EE046746 Computer Vision Class Notes - Spring 2020

Written mostly by Guy Ohayon
Reviewed by Matan Gelber

August 14, 2020

Contents

1 Introduction

Most of these notes are based on Prof. Anat Levin's recorded lectures, with additional necessary materials and explanations I found online. Some of the last sections (such as geometric camera models and epipolar geometry) are based on Stanford's CS231A notes, with some corrections and changes to the order of the subjects (so that they fit this course).

With that being said, I worked hard to write these course notes and to make them as clear as possible, but I don't guarantee that there are no mistakes. I hope these will help you and that you'll succeed in the course. Enjoy :)

2 The Camera's Sensor

In this course we are going to talk about digital images. Nonetheless, we should first understand how cameras create digital images.

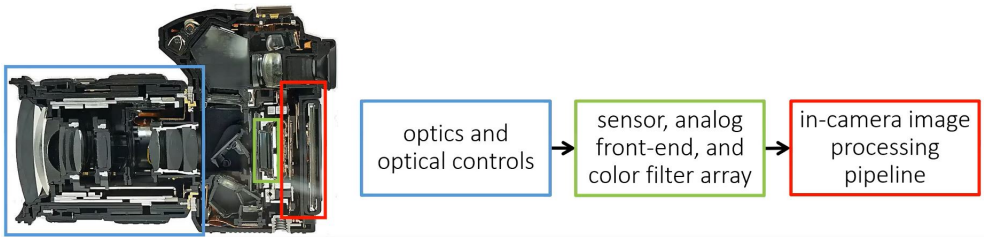


Figure 1: The main contents of a digital camera.

We won't talk much about the optics and the sensor, but we'll talk in depth about the image processing pipeline. Each sensor is a 2D array of "photon buckets" (pixels). Pixels are designed as follows:

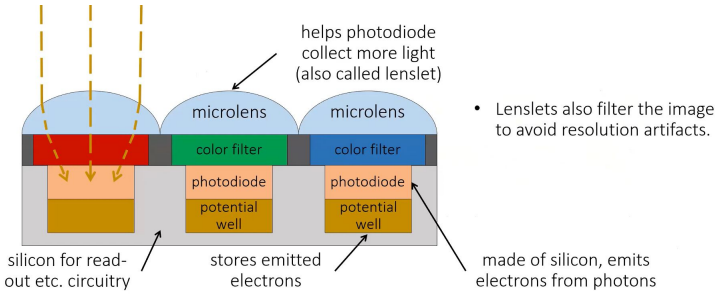


Figure 2: A closer look to the design of three pixels. Each of these pixels has a different color filter

As we can see, each pixel has 3 color filters and 3 microlenses. The microlenses help achieve better light concentration in each photodiode, because they diverge the light into their corresponding photodiode, as we can see in the illustration. The number of electrons emitted by the photodiode is usually linear in the illumination level, but is non-linear when the potential well is saturated (this is also called sensor saturation. Happens at over-exposure) or when there's not enough light (under-exposure. A result of sensor noise).

2.1 Vignetting

Vignetting is a reduction of an image's brightness or saturation toward the periphery compared to the image center. In other words, it's a phenomena when pixels far off from the center of the image receive less light:

Figure 3: Two examples of vignetting

There are four main types of vignetting:

1. **Mechanical**: light rays are blocked by hoods, filters, and other objects. Happens abruptly and only in the corners, since this is caused by matte boxes, filter rings or other objects physically blocking light in front of the lens.

2. **Lens**: light rays are blocked by lens elements. Caused by intrinsic lens characteristics and shading from the lens barrel itself.

3. **Natural**: appears as a gradual darkening and is primarily caused by light reaching different locations on the camera sensor at different angles. This type of vignetting is most significant with wide angle lenses.

4. **Pixel**: angle-dependent sensitivity photodiodes.

2.2 Color

Color is an artifact of human perception, and light is characterized by its wavelength. Each color filter (as in Figure 2) is responsible to collect light (photons) of a specific color (specific wavelength).

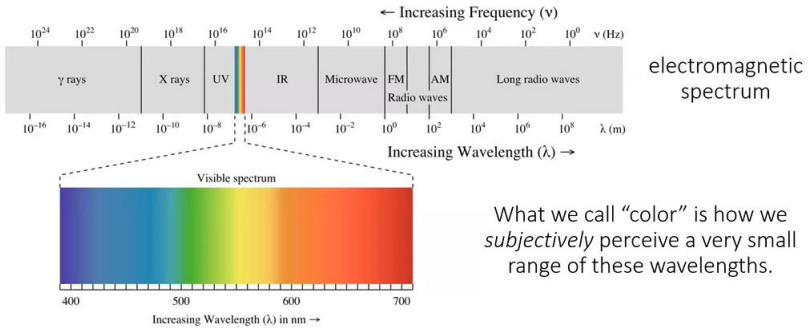


Figure 4: The electromagnetic spectrum

2.3 Light-Material Interaction

We'll describe the light-material interaction scheme:

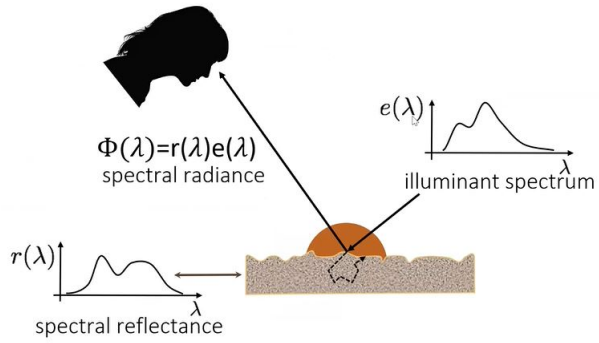


Figure 5: An illustration of the light-material interaction.

In Figure 5, λ is the wavelength, the energy of the incoming light is $e(\lambda)$, and the spectral reflectance is $r(\lambda)$. The illuminant spectrum (energy spectrum) is the wavelength distribution of the incoming light, and the spectral reflectance is the "sensitivity" of the surface to each incoming wavelength (how much the surface reflects and how much it swallows each wavelength). Thus, the returning light from the surface can be described as $\Phi(\lambda) = r(\lambda)e(\lambda)$. This happens because each surface swallows some wavelength more than others (that's why we see surfaces in different colors).

2.4 Spectral Power Distribution (SPD)

Most types of light contain more than one wavelength. We can describe light based on its power distribution over different wavelengths, and we'll call this description (function) the SPD

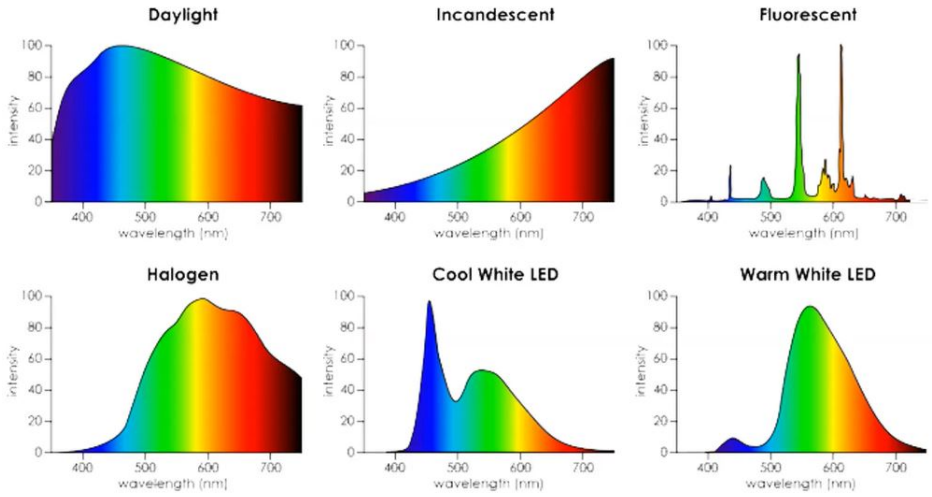


Figure 6: Different kinds of lights and their spectral power distribution

Remember: if you see white light, it's usually not coming with uniform SPD. This means that different cameras might capture the same white light as a slightly different color. We'll better understand it later on.

2.5 Spectral Sensitivity Function (SSF)

Any light sensor (digital or not - our eyes also) has different sensitivity to different wavelengths. The sensitivity is described by the sensor's spectral sensitivity function $f(\lambda)$. When measuring light of some SPD $\Phi(\lambda)$, the sensor produces a scalar response:

$$R = \int_{\lambda} \Phi(\lambda) f(\lambda) d\lambda$$

Where R is the sensor's response. This can be thought of as a weighted combination of the light's SPD: the light contributes more at wavelengths where the sensor has higher sensitivity.

The Human eye is a collection of light sensors called cone cells. There are three types of these cells, each with different SSF. The human's color perception is thus three-dimensional:

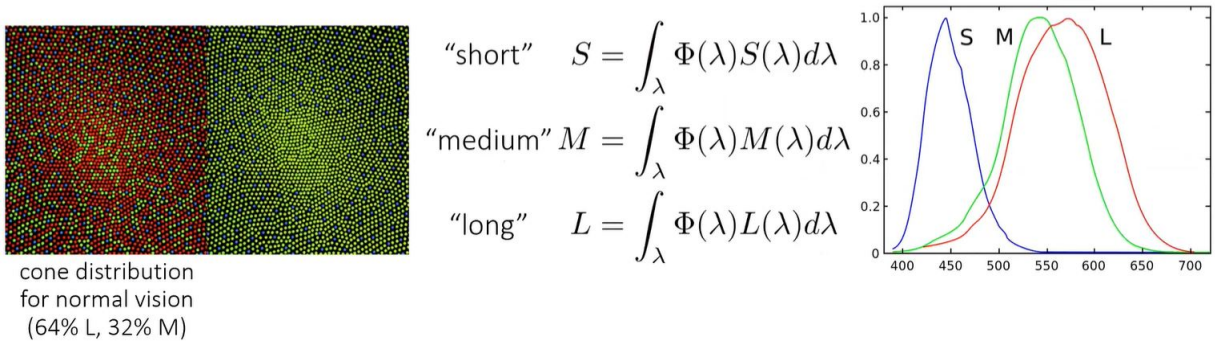


Figure 7: The three types of cones in the human's eye, and their spectral sensitivity functions. As we can see, the "green" cone is not so different than the "red" cone.

As we can understand, if we see a source of light that's very dense around $\lambda = 450$ (impulse), or a source of light that's spread between $\lambda = 400$ and $\lambda = 500$ (when both of these lights have the same total energy, eg the same $\int_\lambda \Phi(\lambda)d\lambda$), we (our eyes) won't tell the difference between these two light sources, because the sensor's response is a weighted average.

2.6 Color Filter Arrays (CFA)

To measure color using digital sensors, we mimic the design of cone cells in the human vision system. Cones correspond to pixels that are covered by different color filters (as in Figure 2), where each filter has its own spectral sensitivity function (SSF).

So, what color filters should we use? There are two main choices we need to make:

- What spectral sensitivity function $f(\lambda)$ should we use for each color filter? For example:

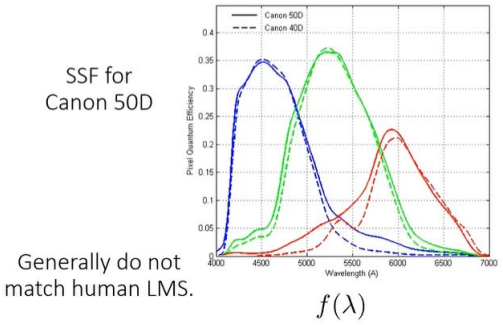


Figure 8: The pixel SSF choice in Canon 50D. This choice is very different than the human eye SSF, as we can see in Figure 7.

- How should we arrange the color filters spatially (mosaic)? For example:

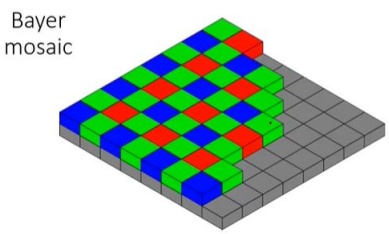


Figure 9: The bayer mosaic spatial pixel arrangement

As we can see in the bayer mosaic filter arrangement, there are more green filters than red or blue filters. This is because the human eye also have more green cones than red or blue cones.

There are many different CFAs. Finding the best CFA is an active area of research.

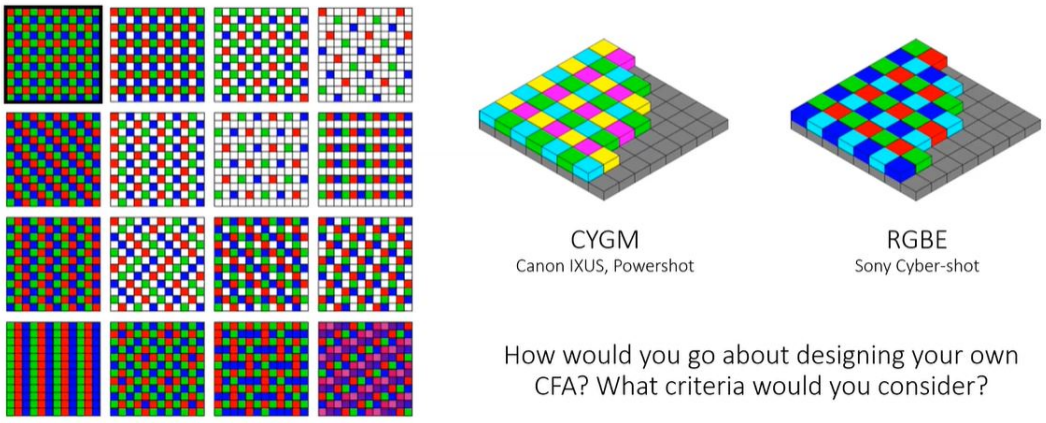


Figure 10: Different kinds of CFA choices

Thus, if we take pictures with cameras, the resulting pictures would be slightly different, which is a problem for computer vision algorithms.

