sum of the products of these new distributions:

$$f(y) = \sum_{u=1}^{m} p'_u(y) q'_u \tag{12.7}$$

Well isn't the sum of element-wise products the definition of the vector dot product? We can thus also express this as:

$$f(y) = \mathbf{p}'(y) \cdot \mathbf{q}' = \left\|\mathbf{p}'(y)\right\| \left\|\mathbf{q}'\right\| \cos\theta_y$$

But since by design these vectors are magnitude 1 (remember, we are treating them as probability distributions), the Bhattacharyya coefficient essentially uses the $\cos\theta_y$ between these two vectors as a similarity comparison value.

12.5.2 Kernel Choices

Now that we have a suitable similarity function, we also need to define what region we'll be using to calculate it. Recall that in Figure 12.11 we used a circular region for simplicity. This was a fixed region with a hard drop-off at the edge; mathematically, this was a uniform kernel:

$$K_U(\mathbf{x}) = \begin{cases} c & \|\mathbf{x}\| \leq 1 \\ 0 & \text{otherwise} \end{cases}$$

Ideally, we'd use something with some better mathematical properties. Let's use something that's differentiable, isotropic, and monotonically decreasing. Does that sound like anyone we've gotten to know really well over the last 188 pages?

That's right, it's the Gaussian. Here, it's expressed with a constant falloff, but we can, as we know, also have a "scale factor" σ to control that:

$$K_U(\mathbf{x}) = c \cdot \exp\left(-\frac{1}{2}\|\mathbf{x}\|^2\right)$$

The most important property of a Gaussian kernel over a uniform kernel is that it's *differentiable*. The spread of the Gaussian means that new points introduced to the kernel as we slide it along the image have a very small weight that slowly increases; similarly, points in the center of the kernel have a constant weight that slowly decreases. We would see the most weight change along the slope of the bell curve.

We can leverage the Gaussian's differentiability and use its gradient to see how the *overall* similarity function changes as we move. With the gradient, we can actually optimally "hill climb" the similarity function and find its local maximum rather than blindly searching the neighborhood.

This is the big idea in mean-shift tracking: the similarity function helps us determine the new frame's center of mass, and the search space is reduced by following the kernel's gradient along the similarity function.

12.5.3 Disadvantages

Much like the Kalman Filter from before, the biggest downside of using mean-shift as an exclusive tracking mechanism is that it operates on a single hypothesis of the "best next point."

A convenient way to get around this problem while still leveraging the power of mean-shift tracking is to use it as the sensor model in a particle filter; we treating the mean-shift tracking algorithm as a measurement likelihood (from before, $\Pr[z \mid X]$).

12.6 Issues in Tracking

These are universal problems in the world of tracking.

Initialization How do we determine our initial state (positions, templates, scales, etc.)?

> In the examples we've been working with, the assumption has always been that it's determined **manually**: we were *given* a template, or a starting location, or a ...

Another option is **background subtraction**: given a sufficiently-static scene, we can isolate moving objects and use those to initialize state, continuing to track them once the background is restored or we enter a new scene.

Finally, we could rely on a specialized **detector** that finds our interesting objects. Once they are detected, we can use the detection parameters and the local image area to initialize our state and continue tracking them. You might ask, "Well then why don't we just use the detector on every frame?" Well, as we've discussed, detectors rely on a clear view of the object, and generalizing them too much could introduce compounding errors as well as false positives. Deformations and particularly occlusions cause them to struggle.

Sensors and Dynamics How do we determine our dynamics and sensor models?

If we learned the dynamics model from real data (a difficult task), we wouldn't even need the model in the first place! We'd know in advance how things move. We could instead learn it from "clean data," which would be more empirical and an approximation of the real data. We could also use domain knowledge to specify a dynamics model. For example, a security camera pointed at a street can reasonably expect pedestrians to move up and down the sidewalks.

The sensor model is much more finicky. We do need some sense of absolute truth to rely on (even if it's noisy). This could be the reliability of a sonar sensor for distance, a preconfigured camera distance and depth, or other reliable truths.

Prediction vs. Correction Remember the fundamental trade-off in our Kalman Filter: we needed to decide on the relative level of noise in the measurement (correction) vs. the noise in the process (prediction). If one is too strong, we will ignore the other. Getting this balance right is unfortunately just requires a bit of magic and guesswork based on any existing data.

Data Association We often aren't tracking just one thing in a scene, and it's often not a simple scene. How do we know, then, which measurements are associated with which objects? And how do we know which measurements are the result of visual clutter? The camoflauge techniques we see in nature (and warfare) are designed to intentionally introduce this kind of clutter so that even *our* vision systems have trouble with detection and tracking. Thus, we need to reliably associate *relevant* data with the state.

The simple strategy is to only pay attention to measurements that are closest to the prediction. Recall when tracking hand contours (see Figure 12.8) we relied on the "**nearest** high-contrast features," as if we knew those were truly the ones we were looking for.

There is a more sophisticated approach, though, which relies on keeping multiple hypotheses. We can even use particle filtering for this: each particle becomes a hypothesis about the state. Over time, it becomes clear which particle corresponds to clutter, which correspond to our interesting object of choice, and we can even determine when *new* objects have emerged and begin to track those independently.

Drift As errors in each component accumulate and compound over time, we run the risk of drift in our tracking.

One method to alleviate this problem is to update our models over time. For example, we could introduce an α factor that incorporates a blending of our "best match" over time with

a simple linear interpolation:

$$\text{Model}(t) = \alpha\text{Best}(t) + (1 - \alpha)\text{Model}(t - 1) \tag{12.8}$$

There are still risks with this adaptive tracking method: if we blend in too much noise into our sensor model, we'll eventually be tracking something completely unrelated to the original template.

That ends our discussion of tracking. The notion we introduced of tracking state over time comes up in computer vision a lot. This isn't image processing: things change often!

We introduced probabilistic models to solve this problem. Kalman Filters and Mean-Shift were methods that rendered a single hypothesis for the next best state, while Particle Filters maintained multiple hypotheses and converged on a state over time.

Planning

> *Life is what happens to us while we are making other plans.*
>
> — Allen Saunders

The various tracking methods we covered in the previous chapter gives us information about the world. Now that we are "localized," how can we use that information to actually take action and achieve some goals? Enter **planning**.

There are many situations in which we have to find the optimal path from a starting location to some destination. It could come from a literal use-case, like how a mapping application tries to find you the optimal way to stop at In-n-Out on your trip across California, or it could come from a more figurative use-case, like deciding the optimal chess move by searching the "game tree." As far as the algorithms are concerned, these are equivalent: you are searching through a set of possible states and transitions between them to find an optimal path.

A Tale of Two Apps: **Google vs. Waze**

Fascinatingly-enough, these two routing applications (which ultimately belong to the same company, Alphabet) often offer completely different routes from one destination to another. This can largely be attributed to a philosophical difference that drives the two applications: Google Maps is more concerned with a simple, familiar, and easy-to-follow route from one place to another that takes big, recognizable streets and sometimes offers slight alternatives that may save time but typically don't introduce extra complexity; on the other hand, Waze is community-driven, meaning you can use it to aggressively find convoluted shortcuts that will shave milliseconds off of your trip time at your tires' expense (given all of the extra turning you'll be doing).

When we get into the idea of a cost function shortly, we will see how different choices for this can greatly affect the plan.

Check out this discussion for a little more on Waze and Google if you're interested.

When discussing these planning, path-finding, or search algorithms (these terms are often interchangeable) we typically use an example of a maze. Given that we are working with robots and

have real contexts that we can apply this to, we'll be considering how an autonomous vehicle can navigate a handful of intersections without maiming its passengers.

Consider the following situation, in which the blue car wants to get to the specified destination and has two plans:

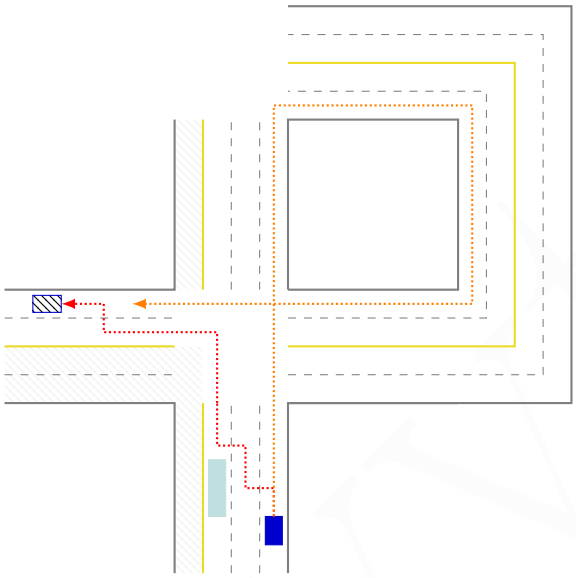


Figure 13.1: A complicated, ever-evolving situation we may commonly face when driving.

The red plan has a high risk: merging across two lanes in such a short span of time (with the optional theoretical teal truck in the way) and will also likely require waiting at a red light, but it's also the shortest path. On the other hand, the orange path is simpler and doesn't run the same risks, however it traverses a further distance. Which path do you take?

> Fun Fact: **UPS Driving Patterns**
>
> If you follow the matra of UPS drivers, the orange path is always the better choice: UPS drivers almost never turn left.

More importantly, what path does an autonomous vehicle take, and how can we encode this algorithmically?

In this chapter, we'll cover robot motion planning methods, starting with *discrete* methods in which the world is "chopped up" into small bins and moving on to *continuous* motion planning methods. Our problem can be defined more concretely. Given:

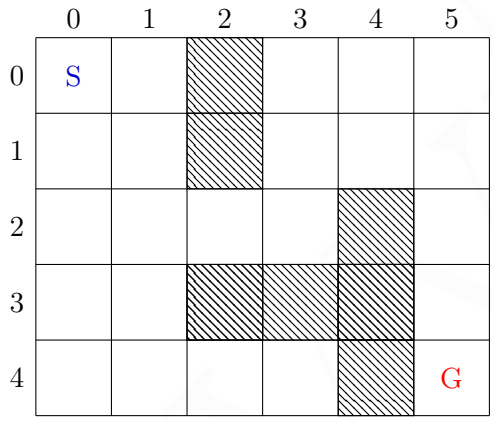
- a map
- a starting location
- a goal location
- a cost function

We want to find **the path from start $\longrightarrow$ goal with the lowest cost**.

Once we have the algorithms in place, we'll see that the only thing that really varies is the cost function. Let's define it a little more. A **cost function** defines an association between an action with a cost. For example, "just go straight" might cost 1, whereas going diagonally might cost $\sqrt{2} \approx 1.4$ (thank u Pythagoras). In such a world, going diagonally would always be better than going straight, turning, and going straight again!

13.1 An Algorithm: Greedy Search

As is tradition, let's imagine a simple grid-based maze:

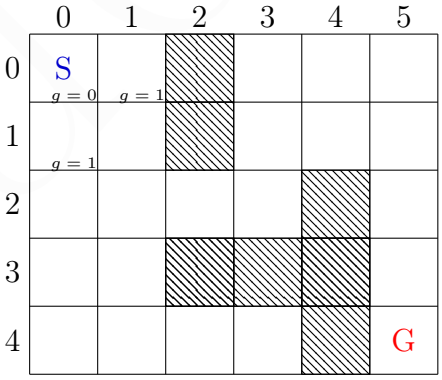


Where we wish to reach the goal from the starting point. Suppose our set of possible actions is movement in any of the 4 directions (no diagonals) and they all cost the same.

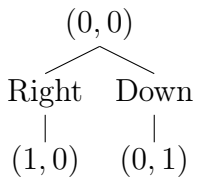
We begin exploration from $(0, 0)$ by *expanding* to neighboring nodes, meaning they get added to what we call the "open list." Furthermore, we've expanded on $(0, 0)$ and returning to it wouldn't make sense, so it gets added to the "closed list":

$$\text{open} = [(1, 0), (0, 1)] \qquad \text{closed} = [(0, 0)]$$

Moving to both of these from our root costs 1 unit which we accumulate into what's called the g-score, so our state is something like this:

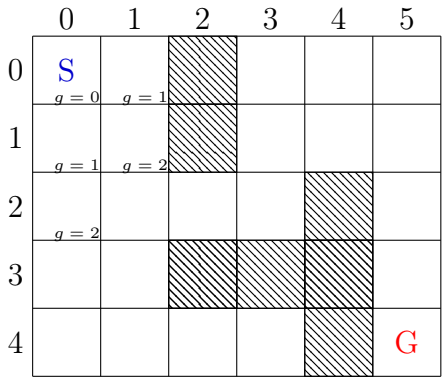


(a) The partially-explored map.

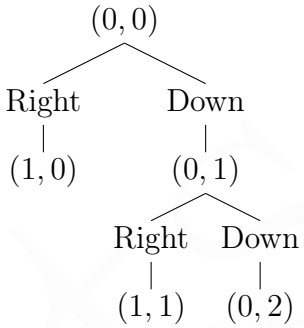


(b) The exploration of the game tree.

From there, we expand on the lowest-cost node in the open list. In this case they have equal scores so the choice is arbitrary; let's go down.

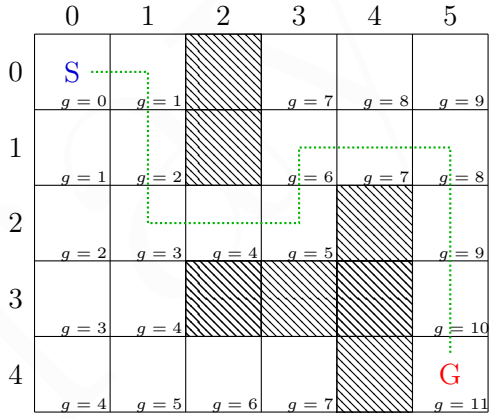


(a) The partially-explored map.



(b) The exploration of the game tree.

We continue this process recursively, continually expanding the node with the lowest g-score, adding its neighbors to the open list while ignoring nodes on the closed list. The whole algorithm is described in algorithm 13.1. Finally, we end up with **a** path to the goal with the lowest possible cost (note that the exact path depends on tie-breaking strategies):



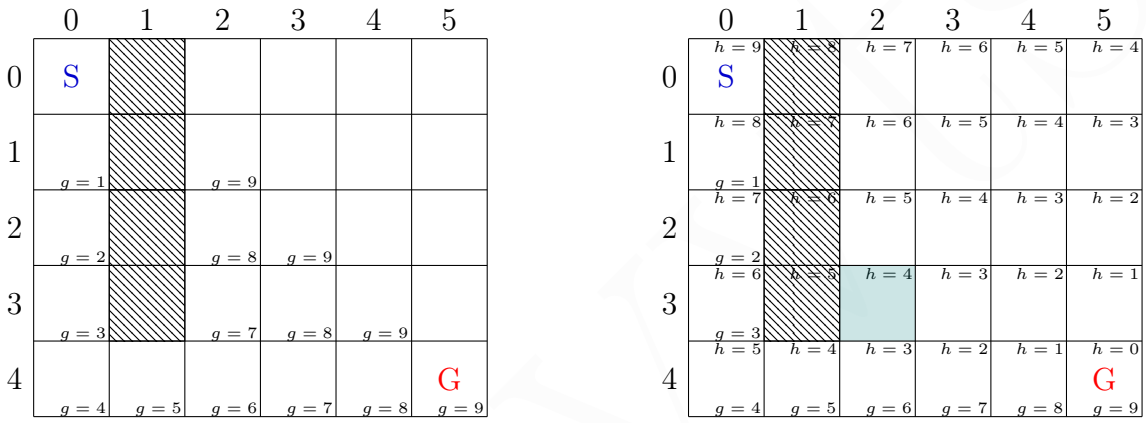
We can (but won't) prove the fact that this is the optimal result because we always (that is, greedily) choose to explore towards the lowest g-score which represents the accumulated cost. This method works but has a massive downside: we had to explore a large chunk of the map that led us nowhere to find this path.

13.2 Heuristic-Based Search

We can avoid searching too much of the map by introducing the concept of a **heuristic** to our search. A hueristic is a rough approximation about how far away we are from the goal at a given cell; it can help break ties and actually help us avoid exploring large swaths of the map. While the g-score is our total cost *from the start*, the heuristic is an approximate cost *to the goal*. Their sum typically provides a good metric for the "optimality" of a cell and is the basis for the **A* search algorithm**.

For grid-based search, for example, a common heuristic is the **Manhattan distance**, which is a cell's $\Delta x + \Delta y$ from the goal.[1] For example, the Manhattan distance at $(1, 2)$ in the grid above is $(5 - 1) + (4 - 2) = 6$. If you wanted to allow diagonal movements, you could just use the Euclidean distance as your heuristic. The most important part about the heuristic is that it **must** be optimistic—it should **never overestimate**. More specifically, $\forall x, y : h(x, y) \leq \text{truth}$, where the *truth* is how long it really would take to reach the goal from (x, y). Following this rule will guarantee that A* will find the optimal solution while doing the minimal amount of work.

To illustrate A*, we will use the same grid as before but with different obstacles. On the left is the exploration after a greedy search, while on the right is the exploration using A* with a Manhattan distance heuristic:



Notice that A* only explores the exact cells it needs to. The incorporation of the heuristic lets it automatically prefer cells that get closer to the goal. There is no sense in exploring $(2, 3)$, for example (marked in teal above) when it takes us further from G than $(3, 4)$.

The only difference between greedy search and A* is that we now track the f-value, the sum of the accumulated cost and the heuristic:

$$f = g + h(x, y)$$

We now choose the node with the lowest f value rather than g value. That's it! That's A*. I highly recommend watching this video on A* in action in a self-driving car simulation that combines localization and search to plan paths that traverse complicated mazes and elaborate scenarios.

13.3 An Algorithm: The All-Seeing Ey

Recall our situation in Figure 13.1: we were unsure if we would have the time to get ahead of (or behind) the truck to make the left turn. As we act, the world around us changes. There is a chance that we won't be able to make the turn, so it's useful to know what the optimal strategy is after we instead cross the intersection.

In fact, it'd be useful to know what the optimal strategy to reach a goal is at *any* point on the map. Of course, we could perform A* from every cell, but that would be computational overkill. Instead,

[1] The term "Manhattan" distance comes from the fact that it's a metric of the amount of blocks you'd need to walk through Manhattan's grid to get to your destination.

we will explore an algorithm that uses dynamic programming to determine the best action at every cell; this will be our "policy" map. For example:

	0	1	2	3	4	5
0	↓	↓		→	↓	↓
1	↓	↓		→	→	↓
2	→	→	→	↑		↓
3	↑	↑				↓
4	↑	↑	←	←		G

To see how we can compute this policy map efficiently, let's introduce a new concept: value. The *value* of a cell is how much it costs to get to the goal from that cell. Naturally, the value at the goal is 0. The value at its immediate neighbors is 1. At their neighbors, 2, and so on... We can compute the value by recursively iterating like this from the goal:

$$f(x, y) = \min_{x', y'} \left[f(x', y') + 1 \right]$$

where our initial values are:

$$f(x, y) = \begin{cases} 0 & \text{if } (x, y) \text{ is the goal} \\ \infty & \text{otherwise} \end{cases}$$

ALGORITHM 13.1: A greedy search algorithm.

Input: A grid-like map of the world, M.
Input: A mapping of actions and their respective costs, $F(a) \mapsto c$.
Input: The start and goal locations, S and G.
Result: An optimal path, P^*.

```
/* A way to track the costs of closed nodes (0=open) in the world.  */
```
$closed := \text{ZEROES}(M.shape)$
$closed[S] := 1$

```
/* Start the open list with our initial location */
```
$p := (S_x, S_y)$
$g := 0$
$open := [(g, p)]$

while *true* **do**
  if $open = \emptyset$ **then**
    return *failure*
  end
  $n_g, n_x, n_y :=$ node in *open* with lowest g-score
  $g = n_g$
  if $(n_x, n_y) = G$ **then**
    break
  end
  `/* Store each unexplored action result in the open list.  */`
  foreach *valid action* $a := (dx, dy)$ **do**
    $p_a := (n_x + dx, n_y + dy)$
    if $closed[p_a] = 0$ **then**
      $g_a := g + F(a)$
      $closed[p_a] := g_a$
      $\text{PUSH}(open, (g_a, p_a))$
    end
  end
end

```
/* Now that we know we have a path with the lowest cost, simply traverse
   the map from G → S while choosing the cheapest node to find it.  */
```
$P^* := \emptyset$
$p := G$
while $p \neq S$ **do**
  $p := \min_{n \in \text{NEIGHBORS}(M[p])} \text{closed}[p]$
  $\text{PUSH}(P^*, p)$
end
return P^*

Recognition

> *Recognizing isn't at all like seeing; the two often don't even agree.*
>
> — Sten Nadolny, *The Discovery of Slowness*

Until this chapter, we've been working in a "semantic-free" vision environment. Though we have detected "interesting" features, developed motion models, and even tracked objects, we have had no understanding of what those objects actually *are*. Yes, we had Template Matching early on in chapter 2, but that was an inflexible, hard-coded, and purely mathematical approach of determining the existence of a template in an image. In this chapter, we'll instead be focusing on a more human-like view of recognition. Our aim will be to dissect a scene and label (in English) various objects within it or describe it somehow.

There are a few main forms of recognition: **verification** of the class of an object ("Is this a lamp?"), **detection** of a class of objects ("Are there any lamps?"), and **identification** of a specific instance of a class ("Is that the Eiffel Tower?"). More generally, we can also consider **object categorization** and label specific general areas in a scene ("There are trees here, and some buildings here.") without necessarily identifying individual instances. Even *more* generally, we may want to describe an image as a whole ("This is outdoors.")

We'll primarily be focusing on generic object categorization ("Find the cars," rather than "Find Aaron's car"). This task can be presented as such:

> *Given a (small) number of training images as examples of a category, recognize "a-priori" (previously) unknown instances of that category and assign the correct category label.*

This falls under a large class of problems that machine learning attempts to solve, and many of the methods we will discuss in this chapter are applications of general machine learning algorithms. The "small" aspect of this task has not gotten much focus in the machine learning community at large; the massive datasets used to train recognition models present a stark contrast to the handful of examples a human may need to reliably identify and recognize an object class.

Categorization

Immediately, we have a problem. Any object can be classified in dozens of ways, often following some hierarchy. Which category should we take? Should a hot dog be labeled as food, as a sandwich, as protein, as an inanimate object, or even just as something red-ish? This is a manifestation of prototype theory, tackled by psychologists and cognitive scientists. There are many writings on the

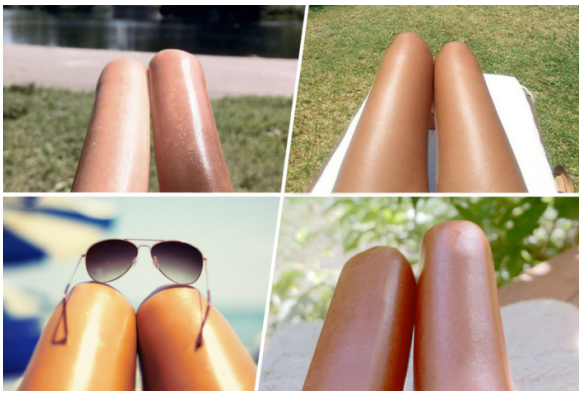


Figure 14.1: Hot dogs or legs? You might be able to differentiate them instantly, but how can we teach a computer to do that? (Image Source)

topic, the most impactful of which was Rosch and Lloyd's *Cognition and categorization*, namely the Principles of Categorization chapter, which drew a few conclusions from the standpoint of human cognition. The basic level category for an object can be based on...

- the highest level at which category members have a similar perceived shape. For example, a German shepherd would likely be labeled that, rather than as a dog or a mammal, which have more shape variation.
- the highest level at which a single mental image can be formed that reflects the entire category.
- the level at which humans identify category members fastest.
- the first level named and understood by children.
- the highest level at which a person uses similar motor actions for interaction with category members. For example, the set of actions done with a dog (petting, brushing) is very different than the set of actions for all animals.

Categorization Reaction Time

Experiments have been done that give us insight to the categorization layout in the human brain.

Given a series of images, participants were asked to answer a question about the image: "Is it an animal?" and "Is it a dog?" Measuring the subsequent response reaction time has robustly shown us that humans respond faster to whether or not something is a dog than whether or not something is an animal.

This might make sense intuitively; the "search space" of animals is far greater than that of dogs, it should take longer. Having scientific evidence to back that intuition up is invaluable, though, and showing that such categories must somehow exist—and even have measurable differences—is a significant insight into human cognition.

Even if we use some of these basic level categories, what scope are we dealing with? Psychologists

have given a range of 10,000 to 30,000 categories for human cognition. This gives us an idea of scale when dealing with label quanities for recognition.

Recognition is important in computer vision because it allows us to create relationships between specific objects. Things are longer just an "interesting feature," but something far more descriptive that can then influence subsequent logic and behavior in a larger system.

Challenges

Why is this a hard problem? There are many multivariate factors that influence how a particular object is perceived, but we often don't have trouble working around these factors and still labeling it successfully.

Factors like illumination, pose, occlusions, and visual clutter all affect the way an object appears. Furthermore, objects don't exist in isolation. They are part of a larger scene full of other objects that may conflict, occlude, or otherwise interact with one another. Finally, the computational complexity (millions of pixels, thousands of categories, and dozens of degrees of freedom) of recognition is daunting.

State-of-the-Art

Things that seemed impossibles years ago are commonplace now: character and handwriting recognition on checks, envelopes, and license plates; fingerprint scans; face *detection*; and even recognition of flat, textured objects (via the SIFT Detector) are all relatively solved problems.

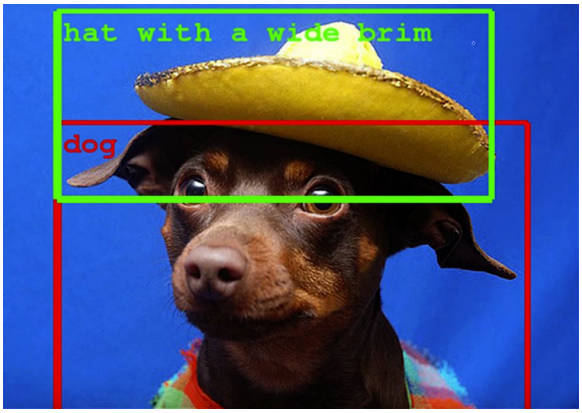


Figure 14.2: An example of GoogleNet labeling different parts of an image, differentiating between a relatively unique pair of objects without any additional context.

The current cutting edge is far more advanced. As demonstrated in Figure 14.2, individual components of an image are being identified and labeled independently. The likelihood of thousands of photos of dogs wearing Mexican hats exist and can be used for training is unlikely, so dissecting this composition is an impressive result.

As we've mentioned, these modern techniques of strong label recognition are really machine learning algorithms applied to patterns of pixel intensities. Instead of diving into these directly (which is a topic more suited to courses on machine learning and deep learning), we'll discuss the general principles of what are called generative vs. discriminative methods of recognition and the image representations they use.

Supervised Classification Methods

Given a collection of labeled examples, we want to come up with a function that will predict the labels of new examples. The existence of this set of labeled examples (a "training set") is what makes this *supervised* learning.

