# Fundamentals of Computer Vision

George Kudrayvtsev
george.ok@pm.me

Last Updated: May 23, 2020

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 and Dr. Irfan Essa, as well as on numerous published papers in the field that are linked where appropriate.

The ultimate goal of this guide is an accessible, searchable introduction to foundational concepts in computer vision with cross-references and additional explanations to minimize assumed knowledge. Though deep learning and convolutional neural networks are ommitted from the discussed topics, many of the algorithms and ideas presented are still widely used today and form the backbone of challenges in computational perception that don't involve classification.

This is a sample of the full guide that shows the full table of contents; it includes the introductory chapters, the chapter on Tracking, and a bonus appendix on the Human Vision System.

# 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, it has recently been expanded to include concepts from *Computational Photography*. There is a lot of overlap between the two.

In either case, it is intentionally structured to follow along with the course lectures. Thus, **if you're here for a standalone introduction** rather than a companion guide to the coursework, this section is irrelevant.

### Computer Vision

If you are here for computer vision, the "reading list" is pretty straightforward: you can (almost) just read this guide from front to back. You can skip over chapter 6, but I recommend reading about Image Pyramids because they will come up later in the chapters on Tracking and Motion. You can also skip the discussion of High Dynamic Range images and the beginning of the chapter on Video Analysis.

### Computational Photography

If you are here for computational photography, the content is a little scattered across chapters. 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];
- refer to High Dynamic Range for a discussion on that, and PhotoSynth is briefly mentioned in Figure 17.4, but a full discussion is forthcoming;
- you can return to Stereo Geometry for a discussion of stereo, and refer to chapter 17 for a brief discussion of structured light and structure from motion (among other, more CV-oriented topics), and camera calibration immediately follows stereo; and
- the beginning of chapter 14 covers video textures.

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

#### Robotics

You may have stumbled upon this guide when studying ideas in robotics. Though I recommend starting with my brief notes on robotics, the chapter on Tracking covers both Kalman filters and particle filters which are fundamental algorithms in robotics.

## 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.
- An item in a maroon box, like...

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

**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.

I also am starting to include margin notes like the one here (which just links to my homepage) that link to the source of the content so that you can easily explore the concepts further. A lot of it will be Udacity content; if you don't have access to that, unfortunately you'll have to stick with these notes. The other times, though, will be links to the original papers being described; keep an eye out for them!

Linky

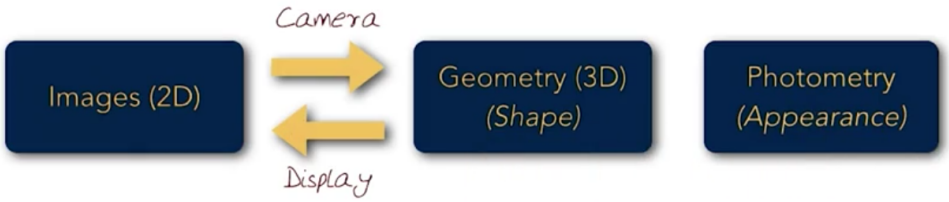
# Introduction

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

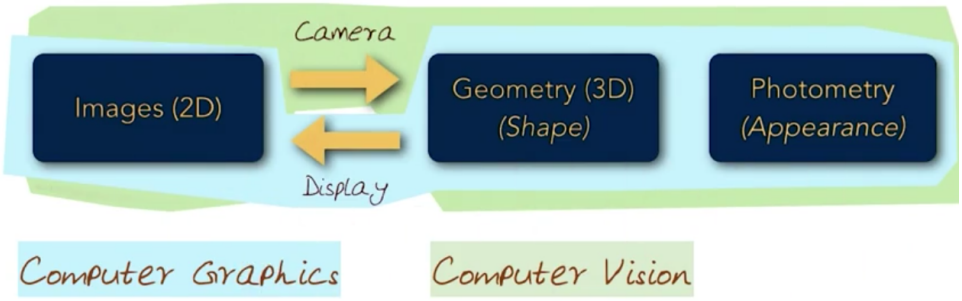
This set of notes is an effort to unify the curriculum across both computer vision and computational photography, cross-pollinating unique ideas from each discipline and consolidating concepts that are shared between the two. Before we begin, it's important to talk about their differences.

## 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:

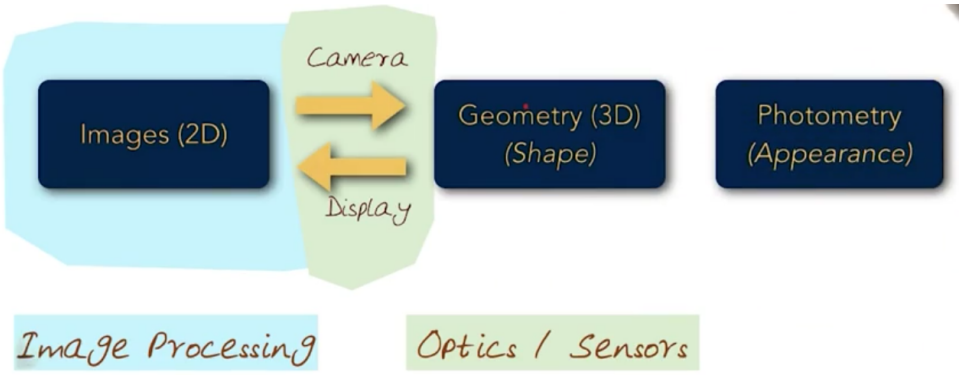



Computer vision is concerned with all of these except *display*. It is the discipline that focuses on looking at 2D images from the real world and inferring information from it, which includes things like geometry as well as a more "high-level" understanding of things like recognizing objects.