So. what does an imaging sensor do?

- Every photodiode converts incident photons into electrons using mosaic's SSF.
- The sensor stores electrons in the photodiode's potential well while it's not full.
- Then, when the camera's shutter closes, the sensor reads the potential wells of each photodiode, row by row, and converts them to analog signals.
- The sensor then applies (possibly non-uniform) gain to these analog signals, converts them to digital signals, and corrects non-linearities.
- Finally, the raw digital image is formed (the image created by exclusively by the sensor, before the camera's post-processing).

So. how does the raw digital image looks like?

Figure 11: An example of a raw digital image.

We can see that there's a lot of noise in the image, and we can also see the artifacts created by the mosaic design of the sensor. Notice that the raw digital image is gray because this image is simply created by $0-1$ output values of each pixel of the sensor (the pixels are not aware that they are capturing a specific color).

Figure 12: Examples of white balancing and different types of whites.

2.7 White Balancing

The human visual system has chromatic adaptation: we can perceive white (and other colors) correctly under different light sources, where cameras can't do that. **White balancing** is the process of removing unrealistic color casts, so that objects which appear white in person are rendered white in your photo. Proper camera white balance has to take into account the "color temperature" of a light source, which refers to the relative warmth or coolness of white light. Our eyes are very good at judging what is white under different light sources, but digital cameras often have great difficulty with auto white balance (AWB) — and can create unsightly blue, orange, or even green color casts. Understanding digital white balance can help you avoid these color casts, thereby improving your photos under a wider range of lighting conditions.

Lets further understand the problem of white balancing. As we learned, the color intensities of the camera and of our eyes are:

$$I_{color}^{camera} = \int_{\lambda} \overbrace{r(\lambda)e(\lambda)}^{\phi(\lambda)} f_{color}^{camera}(\lambda)d\lambda$$

$$I_{color}^{eye} = \int_{\lambda} r(\lambda)e(\lambda)f_{color}^{eye}(\lambda)d\lambda$$

The arising question is, can we match $I_{color}^{eye} = I_{color}^{camera}$? The simplest thing we can try to do is to search for a scalar s such that $I_{color}^{eye} = s \cdot I_{color}^{camera}$, but this wouldn't necessarily work because the relationship is not necessarily linear (this would work though if the light is monochromatic). So, the goal of white balancing is to find a matrix $S = \begin{pmatrix} S_R & 0 & 0 \\ 0 & S_G & 0 \\ 0 & 0 & S_B \end{pmatrix}$ such that for each pixel p:

$$\begin{pmatrix} R_p^{world} \\ G_p^{world} \\ B_p^{world} \end{pmatrix} = S \begin{pmatrix} R_p^{camera} \\ G_p^{camera} \\ B_p^{camera} \end{pmatrix}$$

As we said before, we often can't find such matrix S that works for all pixels, but for simplicity we assume that we can.

Another way to do white balancing is to perform manual white balancing: "tell" the camera which part in the image is white and then the camera finds a transformation that makes this part white.

2.8 CFA Demosaicing

The raw digital image eventually turns into a colored image as part of the camera's post-processing algorithm. This procedure is called CFA Demosaicing.

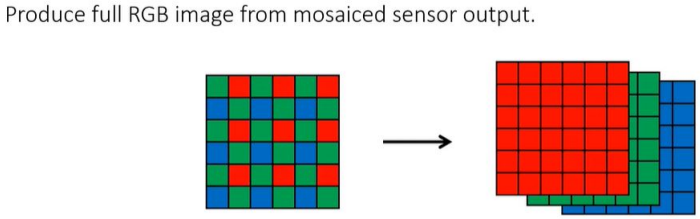


Figure 13: CFA Demosaicing: converting a mosaiced raw digital image into full digital RGB image.

To perform CFA Demosaicing, we have a couple of options:

Automatic white balancing

Grey world assumption:

- Compute per-channel average.
- Normalize each channel by its average.
- Normalize by green channel average.

$$\text{white-balanced RGB} \rightarrow \begin{bmatrix} R' \\ G' \\ B' \end{bmatrix} = \begin{bmatrix} G_{avg}/R_{avg} & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & G_{avg}/B_{avg} \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix} \leftarrow \text{sensor RGB}$$

White world assumption:

- Compute per-channel maximum.
- Normalize each channel by its maximum.
- Normalize by green channel maximum.

$$\text{white-balanced RGB} \rightarrow \begin{bmatrix} R' \\ G' \\ B' \end{bmatrix} = \begin{bmatrix} G_{max}/R_{max} & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & G_{max}/B_{max} \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix} \leftarrow \text{sensor RGB}$$

Three types of sensor noise

1) (Photon) shot noise:

- Photon arrival rates are a random process (Poisson distribution).
- Noise variance scales as mean intensity, the brighter the scene, the higher the SNR.

2) Dark-shot noise:

- Emitted electrons due to thermal activity (becomes worse as sensor gets hotter.)

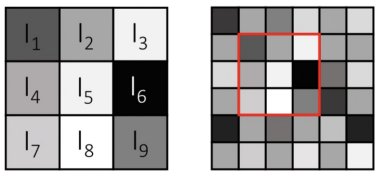
3) Read noise:

- Caused by read-out and AFE electronics (e.g., gain, A/D converter).

Bright scene and large pixels: photon shot noise is the main noise source.

How to denoise?

Look at the neighborhood around you.



- Mean filtering (take average):

$$I'_5 = \frac{I_1 + I_2 + I_3 + I_4 + I_5 + I_6 + I_7 + I_8 + I_9}{9}$$

- Median filtering (take median):

$$I'_5 = \text{median}(I_1, I_2, I_3, I_4, I_5, I_6, I_7, I_8, I_9)$$

Large area of research. Covered in may other classes.

- Bilinear interpolation from neighbors (we need 4 neighbors for that)

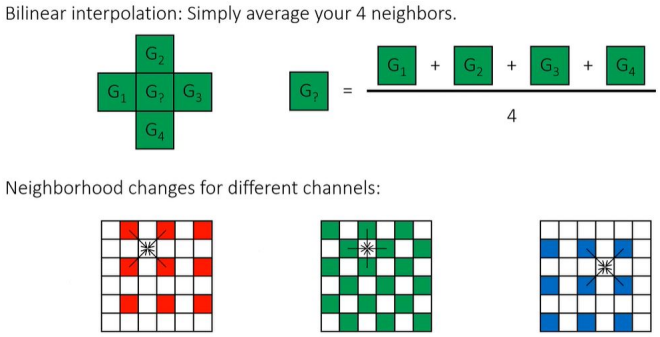


Figure 14: Demosaicing by bilinear interpolation

- Bicubic interpolation from neighbors (we need 16 neighbors for that. The image may over-blur). (קוביות)
- Edge-aware interpolation from neighbors (more on this later).

2.9 Tone Reproduction (Gamma Encoding)

The camera's sensor is usually linear with the light intensity, but monitors (mainly old monitors) doesn't present an image that's linear with the intensities of the digital image. Thus, we should perform the inverse operation of the display (the camera performs this inverse operation), so that for a linear digital image, the display would also display a linear digital image.

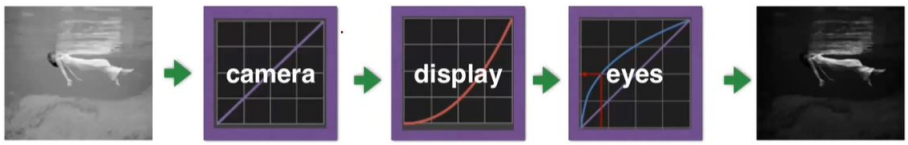
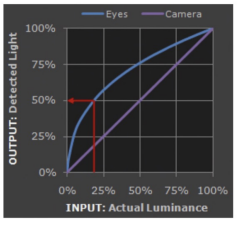


Figure 15: The tone reproduction scheme

So, why should the camera perform this inverse operation? Because the camera also samples the image's data (compression), and so it's better to perform this operation before the compression (to lose less data).

Perceived vs measured brightness by human eye



We have already seen that sensor response is linear.

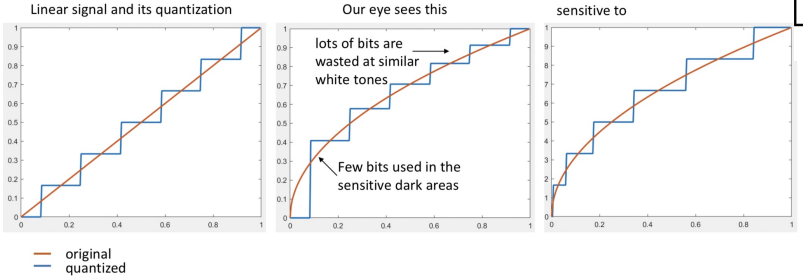
Human-eye *response* (measured brightness) is also linear.

However, human-eye *perception* (perceived brightness) is *non-linear*:

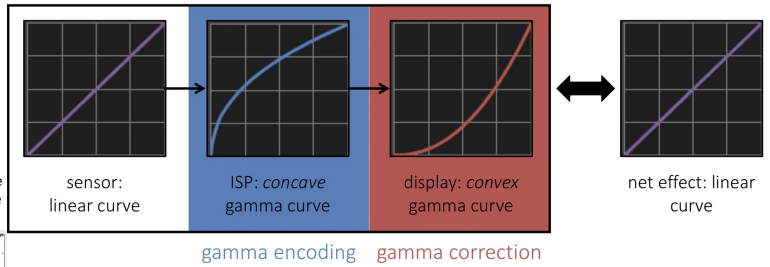
- More sensitive to dark tones.
- Approximately a Gamma function (square root of brightness).

Gamma encoding

We want to perform compression, which includes changing from 12 to 8 bits.



Tone reproduction pipeline



3 Feature Detectors and Descriptors

In computer vision, there are many problems we need to solve, but in most cases we want to find some kind of matching between images. The most basic thing we need to do is to be able to find features in images, and then we'll use these features to find corresponding points between images.

3.1 Edge Detectors

3.1.1 Canny Edge Detector

The Canny edge detector is an edge detection operator that uses a multi-stage algorithm to detect a wide range of edges in images. The Process of Canny edge detection algorithm can be broken down to 5 different steps:

1. **Apply Gaussian filter to smooth the image in order to remove the noise**: since all edge detection results are easily affected by the noise in the image, it is essential to filter out the noise to prevent false detection caused by it.

2. **Find the intensity gradients of the image**: an edge in an image may point in a variety of directions, so the Canny algorithm uses four filters to detect horizontal, vertical and diagonal edges in the blurred image. The edge detection operator (such as Roberts, Prewitt, or Sobel) returns a value for the first derivative in the horizontal direction (Gx) and the vertical direction (Gy). From this the edge gradient and direction can be determined:

$$G = \sqrt{G_x^2 + G_y^2}$$

$$\theta = arctan2(G_y, G_x)$$

3. **Apply non-maximum suppression to get rid of spurious response to edge detection**: non-maximum suppression is an edge thinning technique. It's applied to find the locations with the sharpest change of intensity value. The algorithm for each pixel in the gradient image is:

   (a) Compare the edge strength of the current pixel with the edge strength of the pixel in the positive and negative gradient directions.

   (b) If the edge strength of the current pixel is the largest compared to the other pixels in the mask with the same direction (e.g., a pixel that is pointing in the y-direction will be compared to the pixel above and below it in the vertical axis), the value will be preserved. Otherwise, the value will be suppressed.

4. **Apply double threshold to determine potential edges**: after application of non-maximum suppression, remaining edge pixels provide a more accurate representation of real edges in an image. However, some edge pixels remain that are caused by noise and color variation. In order to account for these spurious responses, it is essential to filter out edge pixels with a weak gradient value and preserve edge pixels with a high gradient value. This is accomplished by selecting high and low threshold values. If an edge pixel's gradient value is higher than the high threshold value, it is marked as a strong edge pixel. If an edge pixel's gradient value is smaller than the high threshold value and larger than the low threshold value, it is marked as a weak edge pixel. If an edge pixel's value is smaller than the low threshold value, it will be suppressed. The two threshold values are empirically determined and their definition will depend on the content of a given input image.

5. **Track edge by hysteresis - Finalize the detection of edges by suppressing all the other edges that are weak and not connected to strong edges**: so far, the strong edge pixels should certainly be involved in the final edge image, as they are extracted from the true edges in the image. However, there will be some debate on the weak edge pixels, as these pixels can either be extracted from the true edge, or the noise/color variations. To achieve an accurate result, the weak edges caused by the latter reasons should be removed. Usually a weak edge pixel caused from true edges will be connected to a strong edge pixel while noise responses are unconnected. To track the edge connection, blob analysis is applied by looking at a weak edge pixel and its 8-connected neighborhood pixels. As long as there is one strong edge pixel that is involved in the blob, that weak edge point can be identified as one that should be preserved.

3.2 Corner Detectors

Corner detection is an approach used within computer vision systems to extract certain kinds of features and infer the contents of an image. Corner detection is frequently used in motion detection, image registration, video tracking, image mosaicing, panorama stitching, 3D reconstruction and object recognition. Corner detection overlaps with the topic of interest point detection.