The function we come up with is called a **classifier**. Whether or not the classifier can be considered "good" depends on a few things, including the mistakes that it makes and the cost associated with these mistakes. Since we know what the desired outputs are from our training set, our goal is to minimize the expected misclassification. To achieve this, there are two primary strategies:

- **generative**: use the training data to build a representative probability model for various object classes. We model conditional densities and priors separately for each class, meaning that the "recognized" class of an object is just the class that fits it best.
- **discriminative**: directly construct a good decision boundary between a class and everything not in that class, modeling the posterior.

Word to the Wise

There is a lot of probability in this chapter. I try to work through these concepts slowly and intuitively both for my own understanding and for others who have a weak (or non-existent) foundation in statistics. If you have a knack for probability or have been exposed to these ideas before, such explanations may become tedious; for that, I apologize in advance.

14.1 Generative Supervised Classification

Before we get started, here is some trivial notation: we'll say that $4 \to 9$ means that the object is a 4, but we called it a 9. With that, we assume that the **cost** of $X \to X$ is 0.

Consider, then, a binary decision problem of classifying an image as being of a 4 or a 9.

$L(4 \to 9)$ is the loss of classifying a 4 as a 9, and similarly $L(9 \to 4)$ is the loss of classifying a 9 as a 4. We define the **risk** of a classifier strategy S as being the expected loss. For our binary decision, then:

$$R(S) = \Pr[4 \to 9 \,|\, \text{using } S] \cdot L(4 \to 9) + \Pr[9 \to 4 \,|\, \text{using } S] \cdot L(9 \to 4)$$

Figure 14.3: An arrangement of 4s to 9s.

We can say that there is some feature vector $\mathbf{x}$ that describes or measures the character. At the *best* possible decision boundary (i.e. right in the middle of Figure 14.3), either label choice will yield the same expected loss.

If we picked the class "four" at that boundary, the expected loss is the probability that it's actually a 9 times the cost of calling it a 9 plus the probability that it's actually a 4 times the cost of calling it a 4 (which we assumed earlier to be zero: $L(4 \to 4) = 0$). In other words:

$$\begin{aligned} &= \Pr[\text{class is } 9 \,|\, \mathbf{x}] \cdot L(9 \to 4) + \Pr[\text{class is } 4 \,|\, \mathbf{x}] \cdot L(4 \to 4) \\ &= \Pr[\text{class is } 9 \,|\, \mathbf{x}] \cdot L(9 \to 4) \end{aligned}$$

Similarly, the expected loss of picking "nine" at the boundary is based on the probability that it was actually a 4:

$$= \Pr[\text{class is } 4 \,|\, \mathbf{x}] \cdot L(4 \to 9)$$

Thus, the best decision boundary is the $\mathbf{x}$ such that:

$$\Pr[\text{class is } 9 \,|\, \mathbf{x}] \cdot L(9 \to 4) = \Pr[\text{class is } 4 \,|\, \mathbf{x}] \cdot L(4 \to 9)$$

With this in mind, classifying a new point becomes easy. Given its feature vector $\mathbf{k}$, we choose the class with the lowest expected loss. We would choose "four" if:

$$\Pr[4 \,|\, \mathbf{k}] \cdot L(4 \to 9) > \Pr[9 \,|\, \mathbf{k}] \cdot L(9 \to 4)$$

How can we apply this in practice? Well, our training set encodes the loss; perhaps it's different for certain incorrect classifications, but it could also be uniform. The difficulty comes in determining our conditional probability, $\Pr[4 \,|\, \mathbf{k}]$. We need to somehow know or *learn* that. As with all machine learning, we trust the data to teach us this.

Suppose we have a training dataset in which each pixel is labeled as being a skin pixel and a non-skin pixel, and we arrange that data into the pair of histograms in Figure 14.4. A new pixel p comes along and it falls into a bin in the histograms that overlap (somewhere in the middle). Suppose $p = 6$, and its corresponding probabilities are:

$$\begin{aligned} \Pr[x = 6 \,|\, \text{skin}] &= 0.5 \\ \Pr[x = 6 \,|\, \neg\text{skin}] &= 0.1 \end{aligned}$$

Is this enough information to determine whether or not p is a skin pixel? In general, can we say that p is a skin pixel if $\Pr[x \,|\, \text{skin}] > \Pr[x \,|\, \neg\text{skin}]$?

No!

Remember, this is the **likelihood** that the hue is some value *already given whether or not it's skin.* To extend the intuition behind this, suppose the likelihoods were equal. Does that suggest we can't tell if it's a skin pixel? Again, absolutely not. Why? Because any given pixel is *much* more likely to not be skin in the first place! Thus, if their hue likelihoods are equal, it's most likely not skin because "not-skin pixels" are way more common.

What we *really* want is the probability of a pixel being skin given a hue: $\Pr[\text{skin} \,|\, x]$, which isn't described in our model. As we did in Particle Filters, we can apply Bayes' Rule, which expresses

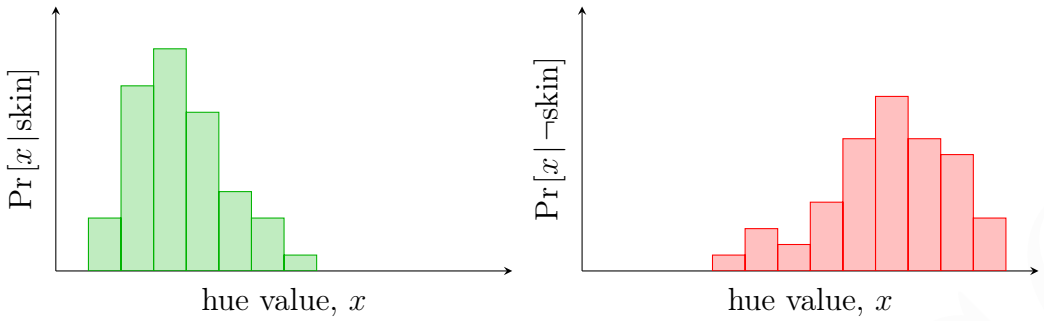


Figure 14.4: Histograms from a training set determining the likelihood of a particular hue value, x, occurring given that a pixel **is** (left) or **is not** (right) skin.

exactly what we just described: the probability of a pixel being skin given a hue is proportional to the probability of that hue representing skin AND the probability of a pixel being skin in the first place (see Equation 12.4).

$$\underbrace{\Pr[\text{skin}\,|\,x]}_{\text{posterior}} \propto \underbrace{\Pr[x\,|\,\text{skin}]}_{\text{likelihood}} \cdot \underbrace{\Pr[\text{skin}]}_{\text{prior}}$$

Thus, the comparison we really want to make is:

$$\Pr[x\,|\,\text{skin}] \cdot \Pr[\text{skin}] \overset{?}{>} \Pr[x\,|\,\neg\text{skin}] \cdot \Pr[\neg\text{skin}]$$

Unfortunately, we don't *know* the prior, $\Pr[\text{skin}]$, ... but we can assume it's some constant, Ω. Given enough training data marked as being skin,[1] we can *measure* that prior Ω (hopefully at the same time as when we measured the histogram likelihoods from before in Figure 14.4). Then, the binary decision can be made based on the measured $\Pr[\text{skin}\,|\,x]$.

Generalizing the Generative Model We've been working with a binary decision. How do we generalize this to an arbitrary number of classes? For a given measured feature vector $\mathbf{x}$ and a set of classes c_i, choose the best c^* by maximizing the posterior probability:

$$c^* = \arg\max_c \Pr[c\,|\,\mathbf{x}] = \arg\max_c \Pr[c]\Pr[\mathbf{x}\,|\,c]$$

Continuous Generative Models If our feature vector $\mathbf{x}$ is continuous, we can't rely on a discrete histogram. Instead, we can return to our old friend the Gaussian (or a mixture of Gaussians) to create a smooth, continuous approximation of the likelihood density model of $\Pr[\mathbf{x}\,|\,c]$.

Using Gaussians to create a parameterized version of the training data allows us to express the probability density much more compactly.

14.2 Principal Component Analysis

One thing that may have slipped by in our discussion of generative models is that they work best in low-dimensional spaces. Since we are building probabilistic descriptions (previously, we used

[1] from graduate students or mechanical Turks, for instance...

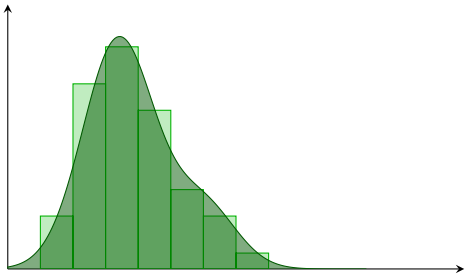


Figure 14.5: A mixture of Gaussians approximating the histogram for $\Pr[x|\text{skin}]$ from Figure 14.4.

histograms to build density approximations) from data, building these robustly for high-dimensional classification requires a *lot* of data. A method for reducing the necessary dimensionality is called **principal component analysis**. Despite being used elsewhere, this is a very significant idea in the computer vision space.

A *component* is essentially a direction in a feature space. A *principal* component, then, is the direction along which points in that feature space have the greatest variance.

The first principal component is the direction of maximum variance. Subsequent principal components are orthogonal to the previous principal components and describes the "next" direction of maximum residual variance.

> **Ackshually...**
>
> Principal components point in the direction of maximum variance *from the mean* of the points. Mathematically, though, we express this as being a direction from the origin. This means that in the nitty-gritty, we translate the points so that their mean is at the origin, do our PCA math, then translate everything back. Conceptually (and graphically), though, we will express the principal component relative to the mean.

We'll define this more rigorously soon, but let's explain the "direction of maximum variance" a little more more intuitively. We know that the variance describes how "spread out" the data is; more specifically, it measures the average of the squared differences from the mean. In 2 dimensions, the difference from the mean (the target in Figure 14.6) for a point is expressed by its distance. The direction of maximum variance, then, is the line that most accurately describes the direction of spread. The second line, then, describes the direction of the remaining spread relative to the first line.

Okay, that may not make a lot of sense yet, but perhaps diving a little deeper into the math will explain further. You may have noticed that the first principal component in Figure 14.6 resembles the line of best fit. We've seen this sort of scenario before... Remember when we discussed Error Functions, and how we came to the conclusion that minimizing perpendicular distance was more stable than measuring vertical distance when trying to determine whether or not a descriptor is a true match to a feature?

We leveraged the expression of a line as a normal vector and a distance from the origin (in case you

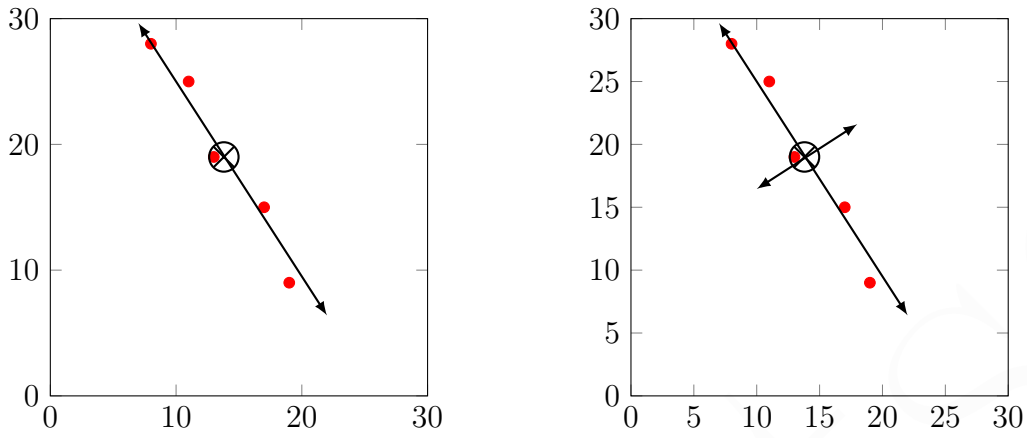


Figure 14.6: The first principal component and its subsequent orthogonal principal component.

forgot how we expressed that, it's replicated again in Figure 14.7). Then, our error function was:

$$E(a, b, d) = \sum_i (ax_i + by_i - d)^2$$

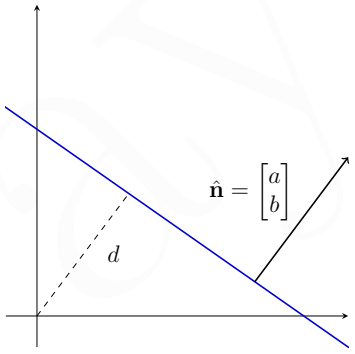


Figure 14.7: Representing a line, $ax + by = d$ as a normal vector and distance from the origin.

We did some derivations on this perpendicular least-squares fitting—which were omitted for brevity—and it gave us an expression in matrix form, $\|\mathbf{B}\hat{\mathbf{n}}\|^2$, where

$$\mathbf{B} = \begin{bmatrix} x_1 - \overline{x} & y_1 - \overline{y} \\ x_2 - \overline{x} & y_2 - \overline{y} \\ \vdots & \vdots \\ x_n - \overline{x} & y_n - \overline{y} \end{bmatrix}$$

We wanted to minimize $\|\mathbf{B}\hat{\mathbf{n}}\|^2$ subject to $\|\hat{\mathbf{n}}\| = 1$, since $\hat{\mathbf{n}}$ is expressed as a unit vector. What we didn't mention was that geometrically (or physically?) this was the **axis of least inertia**. We could imagine spinning the xy-plane around that axis, and it would be the axis that resulted in the least movement of all of those points. Perhaps you've heard this referenced as the *moment* of inertia? I sense eigenvectors in our near future...

How about *another* algebraic interpretation? In **??**, we used the projection to explain how the standard least squares method for solving a linear system aims to minimize the distance from the projection to the solution vector. Again, that visual is reiterated in Figure 14.8.

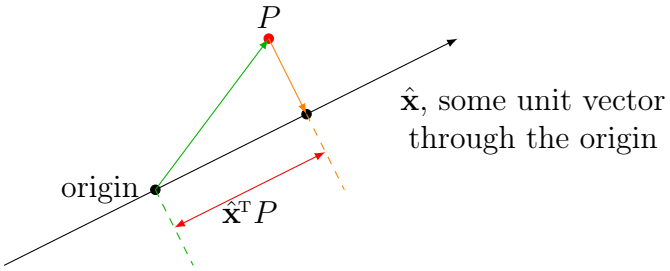


Figure 14.8: The projection of the point P onto the vector $\hat{\mathbf{x}}$.

Through a basic understanding of the Pythagorean theorem, we can see that minimizing the distance from P to the line described by $\hat{\mathbf{x}}$ is the same thing as *maximizing* the projection $\hat{\mathbf{x}}^{\mathrm{T}}P$ (where $\hat{\mathbf{x}}$ is expressed a *column* vector). If we extend this to all of our data points, we want to maximize the sum of the squares of the projections of those points onto the line described by the principal component. We can express this in matrix form through some clever manipulation. If $\mathbf{B}$ is a matrix of the points, then:

$$\mathbf{x}^{\mathrm{T}}\mathbf{B}^{\mathrm{T}}\mathbf{B}\mathbf{x} =$$

$$\begin{bmatrix} a & b \end{bmatrix} \begin{bmatrix} x_1 & x_2 & \dots x_n \\ y_1 & y_2 & \dots y_n \end{bmatrix} \begin{bmatrix} x_1 & y_1 \\ x_2 & y_2 \\ \vdots & \vdots \\ x_n & y_n \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix} = \sum_i (\mathbf{x}^{\mathrm{T}}P_i)^2$$

More succinctly, our goal is to maximize $\mathbf{x}^{\mathrm{T}}\mathbf{B}^{\mathrm{T}}\mathbf{B}\mathbf{x}$ subject to $\mathbf{x}^{\mathrm{T}}\mathbf{x} = 1$. We say $\mathbf{M} = \mathbf{B}^{\mathrm{T}}\mathbf{B}$. Then, this becomes a constrained optimization problem to which we can apply the **Lagrange multiplier** technique:[2]

$$E = \mathbf{x}^{\mathrm{T}}\mathbf{M}\mathbf{x}$$
$$E' = \mathbf{x}^{\mathrm{T}}\mathbf{M}\mathbf{x} + \lambda(1 - \mathbf{x}^{\mathrm{T}}\mathbf{x})$$

Notice that we expect, under the correct solution, for $1 - \mathbf{x}^{\mathrm{T}}\mathbf{x} = 0$. We take the partial derivative of our new error function and set it equal to 0:

$$\frac{\partial E'}{\partial \mathbf{x}} = 2\mathbf{M}\mathbf{x} + 2\lambda\mathbf{x}$$
$$0 = 2\mathbf{M}\mathbf{x} + 2\lambda\mathbf{x}$$
$$\mathbf{M}\mathbf{x} = \lambda\mathbf{x}$$

λ is any constant, so it absorbs the negative

That's a very special $\mathbf{x}\dots$ it's the definition of an eigenvector![3] $\mathbf{x}$ is an eigenvector of $\mathbf{B}^{\mathrm{T}}\mathbf{B}$, our matrix of points.

[2] I discuss this method in further detail (far beyond the scope to which it's described in lecture) in the **??** section of **??**.

[3] Briefly recall that an eigenvector of a matrix is a vector that, when multiplied by said matrix, is just a scalar multiple of itself.

How about yet another (last one, we promise) algebraic interpretation? Our aforementioned matrix product, $\mathbf{B}^T\mathbf{B}$, if we were to expand it and express it with the mean at the origin, would look like:

$$\mathbf{B}^T\mathbf{B} = \begin{bmatrix} \sum_{i=1}^{n} x_i^2 & \sum_{i=1}^{n} x_i y_i \\ \sum_{i=1}^{n} x_i y_i & \sum_{i=1}^{n} y_i^2 \end{bmatrix}$$

...that's the covariance matrix of our set of points, describing the variance in our two dimensions. What we're looking for, then, are the eigenvectors of the covariance matrix of the data points.

Generally, for an $n \times n$ matrix, there are n distinct eigenvectors (except in degenerate cases with duplicate eigenvalues, but we'll ignore those for now). Thus, we can describe principal components as being the eigenvectors of the points.

14.2.1 Dimensionality Reduction

With our relationship between principal components and eigenvectors out of way, how can we use this to solve the low-dimensionality limitations of generative models we described earlier?

The idea is that we can collapse a set of points to their largest eigenvector (i.e. their primary principal component). For example, the set of points from Figure 14.6 will be collapsed to the blue points in Figure 14.9; we ignore the second principal component and only describe where on the line the points are, instead.

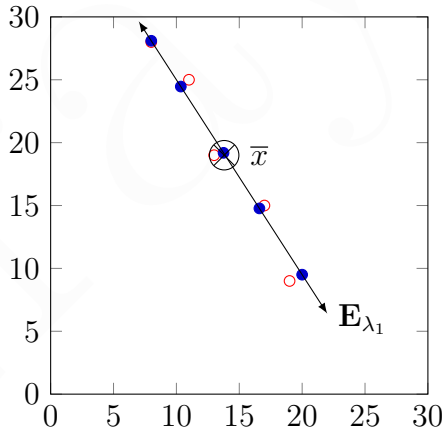


Figure 14.9: Collapsing a set of points to their principal component, $\hat{\mathbf{E}}_{\lambda_1}$. The points can now be represented by a coefficient—a scalar of the principal component unit vector.

Collapsing our example set of points from two dimensions to one doesn't seem like that big of a deal, but this idea can be extended to however many dimensions we want. Unless the data is uniformally random, there will be directions of maximum variance; collapsing things along them (for however many principal components we feel are necessary to accurately describe the data) still results in massive dimensionality savings.

Suppose each data point is now n-dimensional; that is, $\mathbf{x}$ is an n-element column vector. Then, to acquire the maximum direction of projection, $\hat{\mathbf{v}}$, we follow the same pattern as before, taking the

sum of the magnitudes of the projections:

$$\text{var}(\hat{\mathbf{v}}) = \sum_{\mathbf{x}} \|\hat{\mathbf{v}}^{\mathrm{T}}(\mathbf{x} - \overline{\mathbf{x}})\|$$

We can isolate our terms by leveraging the covariance matrix from before; given $\mathbf{A}$ as the **outer product**:[4] $\mathbf{A} = \sum_{\mathbf{x}} (\mathbf{x} - \overline{\mathbf{x}})(\mathbf{x} - \overline{\mathbf{x}})^{\mathrm{T}}$, we can say

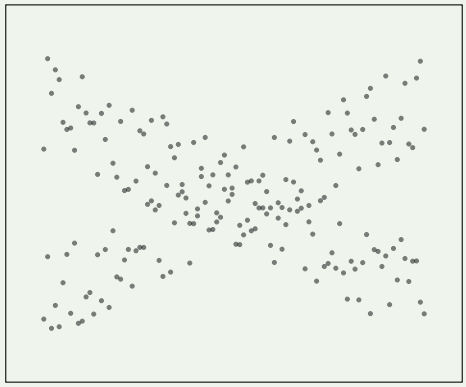
$$\text{var}(\mathbf{v}) = \mathbf{v}^{\mathrm{T}}\mathbf{A}\mathbf{v}$$

As before, the eigenvector with the largest eigenvalue λ captures the most variation among the training vectors $\mathbf{x}$.

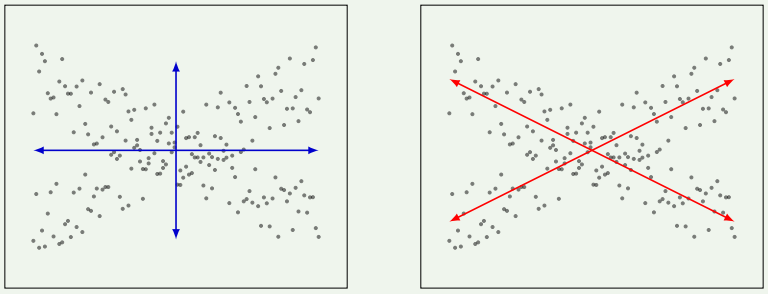
With this background in mind, we can now explore applying principal component analysis to images.

Example 14.1: Understanding Check

What eigenvectors make up the principal components in the following dataset? Remember, we're looking for the directions of maximum variance.



Is it the ones on the right or on the left?



The eigenvectors found by PCA would've be the blue vectors; the red ones aren't

[4] Remember, $\mathbf{x}$ is a *column* vector. Multiplying a column vector $\mathbf{n}$ by its transpose: $\mathbf{n}\mathbf{n}^{\mathrm{T}}$ is the definition of the outer product and results in a matrix. This is in contrast to multiplying a *row* vector by its transpose, which is the definition of the **dot** product and results in a scalar.

even perpendicular! Intuitively, we can imagine the points above the blue line "cancelling out" the ones below it, effectively making their average in-between them. Principal component analysis works best when training data comes from a single class. There are other ways to identify classes that are not orthogonal such as ICA, or **independent component analysis**.

14.2.2 Face Space

Let's treat an image as a one-dimensional vector. A 100×100 pixel image has 10,000 elements; it's a 10,000-dimensional space! But how many of those vectors correspond to valid face images? Probably much less... we want to effectively model the **subspace** of face images from the general vector space of 100×100 images.

Specifically, we want to construct a *low-dimensional* (PCA anyone?) *linear* (so we can do dot products and such) *subspace* that best explains the variation in the set of face images; we can call this the **face space**.

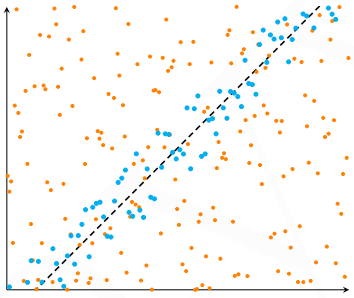


Figure 14.10: Cyan points indicate faces, while orange points are "non-faces." We want to capture the black principal component.

We will apply principal component analysis to determine the principal component(s) in our 10,000-dimension feature space (an incredibly oversimplified view of which is shown in Figure 14.10).

We are given M data points, $\{\mathbf{x}_1, \dots, \mathbf{x}_M\} \in \mathbb{R}^d$, where each data point is a *column* vector and d is very large. We want some directions in $\mathbb{R}^d$ that capture most of the variation of the $\mathbf{x}_i$. If $\mathbf{u}$ is one such direction, the coefficients representing locations *along* that directions would be (where $\boldsymbol{\mu}$ is the mean of the data points):

$$u(\mathbf{x}_i) = \mathbf{u}^{\mathrm{T}}(\mathbf{x}_i - \boldsymbol{\mu})$$

So what is the direction vector $\hat{\mathbf{u}}$ that captures most of the variance? We'll use the same expressions as before, maximizing the variance of the projected data:

$$\begin{aligned}
\operatorname{var}(\mathbf{u}) &= \frac{1}{M}\sum_{i=1}^{M} \underbrace{\mathbf{u}^{\mathrm{T}}(\mathbf{x}_i - \boldsymbol{\mu})}_{\text{projection of } \mathbf{x}_i} \left(\mathbf{u}^{\mathrm{T}}(\mathbf{x}_i - \boldsymbol{\mu})\right)^{\mathrm{T}} \\
&= \mathbf{u}^{\mathrm{T}} \underbrace{\left[\frac{1}{M}\sum_{i=1}^{M} (\mathbf{x}_i - \boldsymbol{\mu})(\mathbf{x}_i - \boldsymbol{\mu})^{\mathrm{T}}\right]}_{\text{covariance matrix of data}} \mathbf{u} && \mathbf{u} \text{ doesn't depend on } i \\
&= \hat{\mathbf{u}}^{\mathrm{T}} \boldsymbol{\Sigma} \hat{\mathbf{u}} && \text{don't forget } \hat{\mathbf{u}} \text{ is a unit vector}
\end{aligned}$$

Thus, the variance of the projected data is $\text{var}(\hat{\mathbf{u}}) = \hat{\mathbf{u}}^{\mathrm{T}}\boldsymbol{\Sigma}\hat{\mathbf{u}}$, and the direction of maximum variance is then determined by the eigenvector of $\boldsymbol{\Sigma}$ with the largest eigenvalue. Naturally, then, the top k orthogonal directions that capture the residual variance (i.e. the principal components) correspond to the top k eigenvalues.

Let $\boldsymbol{\Phi}_i$ be a known face image I with the mean image subtracted. Remember, this is a d-length column vector for a massive d.

Define C to be the average squared magnitude of the face images:

$$\mathbf{C} = \frac{1}{M}\sum_{i \in M}(\boldsymbol{\Phi}_i \boldsymbol{\Phi}_i^{\mathrm{T}}) = \mathbf{A}\mathbf{A}^{\mathrm{T}}$$

where $\mathbf{A}$ is the matrix of our M faces as column vectors, $\mathbf{A} = \begin{bmatrix}\boldsymbol{\Phi}_1 & \boldsymbol{\Phi}_2 & \dots & \boldsymbol{\Phi}_M\end{bmatrix}$; this is of the dimension $d \times M$. That means $\mathbf{C}$ is a massive $d \times d$ matrix.

Consider, instead, $\mathbf{A}^{\mathrm{T}}\mathbf{A}\dots$ that's only an $M \times M$ matrix, for which finding the eigenvalues is much easier and computationally feasible. Suppose $\mathbf{v}_i$ is one of these eigenvectors:

$$\mathbf{A}^{\mathrm{T}}\mathbf{A}\mathbf{v}_i = \lambda \mathbf{v}_i$$

Now pre-multiply that boi by $\mathbf{A}$ and notice the wonderful relationship:

$$\mathbf{A}\mathbf{A}^{\mathrm{T}}\mathbf{A}\mathbf{v}_i = \lambda \mathbf{A}\mathbf{v}_i$$

We see that $\mathbf{A}\mathbf{v}_i$ are the eigenvectors of $\mathbf{C} = \mathbf{A}\mathbf{A}^{\mathrm{T}}$, which we previously couldn't feasibly compute directly! This is the **dimensionality trick** that makes PCA possible: we *created* the eigenvectors of $\mathbf{C}$ without needing to compute them directly.