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:

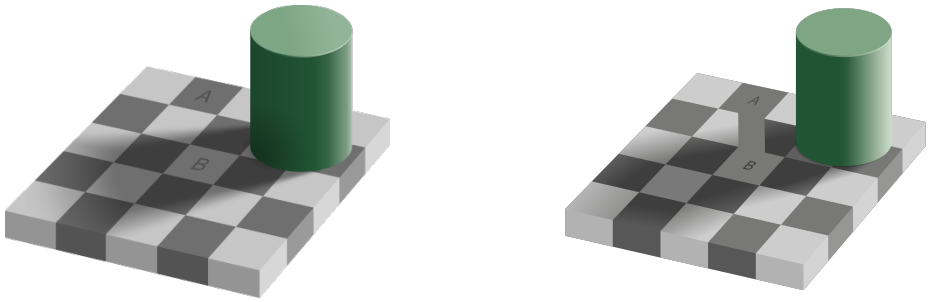



**Figure 1.1:** The checker shadow illusion.

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:

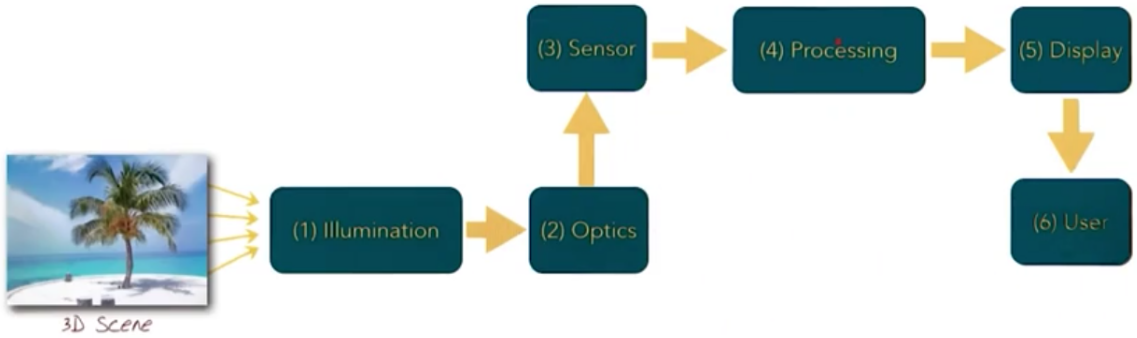
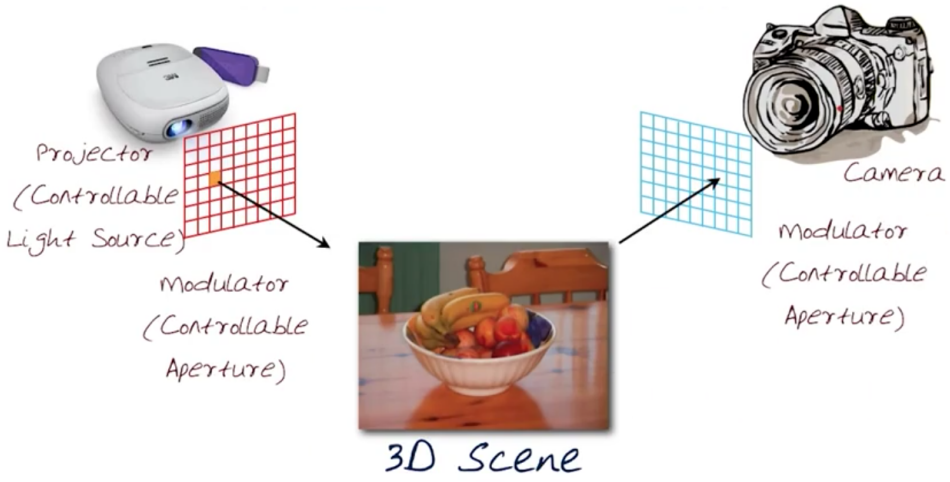



**Figure 1.2:** The photography pipeline.

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:

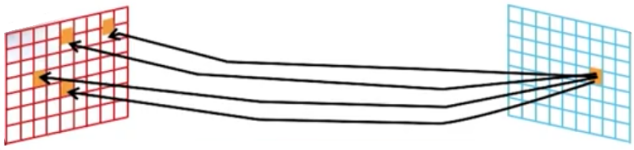



By treating a projector as a controllable light source, we can have it pass through a *modulator* acting as a controllable aperture. We can control which cells in the modulator are open at any given point in time, giving us tight control of the custom lighting we add to the scene. Similarly, we can control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

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

in our setup, allowing us to control exactly what the camera captures from the scene.