A corner is a point whose local neighborhood stands in two dominant and different edge directions. In other words, a corner can be interpreted as the junction of two edges, where an edge is a sudden change in image brightness. Corners are one of the most important features in an image, and they are generally termed as interest points which are invariant to translation, rotation and illumination. Although corners are only a small percentage of the image, they contain the most important features in restoring image information, and they can be used to minimize the amount of processed data for motion tracking, image stitching, building 2D mosaics, stereo vision, image representation and other related computer vision areas.

In order to capture the corners from an image, researchers have proposed many different corner detectors including the Kanade-Lucas-Tomasi (KLT) operator and the Harris operator which are most simple, efficient and reliable for use in corner detection. These two popular methodologies are both closely associated with and based on the local structure matrix. Compared to the Kanade-Lucas-Tomasi corner detector, the Harris corner detector provides good repeatability under changing illumination and rotation, and therefore, it is more often used in stereo matching and image database retrieval. Although there still exists drawbacks and limitations, the Harris corner detector is still an important and fundamental technique for many computer vision applications.

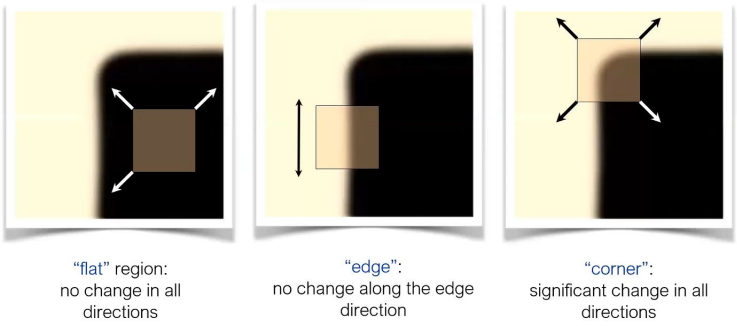


Figure 16: An example of a corner, and edge, and a flat region. Shifting the window over corners should give large change in intensity (note that shifting the window over an edge or a flat region doesn't change the intensity at all in the case above).

3.2.1 Singular Value Decomposition (SVD)

Singular value decomposition (SVD) is a factorization of a real or complex matrix that generalizes the eigendecomposition of a square matrix to any $m \times n$ matrix. Specifically, the singular value decomposition of a $m \times n$ real or complex matrix M is a factorization of the form $M = U\Sigma V^*$, where U is a $m \times m$ real or complex unitary matrix ($UU^* = U^*U = I$, where U^* is the conjugate transpose of U), Σ is a $m \times n$ diagonal matrix with non-negative real numbers on the diagonal, and V is a $n \times n$ real or complex unitary matrix. If M is real, U and $V^T = V^*$ are real orthogonal matrices ($VV^T = V^TV = U^TU = UU^T = I$).

The diagonal entries $\sigma_i = \Sigma^{ii}$ of Σ are called the singular values of M, and the number of non-zero singular values is equal to the rank of M. The columns of U and the columns of V are called the left singular vectors and the right singular vectors of M, respectively.

The SVD is not unique. Nonetheless, it's always possible to choose the decomposition so that the singular values σ_i are in descending order, and in this case, Σ (but not always U and V) is uniquely determined by M.

Intuitive Interpretation of SVD: In the special case where M is a $m \times m$ real matrix, the matrices U and V^* can be chosen to be real $m \times m$ matrices too. In this case, we can interpret the linear transformation $M = U\Sigma V^*$ applied on some vector $x \in R^m$ as a rotation or reflection V^* of x, followed by a coordinate-by-coordinate scaling Σ of V^*x, followed by another rotation or reflection U of ΣV^*x. Thus, the SVD breaks down any invertible linear transformation of R^m into a composition of three geometrical transformations. Note that any orthogonal matrix (such as U and V^*) is a rotation or reflection matrix.

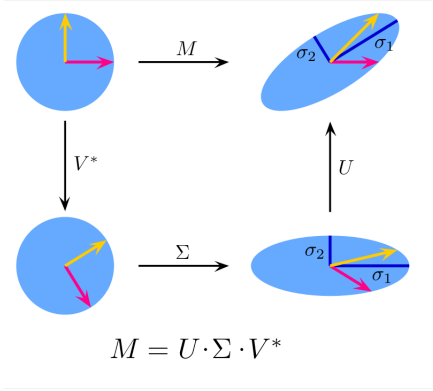


Figure 17: An intuitive illustration of SVD of a real 2×2 matrix M. M is a linear transformation performed on the unit circle

In particular, if M has a positive determinant, then U and V^* can be chosen to be both reflections or both rotations. If the determinant is negative, exactly one of them will have to be a reflection. If the determinant is zero, each can be independently chosen to be of either type.

Singular Values As Semiaxes of An Ellipse: As shown in figure 17, the singular values can be interpreted as the magnitude of the semiaxes of an ellipse in $2D$. This concept can be generalized to $n-dimensional$ Euclidean space, with the singular values of any $n \times n$ square matrix being viewed as the magnitude of the semiaxis of an $n-dimensional$ ellipsoid.

The Columns of U and V Are Orthonormal Bases: Since U and V^* are unitary matrices, the columns of each of them form a set of orthonormal vectors (separately), which can be regarded as basis vectors. The matrix M maps the basis vector V_i (the i_{th} column of V) to the stretched unit vector $\sigma_i U_i$.

3.2.2 Image Gradients

An image gradient is a directional change in the intensity or color in an image. The gradient of the image is one of the fundamental building blocks in image processing. For example, the Canny edge detector uses image gradient for edge detection (more on Canny edge detector later).

Recall that an image is a $2D$ function $f(x, y)$. The gradient of an image f is a vector of its partial derivatives:

$$\nabla f(x,y) = \begin{pmatrix} \frac{\partial f}{\partial x} \\ \frac{\partial f}{\partial y} \end{pmatrix} = \begin{pmatrix} g_x \\ g_y \end{pmatrix}$$

Where g_x is the derivative of f with respect to the x direction, and g_y is the derivative of f with respect to the y direction. Since we are dealing with digital images (which can be represented as a 2D matrix), the horizontal derivative of an image A can be approximated by finite differences, and subsequently as a simple convolution with a filter:

$$\frac{\partial f(x,y)}{\partial x} \approx \frac{f(x+1,y) - f(x-1,y)}{2}$$

$$\Rightarrow g_x = \begin{pmatrix} 1 & 0 & -1 \end{pmatrix} * A$$

And in the same way we can approximate the vertical derivative of f:

$$g_y = \begin{pmatrix} 1 \\ 0 \\ -1 \end{pmatrix} * A$$

The gradient direction is then $\theta = tan^{-1}[\frac{g_y}{g_x}]$, and its magnitude is $\sqrt{g_x^2 + g_y^2}$. There are better/worse derivative filters we can use (there's no single answer to which filter is best). Filters can be sensitive to noise, and in this case we can blur the image before applying the derivative filter.

3.2.3 Harris Corner Detector

Harris Corner Detector is a corner detection operator that is commonly used in computer vision algorithms to extract corners and infer features of an image.

Without loss of generality, we'll assume that a grayscale 2D image is used. Let this image be given by I. Consider taking an image patch $(x, y) \in W$ (W is a window) and shifting it by $(\Delta x, \Delta y)$. The sum of squared differences between these two patches, denoted f, is given by:

$$f(\Delta x, \Delta y) = \sum_{(x_k, y_k) \in W} (I(x_k, y_k) - I(x_k + \Delta x, y_k + \Delta y))^2$$

$I(x_k + \Delta x, y_k + \Delta y)$ can be approximated by a Taylor expansion: let I_x, I_y be the partial derivates of I, such that:

$$I(x_k + \Delta x, y_k + \Delta y) \approx I(x_k, y_k) + I_x(x_k, y_k)\Delta x + I_y(x_k, y_k)\Delta y$$

This produces the following approximation:

$$f(\Delta x, \Delta y) \approx \sum_{(x,y) \in W} (I_x(x, y)\Delta x + I_y(x, y)\Delta y)^2 = \begin{pmatrix} \Delta x & \Delta y \end{pmatrix} M \begin{pmatrix} \Delta x \\ \Delta y \end{pmatrix}$$

Where M is called the structure matrix (also called the covariance matrix):

$$M = \begin{pmatrix} \sum_{(x,y) \in W} I_x^2(x, y) & \sum_{(x,y) \in W} I_x(x, y) I_y(x, y) \\ \sum_{(x,y) \in W} I_x(x, y) I_y(x, y) & \sum_{(x,y) \in W} I_y^2(x, y) \end{pmatrix}$$

The Harris corner detection algorithm is then an algorithm of multiple steps (for each window W surrounding some pixel p, we want to determine if there's a corner in that window):

1. **Color to grayscale**: first, we need to convert the image to grayscale, which will enhance the processing speed. The value of the gray scale pixel can be computed as a weighted sums of the R,G,B values of the colored image.
2. **Spatial Gradients Calculation: compute** $I_x(x, y), I_y(x, y)$ **for each** $(x, y) \in W$

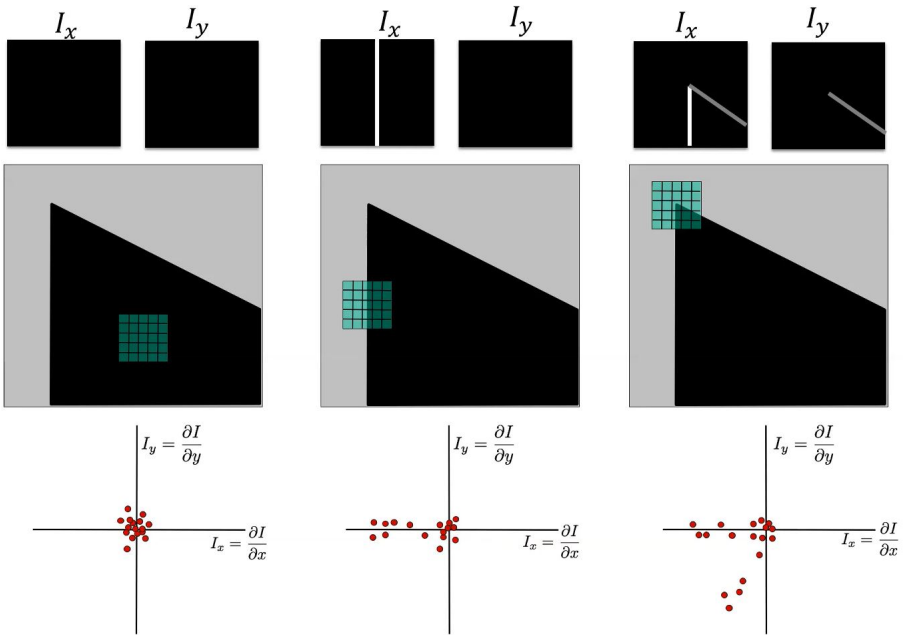


Figure 18: Image gradients I_x and I_y in different scenarios. The bottom plots shows us $I_y(x, y)$ and $I_x(x, y)$ for each $(x, y) \in W$ (the window). In this example, the window is of size $5x5$ and so 25 numbers are plotted in the bottom plots. These distributions reveal edge orientation and magnitude.

3. **Subtract the mean from each image gradient** I_x **and** I_y: $I_x(x, y) = I_x(x, y) - mean_{(x,y) \in W}[I_x(x, y)]$ and $I_y(x, y) = I_y(x, y) - mean_{(x,y) \in W}[I_y(x, y)]$

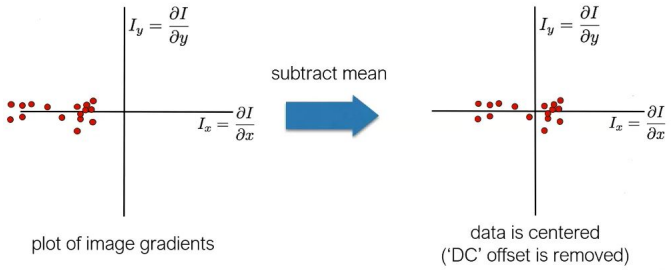


Figure 19: Image gradients I_x and I_y mean subtraction.

4. **Structure Matrix Setup: with $I_x(x, y)$ and $I_y(x, y)$, construct M.** Note that M defines the 2D surface $E(u, v) = \begin{pmatrix} u & v \end{pmatrix} M \begin{pmatrix} u \\ v \end{pmatrix}$. Lets take a look at $E(u, v)$:

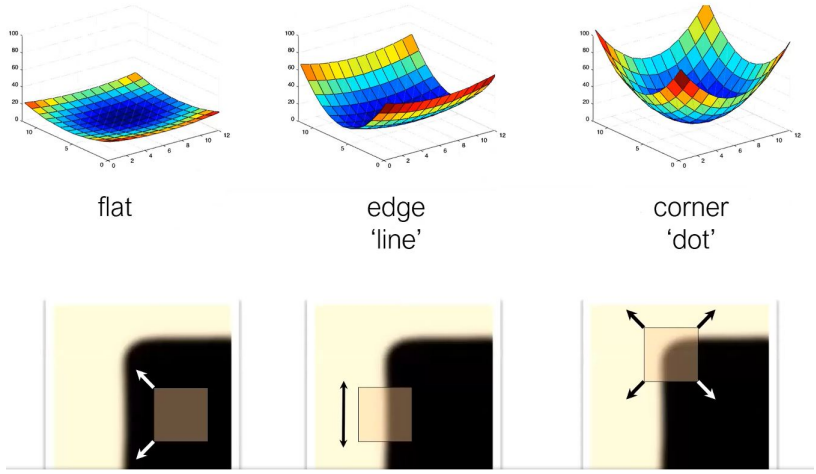


Figure 20: As we can see, we can use $E(u, v)$ to determine whether there's a corner in this window/patch or not.