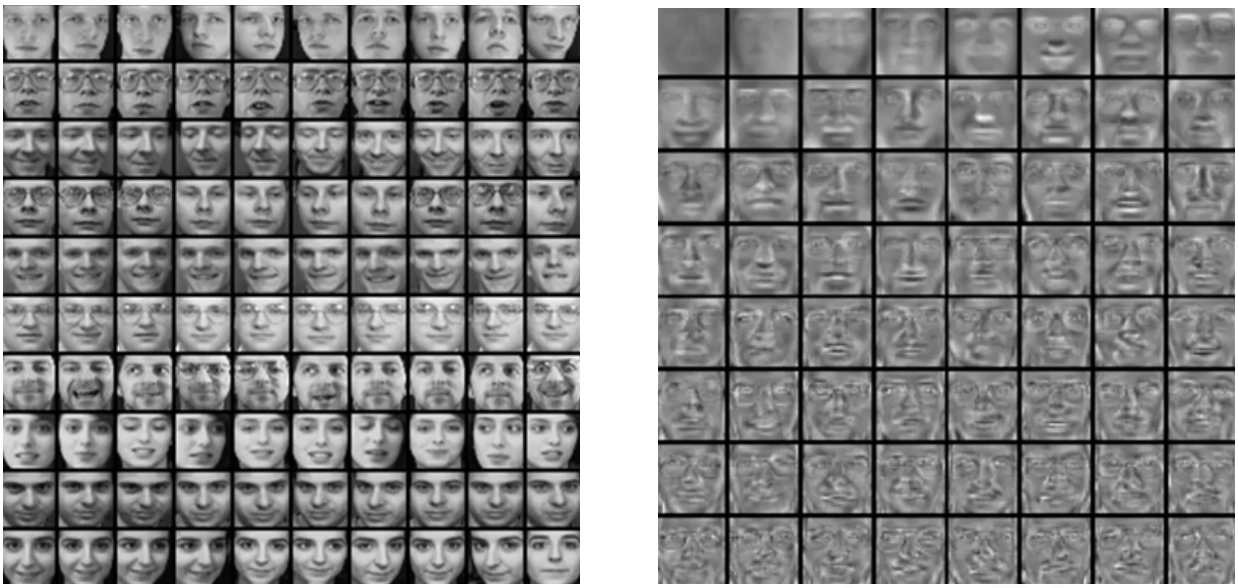
How Many Eigenvectors Are There? Intuition suggests M, but recall that we always subtract out the mean which centers our component at the origin. This means (heh) that if we have a point along it on one side, we know it has a buddy on the other side by just flipping the coordinates. This removes a degree of freedom, so there are $M - 1$ eigenvectors in general.

Yeah, I know this is super hand-wavey.
Just go with it...

14.2.3 Eigenfaces

The idea of **eigenfaces** and a face space was pioneered in this 1991 paper. The assumption is that most faces lie in a low-dimensional subspace of the general image space, determined by some $k \lll d$ directions of maximum variance.

Using PCA, we can determine the eigenvectors that track the most variance in the "face space:" $\mathbf{u}_1, \mathbf{u}_2, \dots, \mathbf{u}_k$. Then any face image can be represented (well, approximated) by a linear combination of these "eigenfaces;" the coefficients of the linear combination can be determined easily by a dot product.



(a) A subset of the training images—isolated faces with variations in lighting, expression, and angle.

(b) The resulting top 64 principal components.

Figure 14.11: Data from Turk & Pentland's paper, "Eigenfaces for Recognition."

We can see in Figure 14.12, as we'd expect, that the "average face" from the training set has an ambigious gender, and that eigenfaces roughly represent variation in faces. Notice that the detail in each eigenface increases as we go down the list of variance. We can even notice the kind of variation some of the eigenfaces account for. For example, the 2nd through 4th eigenfaces are faces lit from different angles (right, top, and bottom).

If we take the average face and add one of the eigenfaces to it, we can see the impact the eigenface has in Figure 14.13. Adding the second component causes the resulting face to be lit from the right, whereas subtracting it causes it to be lit from the left.

Figure 14.12: The mean face from the training data.

We can easily convert a face image to a series of eigenvector coefficients by taking its dot product with each eigenface after subtracting the mean face:

$$\mathbf{x} \to [\mathbf{u}_1 \cdot (\mathbf{x} - \boldsymbol{\mu}), \mathbf{u}_2 \cdot (\mathbf{x} - \boldsymbol{\mu}), \dots, \mathbf{u}_k \cdot (\mathbf{x} - \boldsymbol{\mu})] \tag{14.1}$$

$$= [w_1, w_2, \dots, w_k] \tag{14.2}$$

This vector of weights is now the entire representation of the face. We've reduced the face from some $n \times n$ image to a k-length vector. To reconstruct the face, then, we say that the reconstructed face is the mean face plus the linear combination of the eigenfaces and the weights:

$$\hat{\mathbf{x}} = \boldsymbol{\mu} + w_1\mathbf{u}_1 + w_2\mathbf{u}_2 + \dots$$

Naturally, the more eigenfaces we keep, the closer the reconstruction $\hat{\mathbf{x}}$ is to the original face $\mathbf{x}$. We can actually leverage this as an error function: if $\mathbf{x} - \hat{\mathbf{x}}$ is low, the thing we are detecting is probably actually a face. Otherwise, we may have treated some random patch as a face. This may come in handy for detection and tracking soon...

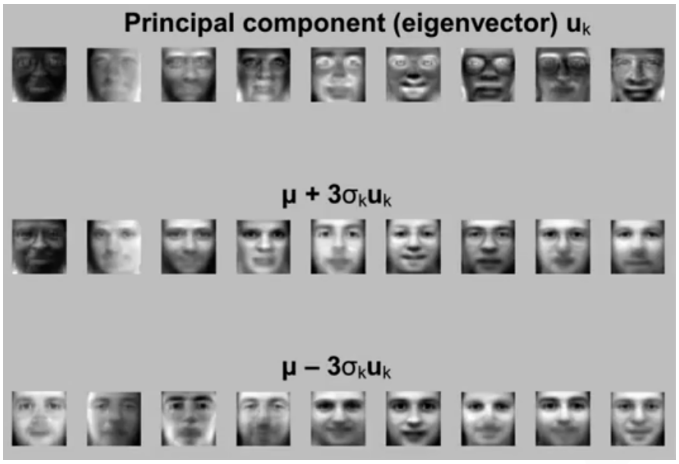


Figure 14.13: Adding or subtracing a principal component from the mean face. σ_k is the standard deviation of the coefficient; we won't worry about it for now.

But we aren't trying to do reconstruction, we're trying to do *recognition*. Given a novel image, $\mathbf{z}$, we project it onto the subspace and determine its weights, as before:

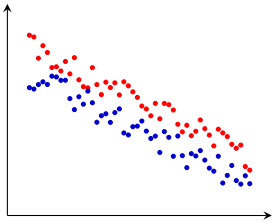
$$[w_1, w_2, \ldots, w_k] = [\mathbf{u}_1 \cdot (\mathbf{z} - \boldsymbol{\mu}), \mathbf{u}_2 \cdot (\mathbf{z} - \boldsymbol{\mu}), \ldots, \mathbf{u}_k \cdot (\mathbf{z} - \boldsymbol{\mu})]$$

Optionally, we can use our above error function to determine whether or not the novel image is a face. Then, we classify the face as whatever our closest training face was in our k-dimensional subspace.

14.2.4 Limitations

Principal component analysis is obviously not a perfect solution; Figure 14.14 visualizes some of its problems.

(a) Traditional PCA has significant limitations if the face image is not aligned with the eigenfaces or has background variation.



(b) PCA falls apart when the principal component itself is the dividing line between two classes, rather than the direction of variation of a single class.

Figure 14.14: Some common limitations of PCA.

With regards to faces, the use of the dot product means our projection requires precise alignment between our input images and our eigenfaces. If the eyes are off-center, for example, the element-wise comparison would result in a poor mapping and an even poorer reconstruction. Furthermore, if the training data or novel image is not tightly cropped, background variation impacts the weight vector.

In general, though, the direction of maximum variance is not always good for classification; in Figure 14.14b, we see that the red points and the blue points lie along the same principal component. If we performed PCA first, we would collapse both sets of points into our single component, effectively treating them as the same class.

Finally, non-linear divisions between classes naturally can't be captured by this basic model.

14.3 Incremental Visual Learning

We want to extend our facial recognition method to a generalized tracking method. Though this method probably belongs in chapter 11, we didn't have the background understanding of principal component analysis there to be able to discuss it. Thus, we introduce it now and will revisit a lot of the principles behind particle filtering (quickly jump to Particle Filters if you need a refresh on those ideas).

What we'll be doing is called **appearance-based tracking**. We'll learn what a target "looks like" through some robust description, and then track the target by finding the image area that results in the most accurate reconstruction. Naturally, though, the appearance of our target (and its surrounding environment) will change from frame to frame, so we need to be robust to a variety of deformations.

Some of our inspiration comes from *eigentracking*, described in this paper, as well as from *incremental learning* techniques described in this paper. The big idea behind eigentracking was to decouple an object's geometry from its appearance. We saw that one of the limitations of eigenfaces was that they weren't robust to imperfect alignments; we can get around this by adding deformations to the subspace model so that eigenvectors also describe rotations, scale changes, and other transformations.

In general, our goal is to have a tracker that...

- ...is not based on a single image. This is the big leagues, not Template Matching.
- ...constantly updates the model, learning the representation while tracking. We touched on this when we introduced α to account for drift in particle filters (see Equation 12.8), but we need something far more robust now.
- ...runs quickly, so we need to be careful of our complexity.
- ...works when the camera itself moves as well.

As before, we'll need a description of our target in advance. Tracking should be robust to pose variations, scale, occlusions, and other deformations such as changes in illumination and viewing angles. We can be robust to these changes by incrementally modifying our description. We don't want to modify it too much, though; that would eventually cause our tracking to drift from the real target.

To achieve this, we'll be using particle filtering. Our particles will represent a distribution of deformations, capturing how the geometry of our target changes from frame to frame. We'll also use the subspace-based reconstruction error to learn how to track our target: our eigenfaces can only accurately reconstruct faces. To adapt to changes, we'll introduce incremental updates to

our subspace; this will dynamically handle deformations by introducing them into the variation described by our subspace automatically.

We say that the location at time t is L_t (though this may include more than just the (x, y) position), and the current observation is F_t (for Faces). Our goal, then, is to predict the target location L_t based on L_{t-1}. This should look very familiar:

$$\Pr[L_t \mid F_t, L_{t-1}] \propto \Pr[F_t \mid L_t] \cdot \Pr[L_t \mid L_{t-1}]$$

Our **dynamics model**, $\Pr[L_t \mid L_{t-1}]$, uses a simple motion model with no velocity.[5] Our **observation model**, $\Pr[F_t \mid L_t]$, is an approximation on an **eigenbasis**.[6] If we can reconstruct a patch *well* from our basis set, that renders a high probability.

14.3.1 Forming Our Model

Let's define the parameters of our particle filtering model a little more thoroughly.

We have a handful of choices when it comes to choosing our state, L_t. We could track a similarity transform: position (x_t, y_t), rotation θ_t, and scaling s_t, or an affine transformation with 6 parameters to additionally allow shearing. For simplicity and brevity, we'll assume the similarity model in this section.

Dynamics Model

The dynamics model is actually quite simple: each parameter's probability is independently distributed along its previous value perturbed by a Gaussian noise function. Mathematically, this means:[7]

$$\Pr[L_1 \mid L_0] = N(x_1; x_0, \sigma_x^2) \cdot N(y_1; y_0, \sigma_y^2) \cdot N(\theta_1; \theta_0, \sigma_\theta^2) \cdot N(s_1; s_0, \sigma_s^2)$$

Given some state L_t, we are saying that the next state could vary in any of these parameters, but the *likelihood* of some variation is inversely proportional to the amount of variation. In other words, it's way more likely to move a little than a lot. This introduces a massive amount of possibilities for our state (remember, particle filters don't scale well with dimension).

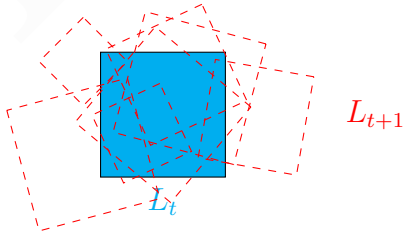


Figure 14.15: A handful of the possible next-states (in red) from the current state L_t (in cyan). The farther L_{t+1} is from L_t, the less likely it is.

[5] The lectures refer to this as a "Brownian motion model," which can be described as each particle simply vibrating around. Check out the Wikipedia page or this more formal introduction for more if you're really that interested ☺ in what that means; it won't come up again.

[6] An eigenbasis is a matrix made up of eigenvectors that form a basis. Our "face space" from before was an eigenbasis; this isn't a new concept, just a new term.

[7] The notation $N(a; b, c)$ states that the distribution for a is a Gaussian around b with the variance c.

Observation Model

We'll be using *probabilistic* principal component analysis to model our image observation process.

Given some location L_t, assume the observed frame was generated from the eigenbasis. How well, then, could we have generated that frame from our eigenbasis? The probability of observing some z given the eigenbasis $\mathbf{B}$ and mean μ is

$$\Pr[z \,|\, \mathbf{B}] = N(z; \mu, \mathbf{B}\mathbf{B}^{\mathrm{T}} + \varepsilon I)$$

This is our "distance from face space." Let's explore the math a little further. We see that our Gaussian is distributed around the mean μ, since that's the most likely face. Its "spread" is determined by the covariance matrix of our eigenbasis, $\mathbf{B}\mathbf{B}^{\mathrm{T}}$, (i.e. variance within face space) *and* a bit of additive Gaussian noise, εI that allows us to cover some "face-like space" as well.

Notice that as $\lim_{\varepsilon \to 0}$, we are left with a likelihood of being purely in face space. Taking this limit and expanding the Gaussian function N gives us:

$$\Pr[z \,|\, \mathbf{B}] \propto \exp\left[-\underbrace{\left\| \underbrace{(z-\mu)}_{\substack{\text{map to} \\ \text{face space}}} - \underbrace{\mathbf{B}\mathbf{B}^{\mathrm{T}}(z-\mu)}_{\text{reconstruction}} \right\|^2}_{\text{reconstruction error}} \right]$$

The likelihood is proportional to the magnitude of error between the mapping of z into face space and its subsequent reconstruction. Why is that the reconstruction, exactly? Well $\mathbf{B}^{\mathrm{T}}(z-\mu)$ results in a vector of the dot product of each eigenvector with our observation z pulled into face space by μ. As we saw in Equation 14.1, this is just the coefficient vector (call it $\boldsymbol{\gamma}$) of z.

Thus, $\mathbf{B}\boldsymbol{\gamma}$ is the subsequent reconstruction, and our measurement model gives the likelihood of an observation z fitting our eigenbasis.

Incremental Learning

This has all been more or less "review" thus far; we've just defined dynamics and observation models with our newfound knowledge of PCA. The cool part comes now, where we allow incremental updates to our object model.

The blending introduced in Equation 12.8 for updating particle filter models had two extremes. If $\alpha = 0$, we eventually can't track the target because our model has deviated too far from reality. Similarly, if $\alpha = 1$, we eventually can't track the target because our model has incorporated too much environmental noise.

Instead, what we'll do is compute a new eigenbasis $\mathbf{B}_{t+1}$ from our previous eigenbasis $\mathbf{B}_t$ and the new observation $\mathbf{w}_t$. We still track the general class, but have allowed some flexibility based on our new observed instance of that class. This is called an **incremental subspace update** and is doable in real time.[8]

[8] *Matrix Computations, 3rd ed.* outlines an algorithm using recursive singular value decomposition that allows us to do these eigenbasis updates quickly. [link]

14.3.2 All Together Now

We can finally incorporate all of these parts into our tracker:

1. Optionally construct an initial eigenbasis for initial detection.
2. Choose an initial location, L_0.
3. Generate all possible locations: $\Pr[L_t \,|\, L_{t-1}]$.
4. Evaluate all possible locations: $\Pr[F_t \,|\, L_{t-1}]$.
5. Select the most likely location: $\Pr[L_t \,|\, F_t, L_{t-1}]$.
6. Update the eigenbasis using the aforementioned R-SVD algorithm.
7. Go to Step 3.

Handling Occlusions

Our observation images have read the Declaration of Independence: all pixels are created equal. Unfortunately, though, this means occlusions may introduce massive (incorrect!) changes to our eigenbasis.

We can ask ourselves a question during our normal, un-occluded tracking: which pixels are we reconstructing *well*? We should probably weigh these accordingly... we have a lot more confidence in their validity. Given an observation I_t, and assuming there is no occlusion with our initial weight mask W_0, then:[9]

$$\mathbf{D}_i = \mathbf{W}_i \otimes (I_t - \mathbf{B}\mathbf{B}^{\mathrm{T}} I_t)$$
$$\mathbf{W}_{i+1} = \exp\left(-\frac{\mathbf{D}_i^2}{\sigma^2}\right)$$

This allows us to weigh each pixel individually based on how accurate its reconstruction has been over time.

14.4 Discriminative Supervised Classification

Generative models were some of the first popular methods for pattern recognition because they could be modeled analytically and could handle low-dimensional space fairly well. As we discussed, though, it came with some liabilities that prevented it from succeeding in the modern world of Big Data™:

- Many signals are extremely high-dimensional—hundreds or tens of thousands of dimensions. Representing the density of these classes is "data-hard:" the amount of data necessary is exponential.
- With a generative model, we were trying to model the *entire* class. Differentiating between skin and "not skin" is too broad of a goal; what we'd rather do is draw a fine, definitive line

[9] We use $\otimes$ to represent element-wise multiplication.

between skin and "things that are very very similar to skin." In other words, we only care about making the right decisions in the hard cases near the "decision boundary", rather than creating general models that describe a class entirely.

- We typically don't know which features in our feature space are useful in discriminating between classes. As a result, we need to have good **feature selection**: which features are informative and useful in differentiation? A generative model has no way of doing that.

All of these principles lead us to **discriminative** methods of supervised classification. As with all of our forays into modeling the world around us coherently, we begin with a series of assumptions.

- There are a fixed number of known classes.

  We can count on the fact the amount of classes we're modeling will not change over time; from the get-go, we will have class A, B, and C, for example. Of course, one of these classes may be a catch-all "none of the above" class.

- We are provided an ample number of training examples of each class.

  We need sufficient training examples to get really fine boundaries. Realistically, this is hard to guarantee, but it's an important assumption and explains the Big Data™ craze of our modern world.

- All mistakes have the same cost. The only thing we are concerned with is getting the label right; getting it wrong has the same cost, regardless of "how wrong" it is.

- We need to construct a representation or description of our class instance, but we don't know *a priori* (in advance) which of our features are representative of the class label. Our model will need to glean the diagnostic features.

14.4.1 Discriminative Classifier Architecture

A discriminative system has two main parts: a **representation** of the object model that describes the training instances and a **classifier**. Since these are notes on computer vision, we will be working with images. Given those, we then move on to testing our framework: we generate candidate images and score our classifier based on how accurately it classified them.

Building a Representation

Let's tackle the first part: representation. Suppose we have two classes: koalas and pandas. One of the simplest ways to describe their classes may be to generate a color (or grayscale) histogram of their image content. Naturally, our biggest assumption is that our images are made up of *mostly* the class we're describing.

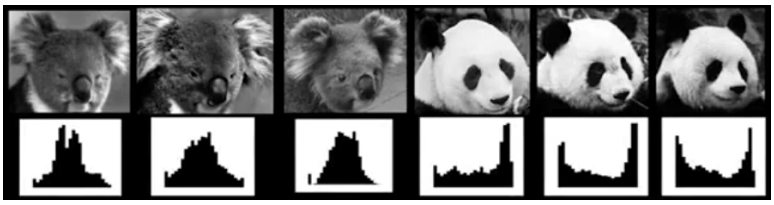


Figure 14.16: Histograms for some training images for two classes: koalas and pandas.

There is clear overlap across the histograms for koalas, and likewise for pandas. Is that sufficient in discriminating between the two? Unfortunately, no! Color-based descriptions are extremely sensitive to both illumination and intra-class appearance variations.

How about feature points? Recall our discussion of Harris Corners and how we found that regions with high gradients and high gradient direction variance (like corners) were robust to changes in illumination. What if we considered these edge, contour, and intensity gradients?

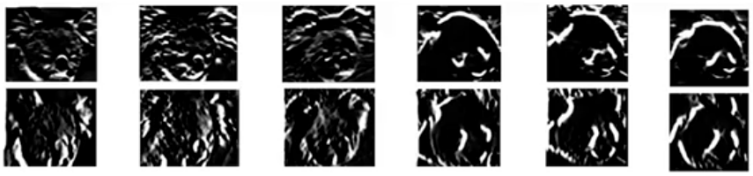


Figure 14.17: Edge images of koala and pandas.

Even still, small shifts and rotations make our edge images look completely different (from an overlay perspective). We can divide the image into pieces and describe the local distributions of gradients with a histogram.

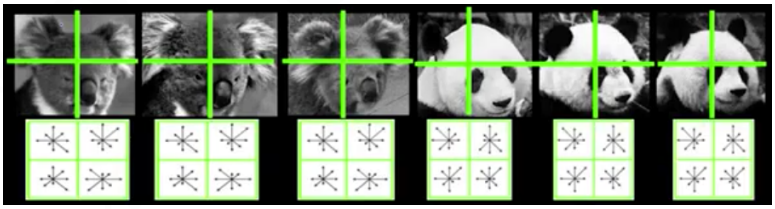


Figure 14.18: Subdivision of each picture into a local distribution of gradients.

This has the benefit of being locally order-less, offering us invariance in our aforementioned small shifts and rotations. If, for example, the top-left ear moves in one of the pandas, the *local* gradient distribution will still stay the same. We can also perform constrast normalization to try to correct for illumination differences.

This is just one form of determining some sort of feature representation of the training data. In fact, this is probably the most important form; analytically determining an approximate way to differentiate between classes is the crux of a discriminative classifier. Without a good model, our classifier will have a hard time finding good decision boundaries.

Train a Classifier

We have an idea of how to build our representation, now how can we use our feature vectors (which are just flattened out $1 \times n$ vectors of our descriptions) to do classification? Given our feature vectors describing pandas and describing koalas, we need to learn to differentiate between them.

To keep things simple, we'll stick to a **binary classifier**: we'll have koalas and non-koalas, cars and non-cars, etc. There are a *massive* number of discriminative classification techniques recently developed in machine learning and applied to computer vision: nearest neighbor, neural networks, support vector machines (SVMs), boosting, and more. We'll discuss each of these in turn shortly.

Generating and Scoring Candidates

Window-based models using a "sliding window" across the image and apply the classifier to *every patch*. There's nothing clever about it, but it works well.

14.4.2 Nearest Neighbor

The **c**lassification strategy is very simple: just choose the label of the nearest training data point. When we are given a novel test example, we finding the closest training example and label it accordingly. Each point corresponds to a Voronoi partition which discretizes the space based on the distance from that point.

Nearest-neighbor is incredibly easy to write, but is not ideal. It's very data intensive: we need to remember all of our training examples. It's computationally expensive: even with a *kd*-tree (we've touched on these before; see 1 for more), searching for the nearest neighbor takes a bit of time. Most importantly, though, it simply does not work that well.

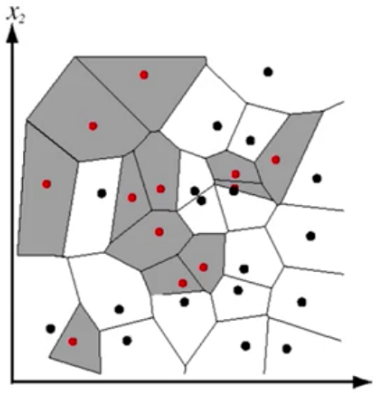


Figure 14.19: Classification of the negative (in black) and positive (in red) classes. The partitioning is a Voronoi diagram.

We can use the k-nearest neighbor variant to make it so that a single training example doesn't dominate its region too much. The process is just as simple: the k nearest neighbors "vote" to classify the new point, and majority rules. Surprisingly, this small modification works *really* well. We still have the problem of this process being data-intensive, but this notion of a *loose consensus* is very powerful and effective.

Let's look as some more sophisticated discriminative methods.

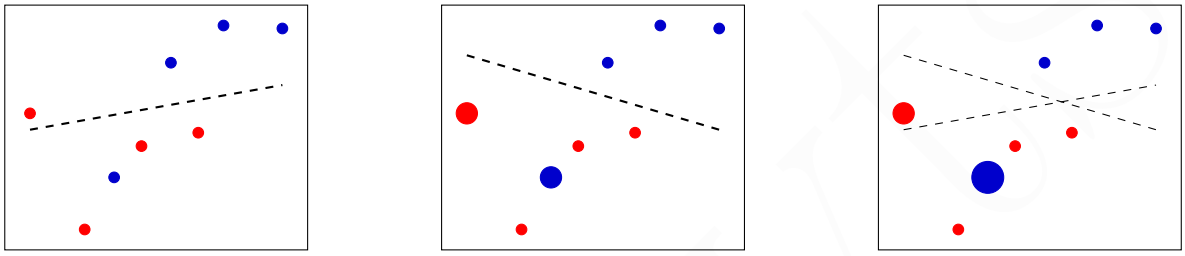
14.4.3 Boosting

This section introduces an iterative learning method: **boosting**. The basic idea is to look at a weighted training error on each iteration.

Initially, we weigh all of our training examples equally. Then, in each "boosting round," we find the weak learner (more in a moment) that achieves the lowest weighted training error. Following that,

we raise the weights of the training examples that were *misclassified* by the current weak learner. In essence, we say, "learn these better next time." Finally, we combine the weak learners from each step in a simple linear fashion to end up with our final classifier.

A **weak learner**, simply put, is a function that partitions our space. It doesn't necessarily have to give us the "right answer," but it does give us *some* information. Figure 14.20 tries to visually develop an intuition for weak learners.



(a) Our set of training examples and the first boundary guess.

(b) Reweighing the error examples and trying another boundary.

(c) Final classifier is a combination of the weak learners.

Figure 14.20: Iteratively applying weak learners to differentiate between the red and blue classes.

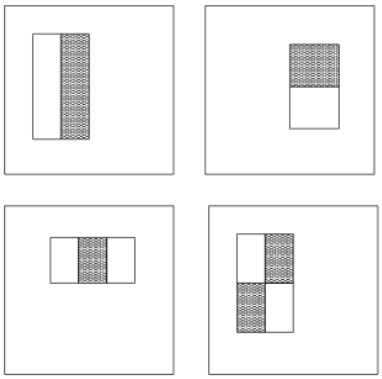
The final classifier is a linear combination of the weak learners, and their weight directly proportional to their accuracy. The exact formulas depend on the "boosting scheme;" one of them is called AdaBoost; we won't dive into it in detail, but a simplified version of the algorithm is described in algorithm 14.1.