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).

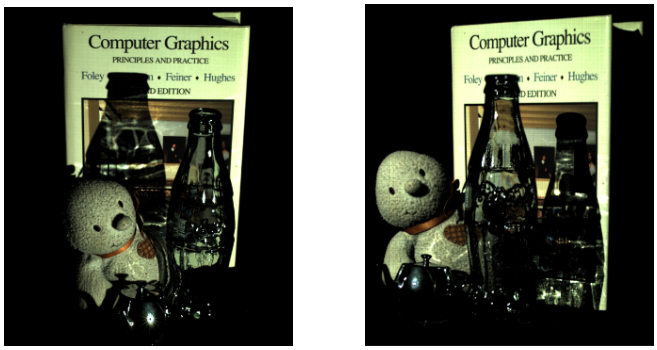



**Figure 1.3:** On the left is the "primal" image captured by the camera under full illumination of the scene by the projector. On the right is a *recreation* of the scene from a number of controlled illuminations from the perspective of the projector.

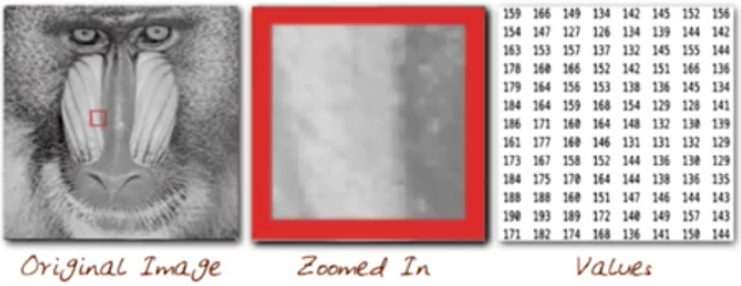
This is an excellent example of how we can use computing to create a novel image from a scenario that would otherwise be impossible to do with traditional film.

[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.

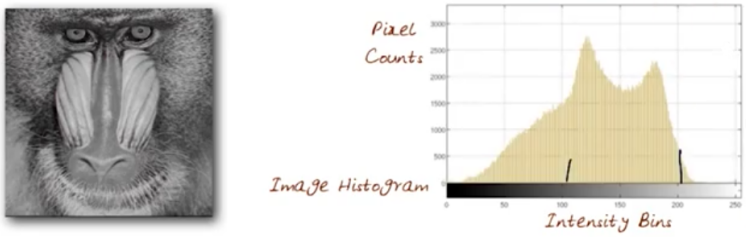
# Basic Image Manipulation

> *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:

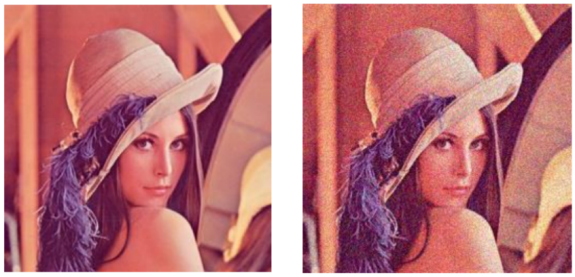
$$I_a - I_b = -(I_b - I_a)$$

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.



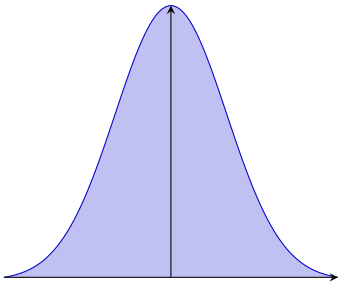


**Figure 2.1:** A Gaussian (normal) distribution centered at a mean, $\mu$.

**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 $\sigma$ – the standard deviation – this will affect the amount of noise: a higher $\sigma$ means a noisier image. Of course, when working with image manipulation libraries, the choice of $\sigma$ 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

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

"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 $[195, 255]$!

### 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 $100 \times 100$ image, you may want to take an average over a moving $5 \times 5$ square. So, disregarding edge values, the value of the pixel at $(2, 2)$ would be:

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

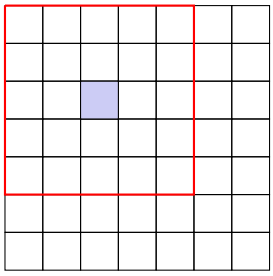


**Figure 2.2:** A $5 \times 5$ averaging window applied to the pixel at $(2, 2)$.

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 $G = H \otimes F$.

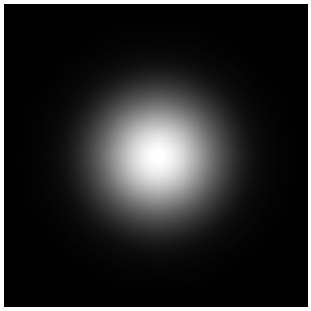
**Handling Borders** Notice that we conveniently placed the red averaging window in Figure 2.2 so that it fell completely within the bounds of the image. What would we do along the top row, for example? We'll discuss this in a little bit, in Boundary Issues.

**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

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** $\sigma$, 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."

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

**(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 $F \otimes H$.

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.

### 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]

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

- **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.