- If we have a flat surface, $I_x, I_y \approx 0$, and so $M \approx 0 \Rightarrow E(u, v) \approx 0$.
- If we have an edge, and suppose $I_y \approx 0$, then M will be approximately a matrix where M_{11} is the only non-zero value. In this case, $E(u, v) \approx u^2$, which is a parabola.
- If we have a corner, then I_x and I_y are non-zero, and in this case $E(u, v)$ is approximately a paraboloid.

Since M is symmetric, we can write $M = R^{-1} \begin{pmatrix} \lambda_1 & 0 \\ 0 & \lambda_2 \end{pmatrix} R$ (eigendecomposition). We can visualize M as an ellipse with axis lengths determined by the eigenvalues λ_1, λ_2 and orientation determined by R. To understand how $E(u, v)$ looks like, R is not important because it's only responsible for rotation or reflection. Thus, we only need M to determine if there's a corner in the window. On a flat window we expect to have $\lambda_1, \lambda_2 \approx 0$. On a window containing an edge we expect to have $\lambda_{min} \ll \lambda_{max}$, and on a window containing a corner we expect to have $\lambda_{min} \approx \lambda_{max} \neq 0$:

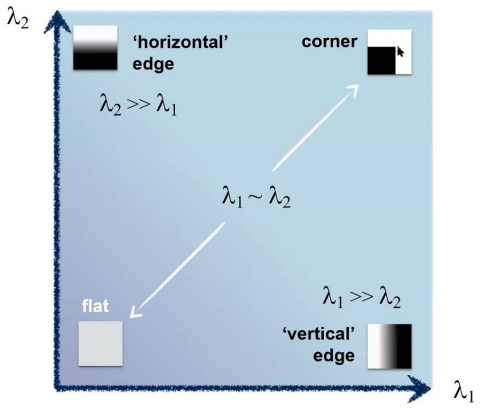


Figure 21: Interpreting Eigenvalues

5. **Harris Response Calculation - Threshold On Eigenvalues**: using what we have learned, we can understand that we should threshold λ_1, λ_2 to determine if there's a corner in the window. We call this the "response", and we'll use the response to determine if there's a corner. There are multiple kinds of responses:
   - The simplest way to define the response is $R = min(\lambda_1, \lambda_2)$. Then, we can decide that there's a corner in the window if $R > t$ for some threshold t. This method is expensive because we need to calculate λ_1 and λ_2.
   - Another way of calculating the response is $R = det(M) - k \cdot trace^2(M)$ for some parameter k. The reason to use this type of response is that we can calculate it a lot faster.

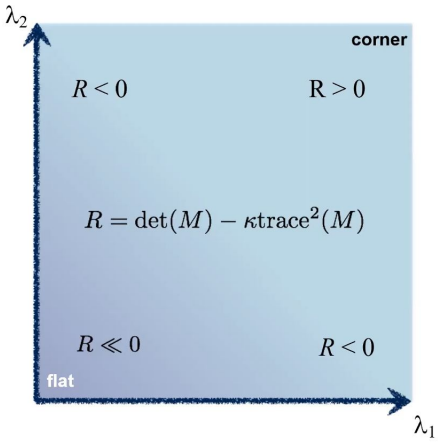


Figure 22: Another kind of response

   - The last way to calculate the response is the following: for $x \ll y$, one has $\frac{xy}{x+y} = \frac{x}{1+x/y} \approx x$. Thus, we can approximate λ_{min} as follows:

$$R = \lambda_{min} \approx \frac{\lambda_1 \lambda_2}{\lambda_1 + \lambda_2} = \frac{det(M)}{trace(M)}$$

     This is useful because $\frac{x}{1+x/y} \leq x$ (assuming $x > 0, y > 0$), and so putting a threshold on $\frac{xy}{x+y}$ means putting a threshold on λ_{min}.

Notice two properties of the Harris corner response:

- It's invariant to rotation (because it only depends on M).
- It's invariant to intensity changes (because it's based on derivatives).
- It's **not** invariant to scale! (because it depends on the corner size, and we use a specific window size).

So, how can we make a scale-invariant corner detector? We'll talk about it later.

3.2.4 Laplacian Feature Detector

So far, we have seen the Harris corner detector. Lets take one step back and talk about a simpler type of feature: the Laplacian.

The Laplacian filter can be approximated as a difference between the image convolved with a Gaussian filter and the image convolved with another Gaussian filter: $L = G_{\sigma_1} * I - G_{\sigma_2} * I = (G_{\sigma_1} - G_{\sigma_2}) * I$. This is called the difference of Gaussians.

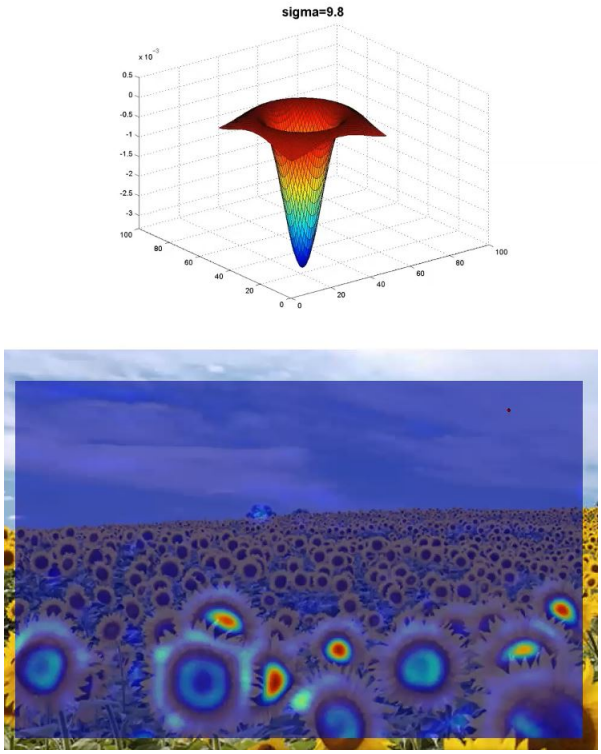


Figure 23: An example of a Laplacian filter applied on an image. Sunflowers that are close to the size of the Laplacian filter has larger convolution response.

This brings us to multi scale feature detection: how can we make a scale-invariant feature detector? We can apply Laplacian filters in multiple scales:

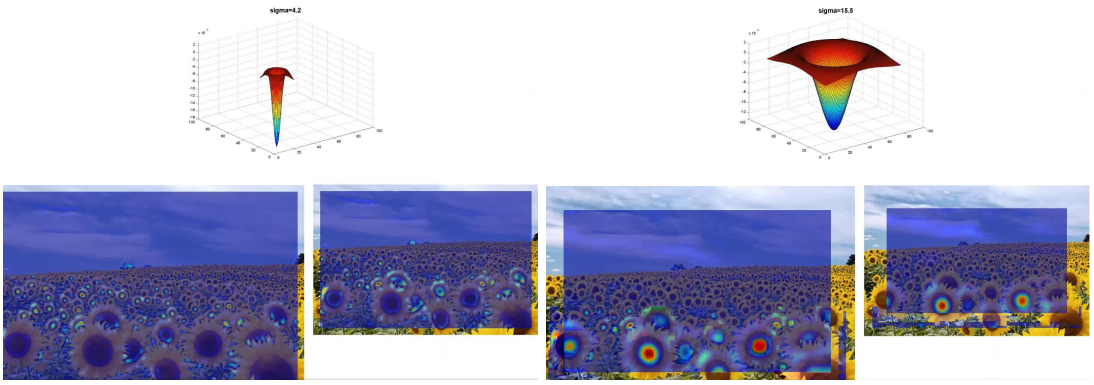


Figure 24: Laplacian filter applied at two different scales. We can see that the convolution result (response) depends on the scale of the Laplacian filter.

Intuitively, we want to find local maxima in both image space and filter scale (we want to find (x, y, s) points such that the Laplacian filter at scale s achieves maximum response at the spatial location (x, y)). For each window, lets look at a graph of the filter's response as a function of the filter's scale:

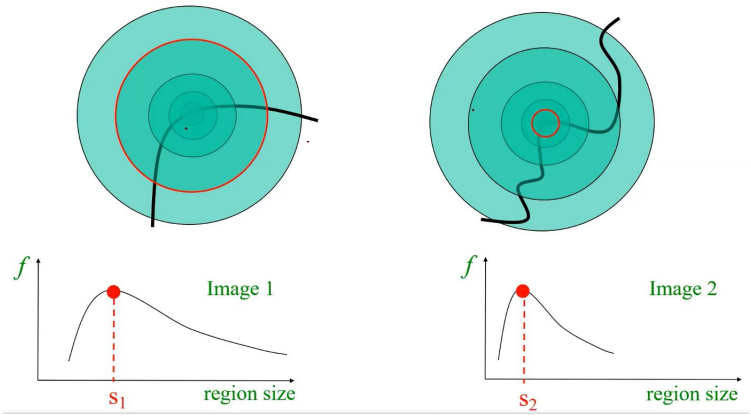


Figure 25: The Laplacian filter response as a function of the filter's scale

Then, our features would have a scale that attains local maxima in this graph. To do that, we calculate the Laplacian filter's response at multiple filter scales (σ). Since we want to avoid noise sensitivity, we also apply a Gaussian filter on each scale, and so we receive the Laplacian of Gaussians 3D function, which can be approximated as follows:

$$D(x, y, \sigma) = (G(x, y, k\sigma) - G(x, y, \sigma)) * I(x, y)$$

where $G(x, y, \sigma)$ is a Gaussian filter (this is called the Difference of Gaussians, which approximates the Laplacian of Gaussians). Notice that the factor k controls the scale of the Laplacian filter. We then choose our features to be locations that are simultaneously extrema in the Laplacian's response image plane and along the scale coordinate of $D(x, y, \sigma)$. Such feature points are found by comparing the $D(x, y, \sigma)$ value of each point with its 8-neighborhood on the same scale level, and with the 9 closest neighbors on each of the two adjacent levels, as shown in figure 26.

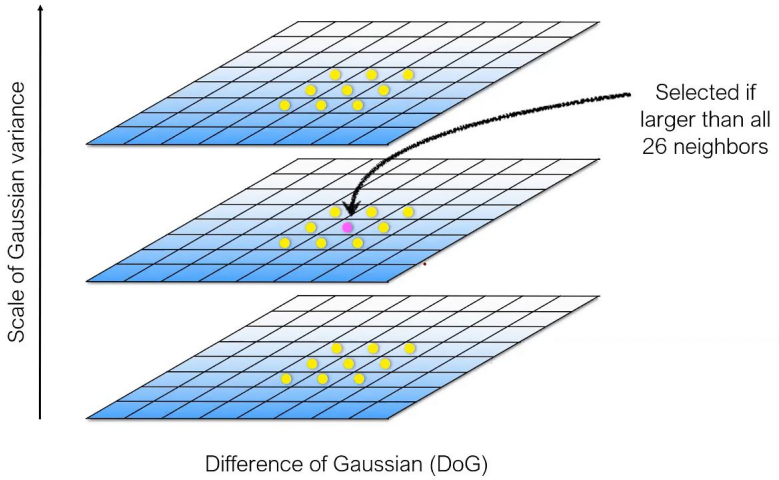


Figure 26: A multi-layer pyramid of Laplacian responses applied at three different scales on the same image. A pixel is selected as a feature if it's larger than all its 26 neighbors.

For each selected feature, we keep both its spatial location (x, y) and its scale s, and so each feature is described as (x, y, s).

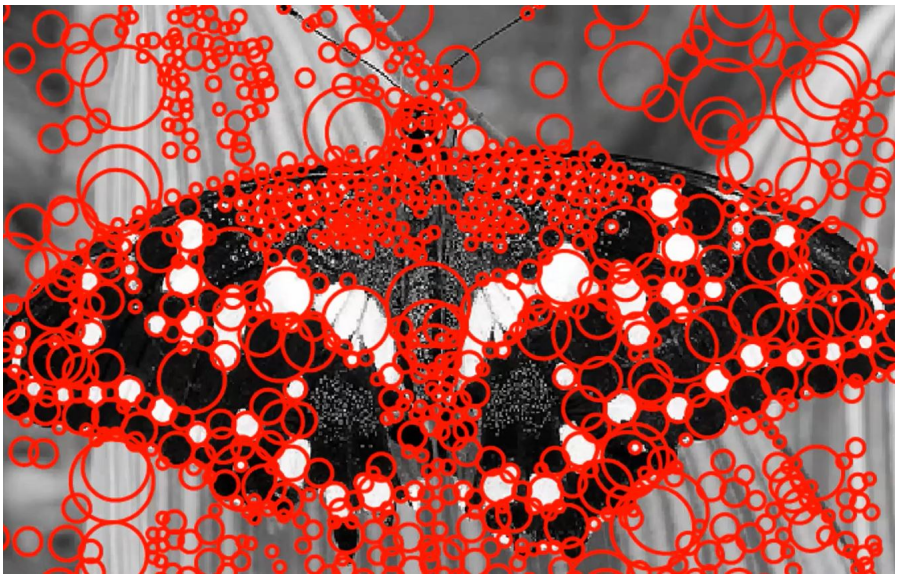


Figure 27: An example of detected Laplacian features from an image. We can see that this is not quite successful: this feature detector detects edges, which don't serve as good image features.