Viola-Jones Face Detector

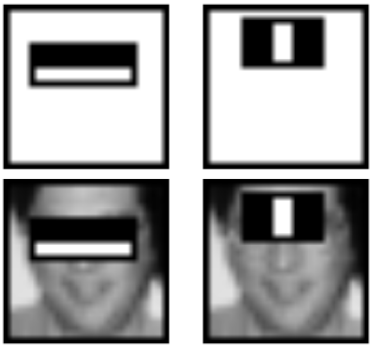
There was an application of boosting to object detection that took the computer vision field by storm: a real-time face detector called the **Viola-Jones detector** (from this 2001 paper).

There were a few big ideas that made this detector so accurate and efficient:

- Brightness patterns were represented with efficiently-computable "rectangular" features within a window of interest. These features were essentially large-scale gradient filters or Haar wavelets.



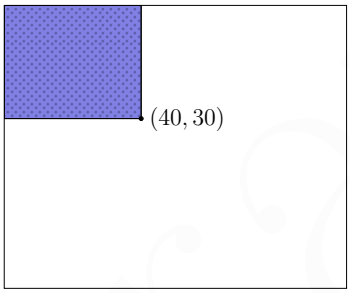
(a) These are some "rectangular filters;" the filter takes the sum of the pixels in the light area and subtracts the sum of the pixels in the dark area.

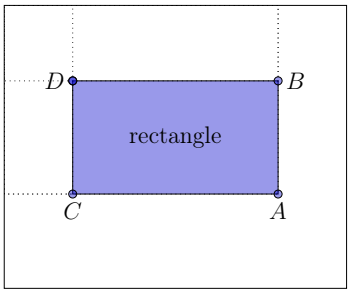


(b) These filters were applied to different regions of the image; "features" were the resulting differences.

Figure 14.21: Feature detection in the Viola-Jones detector.

The reason why this method was so effective was because it was incredibly efficient to compute if performed many times to the same image. It leveraged the **integral image**, which, at some (x, y) pixel location, stores the sum of all of the pixels spatially before it. For example, the pixel at $(40, 30)$ would contain the sum of the pixels in the quadrant from $(0, 0)$ to $(40, 30)$:





Why is this useful? Well with the rectangular filters, we wanted to find the sum of the pixels within an arbitrary rectangular region: (A, B, C, D). What is its sum with respect to the integral image? It's simply $A - B - C + D$, as above.

Once we have the integral image, this is only 3 additions to compute the sum of **any size rectangle**. This gives us really efficient handling of scaling as a bonus, as well: instead of scaling the images to find faces, we can just scale the features, recomputing them efficiently.

- We use a boosted combination of these filter results as features. With such efficiency, we can look at an absurd amount of features quickly. The paper used **180,000 features** associated with a 24×24 window. These features were then run through the boosting process in order to find the best linear combination of features for discriminating faces. The top two weak learners were actually the filters shown in Figure 14.21b. If you look carefully, the first appears useful in differentiating eyes (a darker patch above/below a brighter one) and dividing the face in half.
- We'll formulate a *cascade* of these classifiers to reject clear negatives rapidly. Even if our filters are blazing fast to compute, there are still a *lot* of possible windows to search if we have just a 24×24 sliding window. How can we make detection more efficient?

Well... almost *everywhere* in an image is **not** a face. Ideally, then, we could reduce detection time significantly if we found all of the areas in the image that were definitely not a face.

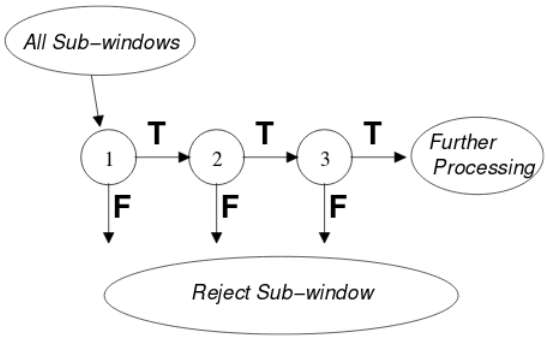


Figure 14.23: The cascading classifiers in the Viola-Jones face detector.

Each stage has no false negatives: you can be completely sure that any rejected sub-window was not a face. All of the positives (even the false ones) go into the training process for the next classifier. We only apply the sliding window detection if a sub-window made it through the entire cascade.

To summarize this incredibly impactful technique, the Viola-Jones detector used rectangular features optimized by the integral image, AdaBoost for feature selection, and a cascading set of classifiers to discard true negatives quickly. Due to the massive feature space (180,000+ filters), training is very slow, but detection can be done in real-time. Its results are phenomenal and variants of it are in use today commercially.

Advantages and Disadvantages

Boosting integrates classification with feature selection in a flexible, easy-to-implement fashion. Its flexibility enables a "plug-and-play" nature of weak learners; as long as they support some form of iterative improvement, boosting can select the best ones and optimize a linear combination. We mentioned detection efficiency for the Viola-Jones detector specifically (testing is *fast*) but in general, boosting has a training complexity linear in the number of training examples; that's a nice property.

Some of the downsides of boosting include needing a large amount of training examples. This was one of our earlier assumptions, though. More significantly, boosting has been found to not work as well as Support Vector Machines and random forests which are newer approaches to discriminative

learning, especially on many-class problems.

ALGORITHM 14.1: The simplified AdaBoost algorithm.

Input: X, Y: M positive and negative training samples with their corresponding labels.
Input: H: a weak classifier type.
Result: A boosted classifier, H^*.

```
/* Initialize a uniform weight distribution.  */
```
$\hat{\mathbf{w}} = \begin{bmatrix} 1/M & 1/M & \dots \end{bmatrix}$
$t =$ some small threshold value close to 0

foreach *training stage* $j \in [1..n]$ **do**
  $\hat{\mathbf{w}} = \mathbf{w}/\|\mathbf{w}\|$
  `/* Instantiate and train a weak learner for the current weights.  */`
  $h_j = H(X, Y, \hat{\mathbf{w}})$
  `/* The error is the sum of the incorrect training predictions.  */`
  $\varepsilon_j = \sum_{i=0}^{M} w_i \qquad \forall w_i \in \hat{\mathbf{w}}$ where $h_j(X_i) \neq Y_i$
  $\alpha_j = \frac{1}{2} \ln\left(\frac{1-\varepsilon_j}{\varepsilon_j}\right)$
  `/* Update the weights only if the error is large enough.  */`
  if $\varepsilon > t$ **then**
    $w_i = w_i \cdot \exp\left(-Y_i \alpha_j h_j(X_i)\right) \qquad \forall w_i \in \mathbf{w}$
  else
    break
end

```
/* The final boosted classifier is the sum of each h_j weighed by its
   corresponding α_j.  Prediction on an observation x is then simply:  */
```
$H^*(x) = \text{sign}\left[\sum_{j=0}^{M} \alpha_j h_j(x)\right]$
return H^*

14.5 Support Vector Machines

In contrast with boosting, **support vector machine**s (SVMs for short) are tougher to implement. Despite that, SVMs are a recent development in machine learing that make for incredibly reliable discriminative classifiers. Before we begin discussing them in further detail, we need to talk about linear classifiers.

14.5.1 Linear Classifiers

Given a set of points, we want to automatically find the line that divides them the best. For example, given the blue and pink points in Figure 14.24, we can easily (as humans) see that the line defines their boundary. We want to adapt this idea to higher-dimensional space.

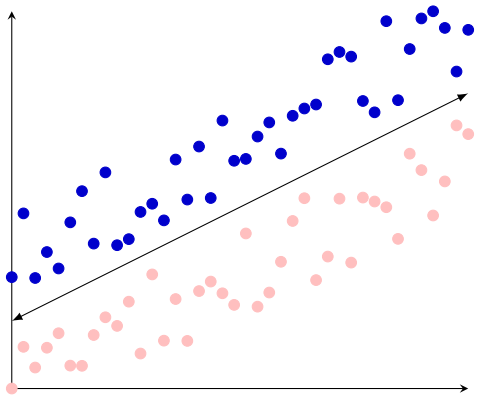


Figure 14.24: Finding the boundary line between two classes of points

We've talked a lot about lines, but let's revisit them in $\mathbb{R}^2$ once more. Lines can be expressed as a normal vector and a distance from the origin. Specifically, given[10] $px + qy + b = 0$, if we let $\mathbf{w} = \begin{bmatrix} p \\ q \end{bmatrix}$ and $\mathbf{x}$ be some arbitrary point $\begin{bmatrix} x \\ y \end{bmatrix}$, then this is equivalent to $\mathbf{w} \cdot \mathbf{x} + b = 0$, and $\mathbf{w}$ is normal to the line. We can scale both $\mathbf{w}$ and b arbitrarily and still have the same result (this will be important shortly). Then, for an arbitrary point (x_0, y_0) *not* on the line, we can find its (perpendicular) distance to the line relatively easily:

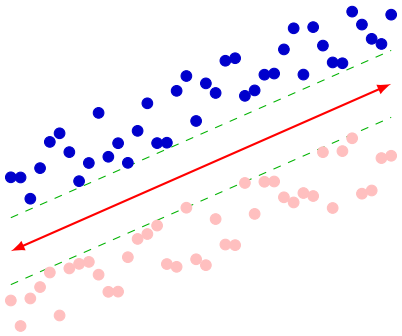
$$D = \frac{px_0 + qy_0 + b}{\sqrt{p^2 + q^2}} = \frac{\mathbf{x}_0 \cdot \mathbf{w} + b}{\|\mathbf{w}\|}$$

None of this should be new information.

If we want to find a linear classifier, we want the line that separates the positive and negative examples. For the points in Figure 14.24, if $\mathbf{x}_i$ is a positive example, we want $\mathbf{x}_i \cdot \mathbf{w} + b \geq 0$, and likewise if $\mathbf{x}_i$ is negative, we want $\mathbf{x}_i \cdot \mathbf{w} + b < 0$.

Naturally, though, there are quite a few lines that separate the blue and pink dots in Figure 14.24. Which one is "best"? Support vector machines are a discriminative classifier based on this "optimal separating line."

In the 2D case, an SVM wants to maximize the **margin** between the positive and negative training examples (the distance between the dotted lines):

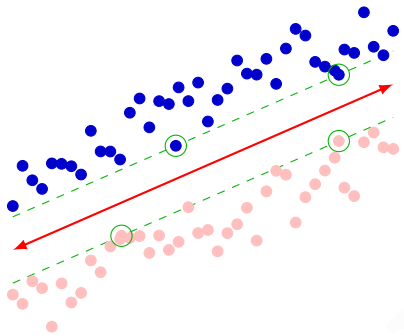


[10] We switch notation to p, q, b instead of the traditional a, b, c to stay in line with the SVM literature.

This extends easily to higher dimensions, but that makes for tougher visualizations.

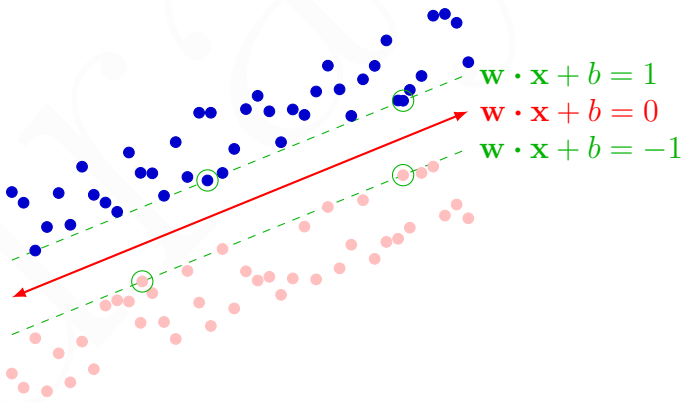
14.5.2 Support Vectors

There are some special points in the above visualization, points where the dashed lines intersect a



The margin lines between two classes will always have such points; they are called the **support vectors**. This is a huge reduction in computation relative to the generative methods we covered earlier, which analyzed the entire data set; here, we only care about the examples near the boundaries. So. How can we use these vectors to maximize the margin?

Remember that we can arbitrarily scale $\mathbf{w}$ and b and still represent the same line, so let's say that the separating line $= 0$, whereas the lines touching the positive and negative examples are $= 1$ and $= -1$, respectively.



The aforementioned support vectors lie on these lines, so $\mathbf{x}_i \cdot \mathbf{w} + b = \pm 1$ for them, meaning we can compute the margin as a function of these values. The distance from a support vector $\mathbf{x}_i$ to the separating line is then:

$$\frac{|\mathbf{x}_i \cdot \mathbf{w} + b|}{\|\mathbf{w}\|} = \frac{\pm 1}{\|\mathbf{w}\|}$$

That means that M, the distance between the dashed green lines above, is defined by:

$$M = \left| \frac{1}{\|\mathbf{w}\|} - \frac{-1}{\|\mathbf{w}\|} \right| = \frac{2}{\|\mathbf{w}\|}$$

Thus, we want to find the $\mathbf{w}$ that maximizes the margin, M. We can't just pick any $\mathbf{w}$, though: it has to correctly classify all of our training data points. Thus, we have an additional set of constraints that:

$$\forall \mathbf{x}_i \in \text{positive examples: } \mathbf{x}_i \cdot \mathbf{w} + b \geq 1$$
$$\forall \mathbf{x}_i \in \text{negative examples: } \mathbf{x}_i \cdot \mathbf{w} + b \leq 1$$

Let's define an auxiliary variable y_i to represent the label on each example. When $\mathbf{x}_i$ is a negative example, $y_i = -1$; similarly, when $\mathbf{x}_i$ is a positive example, $y_i = 1$. This gives us a standard **quadratic optimization problem**:

$$\begin{aligned} \text{Minimize:} \quad & \frac{1}{2}\mathbf{w}^{\mathrm{T}}\mathbf{w} \\ \text{Subject to:} \quad & y_i(\mathbf{x}_i \cdot \mathbf{w} + b) \geq 1 \end{aligned}$$

The solution to this optimization problem (whose derivation we won't get into) is just a linear combination of the support vectors:

$$\mathbf{w} = \sum_i \alpha_i y_i \mathbf{x}_i \tag{14.3}$$

The α_is are "learned weights" that are non-zero at the support vectors. For any support vector, we can substitute in $y_i = \mathbf{w} \cdot \mathbf{x}_i + b$, so:

$$y_i = \sum_i \alpha_i y_i \mathbf{x}_i \cdot \mathbf{x} + b$$

Since it $= y_i$, it's always ± 1. We can use this to build our classification function:

$$f(x) = \text{sign}\left(\mathbf{w} \cdot \mathbf{x} + b\right) \tag{14.4}$$
$$= \text{sign}\left(\sum_i \alpha_i y_i \boxed{\mathbf{x}_i \cdot \mathbf{x}} + b\right) \tag{14.5}$$

The highlighted box is a **crucial** component: the entirety of the classification depends *only* on this dot product between some "new point" $\mathbf{x}$ and our support vectors $\mathbf{x}_i$s.

14.5.3 Extending SVMs

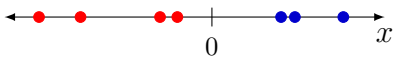
Some questions arise with this derivation, and we'll knock each of them down in turn:

1. We've been working with nice, neat 2D plots and drawing a line to separate the data. This begs the question: **what if the features are not in 2 dimensions?**
2. Similarly, **what if the data isn't linearly separable?**
3. Classifying everything into two binary classes seems like a pipe dream: **what if we have more than just two categories?**

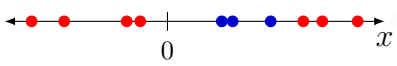
The first one is easy to knock down: nothing in the math of Equation 14.4 requires 2 dimensions: $\mathbf{x}$ could easily be an arbitrary n-dimensional vector. As long as $\mathbf{w}$ is a normal, it defines a structure in that dimensional space: instead of a line, in 3D we'd create a plane; this generalizes to creating an $(n-1)$-dimensional **hyperplane**.

Mapping to Higher-Dimensional Space

The second question is a bit harder to tackle. Obviously, we can find the optimal separator between the following group of points:



But what about these?



No such luck this time. But what if we mapped them to a higher-dimensional space? For example, if we map these to $y = x^2$, a wild linear separator appears!

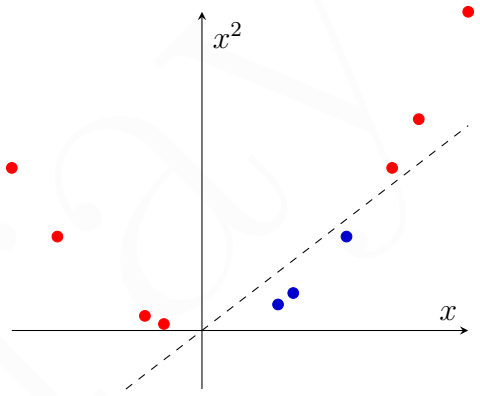


Figure 14.25: Finding a linear separator by mapping to a higher-dimensional space.

This seems promising... how can we find such a mapping (like the arbitrary $x \mapsto x^2$ above) for other feature spaces? Let's generalize this idea. We can call our mapping function Φ; it maps $\mathbf{x}$s in our feature space to another higher-dimensional space $\varphi(\mathbf{x})$, so $\Phi : \mathbf{x} \mapsto \varphi(\mathbf{x})$. We can use this to find the "**kernel trick**."

Kernel Trick Just a moment ago in (14.4), we showed that the linear classifier relies on the dot product between vectors: $\mathbf{x}_i \cdot \mathbf{x}$. Let's define a **kernel function** K:

$$K(\mathbf{x}_i, \mathbf{x}_j) = \mathbf{x}_i \cdot \mathbf{x}_j = \mathbf{x}_i^{\mathrm{T}} \mathbf{x}_j$$

If we apply Φ to our points, we get a new kernel function:

$$K(\mathbf{x}_i, \mathbf{x}_j) \Rightarrow \varphi(\mathbf{x}_i)^{\mathrm{T}} \varphi(\mathbf{x}_j)$$

This kernel function is a "similarity" function that corresponds to an inner (dot) product in some expanded feature space. A similarity function grows if its parameters are similar. This is significant: as long as there is *some* higher dimensional space in which K is a dot product, we can use K in a linear classifier.[11]

The kernel trick is that instead of explicitly computing the lifting transformation (since it lifts us into a higher dimension ☕) $\varphi(\mathbf{x})$, we instead define the kernel function in a way that we *know* maps to a dot product in a higher dimensional space, like the polynomial kernel in the example below. Then, we get the *non-linear* decision boundary in the original feature space:

$$\sum_i \alpha_i y_i \left(\mathbf{x}_i^{\mathrm{T}}\mathbf{x}\right) + b \quad \longrightarrow \quad \sum_i \alpha_i y_i K(\mathbf{x}_i^{\mathrm{T}}\mathbf{x}) + b$$

EXAMPLE 14.2: **A Simple Polynomial Kernel Function**

Let's work through a proof that a particular kernel function is a dot product in some higher-dimensional space. Remember, we don't actually care about what that space *is* when it comes to applying the kernel; that's the beauty of the kernel trick. We're working through this to demonstrate how you would show that some kernel function does have a higher-dimensional mapping.

We have 2D vectors, so $\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$.

Let define the following kernel function: $K(\mathbf{x}_i, \mathbf{x}_j) = (1 + \mathbf{x}_i^{\mathrm{T}}\mathbf{x}_j)^2$

This is a simple **polynomial kernel**; it's called that because we are creating a polynomial from the dot product. To prove that this is a valid kernel function, we need to show that $K(\mathbf{x}_i, \mathbf{x}_j) = \varphi(\mathbf{x}_j)^{\mathrm{T}}\varphi(\mathbf{x}_j)$ for *some* φ.

$$\begin{aligned}
K(\mathbf{x}_i, \mathbf{x}_j) &= (1 + \mathbf{x}_i^{\mathrm{T}}\mathbf{x}_j)^2 \\
&= \left(1 + \begin{bmatrix} x_{i1} & x_{i2} \end{bmatrix} \begin{bmatrix} x_{j1} \\ x_{j2} \end{bmatrix}\right)\left(1 + \begin{bmatrix} x_{i1} & x_{i2} \end{bmatrix} \begin{bmatrix} x_{j1} \\ x_{j2} \end{bmatrix}\right) && \text{expand} \\
&= 1 + x_{i1}^2 x_{ij}^2 + 2x_{i1}x_{j1}x_{i2}x_{j2} + x_{i2}^2 x_{j2}^2 + 2x_{i1}x_{j1} + 2x_{i2}x_{j2} && \text{multiply it all out} \\
&= \begin{bmatrix} 1 & x_{i1}^2 & \sqrt{2}x_{i1}x_{i2} & x_{i2}^2 & \sqrt{2}x_{i1} & \sqrt{2}x_{i2} \end{bmatrix} \begin{bmatrix} 1 \\ x_{j1}^2 \\ \sqrt{2}x_{j1}x_{j2} \\ x_{j2}^2 \\ \sqrt{2}x_{j1} \\ \sqrt{2}x_{j2} \end{bmatrix} && \text{rewrite it as a vector product}
\end{aligned}$$

[11] There are mathematical restrictions on kernel function that we won't get into but I'll mention for those interested in reading further. Mercer's Theorem guides us towards the conclusion that a kernel matrix $\boldsymbol{\kappa}$ (i.e. the matrix resulting from applying the kernel function to its inputs, $\kappa_{i,j} = K(x_i, x_j)$) must be positive semi-definite. For more reading, try this article.

At this point, we can see something magical and crucially important: each of the vectors only relies on terms from *either* $\mathbf{x}_i$ or $\mathbf{x}_j$! That means it's a... wait for it... dot product! We can define φ as a mapping into this new 6-dimensional space:

$$\varphi(\mathbf{x}) = \begin{bmatrix} 1 & x_1^2 & \sqrt{2}x_1x_{n2} & x_2^2 & \sqrt{2}x_1 & \sqrt{2}x_2 \end{bmatrix}^{\mathrm{T}}$$

Which means now we can express K in terms of dot products in φ, exactly as we wanted:

$$K(\mathbf{x}_i, \mathbf{x}_j) = \varphi(\mathbf{x_i})^{\mathrm{T}}\varphi(\mathbf{x_j}) \quad \blacksquare$$

What if the data isn't separable even in a higher dimension, or there is some decision boundary that is actually "better"[12] that a perfect separation that perhaps ignores a few whacky edge cases? This is an advanced topic in SVMs that introduces the concept of **slack variable**s which allow for this error, but that's more of a machine learning topic. For a not-so-gentle introduction, you can try this page.

Example Kernel Functions This entire discussion begs the question: what are some good kernel functions?

Well, the simplest example is just our linear function: $K(\mathbf{x}_i, \mathbf{x}_j) = \mathbf{x}_i^{\mathrm{T}}\mathbf{x}_j$. It's useful when $\mathbf{x}$ is already just a massive vector in a high-dimensional space.

Another common kernel is a Gaussian noise function, which is a specific case of the **radial basis function**:

$$K(\mathbf{x}_i, \mathbf{x}_j) = \exp\left(-\frac{\|\mathbf{x}_i - \mathbf{x}_j\|^2}{2\sigma^2}\right)$$

As $\mathbf{x}_j$ moves away from the support vector $\mathbf{x}_i$, it just falls off like a Gaussian. But how is this a valid kernel function, you might ask? Turns out, it maps us to an infinite dimensional space ☺. You may (or may not) recall from calculus that the exponential function e^x, can be expressed as an infinite series:

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \ldots = \sum_{k=0}^{\infty}\frac{x^k}{k!}$$

Applying this to our kernel renders this whackyboi (with $\sigma = 1$ for simplification):[13]

$$\exp\left(-\frac{1}{2}\left\|\mathbf{x} - \mathbf{x}'\right\|_2^2\right) = \sum_{j=0}^{\infty}\frac{(\mathbf{x}^{\mathrm{T}}\mathbf{x}')^j}{j!} \cdot \exp\left(-\frac{1}{2}\left\|\mathbf{x}\right\|_2^2\right) \cdot \exp\left(-\frac{1}{2}\left\|\mathbf{x}'\right\|_2^2\right)$$

As you can see, it can be expressed as an infinite sum of dot product of $\mathbf{x} \cdot \mathbf{x}'$. Of course, thanks to the kernel trick, we don't actually have to care *how* it works; we can just use it. A Gaussian kernel function is really useful in computer vision because we often use histograms to describe classes and histograms can be represented as mixtures of Gaussians; we saw this before when describing Continuous Generative Models.

[12] For some definition of *better*.

[13] The subscript notation on the norm: $\|\mathbf{x}\|_2$ expresses the p-norm. For $p = 2$, this is our familiar Euclidean distance: $\sqrt{\sum_{x_i \in \mathbf{x}} x_i}$. [from StackExchange]

Another useful kernel function (from this paper) is one that describes histogram intersection, where the histogram is normalized to represent a probability distribution:

$$K(\mathbf{x}_i, \mathbf{x}_j) = \sum_k \min\left(\mathbf{x}_i(k), \mathbf{x}_j(k)\right)$$

We'll leverage this kernel function briefly when discussing Visual Bags of Words and activity classification in video.

Multi-category Classification

Unfortunately, neither of the solutions to this question are particularly "satisfying," but fortunately both of them work. There are two main approaches:

One vs. All In this approach, we adopt a "thing or ¬thing" mindset. More specifically, we learn an SVM for each class versus all of the other classes. In the testing phase, then, we apply each SVM to the test example and assign to it the class that resulted in the highest decision value (that is, the distance that occurs within (14.4)).

One vs. One In this approach, we actually learn an SVM for each possible pair of classes. As you might expect, this blows up relatively fast. Even with just 4 classes, $\{A, B, C, D\}$, we need $\binom{4}{2} = 6$ pairs of SVMs: $\{(A, B), (A, C), (A, D), (B, C), (B, D), (C, D)\}$. During the testing phase, each learned SVM "votes" for a class to be assigned to the test example. Like when we discussed voting in Hough Space, we trust that the most sensible option results in the most votes, since the votes for the "invalid" pairs (i.e. the (A, B) SVM for an example in class D) would result in a random value.

14.5.4 SVMs for Recognition

Recognition and classification are largely interchangeable ideas, so this serves more as a summary of SVMs. To use them, we need to:

1. **Define a representation**. We process our data set, decide what constitutes a feature, etc.
2. **Select a kernel** function. This part involves a bit of magic and guesswork (as is tradition in machine learning).
3. **Compute pairwise kernel values** between labeled examples. This is naturally the slowest part of the training process; it has $O(n^2)$ complexity in the number of training samples.
4. **Use the "kernel matrix"** that results from training to solve for the support vectors and their weights.

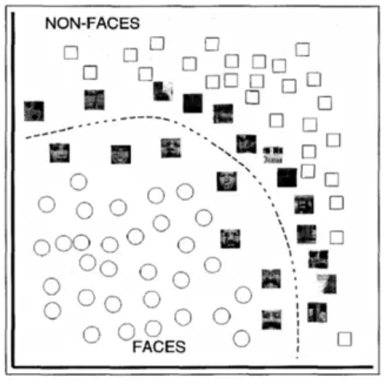


Figure 14.26: Training an SVM to do facial recognition.

The final linear combination of support vectors and weights renders the classification label for any new input vector $\mathbf{x}$ via Equation 14.4.