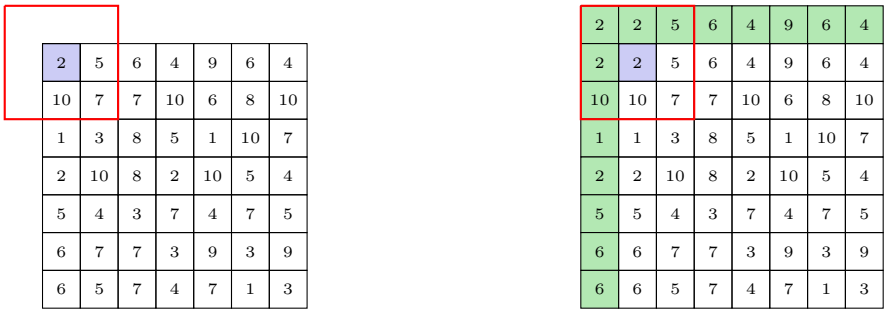



**Figure 2.4:** Padding part of an image to accomodate a $2 \times 2$ kernel by extending its border.

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.

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

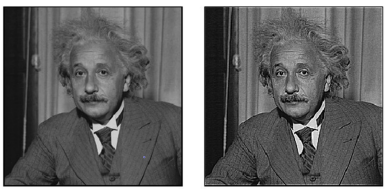


**Figure 2.5:** A sharpening filter applied to an image of Einstein.

- **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.

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

For example, consider a kernel with intensities as follows:

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

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.

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

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 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*!

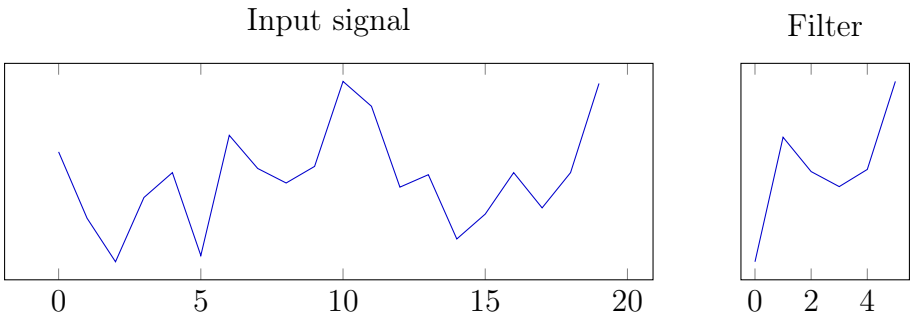


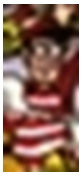

**Figure 2.7:** A signal and a filter which is part of the signal (specifically, from $x \in [5, 10]$).

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 the "solution" image of Waldo from it, like so:



If we perform a correlation between this *template* image as our filter and the image it came from, we will get a correlation map whose maximum tells us where Waldo is!

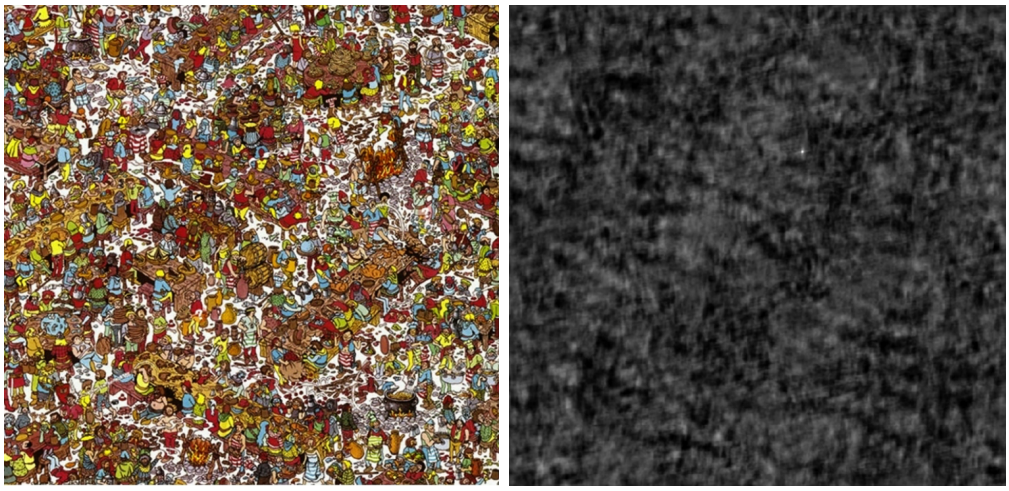


**Figure 2.8:** The original image (left) and the correlation map between it and the template filter from above, with brightness corresponding to similarity.

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

### 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.

# 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?

Shi & Tomasi '94

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. 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.

> 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.

### 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 $Z_t$.

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.

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.