Our next goal is to avoid edge features. To do that, for each detected feature (x, y, s) we can run a corner detector similar to Harris corner detector:

1. Let $D(s)$ be the Difference of Gaussian image at scale s.
2. Calculate $H = \begin{pmatrix} D(s)_{xx} & D(s)_{xy} \\ D(s)_{yx} & D(s)_{yy} \end{pmatrix}$ (on a window surrounding the detected feature).
3. Calculate the response $R = \frac{trace^2(H)}{det(H)} = \frac{(\lambda_{min}+\lambda_{max})^2}{\lambda_{min}\lambda_{max}}$ (this function is minimized when $\lambda_{min} = \lambda_{max}$).
4. Discard the feature (x, y, s) if $R > \theta_r$ (when we have an edge, we get $\lambda_{min} \to 0$ and so $R \to \infty$. This reminds us of Harris corner detector, but the response is a bit different).

The algorithm above can be thought of as we are running Harris corner detector on scale DoG images that contain features, to see if the detected feature is an edge or a corner.

3.3 Feature Descriptors

So, we found feature locations, but now how do we match features found in two images? Our descriptors should have the following properties:

- Invariant to color (photometric transformations).
- Invariant to geometric transformations.
- And more...

The simplest way to describe a feature, is to flatten the patch surrounding it, and then our feature descriptor is the resulting flattened vector:

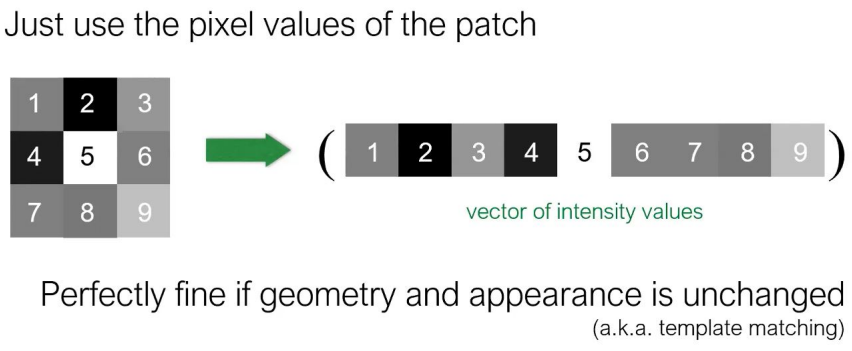


Figure 28: The image patch descriptor

This feature descriptor is sensitive to intensity, rotation, geometry, and many more conditions, and thus this descriptor is not very good.

Another kind of descriptor is to do the same thing, but instead use the vector of derivatives (we look at the sign of the derivatives):

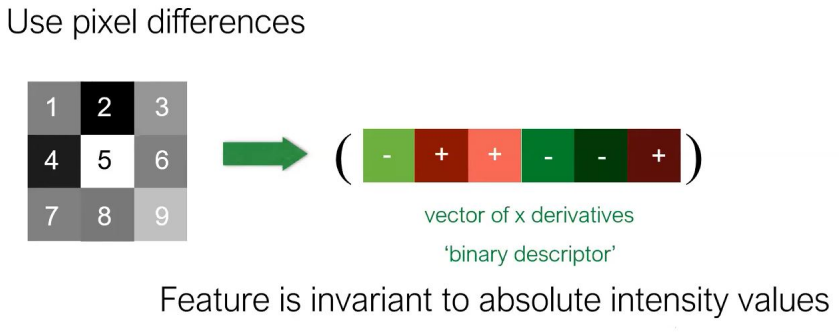


Figure 29: The image patch derivatives descriptor

Why is this helpful? Suppose we take an image I and then perform a transformation $aI + b$ (a, b are scalars). We can understand that our descriptor is invariant to this transformation, meaning that our descriptor is invariant to the image's color values scale and to any bias added to it. Note that this descriptor is very sensitive to rotation and scale!

We'll now build our way to a feature descriptor that is invariant to image deformations (the descriptors we talked about so are are not!).

3.3.1 Color Histogram

A color histogram is simply a histogram of the color levels in an image (or a patch in the image). For example, if we have the patch $\begin{pmatrix} 2 & 2 & 2 \\ 3 & 1 & 3 \\ 3 & 2 & 2 \end{pmatrix}$ then our histogram would be $[1, 5, 3]$ because we have the value 1 once, the value 2 five times, and the value 3 three times.

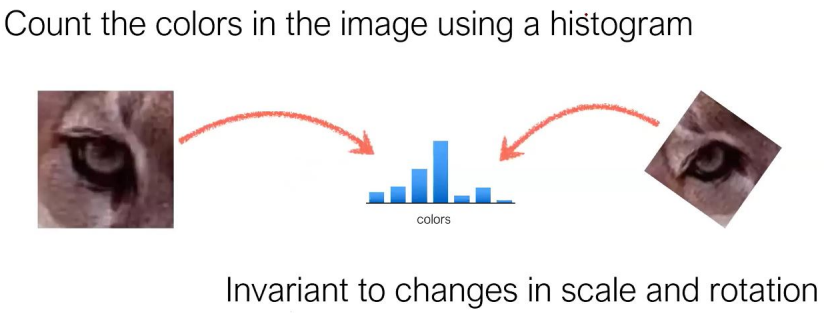


Figure 30: A color histogram. Note that it's invariant to rotation

We can understand that color histograms are invariant to rotation, because rotations doesn't affect the intensity values on an image. However, **there could be two completely different images with the same color histogram!**. A simple way to fix this is to split the image into patches, and compute the color histogram of each patch separately.

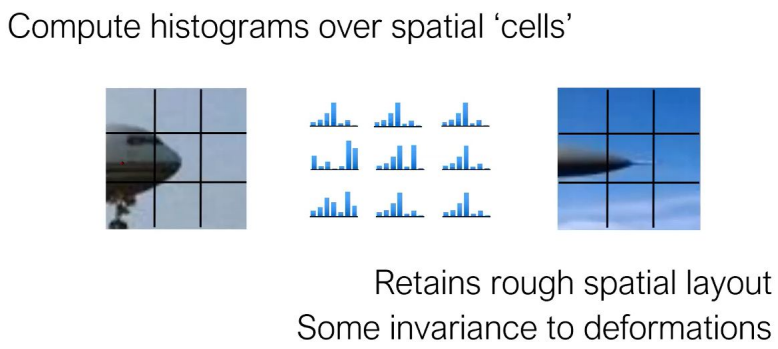


Figure 31: Patch color histogram. This kind of histogram calculation is variant to rotation.

This solution is somewhat invariant to image deformations, but not completely invariant. Also, **this solution, which we call patch color histogram, is variant to rotation!** Fortunately, we have a way to make patch color histogram invariant to rotation:

- For each detected feature, take a patch with the feature at its center.
- Calculate the gradient at each pixel in that patch, and define the dominant image gradient of the patch to be the gradient direction with most pixels pointing at that direction (this is equivalent to performing a histogram on the patch of gradients, and choosing the gradient with the highest histogram value).
- Rotate the patch so that it points to the direction of the dominant image gradient, and now perform a color histogram of that patch. This resulting color histogram is our resulting feature descriptor.

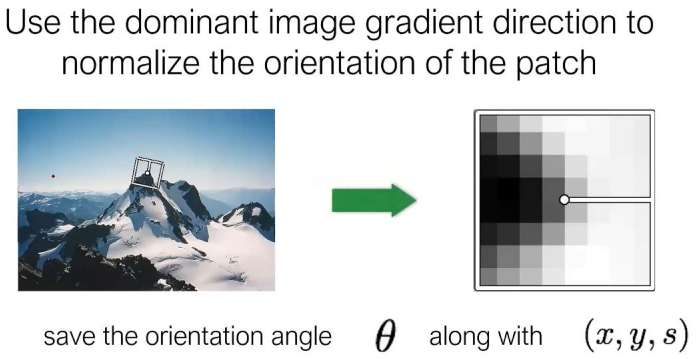


Figure 32: The dominant gradient path re-orientation. We add the direction θ to the feature.

3.3.2 SIFT (Scale Invariant Feature Transform)

SIFT serves both a feature detector and a feature descriptor. As its name suggests, it's a scale-invariant feature detector and descriptor, and it's also invariant to rotation. There are mainly four steps involved in the SIFT algorithm:

1. **Scale-space Extrema Detection:** to detect features in a scale-invariant fashion, SIFT uses the Difference of Gaussian blob detector (as we learned before). This gives us a list of (x, y, σ).
2. **Keypoint Localization:** Once potential keypoint locations are found, they have to be refined to get more accurate results. One refinement example is to remove edges keypoints, which we already learned how to do in the section about Laplacian Feature Detection.
3. **Orientation assignment:** we already discussed this before - this is the dominant image gradient scheme. We'll discuss it again, as it's a bit different this time.

   An orientation is assigned to each potential keypoint to achieve invariance to image rotation. A neighbourhood is taken around the keypoint location depending on its scale, and the gradient magnitude and direction is calculated for each pixel in that region. An orientation histogram with 36 bins covering 360 degrees is created. It is weighted by gradient magnitude and gaussian-weighted circular window with σ equal to 1.5 times the scale of keypoint. The highest peak in the histogram is taken and any peak above 80% of it is also considered to calculate the orientation. This method creates keypoints with the same location and scale, but with possibly different directions (each keypoint is now described as (x, y, σ, θ)). It contributes to the stability of keypoint matching.

4. **Keypoint Description:** now, a keypoint descriptor is created. A 16×16 neighbourhood around the keypoint is taken. It is divided into 16 sub-blocks of 4×4 size. For each sub-block, 8 bin orientation histogram is created. So, a total of 128 bin values are available. It's represented as a vector to form a keypoint descriptor. In addition, several measures are taken to achieve robustness to illumination changes, rotation, etc...

3.3.3 GIST

Another kind of feature descriptor is the GIST. It works as follows:

1. Convolve the image with N Gabor filters at K scales and N-K orientations, producing N feature maps of the same size of the input image.
2. Divide each feature map into 16 regions (by a 4x4 grid), and then average the feature values within each region.
3. Concatenate the 16 averaged values of all N feature maps, resulting in a 16xN GIST descriptor.

3.3.4 MOPS (Multi-scale Oriented Patches)

1. Given a feature (x, y, s, θ), get a 40×40 image patch and sub-sample every 5th pixel (to decrease the resolution, which helps us overcome localization errors).
2. Subtract the mean, and divide by the standard deviation (this makes the descriptor invariant light conditions and intensity changes).
3. Perform Haar Wavelet Transform.

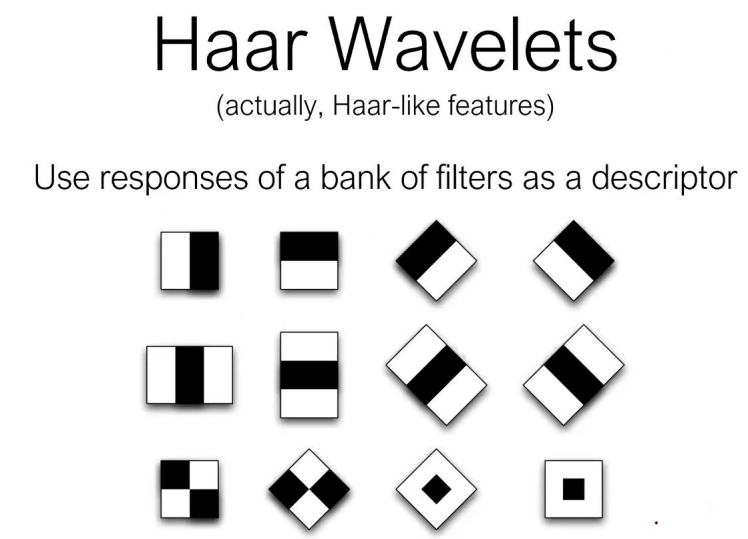


Figure 33: Haar Wavelet Transform. We use the responses of a bank of filters as our descriptor

Performing Haar wavelet response is computationally efficient (we can do it in constant time) and that's why we use it.

3.3.5 BRIEF Descriptor (Binary Robust Independent Elementary Features)

- Select a set of $n = 256$ pixel pairs (x_i, y_i) in patch I (for example, the patch belonging to a detected keypoint), where x_i, y_i are both tuples of pixel coordinates. The list of selected pairs stay constant, as they need to be consistent from image to image.
- Store a binary descriptor: $\rho_n(I, x_n, yx_n) = \begin{cases} 1 & if\ I(x_n) < I(y_n) \\ 0 & otherwise \end{cases}$

This descriptor is useful because it's simple. Note that this descriptor is invariant to affine intensity transformations:

$$\rho_n(I, x_n, y_n) = \rho_n(\alpha I + b, x_n, y_n)$$

This descriptor is not invariant to scale, and not invariant to rotation! But, we can make it scale invariant by first finding the scale of each feature (we can use the Difference of Gaussian pyramid for example) and then performing BRIEF

on the correct scale Laplacian response image of the feature. Also, to make this descriptor robust to noise, we can smooth the image before applying the descriptor.

So, which set of n pixel pairs should we select? We have a couple of options:

- Select the pairs randomly.
- Select the pairs in polar grid.

4 Classification

Image classification is concerned with classifying the types of objects presented in an image: given an image, we want to be able to tell which type of object is in that image (for example, decide if there's a cat or a dog in the image). In image classification, we assume that the possible objects are a finite set of discrete classes ([dog, cat, airplane, baloon, etc...]). This task is quite complicated, as our computer sees an image as an array of numbers, which can change drastically from image to image (even if the images contain the same object).

4.1 Data Driven Approach

The data driven approach is way to perform image classification based on previously classified images:

- Collect a database of preclassified images (images paired with labels).
- Train a machine learning model as an image classifier.
- Evaluate the classifier on test images.