Using SVMs for Gender Classification

An interesting application of SVMs was gender classification of faces, done in this paper in 2002. They used a Gaussian RBF as their kernel function to find the "support faces" that defined the boundary between the genders, a few of which are shown in Figure 14.27.

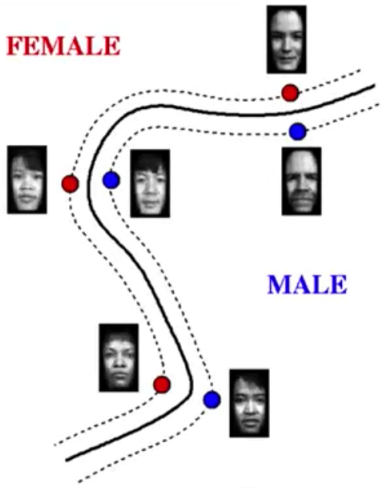


Figure 14.27: The "support faces"—support vectors of faces—determined by the SVM classifier.

The lowest error rate found by the SVM with an RBF kernel was around 3.4% on *tiny* 21×12 images; this performed better than humans *even* when high resolution images were used. Figure 14.28 shows the top five gender misclassifications by humans.

Figure 14.28: The top five gender misclassifications by humans. How would you label them?

Answer: FMMFM

14.6 Visual Bags of Words

Briefly recall what we did when creating panoramas: we found putative matches among feature points using a descriptor (like the SIFT Descriptor), then used something like RANSAC to find the resulting alignment by finding consensus for a particular transformation among our points.

We can do something that is somewhat related for recognition, as well. We'll define and find interesting points, but we'll then talk about describing the patches *around* those points. We can then use the collection of described patches to create general descriptions or categorizations of images as a whole by searching for those patches.

We make an important assumption for these features: if we see points close in feature space (e.g. the patches have similar descriptors), those indicate similar local content. Given a new "query image," then, we can describe its features by finding similar patches from a pre-defined database of images.

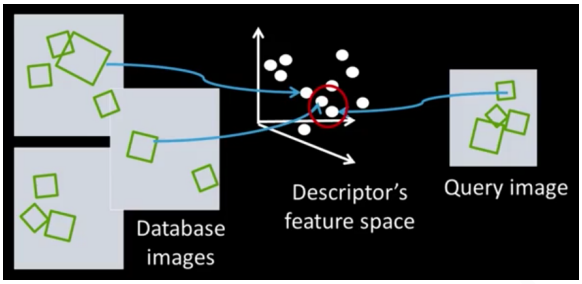


Figure 14.29: Each region in an image has a descriptor, which is a point in some high-dimensional feature space.

The obvious problem with this idea is sheer scalability. We may need to search for *millions* of possible features for a single image. Thankfully, this isn't a unique problem, and we can draw parallels to other disciplines in computer science to find a better search method. To summarize our problem, let's restate it as such:

> *With potentially thousands of features per image, and hundreds of millions of images to search, how can we efficiently find those that are relevant to a new image?*

If we draw a parallel to the real world of text documents, we can see a similar problem. How, from a book with thousands of words on each page, can we find all of the pages that contain a particular "interesting" word? Reference the Index, of course! Suppose we're given a new page, then, which some selection of interesting words in it. How can we find the "most similar" page in our book? The likeliest page is the page with the most references to those words in the index!

Example 14.3: **Find Likely Page**

Suppose we have four pages of text with an index as follows:

Index

ant	1,4	*gong*	2,4
bread	2	*hammer*	2,3
bus	4	*leaf*	1,2,3
chair	2,4	*map*	2,3
chisel	3,4	*net*	2
desk	1	*pepper*	1,2
fly	1,3	*shoe*	1,3

Now we're given a page with a particular set of words from the index: *gong*, *fly*, *pepper*, *ant*, and *shoe*. Which page is this "most like"?

Finding the solution is fairly straightforward: choose the page with the most refer-

ences to the new words:

- **Page 1** contains {*ant, pepper, fly, shoe*} (4).
- **Page 2** contains {*gong, pepper*} (2).
- **Page 3** contains {*fly, shoe*} (2).
- **Page 4** contains {*ant, gong*} (2).

Clearly, then, the new page is most like the 1st page. Naturally, we'd need some tricks for larger, more realistic examples, but the concept remains the same: iterate through the index, tracking occurrences for each page.

Similarly, we want to find all of the images that contain a particular interesting *feature*. This leads us to the concept of mapping our features to "visual words."

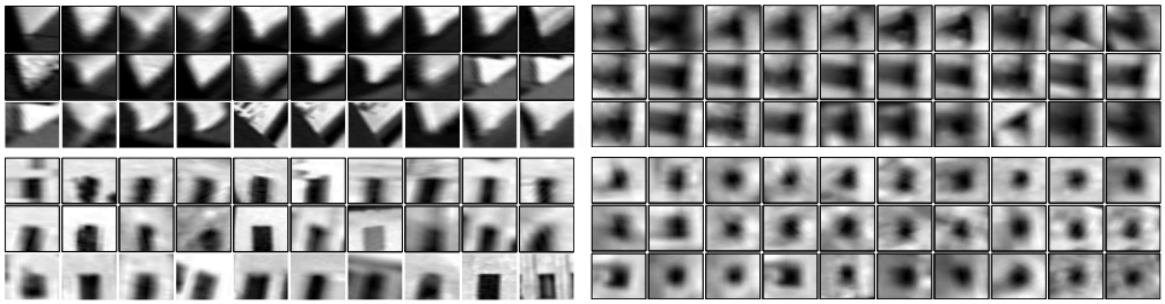


Figure 14.30: An excerpt from the "visual vocabulary" formulated in this paper. Each quadrant represents a selection describing one of the "visual words" in the feature space.

We'll disregard some implementation details regarding this method, such as which features to choose or what size they should be. These are obviously crucial to the method, but we're more concerned with using it rather than implementing it. The academic literature will give more insights (such as the paper linked in Figure 14.30) if you're interested. Instead, we'll be discussing using these "bags of visual words" to do recognition.

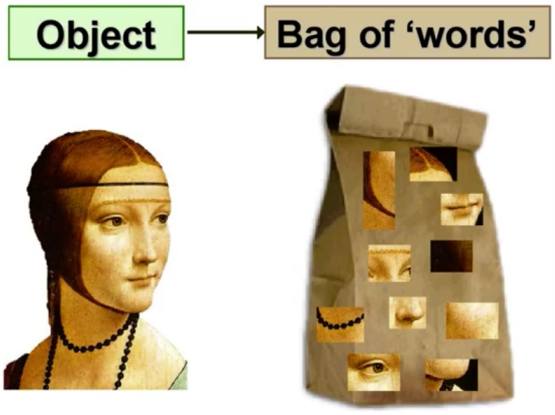


Figure 14.31: We dissect an image into its composite features, which we then use to create our "vocabulary" to describe novel images.

To return to the document analogy, if you had a document and wanted to find similar documents from your library, you likely would pull documents with a similar histogram distribution of words, much like we did in the previous example.

Figure 14.32: Creating a visual vocabulary from a set of objects.

We can use our visual vocabulary to compute a histogram of occurrences of each "word" in each object. Notice in Figure 14.33 that elements that don't belong to the object do sometimes "occur": the bottom of the violin might look something like a bicycle tire, for example. By-and-large, though, the peaks are related to the relevant object from Figure 14.32.

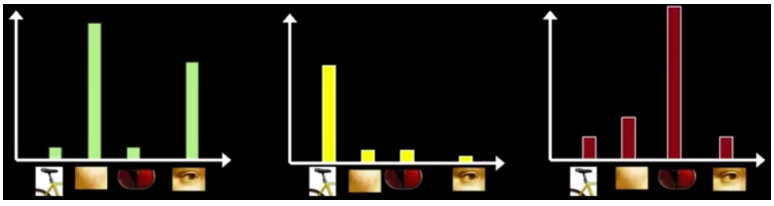


Figure 14.33: Capturing the distribution of each object in the context of our overall vocabulary.

Comparing images to our database is easy. By normalizing the histograms, we can treat them a unit vector. For some database image $\mathbf{d}_j$ and query image $\mathbf{q}$, similarity can then be easily represented by the dot product:

$$\text{sim}(\mathbf{d}_j, \mathbf{q}) = \frac{\mathbf{d}_j \cdot \mathbf{q}}{\|\mathbf{d}_j\| \, \|\mathbf{q}\|}$$

This essentially amounts to a nearest-neighbor lookup. In our initial discussion of nearest-neighbor, we saw that k-nearest-neighbor was fairly effective, but it was slow, inefficient, and data-hungry. Working around this problem was our motivation behind boosting and ultimately support vector machines.

We can apply the same principles here: given our "visual vocabulary" for a set of object classes,[14] we can train an SVM to differentiate between them for novel inputs. This concept was pioneered in this paper.

[14] Such visual bags of words already exist, see the Caltech 101 dataset.

Video Analysis

> *McFarland suggests that we are in the midst of a paradigm shift, and if "a picture is worth a thousand words then maybe a moving picture (video) is worth a million people." Statistics show that in the United States, 75 million people watch videosonline each month and 40 billion videos are streamed monthly.*
>
> — Brian D. Ross, *If "A Picture is Worth a Thousand Words," What is a Video Worth?*

Video is nothing new to us: when we discussed motion models and optic flow, we introduced *time* as a 3rd dimension in our sequence of images. Many of the concepts we've discussed thus far can be extended to the temporal dimension.

15.1 Background Subtraction

We'll be discussing some ideas we introduced in that very chapter on chapter 11. Namely, we open with **background subtraction**. From a sequence of images, our goal is to determine which parts are moving.

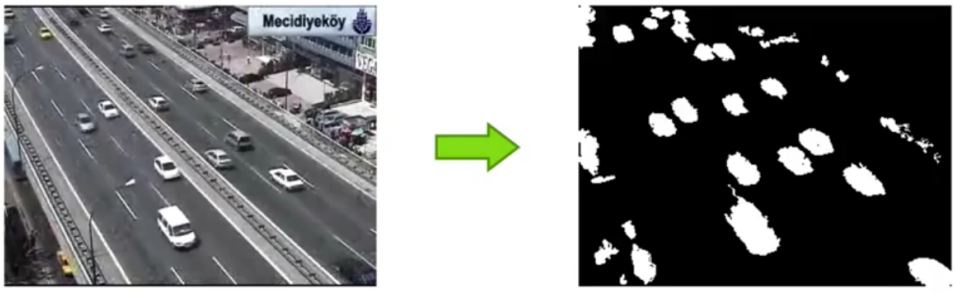


Figure 15.1: Subtracting the background from a highway camera, enabling identification of the "moving blobs" — the vehicles.

Turns out, this is harder than it seems; it's an area of active research. Even with a static camera, this is actually still a relatively tough problem to solve. There are tons of uses, including for traffic monitoring (as seen in Figure 15.1), action recognition, human-computer interaction, and more.

15.1.1 Frame Differencing

Let's walk through a simple approach for background subtraction:

1. Estimate the background at time t, $B(x, y, t)$.
2. Subtract the estimated background from the current frame, $I(x, y, t)$.
3. Apply a threshold to the absolute difference to get the "foreground mask:"

$$|I(x, y, t) - B(x, y, t)| > \text{threshold}$$

Figure 15.2: The previous frame at $t-1$ (right) and the current frame at t (left). Notice the slight forward motion of the far-right white car.

Naturally, the challenge comes from estimating the background. A naïve method is called **frame differencing**: it assumes the background is simply the content of the previous frame. In other words, $B(x, y, t) = I(x, y, t-1)$.

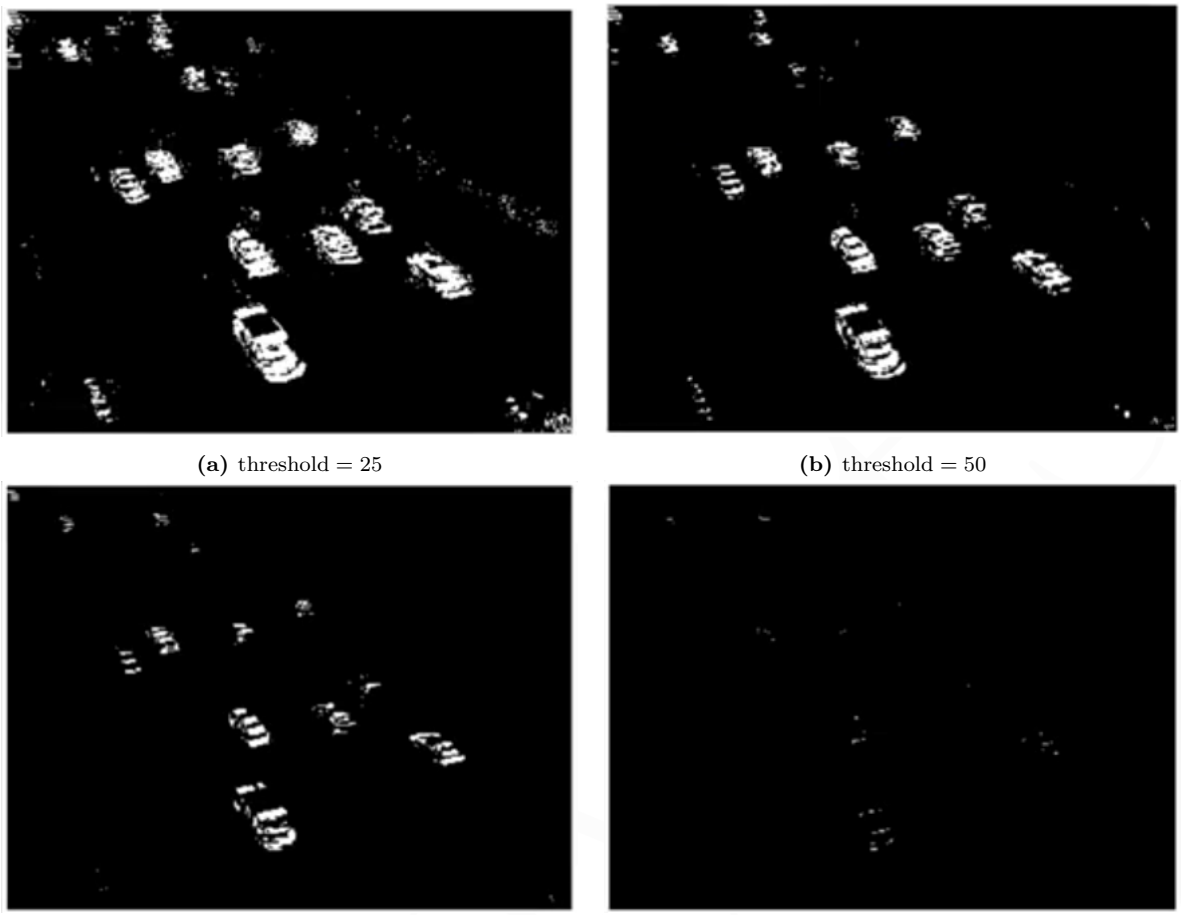
As you might expect, this performs poorly. It's effectiveness (or, usually, lack thereof) is highly dependent on the object structure, speed, frame rate, and threshold. The method is not robust to large areas that don't change much from one frame to another (but are **not** part of the background). For example, the roof of the van in the bottom-middle of Figure 15.2 is solid white in both frames, just slightly offset. In the resulting background images in Figure 15.3, we can see that there is a gap in that very roof due to the large constant-intensity area.

We can do better than this, right? Let's consider **mean filtering**, in which the background is the average of the previous n frames:

$$B(x, y, t) = \frac{1}{n} \sum_{i=1}^{n} I(x, y, t-i)$$

Figure 15.4a shows this method applied to multiple frames from the source video used for Figure 15.2. As you can see, taking the average causes the moving objects to "blend" into the true background. The average isn't really the right thing to calculate...

Let's think about it differently. The background is a constant value with a little bit of variation, but any motion in the background is a *huge* change in intensity. Perhaps this rings a familiar—albeit



(a) threshold = 25 **(b)** threshold = 50

(c) threshold = 100 **(d)** threshold = 200

Figure 15.3: The resulting foreground mask after applying frame differencing using the current and previous frames from Figure 15.2.

faint—bell *way* back from when we discussed how a median filter works way better at removing salt-and-pepper noise in Figure 2.6 than a Gaussian noise filter? Let's assume the *median* of the previous n frames is the background, then:

$$B(x, y, t) = \text{median}\left\{I(x, y, t - i)\right\}$$

We can have a much larger n, now, to allow the median to dominate. Notice that the roof of the white van still influences the resulting background in Figure 15.4b because of its constancy over several frames. A larger n can alleviate this.

Background subtraction via frame differencing is incredibly simple; it's fast and easy to implement. The method is robust to background models that change over time. Unfortunately, though, the accuracy of the method is highly dependent on object speed and frame rate; this results in a reliance on a global threshold value, which is not ideal. Finally, the median method requires storing n frames in memory which can be costly.

(a) The estimated background using mean filtering.

(b) The estimated background using median filtering.

Figure 15.4: A comparison of mean filtering and median filtering on the same scene from Figure 15.2 (though obviously with more frames) using $n = 10$.

15.1.2 Adaptive Background Modeling

When does frame differencing fall apart? As you can imagine, most scenes aren't entirely static. A security camera in a park, for example, would see a specular reflection on the waterfront, or shimmer as wind blows across leaves, revealing the things behind it. This paper pioneered an adaptive method for countering this effect.

The main idea, as demonstrated in Figure 15.5, is that there are multiple values in the background.

The variety of intensities for a single pixel means that each pixel can be modeled as a mixture of Gaussians (like we did for histograms for Generative Supervised Classification, see **??**) that are updated over time, as in Figure 15.6.

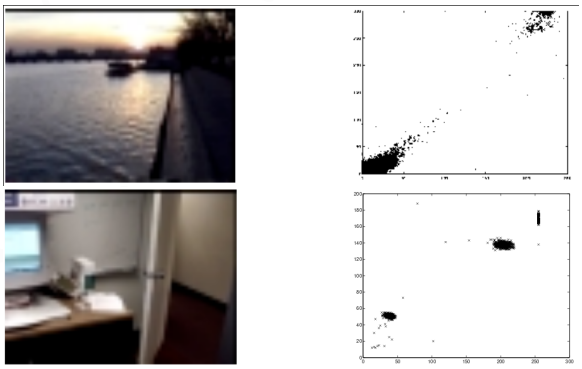


Figure 15.5: Taken from the source, these plots demonstrate the variance in the background intensity value for a handful of different scenes. For example, (b) demonstrates the specular reflection of the sun off of the water and shimmer from waves, whereas (c) demonstrates two states for the door: open or closed.

For each new image, if the pixel falls into the distribution, we will likely consider it to be a background pixel. Otherwise, it's likely to be an object in motion, though we might hold on to it for a few frames to see if it *becomes* background.

15.1.3 Leveraging Depth

There are many modern camera devices (like the Kinect) that measure another dimension: depth. We now can work with RGB and d values. In this case, regardless

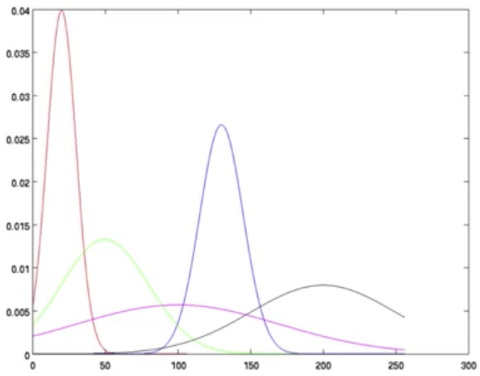


Figure 15.6: A possible mixture of Gaussians for a background pixel.

of how the light changes, we can identify backgrounds based on their depth values which stay fixed. Unless you move your furniture daily, the depth can give reliable information about a scene and determine the background simply based on its geometry.

15.2 Cameras in Motion

Suppose instead of a static camera and a moving scene—to which we applied background subtraction above—we instead have a static scene and a camera that moves around it. What kind of cool stuff can we do there?

A moving camera gives us different views of the same scene... didn't we have an entire chapter dedicated to Multiple Views? That's right, we did. We also talked about Stereo Geometry and showed that disparity is inversely proportional to depth. The closer something is to you, the more motion you perceive when it moves; colloquially, this is understood as parallax motion).

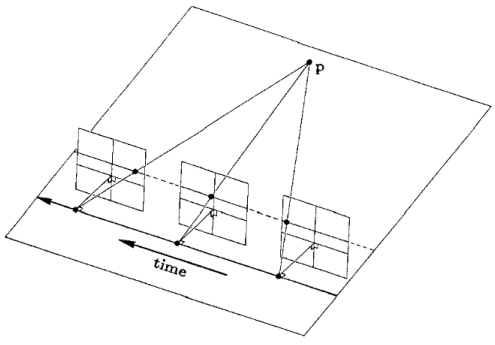
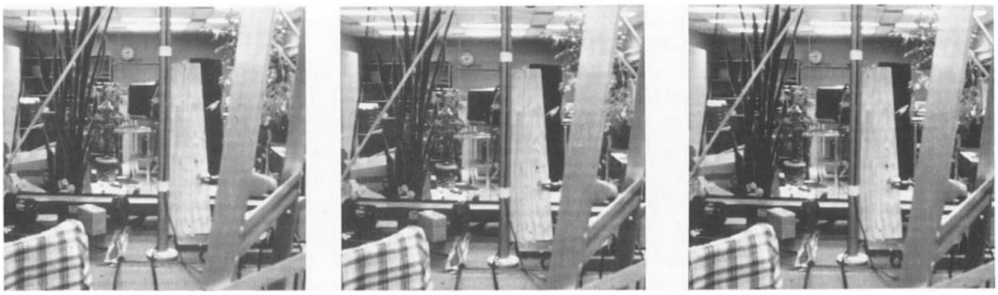


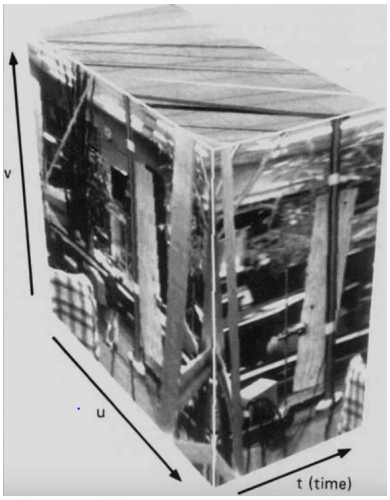
Figure 15.8: As a camera's optical center moves, the point in world space will likewise move across the image plane.

We can use this to learn something about a scene: the rate of motion of a point on an image is a function of its depth. As we said, a closer point in world space will move faster across the image plane with the same camera motion. This form of analysis is known as "epi image analysis" and roots itself in the epipolar

The set of images taken by moving a camera across a scene creates a *volume* of data; we can take slices of the volume in different orientations. Notice the streaks across the volume block in Figure 15.9: they are at different slopes, and those slopes actually correspond to different *depths* in the scene.



(a) Three images from a scene captured with a moving camera.



(b) The resulting volume of data.

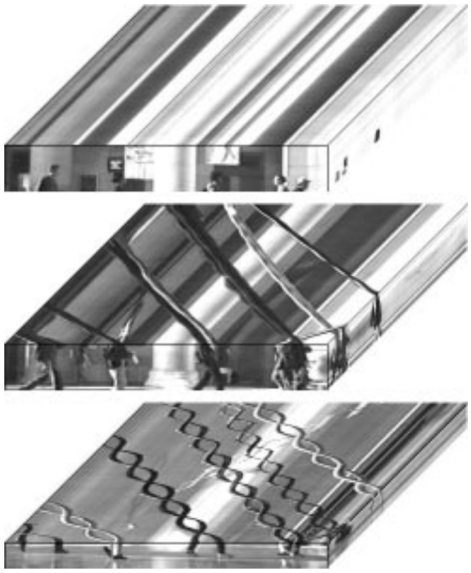
Figure 15.9: Creating a volume of data from a series of images in a static scene, taken with a camera that moves over time.

Researchers applied this understanding in this paper to develop a notion of **epi-gait**. By analyzing the location of feet in a "video volume," they could determine a person's unique gait and use it for identification.

The streaks in the middle image of Figure 15.10b are the heads; you can see a slight bob in some as the streak is not perfectly smooth. The braided pattern in the bottom portion of Figure 15.10b is the visualization of feet in space-time; that is, it's the movement of their feet over time. In fact, just taking a cross-section of the ankles and observing the criss-crossing of the walker's legs gives a **unique** braided signature for walking patterns. The goal wasn't a robust method for person recognition (far more advanced techniques exist for gait analysis), but is just rather an interesting application of viewing video footage as a volume of data.

> The moral of the story? If you want to avoid being recognized, put rocks in your shoes. ☺

(a) A single frame from some video footage.



(b) The resulting volume of data in different regions of the footage.

Figure 15.10: Analysing subject gait through visualizing the video footage as a volume, from Niyogi and Adelson.

15.3 Activity Recognition

Moving on from just *processing* videos, we'll continue with the trend of extracting "good stuff" from our data. In this section, we'll explore labeling videos with activities or general descriptions of the video content. There is no standard universal terminology for this topic, so our nomenclature is (loosely) as follows:

Event An event is a single moment in time, or a single instance in the time detection. For example, the instant a door closes in a video would be an event.

Actions Also called **movements**, these are "atomic motion patterns," which is fancy-talk for actions with a single describable behavior. For example, sitting down would be an action, or waving your arms. It's often a gesture-like behavior and can (possibly, if you're a huge nerd) be described as a movement through a feature space (like the joints of the body).

Activity An activity is the highest level of abstraction and represents the series or composition of actions, such as interactions among a group of people.

There are a handful of basic approaches to activity recognition.

Model-based *action* recognition Build a classifier from data derived or "learned" from techniques that can estimate human poses or movements based on models of the human body or other body tracking techniques. The major challenge is replicating the testing environment in your training environment to get an accurate classifier.

Model-based *activity* recognition Given some lower-level detection of actions (or events), an activity can be recognized by comparing it to some structural representation of an activity. What we're talking about here is essentially a mathematical model for human behavior while playing soccer, for example, which seems like an outlandish notion but is entirely plausible. Soccer is a composition of actions: kicking, jumping, running, etc. If these individual instances can be modeled (likely probabilistically) and detected, we can build them into a larger composite. The major challenge here is handling uncertainty as well as the difficulties of tracking itself: occlusions, ambiguities, and low-resolution data can all hinder tracking.

Activity as Motion Some recent work has been done that disregards both of the previous approaches. Instead video is viewed simply as a space-time volume (as we just did, in Cameras in Motion) and a series of appearance patterns. A classifier is trained on these patterns, though no overall body-tracking is done; it's pure analysis of the volume's content.

Activity from Static Images Interestingly enough, video isn't a prerequisite for activity recognition. You can imagine what a person playing a flute would look like; similarly, you can recognize the activity of a person playing a flute from a simple static image of said player. There is a particular pose, particular objects and colors, etc. that can easily be analyzed. This isn't strictly activity recognition under the context of our nomenclature, per se, it's more of "representative image" recognition or just a more abstract/complex image classifier.

To briefly summarize a technique we're *not* going to discuss, see Figure 15.11. In essence, we can use the same techniques we discussed in Visual Bags of Words to train an SVM from a "feature space" of "interesting" space-time patches that represent our visual video words. It's a 3D (from $(x, y) \to (x, y, t)$) extension of concepts we've already covered.