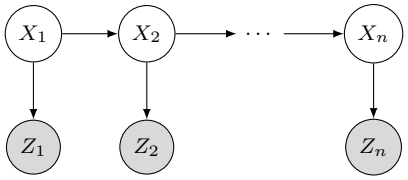



**Figure 12.1:** A graphical model for our assumptions.

### 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: $\Pr[X_{t-1} \mid z_0, \ldots, z_{t-1}]$
Guess: $\Pr[X_t \mid z_0, \ldots, z_{t-1}]$

To solve that, we can apply the **law of total probability** and **marginalization** if we imagine we're working with the joint set $X_t \cap X_{t-1}$.[1] Then:

$$= \int \Pr[X_t, X_{t-1} \mid z_0, \ldots, z_{t-1}] \; dX_{t-1}$$

$$= \int \Pr[X_t \mid X_{t-1}, z_0, \ldots, z_{t-1}] \Pr[X_{t-1} \mid z_0, \ldots, z_{t-1}] \; dX_{t-1}$$

$$= \int \Pr[X_t \mid X_{t-1}] \Pr[X_{t-1} \mid z_0, \ldots, z_{t-1}] \; dX_{t-1}$$

independence assumption from the **dynamics** model

[1] 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 $X_i$s that came before it), so we can apply the law, using the integral instead of the sum.

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 $\Pr[X_t \mid z_0, \ldots, z_{t-1}]$ and the current observation $z_t$, we want to compute $\Pr[X_t \mid z_0, \ldots, z_{t-1}, z_t]$, essentially folding in the new measurement:[2]

$$\begin{aligned}
\Pr[X_t \mid z_0, \ldots, z_{t-1}, z_t] &= \frac{\Pr[z_t \mid X_t, z_0, \ldots, z_{t-1}] \cdot \Pr[X_t \mid z_0, \ldots, z_{t-1}]}{\Pr[z_t \mid z_0, \ldots, z_{t-1}]} \\
&= \frac{\Pr[z_t \mid X_t] \cdot \Pr[X_t \mid z_0, \ldots, z_{t-1}]}{\Pr[z_t \mid z_0, \ldots, z_{t-1}]} && \text{independence assumption from the } \textbf{observation} \text{ model} \\
&= \frac{\Pr[z_t \mid X_t] \cdot \Pr[X_t \mid z_0, \ldots, z_{t-1}]}{\int \Pr[z_t \mid X_t] \Pr[X_t \mid z_0, \ldots, z_{t-1}] \; dX_t} && \text{conditioning on } X_t
\end{aligned} \tag{12.1}$$

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.

### 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.[3]

### 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

[2] We are applying Bayes' rule here, which states that $\Pr[A \mid B] = \frac{\Pr[B \mid A] \cdot \Pr[A]}{\Pr[B]}$.

[3] 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."

$\mathbf{v}$: $\mathbf{p}_1 = \mathbf{p}_0 + \mathbf{v}$. 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 $\Sigma_{d_t}$ which represents uncertainty in how "linear" our model truly is.[4] Mathematically, we have our normal distribution function $N$,[5] and so

$$\mathbf{x}_t \sim \mathcal{N}(\underbrace{\mathbf{D}_t\mathbf{x}_{t-1}}_{\text{mean}}, \underbrace{\Sigma_{d_t}}_{\text{variance}})$$

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:

$$\mathbf{D}\mathbf{x}_t = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} p_x & p_y \\ v_x & v_y \end{bmatrix}_t = \begin{bmatrix} p_x + v_x & p_y + v_y \\ v_x & v_y \end{bmatrix}$$

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:

$$\mathbf{D} = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} \mapsto \begin{cases} \mathbf{p}' = 1 \cdot \mathbf{p} + 1 \cdot \mathbf{v} \\ \mathbf{v}' = 0 \cdot \mathbf{p} + 1 \cdot \mathbf{v} \end{cases}$$

[4] This is "capital sigma," not a summation. It symbolizes the variance (std. dev squared): $\Sigma = \sigma^2$.

[5] $\mathcal{N}(\mu, \sigma^2)$ is a compact way to specify a multivariate Gaussian (since our state can have $n$ variables). The notation $x \sim \mathcal{N}(\mu, \sigma^2)$ means $x$ is "distributed as" or "sampled from" a Gaussian centered at $\mu$ with variance $\sigma^2$.

**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:

$$\mathbf{z}_t \sim \mathcal{N}(\mathbf{M}_t\mathbf{x}_t, \Sigma_{m_t})$$

$\mathbf{M}$ 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 $\mu_t^-$ and $\sigma_t^-$. 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 $\mu_t^+$ and $\sigma_t^+$.

Again, at time $t$,

- $\mu_t^-, \sigma_t^-$ — state mean and variance after *only* prediction.
- $\mu_t^+, \sigma_t^+$ — 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:

$$\begin{aligned} &\text{Update the means:} && \mu_t^- = d\mu_{t-1}^- \\ &\text{Update the variance:} && \left(\sigma_t^-\right)^2 = \sigma_d^2\left(d\sigma_{t-1}^+\right)^2 \end{aligned}$$

### 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:

$$\mu_t^+ = \frac{\mu_t^- \sigma_m^2 + m z_t \left(\sigma_t^-\right)^2}{\sigma_m^2 + m^2\left(\sigma_t^-\right)^2}$$

Update the variance:

$$\left(\sigma_t^+\right)^2 = \frac{\sigma_m^2\left(\sigma_t^-\right)^2}{\sigma_m^2 + m^2\left(\sigma_t^-\right)^2}$$