4.1.1 Bag of Features

In this approach, we assume that an image can be described as a collection of local features. We then use this collection of features of each image to classify the image. This approach is invariant to occlusion, is scale invariant, and is rotation invariant. The underlying assumption of this model is that spatial information of local features can be ignored for object recognition.

Our way to represent a data item (document, image, etc...), is to represent it as a histogram of features:

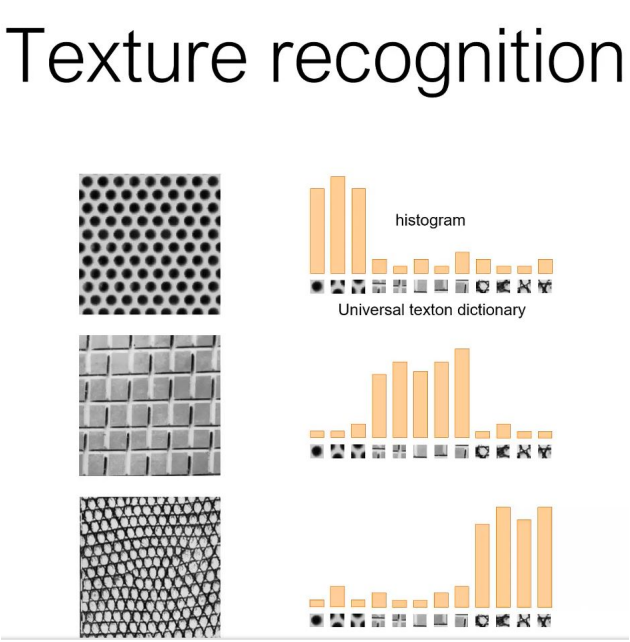


Figure 34: Texture recognition using bag of features

Since this idea is adopted from document classification, it's more natural to explain it using documents. Now, for each data item we have a histogram of features, and to compare one document to another, we want to compare their feature histograms, and to do that we think of their feature histograms as vectors and compare them in vector space:

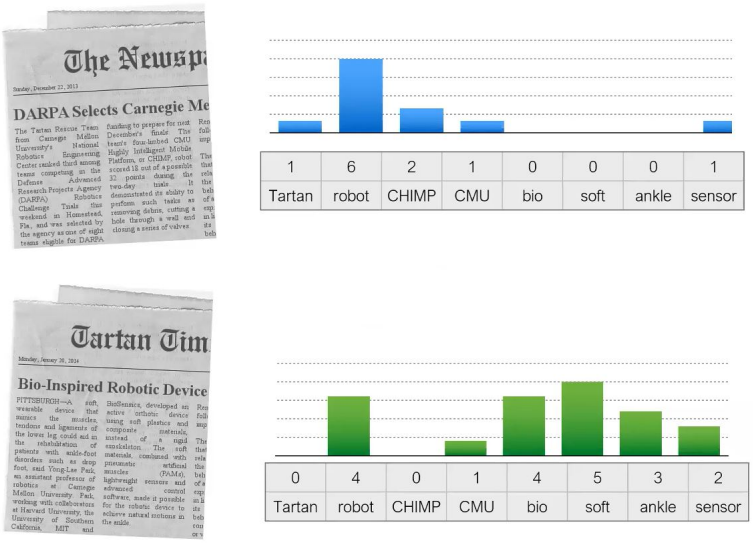


Figure 35: Two examples of feature histogram vectors of text documents. It's common to assume that the order of the words doesn't matter when we classify documents, and so using feature histogram is natural in this case.

A document (data point) is a vector of counts over each word (feature):

$$v_d = \begin{pmatrix} n(w_1, d) & n(w_2, d) & \dots & n(w_T, d) \end{pmatrix}$$

where $n(w_i, d)$ is the number of times the word w_i appears in the document d, and $\{w_i\}_{i=1}^{T}$ is the finite set of words we assume to have in our language dictionary.

To measure similarity between two documents d_1, d_2, we can compare the vectors v_{d_1} to v_{d_2} using some vector distance metric such as the cosine distance:

$$d_{cos}(v_{d_1}, v_{d_2}) = cos(\theta) = \frac{v_{d_1} \cdot v_{d_2}}{||v_{d_1}||\ ||v_{d_2}||}$$

It makes sense to normalize the dot product $v_{d_1} \cdot v_{d_2}$ because we don't want the documents' size to affect our classification decision (we don't wanna be sensitive to the documents' size).

However, it's known for a while that not all words are created equal. Words such as "are" can appear in many different contexts.

4.1.2 TF-IDF (Term Frequency Inverse Document Frequency)

This method takes into account that not all words have the same impact on the classification decision. We can weigh each word with a heuristic:

$$v_d = \begin{pmatrix} \alpha_1 n(w_1, d) & \alpha_2 n(w_2, d) & \dots & \alpha_T n(w_T, d) \end{pmatrix}$$

where $df_i = \sum_{d'} \mathbb{1}_{(w_i \in d')}$ is the number of documents containing the word w_i, D is the total number of documents, and $\alpha_i = \log \frac{D}{df_i}$. Thus, for words w_i that exist in all documents we have $\alpha_i = 0$, and for words w_i that exist in only one document we have $\alpha_i \gg 1$ (common words are irrelevant, and rare words are highly weighed).

4.1.3 Standard Bag of Features Pipeline For Image Classification

1. **Dictionary Learning (learn visual words using clustering):** to find our features dictionary, we first need to extract features from our images. A popular way to do that is to use feature extractors and descriptors such as SIFT, and thus every detected interest point in our images are described as a feature descriptor vector. After we extract features from all of the images in our dataset, we now possibly have a lot of detected features and we need to decide which of these features will be part of our dictionary. One way to do that is to cluster all of the detected features (their descriptors) using K-Means:

   (a) Select initial centroids at random.

   (b) Assign each object to the cluster with the nearest centroid.

   (c) Compute each centroid as the mean of the objects assigned to it. If one of the computed centroid means is different than the previous mean of the centroid, go again to (b). Otherwise, break.

   - Pros of K-means: very simple method. Converges to a local minimum of the error function.

- Cons of K-means: memory-intensive. We need to choose K. Sensitive to initialization. Sensitive to outliers. Only finds "spherical" clusters.

After we cluster our features using K-Means, each centroid is a word in our dictionary. The dictionary should be learned from a training set that is sufficiently representative.

2. **Encode (build bag of features vectors for each image):** On each test image, we detect its features and for each feature we find its corresponding cluster (each image feature gets associated with a cluster (word) that has a mean closest to the feature). We now build a histogram of the image features by counting the number of features belonging to each cluster:

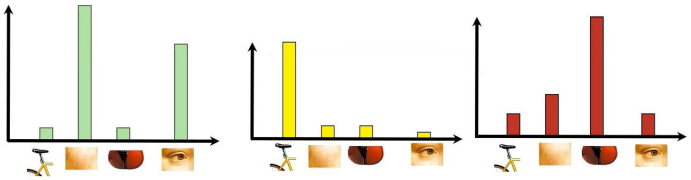


Figure 36: Three Image histograms examples. Each bin in the histogram corresponds to one cluster, which represents a word in our dictionary.

3. **Classify (train a model on the bag of features data and test it):** Now, for each test image we have a histogram of the "words" it contains. There are many ways to classify each of our test images using the histograms we created, and we'll discuss some of these ways in the next sections.

4.1.4 K Nearest Neighbors (KNN)

Suppose we have a training set of (image, class) pairs (for each image in the training set, we know its class), and suppose we have some test image (and we don't know its class). We can look at its K "nearest neighbors" (for example, we can define "near" to be some distance metric in the histogram space we defined before) and choose the test image's class to be the most common class of the image's K nearest neighbors (majority vote). For example, if we have a test image and choose K=5, and the image's 5 nearest neighbors are classified as [1,2,2,2,3], then we choose the test image's class to be 2, since it is the most common class amongst its neighbors.

We can use different distance mertics to measure distance between datapoints, such as:

- Euclidean distance: $d_2(I_1, I_2) = \sqrt{\sum_p (I_1^{(p)} - I_2^{(p)})^2}$
- Manhattan distance: $d_1(I_1, I_2) = \sum_p |I_1^{(p)} - I_2^{(p)}|$
- Locality sensitive distance metrics.

Note that it's important to normalize the data, because different dimensions might have different scales (and we don't want certain dimensions to dominate our classification decision).

Notice that we have some hyperparameters that we need to choose (such as K and the distance metric). So how do we choose them?

- Try out which hyperparameters work best on the test set. This approach is a bad idea because the test set is a proxy for the generalization performance, which means that we'll easily overfit our model on our test data. We should use our test data only at the end - when we evaluate the model.
- **Validation:** we can take our training data and split it to N folds. We then choose one of the folds as our validation data, which we'll use to choose our model's hyperparameters by evaluating the model on this validation data for different choices of our hyperparameters. We choose the hyperparameters that achieve the best performance on the validation data. The rest of the folds (N-1 remaining folds) are our training data. In this approach, we use our test data only at the end - when we evaluate our model.

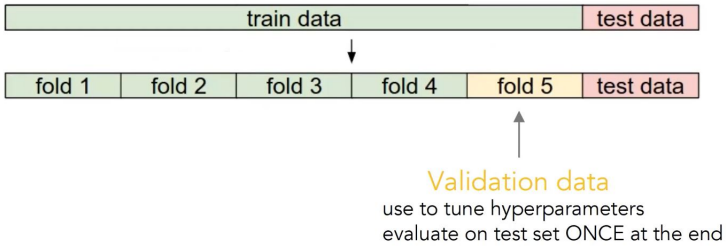


Figure 37: Validation

- **Cross-validation:** it's almost the same thing as validation, but now for each hyperparameters configuration we cycle through the choice of which fold is the validation fold, and we average the the performance results of all validation folds. Then, we choose the hyperparameters that achieved the best average performance on all folds.

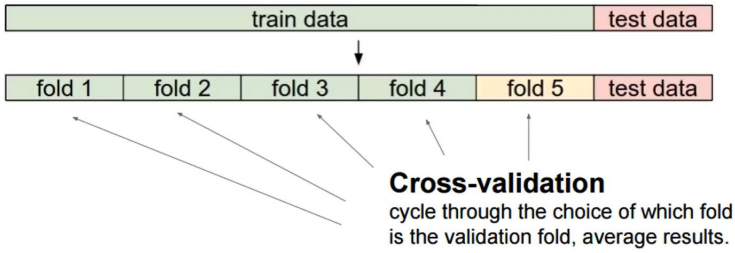


Figure 38: Cross validation

Pros and cons of KNN:

- Pros: simple, yet effective. Fast training time (actually no training at all).
- Cons: search and storage is expensive (testing takes time, and we need to keep all the dataset in memory). Difficulties with high dimensional data.
- Training takes $O(1)$ times. Testing takes $O(MN)$ times, where N is the number of training images, and M is the number of test images. Normally, we want the opposite: testing should be fast and training can be slow.

4.1.5 Naive Bayes

We have a feature descriptor $(x_1, \ldots, x_n)$ (for example, our histogram feature descriptor). The probability that this feature (image) belongs to class y is $p(y|x_1, \ldots, x_n)$. So, our goal is to estimate this probability for each possible class y, and then our classification decision would be the class $\hat{y}$ that maximizes this probability. Naive Bayes methods are a set of supervised learning algorithms based on applying Bayes' theorem with the "naive" assumption of conditional independence between every pair of features given the value of the class variable. Bayes' theorem states the following relationship, given class variable y and dependent feature vector x_1 through x_n:

$$P(y|x_1, \ldots, x_n) = \frac{P(y)P(x_1, \ldots, x_n|y)}{P(x_1, \ldots, x_n)}$$

Using the naive conditional independence assumption that $P(x_i|y, x_1, \ldots, x_n) = P(x_i|y)$, we get:

$$P(y|x_1, \ldots, x_n) = \frac{P(y)\prod_{i=1}^{n} P(x_i|y)}{P(x_1, \ldots, x_n)}$$

Since $P(x_1, \ldots, x_n)$ is constant given the input, we can use the following classification rule:

$$P(y|x_1, \ldots, x_n) \propto P(y)\prod_{i=1}^{n} P(x_i|y)$$

$$\Downarrow$$

$$\hat{y} = \arg\max_y P(y)\prod_{i=1}^{n} P(x_i|y)$$

where $\hat{y}$ is our class prediction. We can use Maximum A Posteriori (MAP) estimation to estimate $P(y)$ and $P(x_i|y)$; the former is the relative frequency of the class y in the training set, and the latter is the relative frequency of the feature x_i amongst all examples of class y in the training set:

$$P(y) \approx \frac{\sum_{k=1}^{N} \mathbb{1}_{(y_k=y)}}{N}$$

$$P(x_i|y) \approx \frac{\sum_{k=1}^{N} \mathbb{1}_{(x_k=x_i, y_k=y)}}{\sum_{k=1}^{N} \mathbb{1}_{(y_k=y)}}$$

where N is the number of training examples.

4.1.6 Hard Margin Support Vector Machine (SVM)

The objective of the SVM algorithm is to find a hyperplane in an N-dimensional space (where N is the number of features) that distinctly classifies our data points.