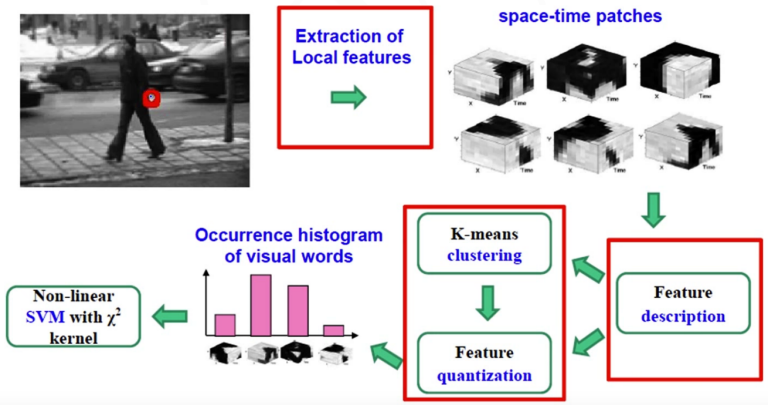


Figure 15.11: Adapting bags of visual words to video.

Instead, we'll be looking at the properties of the raw motion that occurs in a video, mimicking how humans can interpret a scene just given some motion, even if the image they're seeing doesn't show anything very meaningful.

15.3.1 Motion Energy Images

Suppose we examine the cumulative areas that changed in a video over time as in Figure 15.12.

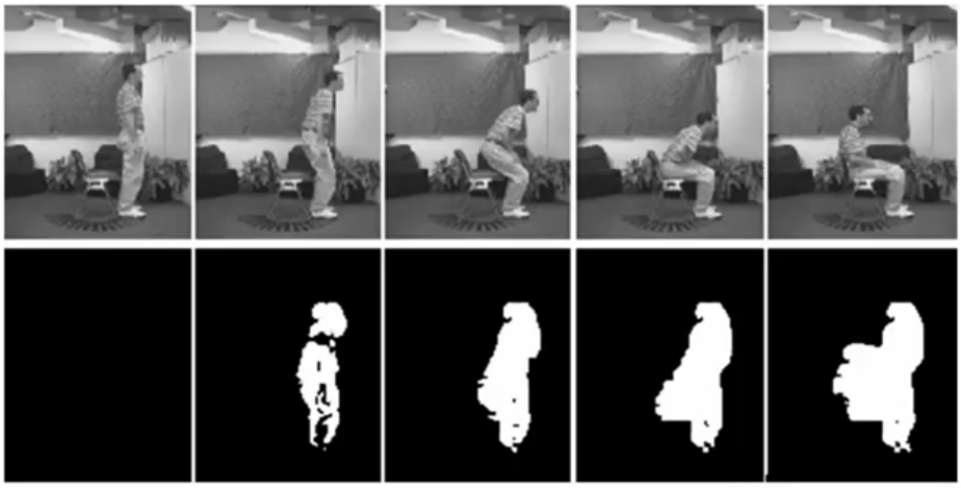


Figure 15.12: The cumulative areas of motion over the course of five video frames.

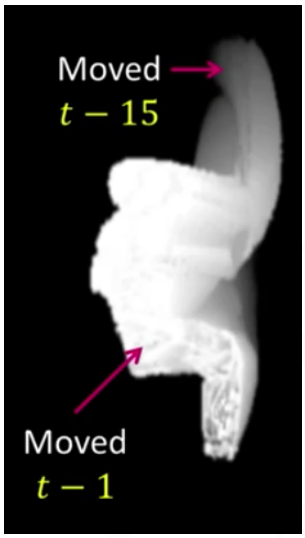


Figure 15.13: Visualizing the temporal domain in the cumulative motion image captured at the end of the last frame in Figure 15.12.

These are called **motion energy image**s. If we further include the temporal domain into the areas of motion, we get the cool cloud-like **motion history image** in Figure 15.13. This image can be formed with a simple recursive relationship:

$$I_\tau(x,y,t) = \begin{cases} \tau & \text{if moving} \\ \max(I_\tau(x,y,t-1)-1,0) & \text{otherwise} \end{cases}$$

Simply put, we set the pixel to some maximal value τ if its in motion; otherwise, we decrement its value, taking care to stay ≥ 0. The result, as seen in Figure 15.13, is that brighter pixels represent more recent change.

Now we can see not only what things have moved, but also how recently they moved. Interestingly enough, we can compare someone sitting quickly and slowly with relative ease: constructing $I_{\tau-k}(x,y,t)$ is trivial.

We can use these temporal representations of motion and return to "old school" computer vision: summarize the statistics of a particular motion pattern, construct a generative classification model, then do recognition. A handful of the motion history images are shown in Figure 15.14.

How do we "summarize" these images? This requires a quick diversion into some ancient computer vision to discuss *image moment*s. Don't worry, this will only take a **moment** of your time. Heh, nice.

Image Moments

In physics, we say that the moment is the distribution of matter around a point or axis. Similarly, the particular ij^{th} **image moment** M_{ij} is related to the distribution of image content by exponentation:

$$M_{ij} = \sum_x \sum_y x^i y^j I(x, y)$$

Naturally, this disregards pixels that are "empty;" only non-zero intensity values are considered. In a binary image such as the the motion energy images from Figure 15.12, this becomes any part of the image with motion content. In the grayscale motion history images from Figure 15.13, the pixel intensities become weights in the image moment.

For example, the 0^{th} moment is just number of non-zero pixels, or the **area** of the image. The problem with moments in general is that they are not translation invariant: the value depends on where x and y actually *are*. To alleviate that, we can use the **central moment**s which leverage the average position of the image's content:

$$\mu_{pq} = \sum_x \sum_y (x - \overline{x})^p (y - \overline{y})^q I(x, y)$$

$$\overline{x} = \frac{M_{10}}{M_{00}} \qquad \overline{y} = \frac{M_{01}}{M_{00}}$$

Even with this invariance, we still have a problem. Things like scaling and rotation will similarly have a big impact on the value of a moment, even though the "thing" that is being described is the same. This is solved by **Hu moment**s, which are seven moments crafted to be translation, rotation, *and* scale invariant (from Hu's 1962 paper).

You would never need the raw equations because libraries will have bug-free implementations for the Hu moments (such as OpenCV), but here they are anyway:[1]

$$h_1 = \mu_{20} + \mu_{02} \tag{15.1}$$

$$h_2 = (\mu_{20} - \mu_{02}) + 4\mu_{11}^2 \tag{15.2}$$

$$h_3 = (\mu_{30} - 3\mu_{12}) + (3\mu_{21} - \mu_{03})^2 \tag{15.3}$$

$$h_4 = (\mu_{30} + \mu_{12}) + (\mu_{21} + \mu_{03})^2 \tag{15.4}$$

$$\begin{aligned} h_5 = (\mu_{30} - 3\mu_{12})(\mu_{30} + \mu_{12}) \left[(\mu_{30} + \mu_{12})^2 - 3(\mu_{21} + \mu_{03})^2\right] \\ + (\mu_{21} - \mu_{03})(\mu_{21} + \mu_{03}) \\ \cdot \left[3(\mu_{30} + \mu_{12})^2 - (\mu_2 1 + \mu_{03})^2\right] \end{aligned} \tag{15.5}$$

$$\begin{aligned} h_6 = (\mu_{20} - \mu_{02}) \left[(\mu_{30} + \mu_{12})^2 - (\mu_{21} + \mu_{03})^2\right] \\ + 4\mu_{11}(\mu_{30} + \mu_{12})(\mu_{21} + \mu_{03}) \end{aligned} \tag{15.6}$$

$$\begin{aligned} h_7 = (3\mu_{21} - \mu_{03})(\mu_{30} + \mu_{12}) \left[(\mu_{30} + \mu_{12})^2 - 3(\mu_{21} + \mu_{03})^2\right] \\ - (\mu_{30} - 3\mu_{12})(\mu_{21} + \mu_{03}) \left[3(\mu_{30} + \mu_{12})^2 - (\mu_{21} + \mu_{03})^2\right] \end{aligned} \tag{15.7}$$

This is irrelevant, though. What we care about is the feature vector that results from computing these Hu moments on the grayscale motion history images above in Figure 15.13 and Figure 15.14 and the binary motion energy image in Figure 15.12. With feature vectors, we can build a classifier.

[1] In fact, I wouldn't be surprised if these were transcribed incorrectly. Stop rolling your own vision code!

Classification with Image Moments

Remember the difference between the two main supervised learning methods? In generative classifiers, we build a model for each class and compare them all, whereas in discriminative classifiers, we build a model of the boundary between classes. How would you build a generative model of each class of actions for activity recognition?

A simple approach might be to construct a Gaussian (or a mixture of Gaussians) in the feature space of Hu moments. Each of the seven moments from (15.1) renders a scalar value; applying these to two images (the motion history image and the motion energy image) gives us a point in a 14-dimensional feature space.

At that point, it's a matter of comparing likelihoods for each model: $\Pr[\text{data} \mid \text{action}_i]$. Remember, this is the probability of that data occurring given the particular model. Then, **if** we have priors, we can apply Bayes' rule (recall, from footnote 3) as we've done before:

$$\Pr[\text{model}_i \mid \text{data}] \propto \Pr[\text{data} \mid \text{model}_i] \cdot \Pr[\text{model}_i]$$

Otherwise, we might select on the highest likelihood, or even fall back to a nearest neighbor.

An interesting thing worth noting is that our model is representative of the *complete* motion. We can't really do "on line" activity recognition; instead, we have to wait until the action is completed (e.g. the person finished sitting down) in order to properly model the activity.

15.3.2 Markov Models

Generative models are so '90s. That was then, and *this* is now. In this section, we'll be diving into techniques for activity modeling and recognition that are more in-line with modern machine learning, so buckle up. As always, diving deeper into these topics is more appropriate for a machine learning course.

We'll be exploring **time series** (which is just our formalization of the idea of a video as being a 2D signal that changes over time) and using **Markov model**s to analyze them. This is an idea rooted in signal processing (for example, audio signals or stock market data), but can likewise be extended to images and vision to understand various activities. For example, we may want to determine whether or not someone is waving or pointing in an image, as in Figure 15.15.

How do we model these problems (such as, "Is that person waving?") and, furthermore, how do we formulate these questions into inference or learning problems? Given a novel input, we should be able to ask, "Which of my models am I most likely seeing?"

So what *is* a Markov model, which is the statistical construct we'll be using to analyze these time series problems? Let's begin with an example: weather. Suppose the following:

- Given a **sunny** day, there's an 80% chance it will stay **sunny**, a 15% chance it will be **rainy**, and a 5% chance that it will be **snowy** on the following day.
- Given a **rainy** day, there's a 60% chance it will stay **rainy**, a 38% chance it will be **sunny**, and a 2% chance it will be **snowy** on the following day.
- Given a **snowy** day, there's a 20% chance it will stay **snowy**, a 75% chance it will be **sunny**, and a 5% chance it will be **rainy** on the following day.

In summary, we have a table of transition probabilities that looks like this:

Currently	Sunny	Rainy	Snowy
sunny	80%	15%	5%
rainy	38%	60%	2%
snowy	75%	5%	20%

This is a 1st order Markovian system because all we need to predict the weather of the *next* day is the information about the *current* day. More formally, the probability of moving to a given state depends only on the current state.

In general, we can abstract our **states** into a set: $\{S_1, S_2, \dots, S_N\}$; in our running example this is the weather. We also have the state transition probabilities: $a_{ij} = \Pr[q_{t+1} = S_i \,|\, q_t = S_j]$, where q_t is the state at time t. For example, the state transition probability from **rainy** to **snowy** is 2%, so $\Pr[q_{t+1} = \text{snowy} \,|\, q_t = \text{rainy}] = 0.02$. We also have an *initial* state distribution, $\pi_i = \Pr[q_1 = S_i]$, which is the state at $t = 1$ (this weren't given for our example).

Let's define each of these for our running example:

$$
\begin{aligned}
S &= \{S_{\text{sunny}}, S_{\text{rainy}}, S_{\text{snowy}}\} \\
\mathbf{A} &= \begin{bmatrix} 0.80 & 0.15 & 0.05 \\ 0.38 & 0.60 & 0.02 \\ 0.75 & 0.05 & 0.20 \end{bmatrix} \\
\boldsymbol{\pi} &= \begin{bmatrix} 0.70, 0.25, 0.05 \end{bmatrix}
\end{aligned}
$$

Given such a model, we can ask certain questions about the state of the system. For example, given the following time series: **sunny**, **rainy**, **rainy**, **rainy**, **snowy**, **snowy**, what is the probability of it occurring? This is just a product of each probability:[2]

$$
\begin{aligned}
&= P(S_{\text{sunny}}) \cdot \Pr[S_{\text{rainy}} \,|\, S_{\text{sunny}}] \cdot \Pr[S_{\text{rainy}} \,|\, S_{\text{rainy}}] \\
&\qquad \cdot \Pr[S_{\text{rainy}} \,|\, S_{\text{rainy}}] \cdot \Pr[S_{\text{snowy}} \,|\, S_{\text{rainy}}] \cdot \Pr[S_{\text{snowy}} \,|\, S_{\text{snowy}}] \\
&= 0.7 \cdot 0.15 \cdot 0.6 \cdot 0.6 \cdot 0.02 \cdot 0.2 \\
&= 0.0001512 \approx 0.015\%
\end{aligned}
$$

That's Markov models in a nutshell, but **hidden Markov models** are far more interesting. In this model, the underlying state probabilities are unknown; instead, what we have are **observables** or **evidence** about the world. Returning to our example, we might not be able to see the weather, but we can see what people outside are wearing. This model requires a relationship between the observables and the states: the **emission probabilities** (denoted with $\mathbf{B}$) describe the chance to see k, an observable, at a particular state S_i: $b_j(k) = \Pr[o_t = k \,|\, q_t = S_i]$. For example, we would need the probability of seeing a **bathing suit** given that it's **sunny** outside.

Now instead of asking about a particular series of states, we can ask about the probability of seeing a particular series of *observations*.

[2] Naturally, multiplying a series of values that are all less than 1 results in a *very* small number a lot of the time. Small numbers give computers trouble (and are harder to reason with for humans), so often the log-likelihood is used instead.

Formally, a hidden Markov model λ, or HMM, is made up of a triplet of components: $\lambda = (\mathbf{A}, \mathbf{B}, \boldsymbol{\pi})$. In detail, these are:

- $\mathbf{A}$, the state transition probability matrix, defined as before:

$$a_{ij} = \Pr[q_{t+1} = S_i \,|\, q_t = S_j]$$

- $\mathbf{B}$, the observation probability distribution, defined as:

$$\begin{aligned} b_j(k) &= \Pr[o_t = k \,|\, q_t = j] \qquad 1 \leq k \leq M \\ b_j(x) &= \Pr[o_t = x \,|\, q_t = j] \end{aligned}$$

  for the discrete and continuous cases, respectively.

- $\boldsymbol{\pi}$, the initial state distribution, defined as before:

$$\boldsymbol{\pi} = \begin{bmatrix} \Pr[q_1 = S_0] & \Pr[q_1 = S_1] & \dots & \Pr[q_1 = S_M] \end{bmatrix}$$

With the (brief summary of the) theory out of the way, let's get back to vision.

(to defeat... the Huns)

Applying HMMs to Vision

How do HMMs relate to vision? Well, our time series of images is a sequence of observations. Now we can ask, "what model generated those?"

Returning to the example of clothing as an observation, suppose we saw some series of outfits: **coat**, **coat**, **umbrella**, **umbrella**, **bathing suit**, **umbrella**, **umbrella**. Now we want to know which of three cities these observations came from: Philadelphia, Boston, or Newark. The underlying assumption is that we have trained models for each of these cities that came from distributions describing observed outfits. The HMM would allow us to evaluate this probability.

Generally speaking, HMMs allow us to model 3 different problems:

Evaluation Given the HMM λ, what is the probability of the occurrence of a particular observation sequence $O = \{o_1, \dots o_t\}$? That is, what is $\Pr[O \,|\, \lambda]$? This is our classification / recognition problem: given a trained model for each class in a set, which one is most likely to generate what we saw?

Decoding What is the optimal state sequence to produce an observation sequence $O = \{o_1, \dots o_t\}$? This is useful in recognition; it helps give meaning to states.

Learning Determine (or "learn") the model λ given a training set of observations. In other words, find λ such that $\Pr[O \,|\, \lambda]$ is maximal; this involves finding the maximum likelihood via expectation maximization which we'll discuss briefly later.

Let's work through the math for each of these in turn.

Solving the Evaluation Problem We'll start with the naïve solution and get to the cool one afterwards. Assume first that we actually know our state sequence, $Q = (q_1, \ldots, q_T)$. Then, $\Pr[O \mid q, \lambda]$ (the probability of an observation series given the HMM) is just a product of the independent probabilities:

$$\Pr[O \mid Q, \lambda] = \prod_{t=1}^{T} \Pr[o_t \mid q_t, \lambda] = b_{q_1}(o_1) \cdot b_{q_2}(o_2) \cdot \ldots \cdot b_{q_T}(o_T)$$

Furthermore, we know the probability of *getting* a particular state sequence Q. It's the product of the state transition probabilities from $\mathbf{A}$ for each $q_i \to q_{i+1}$.

$$\Pr[Q \mid \lambda] = \pi_{q_1} a_{q_1 q_2} a_{q_2 q_3} \ldots a_{q_{T-1} q_T}$$

In that case, we can say that

$$\Pr[O \mid \lambda] = \sum_{q \in Q} \Pr[O \mid q, \lambda] \Pr[q \mid \lambda]$$

Unfortunately, this is a summation over **all** possible paths through the state sequence. There are N^T state paths each costing $O(T)$ calculations, so that's a **massive** $O(TN^T)$ time complexity.

We can do so much better...

Recall the **Markovian property**: the probability of the next state *only* depends on the current state, not how we got to the current state. To take advantage of that, let's define an auxiliary forward variable α, which describes the probability of observing a particular sequence of observables $o_1, \ldots, o_t$ **and** the being in state i at time t:

$$\alpha_t(i) = \Pr[o_1, \ldots, o_t, q_t = i \mid \lambda]$$

With this variable, we can define a recursive relationship for α.

First, initialize:

$$\alpha_1(i) = \pi_i b_i(o_1) \tag{15.8}$$

Then, we should realize that the next value is one of some set of previous values: either we were at some state $q_t = i$, or $q_t = j$, or ... These probabilities are mutually exclusive, so we can just sum them up:

$$\alpha_{t+1}(j) = \left[\sum_{i=1}^{N} a_{ij} \alpha_t(i) \right] b_j(o_{t+1}) \tag{15.9}$$

To explain further, we have a sum of the possible previous αs (which include the observations up to time t) weighed by their transition probabilities from $\mathbf{A}$, all multiplied by the *new* observation probability.

Now if we don't actually care about the state we end up in, and only want the probability of the sequence of observations, we can conclude:

$$\Pr[O \mid \lambda] = \sum_{i=1}^{N} \alpha_T(i) \tag{15.10}$$

...which is what we wanted to find in the first place.[3] The complexity of this is $O(N^2T)$, which is *much* more manageable.

This is the meaning of "training" the HMM. With these probabilities and αs, we can now determine which model a novel input is most likely to have come from.

Solving the Decoding and Learning Problems In the previous bit, we defined a *forward* recursive algorithm that gave us $\Pr[O \mid \lambda]$, or the likelihood of being in a state i at time t and having observed all of the evidence up to time t. We can similarly define a *backward* recursive algorithm that computes the likelihood of being in a state i at time t and observing the **remainder** of the evidence.

With both of these algorithms *and* knowledge of the HMM λ, we can **estimate** the distribution over which state the system is in at time t. Then, with those distributions and having *actually* observed the evidence (as opposed to just calculating the "What *if* we observed this evidence?" scenario), we can determine the emission probabilities $b_j(k)$ that would **maximize** the probability of the sequence. Furthermore, we can also determine the transition probabilities a_{ij} that maximize the probability of the sequence.

Given our newfound a_{ij}s and $b_j(k)$s, we can get a new estimate of the state distributions and start again. This iterative process is called **expectation maximization**, though we won't get into the mathematics beyond what was discussed above.

TODO: I'm actually short a video or two on this section.

[3] This high-level discussion excluded details about Trellis diagrams and the Baum-Welch algorithm that are core to this efficient computation method; these are avenues to explore if you want more information.

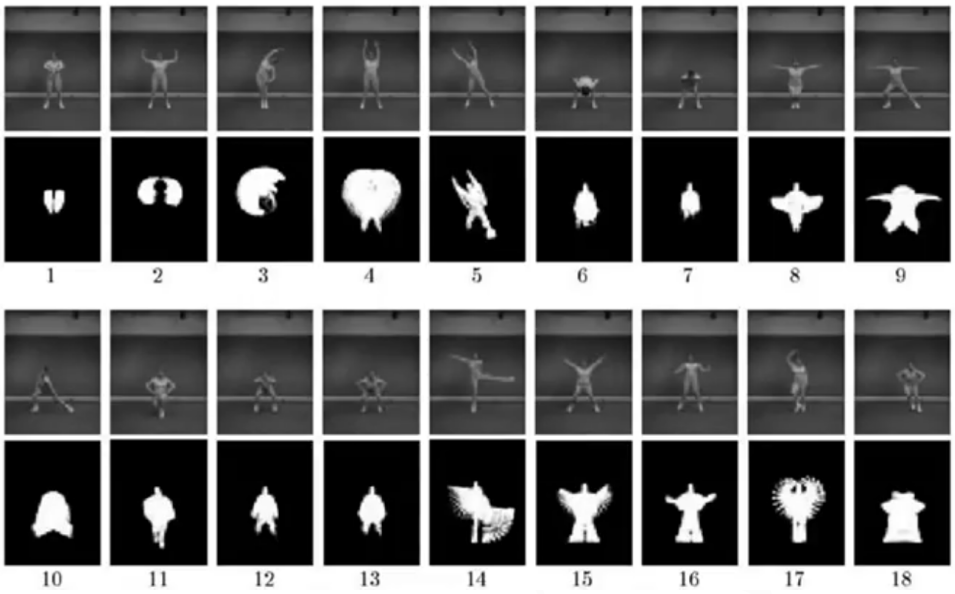


Figure 15.14: A series of motion history images for 18 different actions, from Davis & Bobick '99.

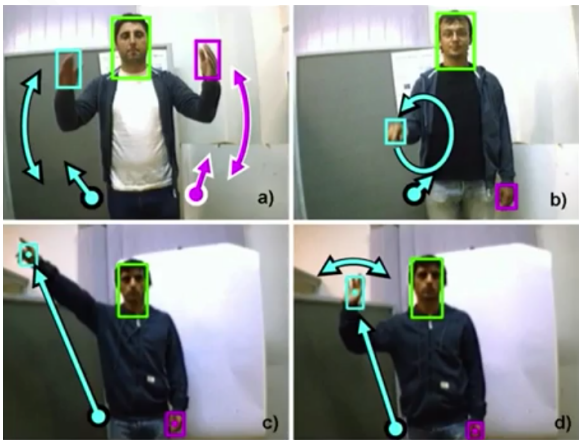


Figure 15.15: Various actions being recognized as part of a time series.

Segmentation

> *I stand at the window and see a house, trees, sky. Theoretically I might say there were 327 brightnesses and nuances of colour. Do I have "327"? No. I have sky, house, and trees. The concrete division which I see is not determined by some arbitrary mode of organization lying solely within my own pleasure; instead I see the arrangement and division which is given there before me.*
>
> — Max Wertheimer, *"Laws of organization in perceptual forms"*

Decomposing an image into separate regions—the process of **segmentation**—is an important step in preprocessing images prior to extracting out the "good stuff." The details and variations on *how* an image is segmented depends on how it's going to be interpreted and the requirements of that subsequent processing.

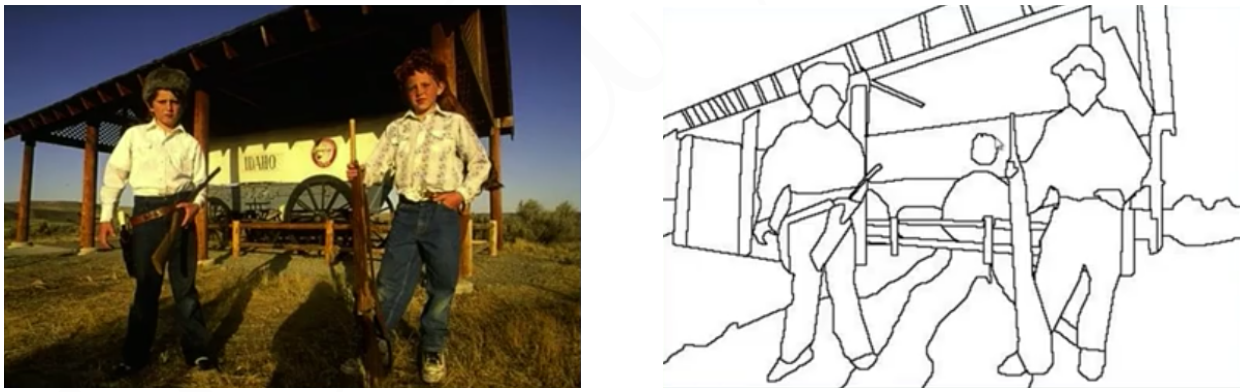
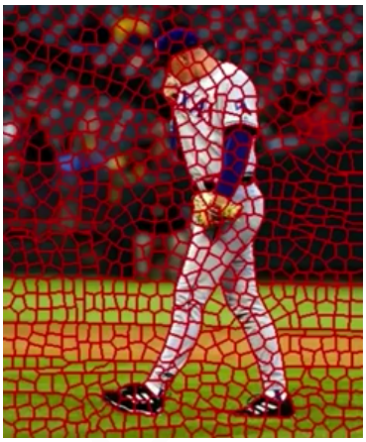


Figure 16.1: Segmentation of a scene into a rough discretization of components or "parts" of the image, from the Berkeley segmentation database.

There are a number of uses for segmentation, including separating a scene out into "parts" as in Figure 16.1, doing Background Subtraction (on individual images instead of a video feed) via **figure ground segmentation** as in Figure 16.2a), or even grouping pixel neighborhoods together to form "**superpixels**."

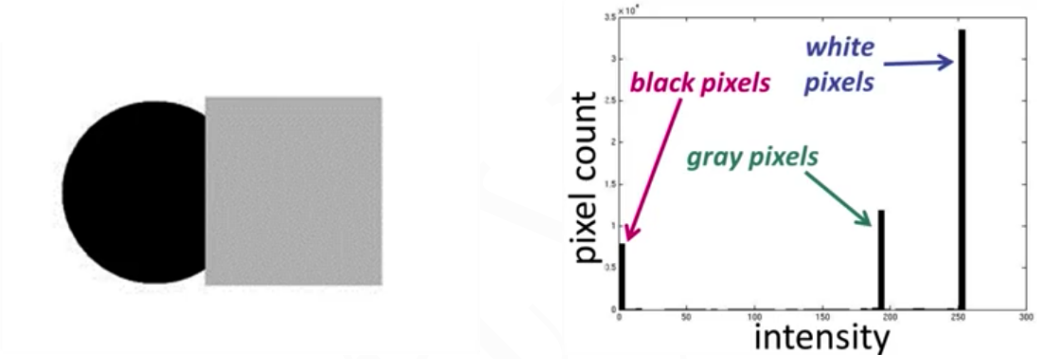
We'll open our discussion of segmentation with a toy example and then make it more robust and realistic. Suppose we begin with a clean input image that is very easy to dissect and its subsequent intensity histogram, as such:

(a) Figure ground segmentation of a foreground object (in this case, a small child) from its background.