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, $\mu_t^+$, really is. Let's first divide the entire thing by $m^2$ to "unsimplify." We get this mess:

$$\mu_t^+ = \frac{\frac{\mu_t^- \sigma_m^2}{m^2} + \frac{z_t}{m}\left(\sigma_t^-\right)^2}{\frac{\sigma_m^2}{m^2} + \left(\sigma_t^-\right)^2} \tag{12.3}$$

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:

$$\mu_t^+ = \frac{a\mu_t^- + b\frac{z_t}{m}}{a+b}$$

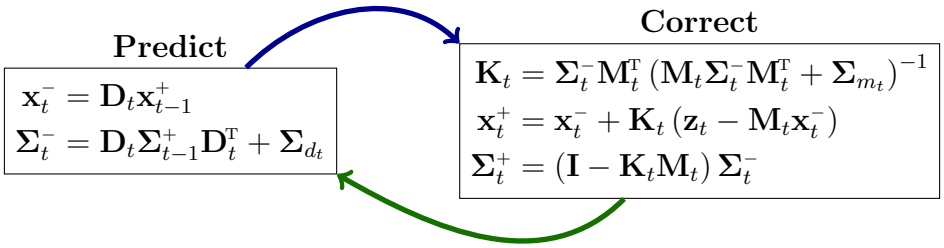
We can do some manipulation (add $b\mu_t^- - b\mu_t^-$ to the top and factor) to get:

$$\begin{aligned} &= \frac{(a+b)\mu_t^- + b\left(\frac{z_t}{m} - \mu_t^-\right)}{a+b} \\ &= \mu_t^- + \frac{b}{a+b}\left(\frac{z_t}{m} - \mu_t^-\right) \\ \mu_t^+ &= \mu_t^- + k\left(\frac{z_t}{m} - \mu_t^-\right) \end{aligned}$$

Where $k$ is known as the **Kalman gain**. What does this expression tell us? Well, the new mean $\mu_t^+$ 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 a lot to take in...

Kalman '60
Udacity '12a
Udacity '12b

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):

$$\mathbf{K}_t = \underbrace{\mathbf{\Sigma}_t^- \mathbf{M}_t^{\mathrm{T}}}_{n \times m} \underbrace{\left( \underbrace{\mathbf{M}_t \mathbf{\Sigma}_t^- \mathbf{M}_t^{\mathrm{T}}}_{m \times m} + \mathbf{\Sigma}_{m_t} \right)^{-1}}_{n \times m}$$

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.

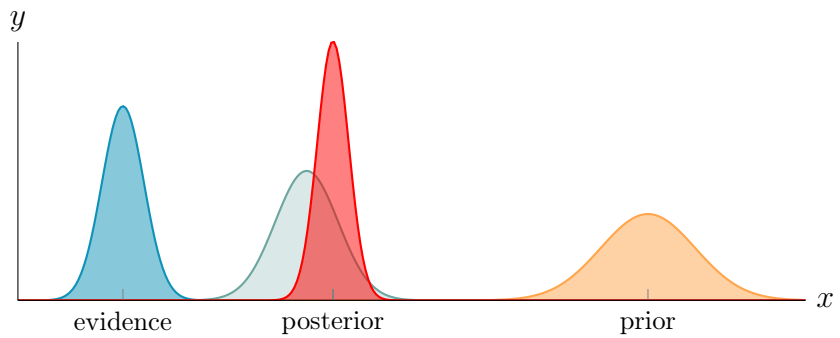



**Figure 12.2:** One of the flaws of the Kalman model is that it is always more confident in its distribution, resulting in a tighter Gaussian. In this figure, the red Gaussian is what the Kalman filter calculates, whereas the blue-green Gaussian may be a more accurate representation of our intuitive 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?

## 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:

$$\mathbf{x}_{t+1} \sim \mathcal{N}(\mathbf{D}\mathbf{x}_t, \mathbf{\Sigma}_t) + \mathbf{u}$$

If the movement has its own uncertainty, it's a sum of Gaussians:

$$\mathbf{x}_{t+1} \sim \mathcal{N}(\mathbf{D}\mathbf{x}_t, \mathbf{\Sigma}_t) + \mathcal{N}(\mathbf{u}, \mathbf{\Sigma}_u)$$

### 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:

$$\Pr[x_t \,|\, u_t, x_{t-1}]$$

  **Note:** We (*always*, 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: $\Pr[z \,|\, X]$. 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**:

$$Bel(x_t) = \Pr[x_t \,|\, u_1, z_2, \ldots, u_t, z_t]$$

The assumptions in our probabilistic model is represented graphically in Figure 12.3, and result in the following simplifications:[6]

$$\Pr[z_t \,|\, X_{0:t}, z_{1:t-1}, u_{1:t}] = \Pr[z_t \,|\, x_t]$$
$$\Pr[x_t \,|\, X_{1:t-1}, z_{1:t-1}, u_{1:t}] = \Pr[x_t \,|\, x_{t-1}, u_t]$$

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.