Assume we have a total of C classes. To classify an image, we can give a score to each class, and then choose the image's class to be the class with the highest score. First, we'll use a linear score function f (a linear classifier):

$$f(x_i, W, b) = Wx_i + b$$

where f gives the score for each class (f results in a vector, where f_c is the score of the class c), x_i is the feature vector of image i (the histogram, for example), W is the classifier's weights, and b is the classifier's bias (these are the parameters of the model). Our goal is to find W and b that gives the correct classes of each image. From now on (for simplicity), we'll assume that $C = 2$, and so we want to separate between two classes:

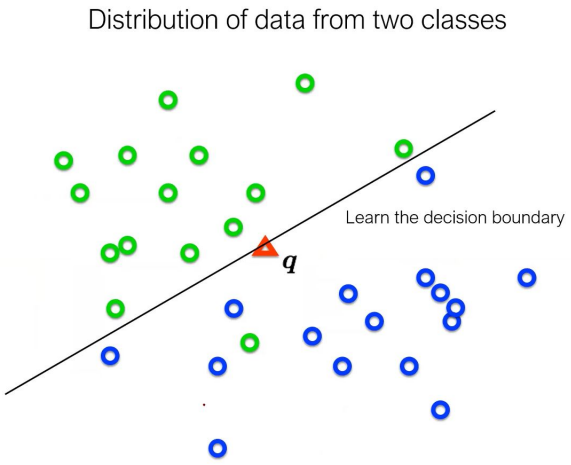


Figure 39: An example of the data distribution of two classes. Our goal is to separate these two classes.

Lets talk about hyperplanes in 2D, which are lines. A line (hyperplane) in 2D can be written as

$$w_1x_1 + w_2x_2 + b = 0 \Rightarrow w^Tx + b = 0, w \in \mathbb{R}^2$$

where the axes are x_1 and x_2. Another way to describe a line is to add a constant 1 to w and to write it as follows:

$$w^Tx = 0, w \in \mathbb{R}^3, w_3 = 1$$

An important property of this representation, is that we are free to choose the scale of w: the lines $w^Tx = 0$ and $\lambda w^Tx = 0$ are exactly the same lines (we can divide by λ). Thus, we'll enforce the constraint $||w|| = 1$ for simplicity.

We know that the distance from a line $w^Tx + b = 0$ to the origin is $\frac{b}{||w||}$. Thus, the distance between the lines $w^Tx + b = 0$ and $w^Tx + b = -1$ is $\frac{1}{||w||}$ (this is also true for $w \in \mathbb{R}^n$), and the distance between the line $w^Tx + b = -1$ and the line $w^Tx + b = 1$ is $\frac{2}{||w||}$ (this will be useful soon).

To separate two data point distributions in 2D (assuming that the data points are linearly separable), we can choose many different lines (with different w) that separates those distributions. These are only two examples:

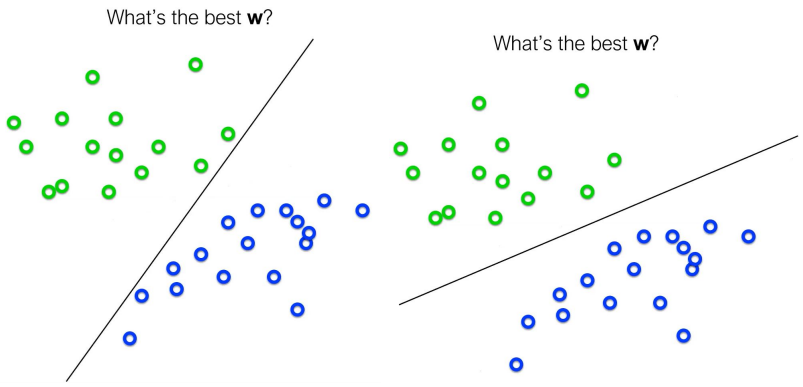


Figure 40: Two different lines that separate data points distributed in 2D space. The data points belong to two classes: the green and the blue classes.

If the training data is linearly separable, we can select two parallel hyperplanes that separate the two classes of data, so that the distance between these hyperplanes is as large as possible (that's the objective of a Support Vector Machine). The region bounded by these two hyperplanes is called the "margin", and the maximum-margin hyperplane is the hyperplane that lies halfway between them. With a normalized or standardized dataset, these hyperplanes can be described by the equations:

$$w^T x - b = 1$$

(anything on or above this boundary is of one class, with label 1) and

$$w^T x - b = -1$$

(anything on or below this boundary is of the other class, with label -1). Geometrically, the distance between these two hyperplanes is $\frac{2}{||w||}$, so to maximize the distance between these hyperplanes we can minimize $||w||$.

The distance is computed using the distance from a point to a plane equation. We also have to prevent data points from falling into the margin, so we add the following constraint: $\forall i = 1, \ldots, N$, if $y_i = 1$ then $w^T x_i - b \geq 1$, and if $y_i = -1$ then $w^T x_i - b \leq -1$. These constraints state that each data point must lie on the correct side of the margin (classified correctly). These constraints can also be written as:

$$\forall i = 1, \ldots, N : y_i(w^T x_i - b) \geq 1$$

We can put all of this together and get the following optimization problem (that's the optimization problem that SVM solves):

$$\min_w ||w||, \ \textit{subject to } \forall i = 1, \ldots, N : y_i(w^T x_i - b) \geq 1$$

The w and b that solve this problem determine our classifier: for a given feature vector x, we classify it as $\hat{y} = sign(w^T x - b)$.

An important consequence of this geometric description is that the maximum margin hyperplane is completely determind by those x_i that lie nearest to it. These x_i are called the support vectors:

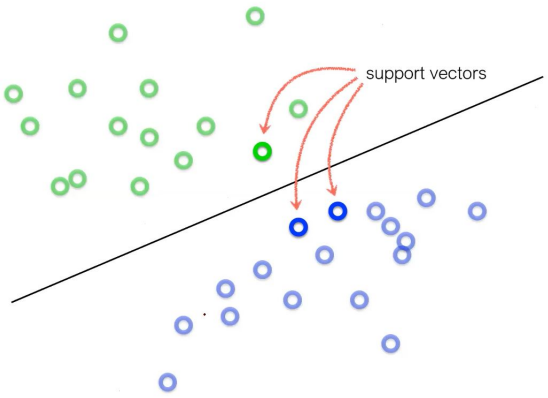


Figure 41: A line and its support vectors.

As we see, the SVM problem formulation is a convex quadratic (QP) problem. A unique solution exists, and there are many efficient algorithms that solves this optimization problem.

4.1.7 Soft Margin Support Vector Machine

This formulation of support vector machines is concerned with the case in which data points are not linearly separable. In this case, we have to allow for some misclassifications. Our new optimization objective is then:

$$\min_w \frac{1}{2}||w||^2 + C\sum_i \xi_i, \ \textit{subject to } \forall i = 1, \ldots, N : y_i(w^T x_i - b) \geq 1 - \xi_i \textit{ and } \xi_i \geq 0$$

where ξ_i are called the "slack variables". Notice that each of the constraints can be satisfied if its slack variabe is large enough. C is a regularization parameter:

- A small C means that we ignore the constraints (large margin).
- A large C means that we penalize misclassifications more (small margin).

This is still a QR problem, and thus we can find its unique solution (for a specific C) using any of the available QR solvers.

5 Convolutional Neural Networks

In deep learning, a convolutional neural network (CNN, or ConvNet) is a class of deep neural networks, most commonly applied to analyzing visual imagery. They are also known as shift invariant or space invariant artificial neural networks (SIANN), based on their shared-weights architecture and translation invariance characteristics. They have applications in image and video recognition, recommender systems, **image classification**, medical image analysis, natural language processing, and financial time series.

CNNs are regularized versions of multilayer perceptrons. Multilayer perceptrons usually mean fully connected networks, that is, each neuron in one layer is connected to all neurons in the next layer. The "fully-connectedness" of these networks makes them prone to overfitting data. Typical ways of regularization include adding some form of magnitude measurement of weights to the loss function. CNNs take a different approach towards regularization: they take advantage of the hierarchical pattern in data and assemble more complex patterns using smaller and simpler patterns. Therefore, on the scale of connectedness and complexity, CNNs are on the lower extreme.

Convolutional networks were inspired by biological processes in that the connectivity pattern between neurons resembles the organization of the animal visual cortex. Individual cortical neurons respond to stimuli only in a restricted region of the visual field known as the receptive field. The receptive fields of different neurons partially overlap such that they cover the entire visual field.

CNNs use relatively little pre-processing compared to other image classification algorithms. This means that the network learns the filters that in traditional algorithms were hand-engineered. This independence from prior knowledge and human effort in feature design is a major advantage.

5.1 General Architecture

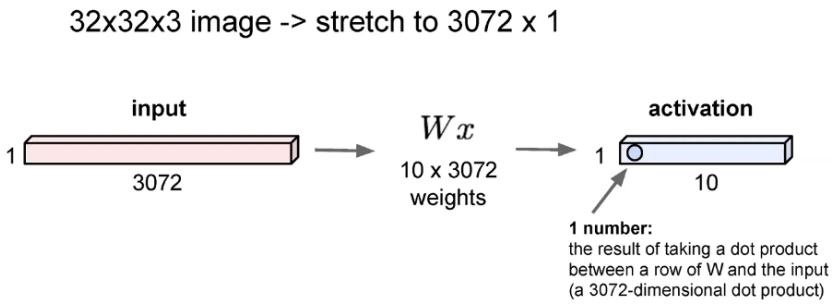
A convolutional neural network consists of an input and an output layer, as well as multiple hidden layers. The hidden layers of a CNN typically consist of a series of convolutional layers that convolve with a multiplication or other dot product. The activation function is commonly a RELU layer, and is subsequently followed by additional convolutions such as pooling layers, fully connected layers and normalization layers, referred to as hidden layers because their inputs and outputs are masked by the activation function and final convolution.

Though the layers are colloquially referred to as convolutions, this is only by convention. Mathematically, it is technically a sliding dot product or cross-correlation. This has significance for the indices in the matrix, in that it affects how weight is determined at a specific index point.

5.2 Types of Layers

5.2.1 Fully Connected Layer

A fully connected layer receives a vector input, performs linear transformation on the vector, and then possibly adds bias, and then applies non-linearity on the outputs. More concretely, on an input vector x, a fully connected layer performs: $out = f(Wx + b)$ where f is an activation function, b is the bias, and W is the weights matrix.



Suppose that $x \in \mathbb{R}^n$, and that the fully connected layer has m neurons. Then, the total number of learnable parameters is $n \cdot m + m$.

5.2.2 Convolutional Layer

The input to a convolutional layer is a tensor with shape $H_{in} \times W_{in} \times C_{in}$ where C_{in} is the number of input channels (RGB for example), and H_{in} and W_{in} are the height and width of the matrix of each channel, respectively. Then, after passing through the layer, the tensor becomes a stack of what is called "feature maps", where the stack's shape is $H_{out} \times W_{out} \times C_{out}$. C_{out} is the number of filters (number of feature maps) in the layer (will be explained soon), and H_{out} and W_{out} are the height and width of each feature map, respectively.

The operation of convolutional layer is actually to perform C_{out} cross-correlation operations on the input tensor. The layer consists of C_{out} filters (kernels), each with size $H_k \times W_k \times C_{in}$. Each of these kernels perform a 3D cross-correlation operation on the input tensor, and the results from all the kernels are stacked together to form an output tensor, and as we said, this is a tensor of stacked feature maps.

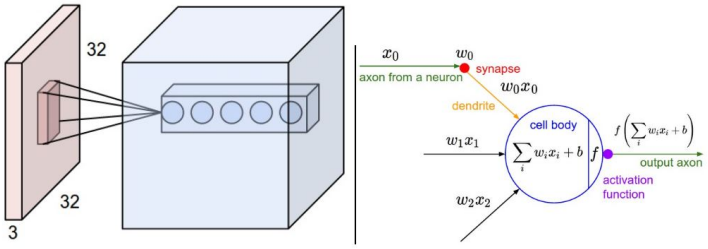


Figure 42: The convolutional layer operation. The shape of the input tensor in this case is $32 \times 32 \times 3$.

We can think of the cross-correlation operation each filter performs as follows:

1. Take a 3D patch of size $H_k \times W_k \times C_{in}$ from the input tensor, and flatten it.
2. Take a convolutional layer filter and flatten it.
3. Perform a dot product between the two, and return the result.

The result is one element in the output tensor, as can be seen in Figure 42. Just think of it as a 2D cross correlation operation, with an additional dimension - the depth/channel dimension.

As we said, each cross-correlation output result is one element in the output tensor. As the filter passes through the height and width dimensions of the input tensor, the output elements of each filter operation results in the output 2D matrix.

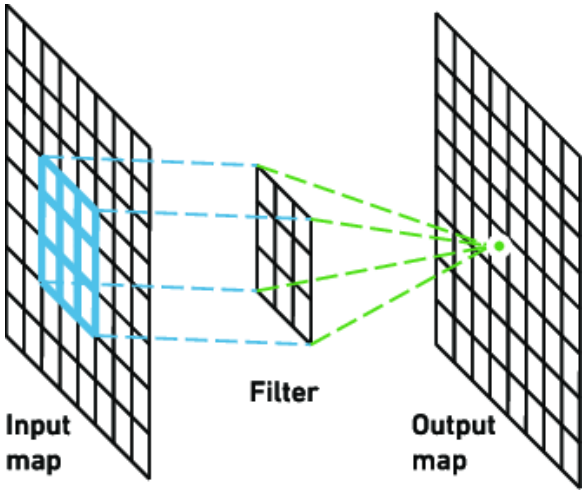


Figure 43: The output of each filter is a matrix. Here, the input tensor doesn't have depth: it's also a 2D matrix, but this is only to illustrate the output.

Additionally, each filter can also add bias to each of its "convolution" results, and so each filter can also hold a bias parameter. As in a fully connected layer, the resulting output (the convolution result with the added bias) can pass through a non-linear activation function, which is typically a ReLU activation function.