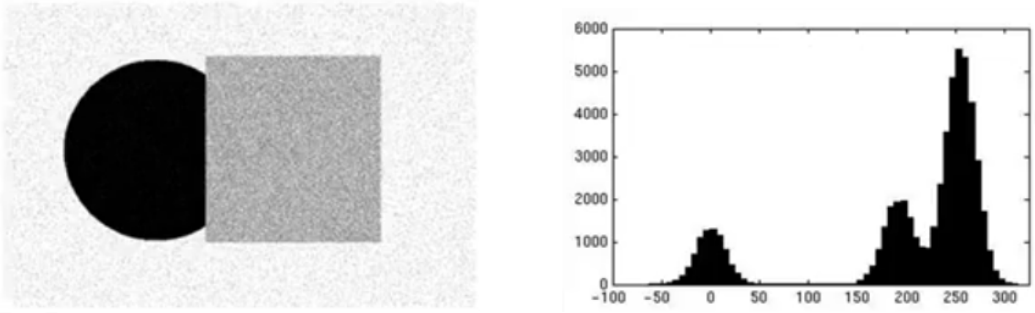


(b) Grouping of similar neighbors into "superpixels."

Figure 16.2: Other applications of segmentation.



Obviously, this is absolutely trivial to segment into 3 discrete components: the black circle, the gray square, and the white background. We've correlated the groups in the histogram to segments in the picture. What if we had a noisier image, though?



Suddenly, things are a little murkier. There are clear peaks in the histogram to the human eye, but that's insufficient; evolution has already solved the *human* vision problem. How can we make a computer find the three peaks? How can we let it know that there even *are* precisely three peaks? We want to *cluster* our histogram into the three intensities and determine which pixels belong together.

16.1 Clustering

Given a histogram of intensities as above, our simplified goal is to choose three "centers" as the representative intensities, then label every pixel according to which of these centers it's nearest to, defining three clusters.

What defines a "good" cluster center? Well, the best ones will minimize the SSD (squared sum of differences) between the points and their nearest cluster center, c_i:

$$\text{SSD} = \sum_{\text{cluster } C_i} \sum_{p \in C_i} \|p_j - c_i\|^2$$

We have a bit of a "chicken and egg" problem:

- If we knew the cluster *centers*, how do we determine which points are associated with each c_i? Naturally, we'd choose the closest c_i for each point p.
- If we knew the cluster *memberships*, how do we get the centers? Naturally, we'd make c_i the mean of all of the points in the cluster.[1]

So we have way of finding one if we have the other, but we start with neither... This actually leads us right into the ***k*-means clustering** algorithm (described in algorithm 16.1) which essentially solves the chicken-egg debate by starting with random cluster centers and iteratively improving on them; it can be thought of as a quantization of the feature space.

Figure 16.3: k-means clustering of a panda image with $k = 2$ and $k = 3$.

A natural question follows: how do we choose k? Well depending on what we choose as the "feature space," we can group pixels in different ways. If we chose intensity, we can discretize based on the "lightness" or "darkness" of an image as in Figure 16.3. Unfortunately, though, a fundamental problem of k-means is that the number of clusters must be specified in advance.

Of course, even pandas aren't purely black and white. We can cluster the same way in RGB space, grouping pixels based on their color similarity. This is especially helpful when objects differ in their chroma but not so much in their intensity.

[1] The mean (or average) actually minimizes the SSD, since you are implicitly assuming that your data set is distributed around a point with a Gaussian falloff; the mean then maximizes the likelihood.

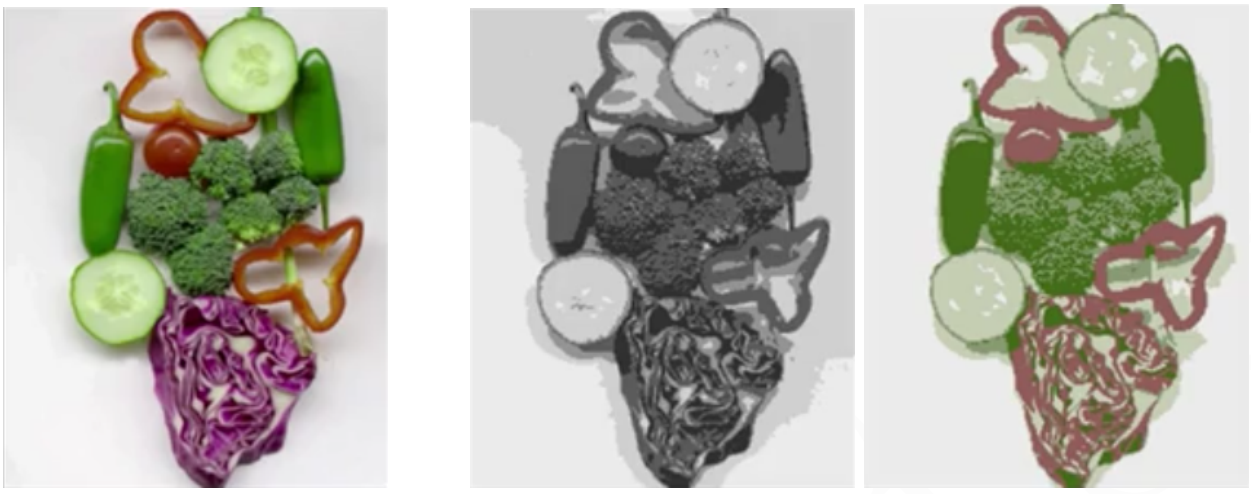


Figure 16.4: Applying k-means clustering on an input image using both intensity and color as the feature space.

Notice, of course, that k-means clustering doesn't actually care about image content. The two green peppers on either side of the above image are not recognized as being distinct clusters; they're grouped based purely on their color values. We can alleviate this trivially by again altering our feature space! By grouping pixels based additionally on (x, y) *position*, we can cluster features together based on color (5D) or intensity (3D) similarity *and* locality in the image.

16.1.1 Pros and Cons

Most of these should be obvious, but we'll cover them anyway. The main benefit of k-means clustering is that it's an incredibly simple and straightforward method; as you can see in algorithm 16.1, it requires a small handful of trivial operations. Furthermore, it actually provably converges to a *local* (not a global, mind you) minimum, so you're guaranteed a level of correctness.

There are some significant downsides, though, the biggest of which is that k must be specified in advance. It's a fairly memory intensive algorithm since the entire set of points needs to be kept around. Furthermore, it's sensitive to initialization (remember, it finds the *local* minimum) and outliers since they affect the mean disproportionately.

Another important downside to k-means is that it only finds "spherical" clusters. In other words, because we rely on the SSD as our "error function," the resulting clusters try to balance the centers to make points in any direction roughly evenly distributed. This is highlighted in the following figure:

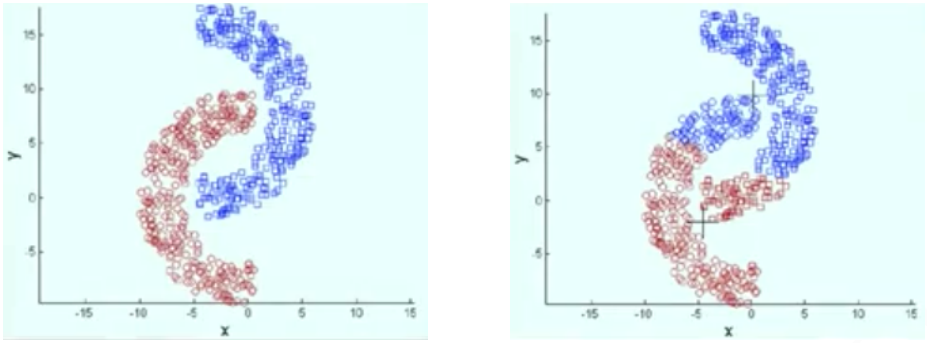


Figure 16.5: A problem of k-means clustering is that it prefers "spherical" clusters, resulting in an unintuitive (and partially incorrect) segmentation of the red and blue regions.

The red and blue segments are artificially colored to differentiate them in feature space, and as a human we would intuitively group them into those clusters, but k-means finds spherical clusters resulting in an incorrect segmentation.

ALGORITHM 16.1: The k-means clustering algorithm.

Input: A set of points, P.
Result: A set of cluster centers and their constituent points.

1 $C = \{c_1, \ldots, c_K\}$ `// Randomly initialize cluster centers.`
2 $V = [\emptyset_1, \ldots, \emptyset_K]$ `// A set of points corresponding to its cluster.`
3 **foreach** $p \in P$ **do**
4 | Find the closest c_i
5 | $V[i] \leftarrow p$ `// Put` p `into cluster` i
6 **end**
`// Given the points in each cluster, solve for` c_i `by setting it to the mean.`
7 **foreach** $c_i \in C$ **do**
8 | $c_i = \frac{\sum_j V[i]_j}{|V[i]|}$
9 **end**
10 **if** *any* c_i *has changed* **then**
11 | **goto** Line 3
12 **end**

16.2 Mean-Shift, Revisited

Let's move on from k-means, whose biggest flaw, as we mentioned, was the reliance on an *a priori* k value, and try to develop a better clustering method. In k-means, we're working with the assumption

that there are some "spherical clumps" in the feature space that we're trying to segment. If we think of feature space as a probability density, these clumps are like **modes** or peaks in our feature space with some falloff. If we could find those modes—local maxima in feature space—we might be able to segment the image that way.

We've discussed seeking the modes already in our discussion of Tracking when we introduced the mean-shift algorithm (see Mean-Shift). Let's return to it now except with an application to intensity or chroma values, which was an idea pioneered by this paper in 2002.

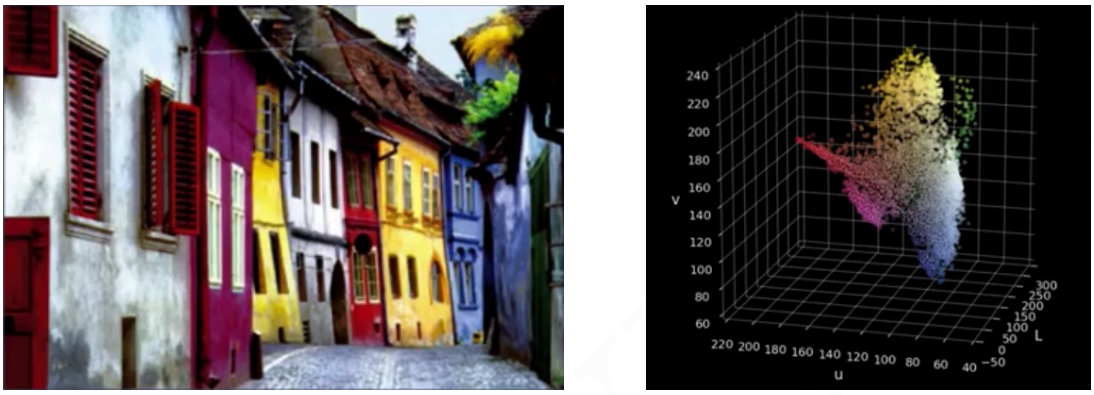


Figure 16.6: A mapping of an input image to a feature space of its *Luv* ♡ color values here.

We'll use the mean-shift algorithm to find the modes in our feature space. To revisit the algorithm, see Figure 16.7 which is a replication of Figure 12.11 from chapter 11. In summary, we continually shift our region of interest towards the weighted center of mass by the mean-shift vector until we converge to the local maximum.

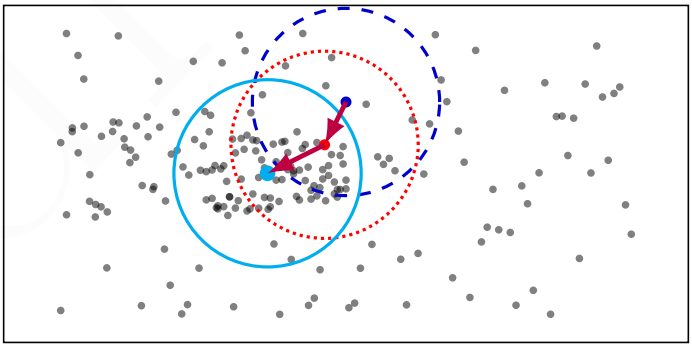
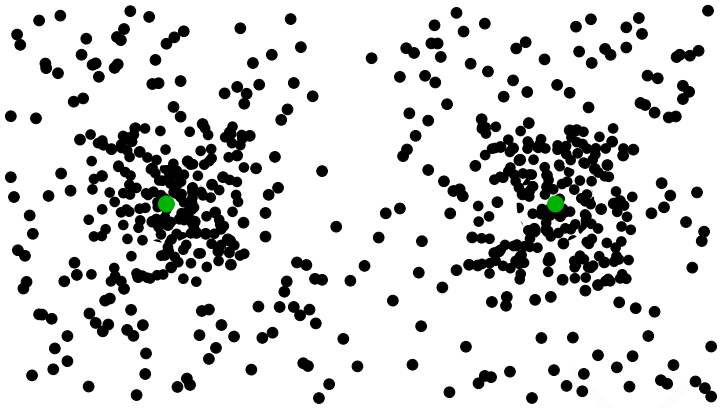


Figure 16.7: Performing mean-shift 2 times to find the area in the distribution with the most density (an approximation of its mode).

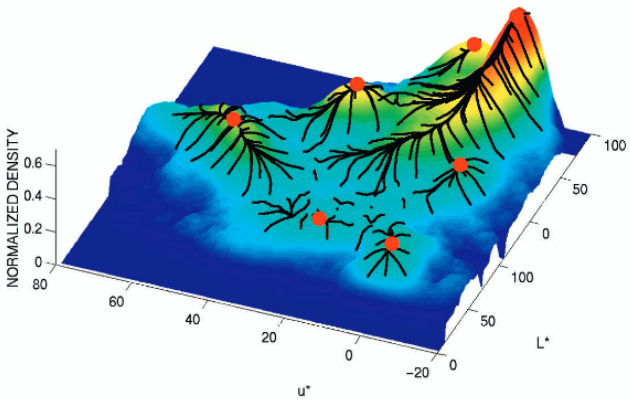
How do we apply this to clustering? We can define a cluster as being all of the data points in the "attraction basin" of a mode. The **attraction basin** is the region for which all trajectories lead to the same node.



We can clearly see two modes (approximately marked in green) in the above scatter plot, but how can we determine which mode which points belong to? Well if we start our region of interest at a particular point, it will shift towards one of the modes! That's the basin it belongs in. Thus we have an algorithm for using mean-shift for clustering and segmentation:

1. Define the feature space and find the values in the image (color, gradients, texture, intensity, etc.)
2. Initialize windows at individual feature points (a sample of them, not all, unless you can spare the compute resources) or pixels.
3. Perform mean-shift for each window until convergence.
4. Merge all of the windows (or pixels) that end up near the same "peak" or mode.

For the input image in Figure 16.6, we get a normalized probability density with seven basins of attractions (local maxima) that looks like this:



As you can see in Figure 16.8, the mean-shift segmentation algorithm results in some great results (and pretty artistic ones, at that!).

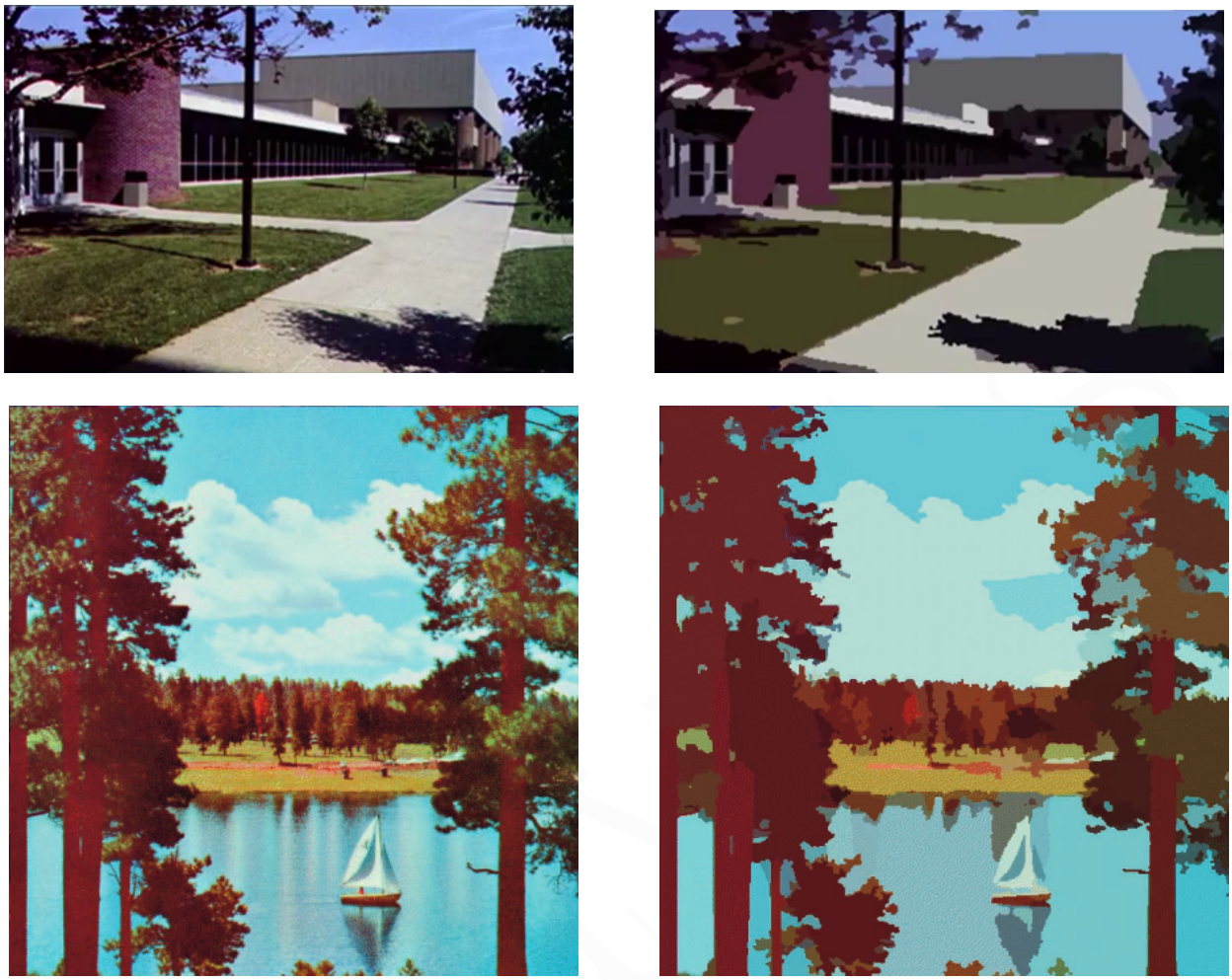


Figure 16.8: Some results of mean-shift segmentation (from this paper).

16.2.1 Pros and Cons

Unlike in *k*-means clustering, mean-shift *automatically* finds the basins of attraction; this is a huge automation benefit. Furthermore, it doesn't assume shape on the clusters like the spherical limitations of *k*-means and is still a general-purpose technique. These benefits are not gained at the cost of complexity: the technique is just as simple and window size is the only parameter choice.

Unfortunately, there ain't no free lunch. Mean-shift doesn't scale well with increasing dimensionality of the feature space. Higher-dimensional space will have a large amount of empty space due to the sheer scale of possible values[2], but mean-shift requires you to have a reasonable amount of points in the region of interest.

[2] For example, a 50-dimensional feature space discretized into 10 buckets has 10^{50} possible values!

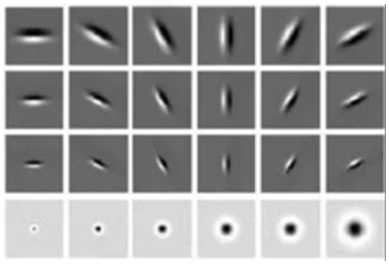
16.3 Texture Features

The features we've discussed (color, brightness, position) aren't sufficient for distinguishing all regions across all images. Some images, like those in Figure 16.9, have areas that clearly belong to the different segments but would look similar to an algorithm. For example, the average color of the background of the right-most image might be the same as the average color of an area on the woman's dress: something gray-ish.

Figure 16.9: A series of examples in which segmentation in a feature space defined just by color and position is insufficient.

As humans, it's trivial to segment the deer from the pile of leaves because we understand **texture**. Let's teach a computer about it based on this 2001 paper.

One approach might be to develop a **filter bank** based on patterns in the image. For example, for the woman on the right, we could have:



We have a few orientations and scales of the spots on her dress. The feature space is then the responses of the filter bank (i.e. convolution of the filter with the image). We'll perform clustering on the vectors of these filter bank outputs to find the **texton** map like the one in Figure 16.10. The value of the pixel at the image represents the presence of a particular texture. This map has some nice clusters which correspond to consistent textures, like the arm, but the dress is still littered with a variety of textons representing despite being a consistent polka dot texture.

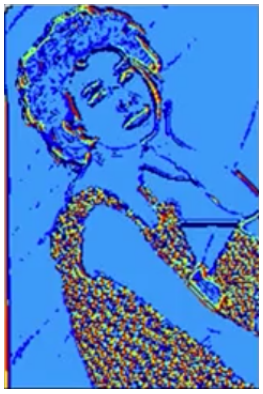
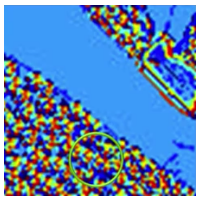


Figure 16.10: The texton map resulting from applying a filter bank to the image of the woman in Figure 16.9.

To solve this, we can do segmentation *again* on individual windows in the image.



Each window, like the circled area above has a particular histogram of textons. Areas like the woman's shirt will have similar, seemingly-random histograms whereas areas in which texture has already been consistently identified, like the arms and background, will have a particular dominant texton. We now do a second iteration of clustering based on the histogram space of the image.

For highly-textured images, this gives a much better result than the other feature spaces we used previously:

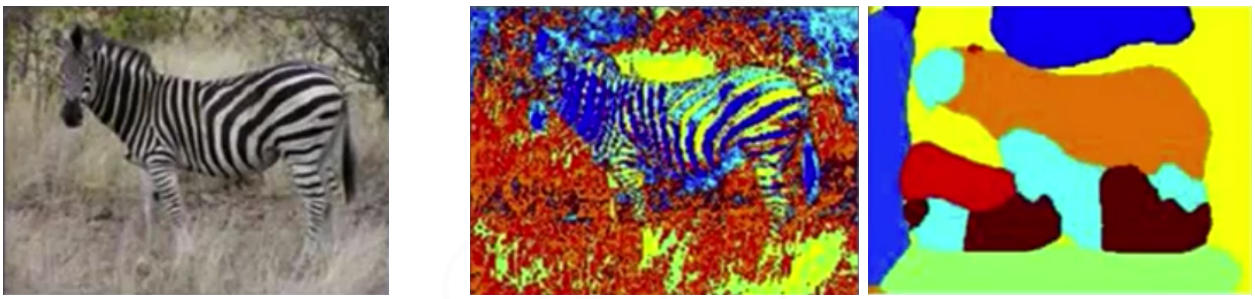
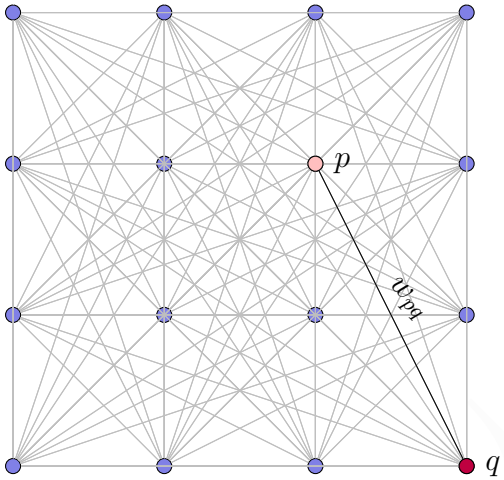


Figure 16.11: Results of image segmentation based on color (middle) vs. texture (right).

16.4 Graph Cuts

Until now, we've been approaching segmentation as clustering within some "feature space" that discretized our image based on particular properties of individual pixels. Whether color, position, or texture, none of these techniques took into account the actual connections between individual. Let's look at how we can treat the image as a cohesive whole instead of an arbitrary arrangement of points in an arbitrary feature space.

To do this, we need to treat images as fully-connected graphs, as described in this paper. Specifically, we're working with a graph that has one node (or vertex) for each pixel and a link between *every* pair of pixels, $\langle p, q \rangle$. Furthermore, each link (or edge) has an **affinity** weight w_{pq} which measures similarity: it's inversely proportional to difference in color and position.



A standard affinity function could be our friend the Gaussian, where the affinity falls off with distance:

$$\text{aff}(\mathbf{x}_i, \mathbf{x}_j) = \exp\left(-\frac{1}{2\sigma^2}\text{dist}(\mathbf{x}_i, \mathbf{x}_j)^2\right)$$

Obviously, σ is our parameter of choice here. Smaller sigmas group only nearby points together, whereas large sigmas group distant points in our "distance space," which doesn't necessarily have to include just Euclidean distance.

Naturally, we can achieve segmentation by partitioning the graph into segments. We do this by deleting links that cross between segments, which we can define as links with low affinity.

The standard algorithm for partitioning a graph is called the graph cut algorithm; we've alluded to it before when doing energy minimization in Better Stereo Correspondence. The graph cut algorithm determines the set of edges whose removal makes a graph disconnected.

The cost of a particular cut is the sum of the weights of the cut edges:

$$\text{cut}(A, B) = \sum_{p \in A, q \in B} w_{pq}$$

Figure 16.12: An example of segmentation by graph partitioning on an image of a tiger.

The graph cut gives us a segmentation, but what defines a "good" cut and how can we find one? The most-common graph cut is called the minimum cut (also called maximum flow); it's fast. Unfortunately, the whole purpose of min cut is to find small cuts with few connections; this often results in small, isolated components that are no good when it comes to image segmentation.

What we want to do instead is a *normalized* cut. This fixes the bias in min cut by normalizing for the size of the segments:

$$\text{Ncut}(A, B) = \frac{\text{cut}(A, B)}{\text{assoc}(A, V)} + \frac{\text{cut}(A, B)}{\text{assoc}(B, V)}$$

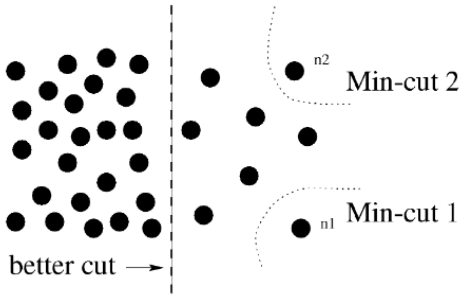


Figure 16.13: A case where minimum cut gives a bad partition.

where $\text{assoc}(A, V)$ describes how "well-connected" a vertex A is. Specifically, it's defined by the sum of the weights of all edges that touch A.