As a reminder, Bayes' rule (described more in footnote 2) can also be viewed as a proportionality ($\eta$ is the normalization factor that ensures the probabilities sum to one):

$$\Pr[x \,|\, z] = \eta \Pr[z \,|\, x] \Pr[x] \tag{12.4}$$

[6] The notation $n_{a:b}$ represents a range; it's shorthand for $n_a, n_{a+1}, \ldots, n_b$.

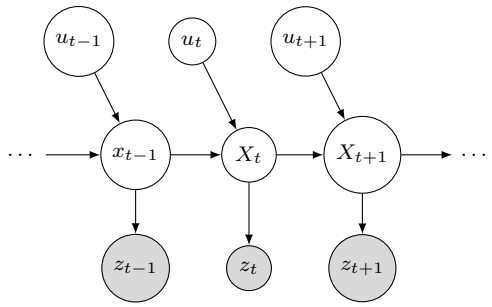



**Figure 12.3:** A graphical model for Bayes filters.

$$\propto \Pr[z \mid x] \Pr[x] \tag{12.5}$$

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:

$$\begin{aligned}
Bel(x_t) &= \Pr[x_t \mid u_1, z_2, \ldots, u_t, z_t] \\
&= \eta \Pr[z_t \mid x_t, u_1, z_2, \ldots, u_t] \Pr[x_t \mid u_1, z_2, \ldots, u_t] && \text{Bayes' Rule} \\
&= \eta \Pr[z_t \mid x_t] \Pr[x_t \mid u_1, z_2, \ldots, u_t] && \text{sensor independence} \\
&= \eta \Pr[z_t \mid x_t] \int \Pr[x_t \mid x_{t-1}, u_1, z_2, \ldots, u_t] \cdot \Pr[x_{t-1} \mid u_1, z_2, \ldots, u_t] \; dx_{t-1} \\
& && \text{total probability} \\
&= \eta \Pr[z_t \mid x_t] \int \Pr[x_t \mid x_{t-1}, u_t] \cdot \Pr[x_{t-1} \mid u_1, z_2, \ldots, u_t] \; dx_{t-1} && \text{Markovian assumption} \\
&= \eta \Pr[z_t \mid x_t] \underbrace{\int \Pr[x_t \mid x_{t-1}, u_t] \cdot Bel(x_{t-1}) \; dx_{t-1}}_{\text{predictions } \textit{before} \text{ measurement}} && \text{substitution}
\end{aligned}$$

This results in our final, beautiful recursive relationship between the previous belief and the next belief based on the sensor likelihood:

$$Bel(x_t) = \eta \Pr[z_t \mid x_t] \int \Pr[x_t \mid x_{t-1}, u_t] \cdot Bel(x_{t-1}) \; dx_{t-1} \tag{12.6}$$

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.

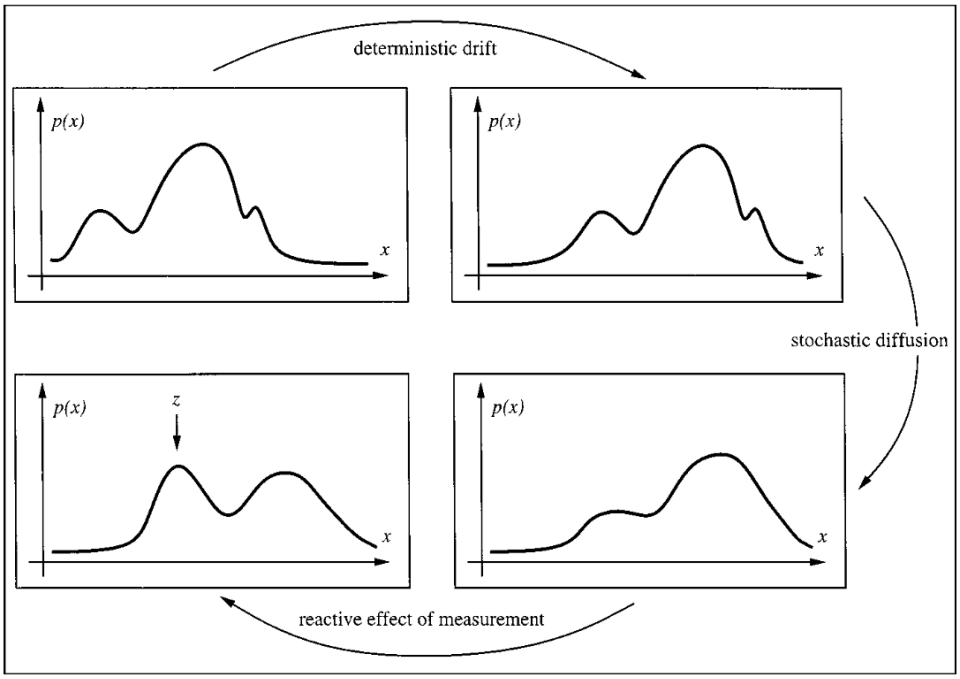



**Figure 12.4:** Propogation of a probability density in three phases: drift due to the deterministic object dynamics, diffusion due to noise, and reactive reinforcement due to observations.