Finding the normalized cut is computationally intractible, but there's an approximate solution that is based on solving the "generalized eigenvalue problem." To briefly summarize (and for more, you can again refer to the paper) the process of finding the normalized cut:

- Let $\mathbf{W}$ be the **adjacency matrix** of the graph, so $\mathbf{W}(i, j)$ is the adjacency between pixel i and pixel j.
- Let $\mathbf{D}$ be the **degree matrix**, so that $\mathbf{D}(i, i)$ describes the number of vertices connected to the pixel i. Specifically, a diagonal matrix with entries:

$$\mathbf{D}(i, i) = \sum_j \mathbf{W}(i, j)$$

- Then, the normalized cut cost can be written as:

$$\frac{\mathbf{y}^\text{T}(\mathbf{D} - \mathbf{W})\mathbf{y}}{\mathbf{y}^\text{T}\mathbf{D}\mathbf{y}}$$

  where $\mathbf{y}$ is an **indicator vector** with 1 in the i^th position if the i^th feature point belongs to segment A and a negative constant otherwise.

As we said, though, finding $\mathbf{y}$ is intractible. Instead, we can approximate by solving the following eigenvector equation for the eigenvector of the second-smallest eigenvalue:[3]

$$(\mathbf{D} - \mathbf{W})\mathbf{y} = \lambda\mathbf{D}\mathbf{y}$$

We then use the entries of the eigenvector to bipartition the graph. Again, this is very hand-wavy stuff that doesn't go into rigorous detail; refer to the paper if you're interested in more.

16.4.1 Pros and Cons

We started off with the goal of escaping the arbitrarity of feature space; we're still dependent on it in our choice of distance function, but it no longer exists in isolation. The feature space is now

[3] This is based on spectral partitioning methods if you'd like a starting point for learning more.

intertwined with the pixels themselves, and the introduction of the affinity matrix lets us ~~partition~~ segment the ~~graph~~ image as a whole.

Because of the graph-based approach, we have a generic framework that decouples the choice of distance function and affinity matrix with the actual computation of cuts. It no longer requires a model of the pixel data distribution.

Unfortunately, many graph algorithms aren't very efficient, so the time complexity can be high. This gets worse with dense, highly-connected graphs due to the number of affinity computations. A more fundamental downside to the algorithm is that we intentionally introduced a preference for balanced partitions by the very act of normalizing; normalized cut will struggle with finding small, isolated components.

Binary Image Analysis

> *The existence of these patterns* [fractals] *challenges us to study those forms that Euclid leaves aside as being "formless," to investigate the morphology of the "amorphous." Mathematicians have disdained this challenge, however, and have increasingly chosen to flee from nature by devising theories unrelated to anything we can see or feel.*
>
> — Benoit B. Mandelbrot, *The Fractal Geometry of Nature*

Binary images are an important part of most image processing and computer vision pipelines. We've encountered them many times before: images resulting from Edge Detection (1 – edge) and Background Subtraction (1 – foreground) are just two examples of binary images we've worked with. Even a segmented image like the ones we saw in chapter 16 can be isolated into individual binary segments. In this chapter we'll cover an assortment of topics that work with these types of images.

When it comes to binary operations, some of the interesting operations we want to perform include **separation** (objects from one another), **aggregation** (of pixels for an object), and **computation** (of the features of an object). Recall our discussion of Image Moments and Hu moments (see 15.1), where we computed a value describing a particular blob in an image? That was an example of computation on a binary image.

17.1 Thresholding

The simplest operation we can perform on an image to make it binary is **thresholding**. Libraries like OpenCV support a variety of thresholding variants, but the simplest one makes all pixels above a certain threshold the maximum intensity and those below it 0:

$$\text{threshold}(x, y, t) = \begin{cases} max & \text{if } I(x, y) > t \\ 0 & \text{otherwise} \end{cases}$$

More generally, how can we determine what the appropriate t is? Often, we can reference a histogram to separate the image into 2 or more regions, but there are not always clear threshold values.

A simple automatic thresholding method is called **Otsu's method**. It relies on a fundamental assumption that the histogram is **bimodal**, meaning it has two modes and can be split into two

Figure 17.1: Thresholding an image of a kidney from a CT scan, then identifying and labeling its connected components.

groups. The method finds the threshold t that minimizes the weighted sum of within-group variances for the two groups.

17.2 Connected Components

Once you have a binary image, you can identify and subsequently analyze each connected set of pixels. Of course, you first need to determine which pixels are connected, leading us to **connected component labeling** (CCL).

There are a variety of approaches to connected component labeling of varying efficiency and connectivity.[1] The row-by-row approach is the most common CCL algorithm and has a "classical" two-pass variant as well as an efficient run-length algorithm often used in real-time industrial applications.

To summarize the algorithm briefly before diving into its details and a walk-through, we'll essentially be stepping through each valid (non-zero) pixel, checking its connectivity, then either marking it with a neighbor's label if it's connected or a new label if not. More formally, the algorithm is described in algorithm 17.1.

0	0	0	1	1	1	0	0	0	0	1	1	1	1	0	0	0	0	0	
0	0	0	1	1	1	1	0	0	0	1	1	1	1	0	0	0	0	0	
0	0	0	1	1	1	1	1	0	0	1	1	1	1	0	0	0	1	0	
0	0	0	1	1	1	1	1	1	0	1	0	1	1	0	0	1	1	1	
0	1	0	1	1	1	1	1	1	1	1	0	1	1	0	1	1	1	1	
0	1	0	1	1	1	1	1	1	1	1	1	1	1	0	1	1	1	1	
0	0	0	1	1	1	0	1	1	0	1	1	1	1	0	1	1	1	1	
0	0	0	1	1	1	0	0	1	0	0	0	0	0	0	0	0	0	0	

→

0	0	0	2	2	2	0	0	0	0	2	2	2	2	0	0	0	0	0	
0	0	0	2	2	2	2	0	0	0	2	2	2	2	0	0	0	0	0	
0	0	0	2	2	2	2	2	0	0	2	2	2	2	0	0	0	3	0	
0	0	0	2	2	2	2	2	2	0	2	0	2	2	0	0	3	3	3	
0	1	0	2	2	2	2	2	2	2	2	0	2	2	0	3	3	3	3	
0	1	0	2	2	2	2	2	2	2	2	2	2	2	0	3	3	3	3	
0	0	0	2	2	2	0	2	2	0	2	2	2	2	0	3	3	3	3	
0	0	0	2	2	2	0	0	2	0	0	0	0	0	0	0	0	0	0	

Figure 17.2: The before-and-after result of labeling connected components.

Though an animation would make things much easier, let's do our best in walking through the algorithm to get the result in Figure 17.2. In the first scanline we encounter a foreground pixel

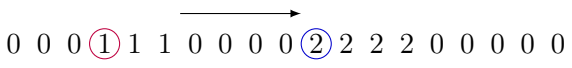
[1] 4-way connectivity implies connectivity to direct neighbors, whereas 8-way connectivity also includes diagonal neighbors. We'll be assuming and working with 4-way connectivity here, but the extension is relatively trivial.

ALGORITHM 17.1: A connected component labeling algorithm.

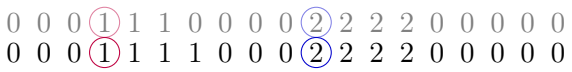
Input: A binary input image, I.
Result: A labeled image, L.

```
label ← 1
L ← an empty image                                  // the labeled output image
C ← ∅                                               // the set of conflicting labels
foreach row r ∈ I do
    if r_i = 0 then
        L[r_i] = 0
        continue
    end
    Determine whether or not r_i is 4- or 8-way connected.
    if not connected then
        L[r_i] = label
        label += 1
    else
        Let N be the set of neighbor labels, if any.
        k = min N
        L[r_i] = k
        foreach n ∈ N do                            // add conflicting labels
            C = C ∪ (k, n)
        end
    end
end
Merge together the conflicting labels.
return L
```

at (0, 3), marking it and its subsequent neighbors with *label* = 1. When we encounter the next foreground pixel, we label it with the next *label* = 2.



Upon reaching the second row, we see a connection occurring at $(3, 1)$ and again at $(10, 1)$, so no new labels are marked.



In the 3rd row, though, we see a new unconnected component appear, so we give it the *label* = 3.

0 0 0 1 1 1 0 0 0 0 2 2 2 2 0 0 0 0 0
0 0 0 1 1 1 1 0 0 0 2 2 2 2 0 0 0 0 0
0 0 0 1 1 1 1 1 0 0 2 2 2 2 0 0 0 3 0

This continues for a bit until we reach the first conflict in row 5, at pixel $(10, 4)$. Its neighbors are $label = 1$ to its left and $label = 2$ above. We set it to the lesser of the two and add the pair $(2, 1)$ to the set of conflicts.

0 0 0 1 1 1 0 0 0 0 2 2 2 2 0 0 0 0 0
0 0 0 1 1 1 1 0 0 0 2 2 2 2 0 0 0 0 0
0 0 0 1 1 1 1 1 0 0 2 2 2 2 0 0 0 3 0
0 0 0 1 1 1 1 1 1 0 2 0 2 2 0 0 3 3 3
0 4 0 1 1 1 1 1 1 1 ? ? ? ? ? ? ? ? ?

At the end of the first pass through the image (pre-conflict resolution), our image would be labeled as follows:

0 0 0 1 1 1 0 0 0 0 2 2 2 2 0 0 0 0 0
0 0 0 1 1 1 1 0 0 0 2 2 2 2 0 0 0 0 0
0 0 0 1 1 1 1 1 0 0 2 2 2 2 0 0 0 3 0
0 0 0 1 1 1 1 1 1 0 2 0 2 2 0 0 3 3 3
0 4 0 1 1 1 1 1 1 1 1 0 2 2 0 3 3 3 3
0 4 0 1 1 1 1 1 1 1 1 1 1 1 0 3 3 3 3
0 0 0 1 1 1 0 1 1 0 1 1 1 1 0 3 3 3 3
0 0 0 1 1 1 0 0 1 0 0 0 0 0 0 0 0 0 0

A subsequent conflict resolution pass would merge $2 \rightarrow 1$ and render the same result (albeit without normalized label numbers) as in Figure 17.2.

17.3 Morphology

There is another set of mathematical operations we can perform on a binary image that are useful for "fixing it up." The two fundamental morphological operations are **dilation** and **erosion**.

17.3.1 Dilation

Dilation *expands* the connected set of 1s in a binary image. It can be useful for growing features as well as filling holes and gaps.

17.3.2 Erosion

On the other side of the morphological spectrum is **erosion**, which *shrinks* the connected sets of 1s in a binary image. It can be used for shrinking features as well as removing bridges, branches, or protrusions in various misshapen binary blobs.

17.3.3 Structuring Element

The structuring element is a shape mask used in basic morphological operations. It has a well-defined origin (which doesn't necessarily have to be at the center) and can be any shape or size that is digitally representable. For example, the following is a hexagonal structuring element with an origin at the center:

$$\begin{bmatrix} 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & \mathbf{1} & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \end{bmatrix}$$

To do a dilation operation, we move the structuring element S over the binary image B, placing the origin at each pixel. Considering only the 1-pixel locations in S, we compute the binary `OR` of corresponding elements in B. If there are any non-zero pixels in the resulting window, we place a 1 in the morphed image. For example, given the following S and B (where the red 1 is the origin of S):

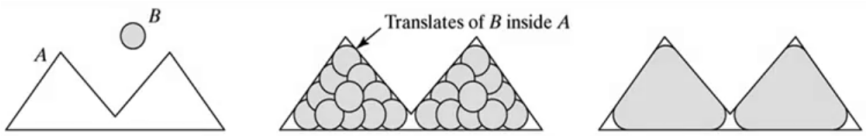
$$\underbrace{\begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}}_{B} \quad \underbrace{\begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix}}_{S} \quad \xrightarrow{\text{dilate}} \quad \underbrace{\begin{bmatrix} 0 & 0 & 0 & 0 \\ 1 & \mathbf{1} & \mathbf{1} & 0 \\ 0 & 1 & 1 & 0 \end{bmatrix}}_{B \oplus S}$$

To do an erosion operation, we instead consider the binary `AND` of the correponding elements in B. In other words, the entirety of the masked area in B should have a 1 where S has a 1 to be preserved. For example:

$$\underbrace{\begin{bmatrix} 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 & 1 \end{bmatrix}}_{B} \quad \underbrace{\begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}}_{S} \quad \xrightarrow{\text{erode}} \quad \underbrace{\begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}}_{B \ominus S}$$

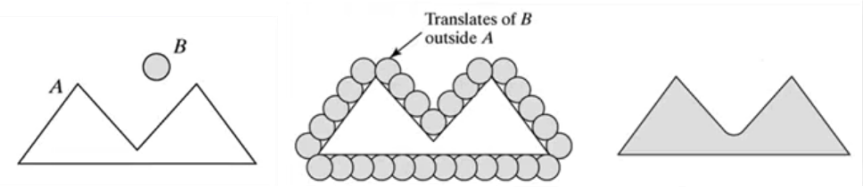
17.3.4 Opening and Closing

We can combine these two fundamental morphological operators into more useful operations. **Opening** is a compound operation of erosion followed by dilation with the same structuring element. It can be thought of formally as the area of translations of the structuring element that fit entirely within the shape, or intuitively as the area in a blob in which a structuring element "brush" can paint within the lines:



Notice that if we were to perform opening on the resulting gray blobs above, nothing would change: the structuring element already fits entirely within the shape. Morphological opening is **idempotent**, meaning repeated operations have no further effects.

On the other side of the coin, we having **closing**, which is dilation followed by erosion with the same structuring element. Formally, it can be shown that closing of A by B is the complement of the union of all translations of B that do not overlap A (lol what?). More intuitively, closing is the area that we can **not** paint with the brush B when we're not allowed to paint inside of A:



Closing is likewise idempotent. It can be used for simple segmentation because it effectively fills in small holes in the image.

Figure 17.3: Simple segmentation of an image by morphologically closing (with a disk of size 20) the thresholded binary image.

Opening typically breaks narrow connections, smooths out or eliminates small peaks, and removes small noisy "islands." Closing smooths contours at concavities, fuses narrow breaks, and fill in small gulfs. We can combine these just like we combined dilation and erosion.

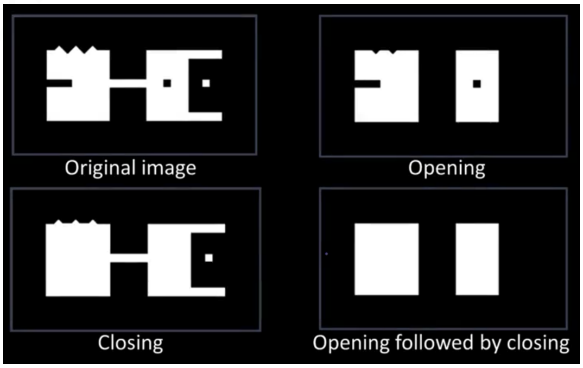


Figure 17.4: A demonstration of combining opening and closing operations.

An opening followed by a closing often gives the "underlying elements," or some very rough discretized approximation of the basic objects. You can think of it as squinting really, *really* hard so that all sense of image fidelity is gone. In Figure 17.4, we can see that the two fundamental elements of the image were two rectangles.

Though we've been looking at fairly unrealistic examples of binary blobs as to demonstrate these concepts, this stuff really does have real-world applications:

By opening and subsequently closing the fingerprint image, we can remove noise and solidify the ridges to create a much cleaner image.

> There is another morphological operation called the hit-or-miss transform we won't cover, but it's useful for things like shape detection and pruning of unwanted patterns.

17.3.5 Basic Morphological Algorithms

There are a number of things we can actually *do* with the aforementioned handful of operations. Some of these include:

- boundary extraction
- region filling
- connected component extraction
- convex hulls
- thinning
- skeletons
- pruning

We'll only talk a little more about boundary extraction. Notation-wise, let $A \ominus B$ denote the erosion of A by B. The boundary of A, then, can be computed as:

$$A - (A \ominus B)$$

where B is a 3×3 structuring element (or some other size, depending on the size of the boundary you want). In other words, we subtract from A an erosion of it to obtain its boundary.

17.3.6 Effectiveness

How powerful or useful is morphology, really? As with all things in life... it depends. If the binary images you start with are fairly "clean," meaning your true inputs behave similarly to the sample inputs you used when designing the system, you can actually do some very powerful stuff to both clean up images and to detect variations from desired images.

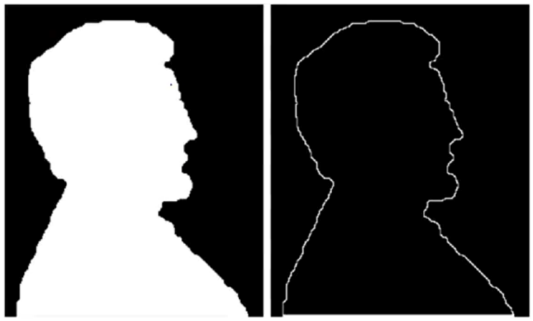
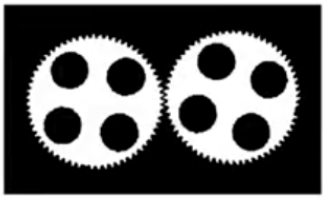


Figure 17.5: Performing boundary extraction on a binary image.

Morphology is used a lot in "industrial vision," meaning vision on assembly lines and the like for quality inspection. One such example is finding the defects in some gears:



Can you identify if there are any defects or faults in the gears? How about in this one:

Much easier! The latter image is a product of a series of morphological operations to extract the parts of the gears that don't have teeth.

Depth and 3D Sensing

This chapter introduces a different 3rd dimension to our computer vision pipeline: **depth**. Previously, we've been treating the temporal domain as our 3rd dimension, but now we will introduce new image sensors that allow us to incorporate approximate (to varying degrees of accuracy) depth into our systems. We've obviously discussed depth before when talking about Stereo Correspondence at length, but now we will introduce both active and passive sensors whose sole purpose is to determine depth.

Our motivations are varied. We can use depth to determine the shape and physical 3D structure of an object as well as determine where the object(s) and background(s) begin. We can further use it to map the environment and find obstacles, a task particularly useful in robotics. Tracking, dynamics, and pretty much everything we've discussed up until now can be improved and processed much more accurately with reliable depth information.

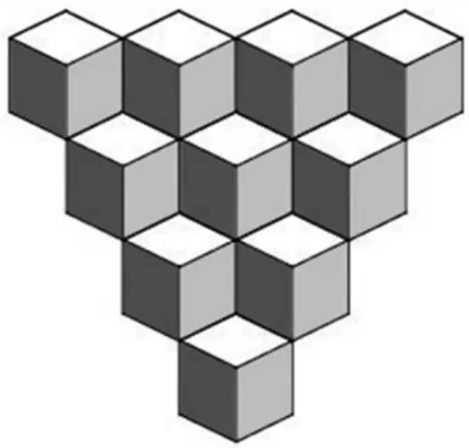


Figure 18.1: It's impossible to know whether or not there is actual 3D structure, or if this is simply a neat flat texture.

A lack of "true" depth information (which is an inevitable product of projecting a 3D scene onto a 2D surface; recall Figure 8.1) introduces geometric ambiguities like the ones in Figure 18.1, Figure 18.2, and our old friend Figure 7.8.

Figure 18.2: Images alone are often insufficient in determining scene geometry. Similarities in foreground and background texture, a lack of texture entirely, and color constancy all introduce difficulties in "understanding" an image.

18.1 3D Sensing

We'll open up our discussion of depth with the hardware. There are two types of 3D sensing: **passive** and **active**.

18.1.1 Passive Sensing

Passive 3D sensing works with naturally occurring light (i.e. taking the world as its given to you) and exploits known scene geometry or properties.

(a) A stereo camera setup, enabling depth approximation much like the binocular fusion of the human vision system.

(b) The differences in focus among objects hints at depth information.

Figure 18.3: A variety of passive 3D sensing methods.

A familiar example of a passive sensing model is the one used to produce the images we used in Stereo Correspondence. Another one that is less familiar but perhaps more intuitive is depth sensing via focal pull. As the focal plane changes, so does the fidelity of various objects in the scene. We can use this to approximate relative distances by continually changing the focus.

18.1.2 Active Sensing

In contrast with passive sensing, active sensing works by interacting directly with its environment. Well-known examples of active sensing are known as **time-of-flight sensors**. These in-

clude SONAR—emitting sound waves or "chirps"—and LIDAR—emitting lasers—which function by recording the time it takes for the impulses to echo via the simple formula $d = v \cdot t/2$.

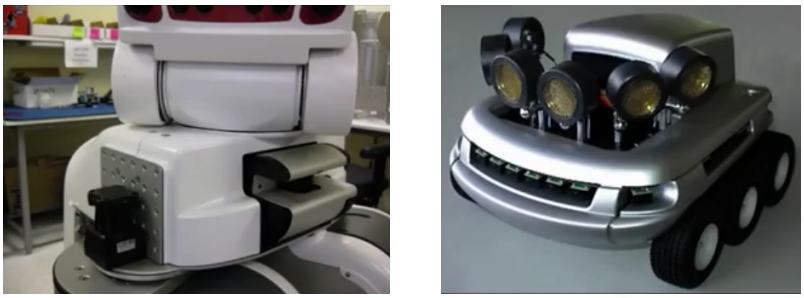


Figure 18.4: Both LIDAR and SONAR are forms of active 3D sensing.

A more fundamentally "computer vision-esque" method of active depth sensing uses **structured light**, also called **light striping**. We begin with a setup with coplanar image planes much like in stereo as in Figure 7.10, but with a projector instead of a second camera. The projector sends out a particular color of light for a particular band of pixels. This is picked up by the camera in the scene geometry, and the correspondence between the colors gives depth insights. The curvature of each band changes based on the depth in the scene.

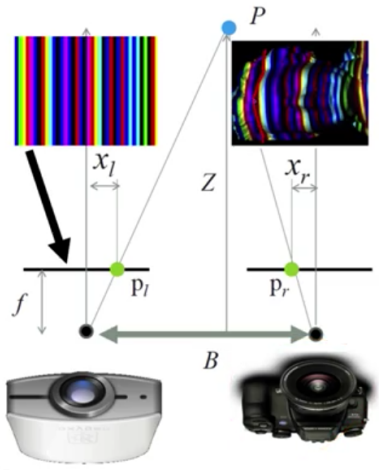


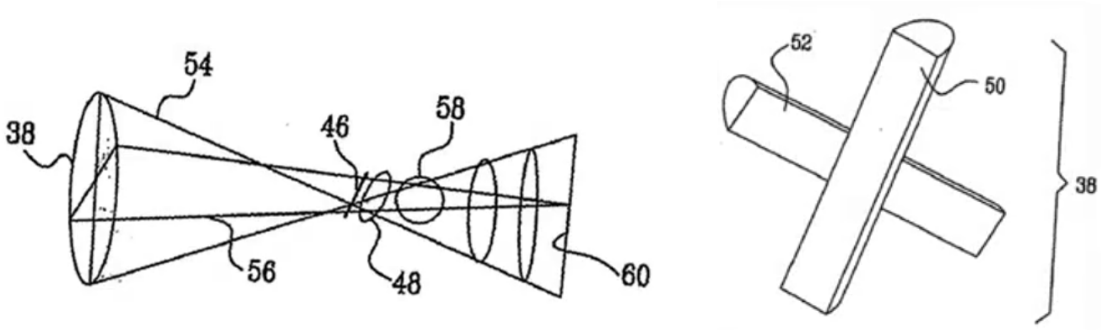
Figure 18.5: A visual explanation of the structured light active sensing method.

A recent explosion in this arena has been the use of **infrared structured light**—largely due to the release and subsequent popularity of the Microsoft Kinect—which is an obvious improvement over using colored bands because infrared is imperceptible to the human eye.

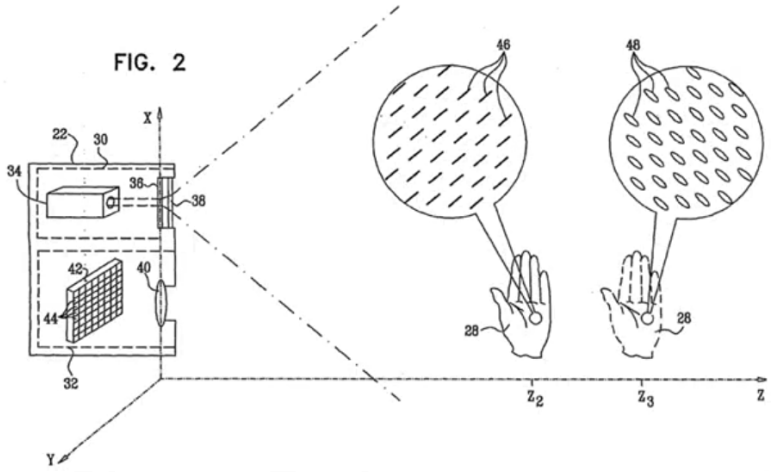
Though the precise methods that the Kinect uses to sense depth are not public information, we have solid insights from patent contents and reverse engineering. There are two main (surmised) methods: a clever optics trick and the aforementioned light striping method.

The optic trick is so clever that even Prof. Bobick can't explain it in full detail. We start with a cylindrical lens: it only focuses light in one direction.[1] Given two of these, configured at a particular orientation, we end up with a situation that results in different focal lengths that depend on which way you move.

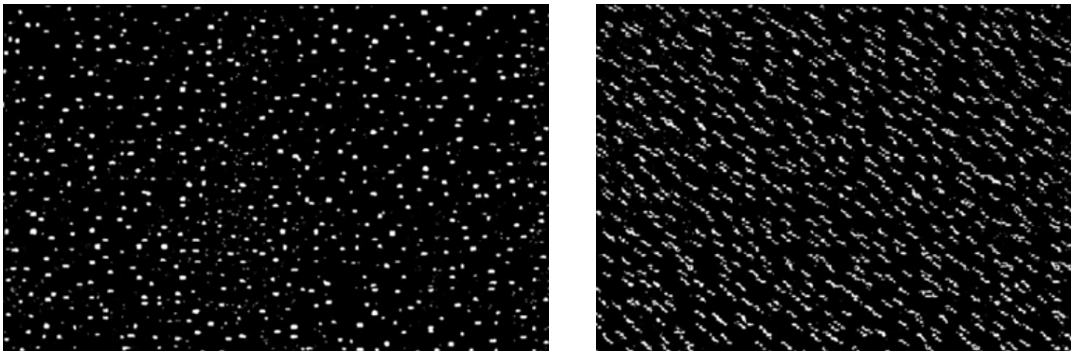
[1] These are called astomatic (sp?) or isometric lenses.



You have a particular focal length horizontally and another vertically (and, obviously, particular changes as you rotate around). Remember a *looong* time ago when we surmised about what an out-of-focus point of light would look like? We concluded that it would be a blurry circle. Well, under these lens conditions, it would actually be an ellipse. Not only that, it would additionally be an *rotated* ellipse whose orientation depends on the depth.



This means the projection of some psueodrandom speckle pattern would result in speckles that are rotated and scaled based on the depth of the scene geometry. Your resulting image, then, maps **orientation as a function of distance**! This is an incredibly clever construction.



It's unlikely that this is used (exclusively, at the very least) in the Kinect because it requires some incredibly precise lenses for reliable results. The simpler method is the one we described above, using infrared light striping. The algorithms are the same as in stereo: the epipolar constraint reduces the search space for a particular corresponding pixel to a line in the image.

TODO: Describe the algorithm.

Index of Terms

F

G

H

I

S

T

V

W

Y