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

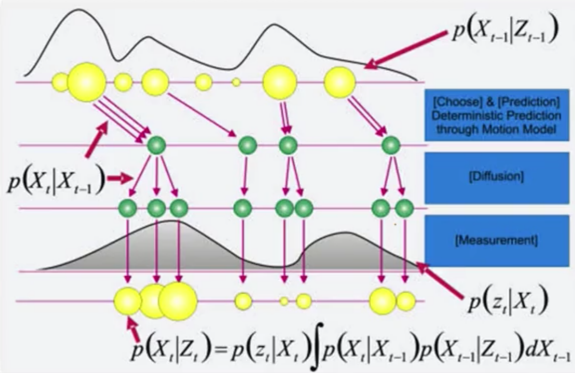



**Figure 12.5:** A graphic representation of particle filtering.

set of new particles, $S_t$.

---

**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 */`
  $x_{i,t} = \texttt{Sample}(\Pr[x_t \mid u_t, x_{j,t-1}])$ `/* sample after incorporating input */`
  $w_{i,t} = \Pr[z_t \mid x_{i,t}]$ `/* 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** $S_t$

---

### 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.

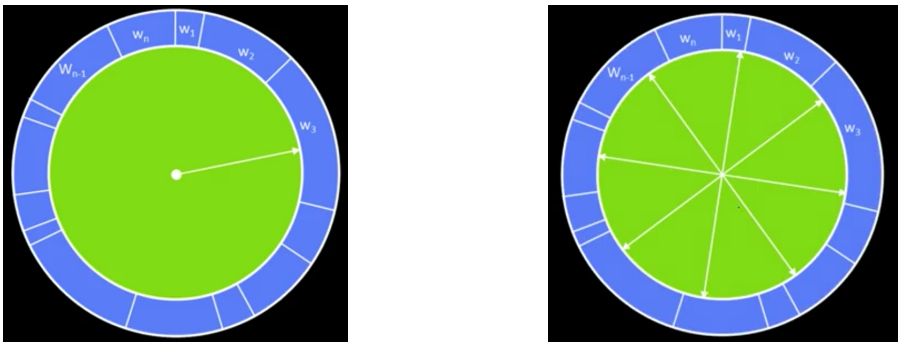
#### Sampling Method

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 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 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 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 $X$?

### 12.4.1 Tracking Contours

Shi & Tomasi '94

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: $\theta$), but it also can change shape. Using principal component analysis, a topic covered soon in chapter 13, 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 A.3a. Specifically,

$$\Pr[\mathbf{z}|X] \propto \exp\left(-\frac{(d \text{ to edge})^2}{2\sigma^2}\right)$$

...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.

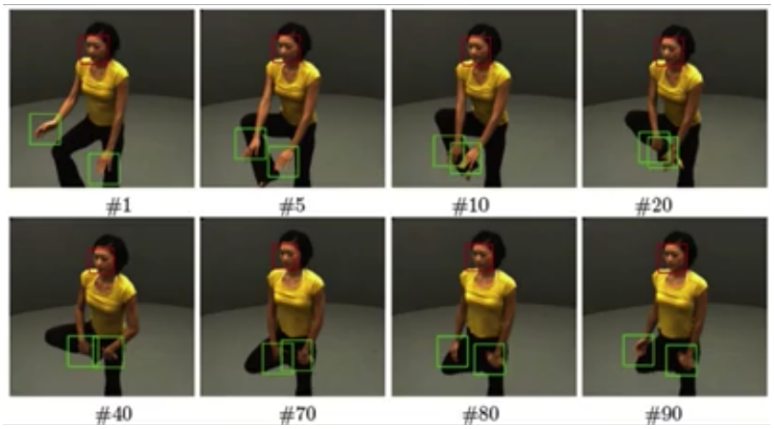



**Figure 12.10:** Using colored blobs to track a head and hands.

### A *Very* Simple Model

Let's make this as simple as we possibly can.

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.

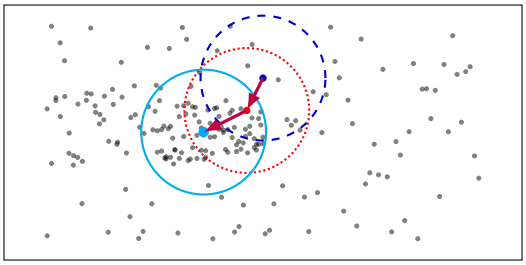


**Figure 12.11:** Performing mean-shift 2 times to find the area in the distribution with the most density (an approximation of its mode).

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:

$$\mathbf{q}' = (\sqrt{q_1}, \sqrt{q_2}, \ldots, \sqrt{q_m})$$

$$\mathbf{p}'(y) = \left(\sqrt{p_1(y)}, \sqrt{p_2(y)}, \ldots, \sqrt{p_m(y)}\right)$$

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:

$$f(y) = \mathbf{p}'(y) \cdot \mathbf{q}' = \|\mathbf{p}'(y)\| \, \|\mathbf{q}'\| \cos\theta_y$$

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 202 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" $\sigma$ 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, 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 A.3a) 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 $\alpha$ 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.

# Index of Terms