As for the number of learnable parameters: suppose a convolutional layer consists of C_{out} filters, each of size $H_k \times W_k \times C_{in}$. Thus, the number of parameters in all filters is $C_{out} \cdot H_k \cdot W_k \cdot C_{in}$. As we said, each filter might also have an additional bias, and so we get a total of $C_{out} \cdot (H_k \cdot W_k \cdot C_{in} + 1) = C_{out} \cdot H_k \cdot W_k \cdot C_{in} + C_{out}$ learnable parameters, as C_{out} is the number of required bias parameters.

A convolutional layer has hyperparameters such as stride, padding, dilation, and more. We won't explain it in detail here, but the size of each dimension in the output result of a convolutional layer (height or width) can be calculated as follows:

$$D_{out} = \frac{D_{in} - D_k + 2p}{s} + 1$$

where D_{out} is the output dimension (for example W_{out} or H_{out}), D_{in} is the input dimension (for example W_{in} or H_{in}), D_k is the kernel size along the dimension D (along the width or height), p is the padding hyperparameter, and s is the stride hyperparameter. More concretely, we have:

$$W_{out} = \frac{W_{in} - W_k + 2p}{s} + 1$$

$$H_{out} = \frac{H_{in} - H_k + 2p}{s} + 1$$

5.2.3 Pooling Layer

It is common to periodically insert a Pooling layer in-between successive convolutional layers in a ConvNet architecture. Its function is to progressively reduce the spatial size of the representation to reduce the amount of parameters and computation in the network, and hence to also control overfitting. The Pooling Layer operates independently on every depth slice of the input and resizes it spatially, using the MAX operation for example. The most common form is a pooling layer with filters of size 2x2 applied with a stride of 2 downsamples every depth slice in the input by 2 along both width and height, discarding 75% of the activations. Every MAX operation would in this case be taking a max over 4 numbers (little 2x2 region in some depth slice). The depth dimension remains unchanged. More generally, the pooling layer:

- Receives an input tensor of size $H_{in} \times W_{in} \times C_{in}$.
- Requires two hyperparameters: the spatial extent F, and the stride S.
- Produces an output tensor of size $H_{out} \times W_{out} \times C_{out}$, where:
  - $W_{out} = \frac{W_{in} - F}{S} + 1$.
  - $H_{out} = \frac{H_{in} - F}{S} + 1$.
  - $C_{out} = C_{in}$.

- Introduces **zero** parameters since it computes a fixed function of the input.
- For pooling layers, it's not common to pad the input using zero-padding.

It's worth noting that there are only two commonly seen variations of the max pooling layer found in practice: a pooling layer with $F = 3$ and $S = 2$ (also called overlapping pooling), and more commonly $F = 2$ and $S = 2$. Pooling sizes with larger receptive fields are too destructive.

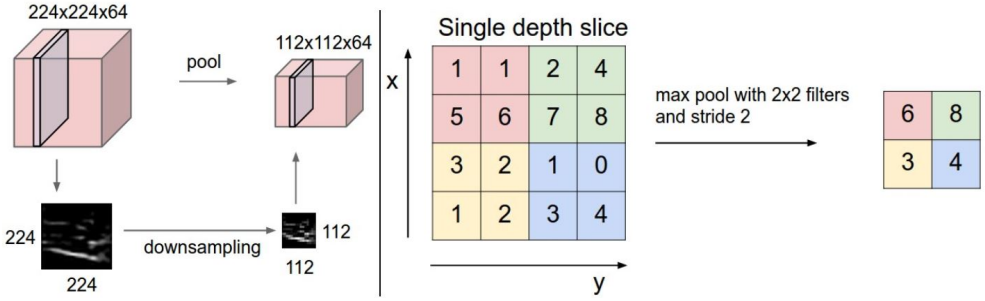
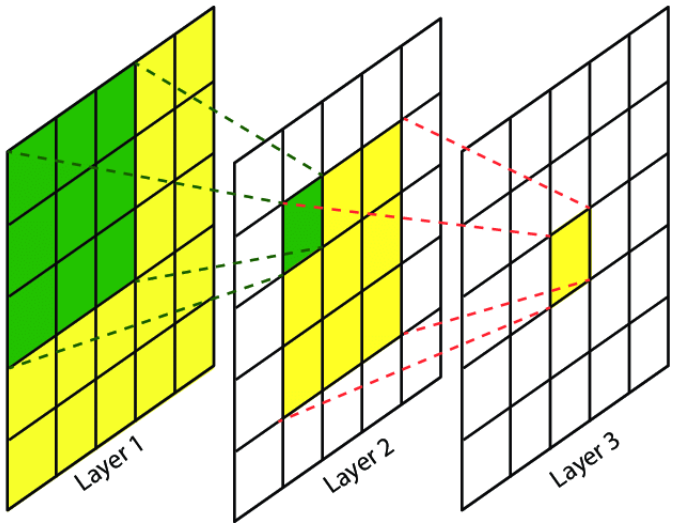


Figure 44: An example of the pooling operation.

5.3 The Receptive Field

Suppose we have a multi layer neural network, with some input tensor. The receptive field of a neuron in some layer is defined as the number of neurons in the input tensor affecting it. We can also define the receptive field of a neuron with respect to part of the network, and not all the way to the beginning of it. In a convolutional layer, the receptive field of a neuron is defined as the number of spatial neurons in the input tensor affecting it (note that this depends on the stride!). For example, suppose we have a convolutional neural network with 3 layers (the first layer is the input layer), where each intermediate layer has a $1 \times 3 \times 3$ sized filter, operates with stride 1:



Thus, each neuron in the second layer has a receptive field of $3 \times 3 = 9$ (because it's defines with respect to the spatial neurons only, and not with respect to the input channels). Also, each neuron in the third layer has a receptive field of $7 \times 7 = 49$, and so on. We came up with the following formula, although we are not sure it's correct: $receptive\ field\ at\ layer\ \ell = (K + 2(\ell - 2))^2$, where ℓ is the layer index (starting from layer 2), and K is the kernel size.

Thus, a two layers conv net with a 7×7 filter (the first layer is the input layer) has the same receptive filed as a 4 layers conv net with 3×3 filters. But, in the 2 layers conv net we have $7 \cdot 7 = 49$ learnable parameters (without considering biases), and in the 3 layers conv net we have $3 \times 3 \times 3 = 27$ learnable parameters, so we got the same effective receptive filed with a lot less learnable parameters.

We can therefore see two advantages of working with smaller filters and deeper layers:

1. The number of learnable parameters is small.
2. The deeper the network, the large the receptive field.
3. We can apply more non-linearities on deeper networks.

6 Segmentation

Image segmentation is the process of partitioning a digital image into multiple segments (sets of pixels, also known as image objects). The goal of segmentation is to simplify and/or change the representation of an image into something that is more meaningful and easier to analyze. Image segmentation is typically used to locate objects and boundaries (lines, curves, etc.) in images. More precisely, image segmentation is the process of assigning a label to every pixel in an image such that pixels with the same label share certain characteristics.

The result of image segmentation is a set of segments that collectively cover the entire image, or a set of contours extracted from the image. Each of the pixels in a region are similar with respect to some characteristic or computed property, such as color, intensity, or texture. Adjacent regions are significantly different with respect to the same characteristic(s). When applied to a stack of images, typical in medical imaging, the resulting contours after image segmentation can be used to create 3D reconstructions with the help of interpolation algorithms like marching cubes.

There are mainly two approaches for segmentation:

- Bottom-top: group pixels that look alike.
- Top-bottom: group pixels that likely belong to the same object.

So what is a good segmentation? There is no single answer.

The first segmentation evaluation idea is to compare a segmentation created by humans (ground truth) to a segmentation created by some algorithm (for the same image), but this is still controversial as different people might segment the same image differently. To compare a segmentation created by an algorithm to its "ground truth" segmentation, we take the segmentation boundaries found by the algorithm, and for each pixel in the segmentation boundary we check its closeness to the true boundary. We then check how many of the boundary pixels are aligned with the correct boundary:

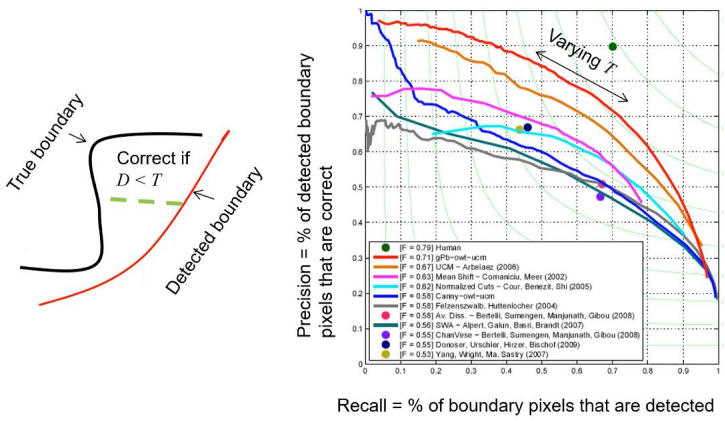


Figure 45: The boundary agreement graph. The threshold T defines the minimum pixel distance to decide if a pixel in on the true boundary or not.

So, a "better" algorithm achieves a higher boundary agreement.
Another way to evaluate a segementation result is to check the overlaps between the segments in the algorithm's segmentation result and the segments in the ground truth.

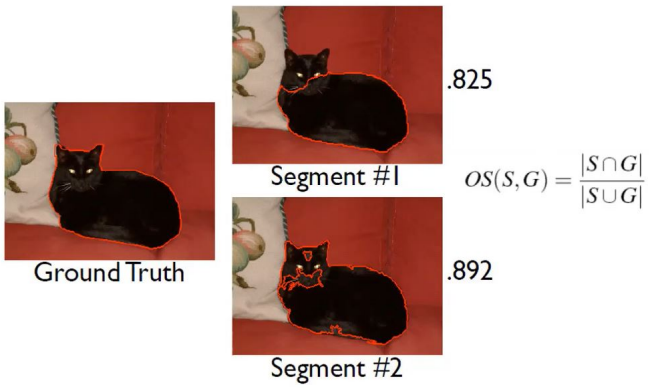


Figure 46: Two segments created by some algorithm, and the overlap result with the ground truth. S is the set of pixels belonging to the segment in the algorithm's result, and G is the set of pixels belonging to the ground truth segment.

6.1 Segmentation Methods

6.1.1 Multiple Segmentation

Almost every segmentation algorithm doesn't have a single correct result. Thus, we can run any segmentation algorithm multiple times under different configurations, hyperparameters, etc..., and now search for some consensus at all segmentation results (such as looking for segmented parts of the image that is repetitively segmented in all of the results).

6.1.2 Clustering

Each pixel in the image has RGB values, so we can think of it as a vector in $\mathbb{R}^3$. Then, we can use any clustering algorithm (such as K-means - we already talked about it) to cluster the pixels, where each cluster represents a segment in the image.

This idea is very naive and has a lot of drawbacks, such as that it doesn't take into account "closeness" between pixels: pixels in very far locations in the image can be in the same segment! (this doesn't make sense). To "fix" it we can add the location coordinates to each pixel (now each pixel is a vector of RGB values and (x, y) location) and then cluster the pixels. This it a bit better but still not very useful.

- Pros: simple and fast. Easy to implement.
- Cons: might be difficult to choose K. Sensitive to outliers.
- It's rarely used for pixel segmentation.

6.1.3 Mean Shift

This is another yet versatile clustering-based method to perform segmentation. In this method, we think of each pixel as a vector in $\mathbb{R}^n$. This vector might contain the RGB values of the pixel, its location, and more information such as texture, dominant gradient, etc...

In this method, we try to find "modes" (dense regions) in $\mathbb{R}^n$, in which many pixel vectors lie:

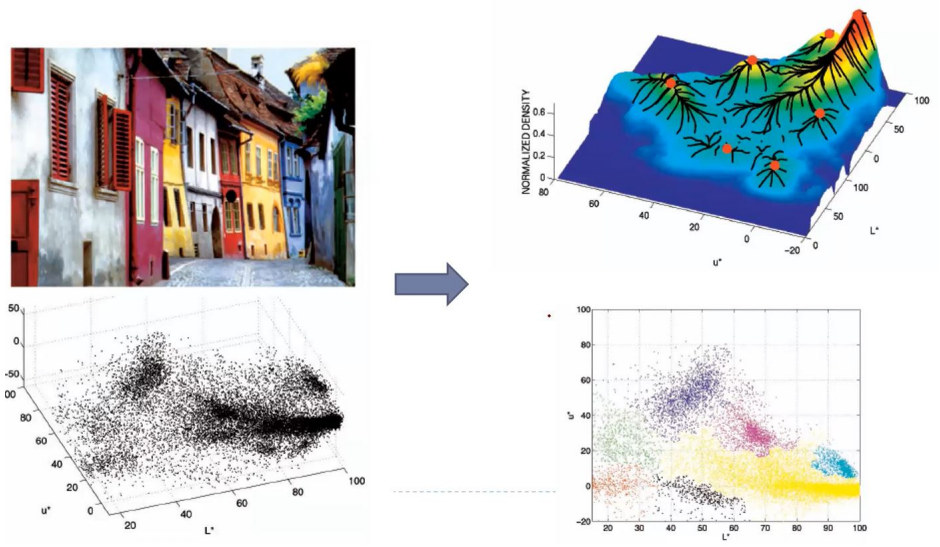


Figure 47: An example of pixels density in vector space. In this algorithm, we are trying to look for regions that are highly densed with pixels (these densed regions are called modes).

Mean shift is a procedure for locating the maxima — the modes — of a density function given discrete data sampled from that function. This is an iterative method, and we start with an initial estimate x (the mode location in the vector space). Let a kernel function $K(x_i - x)$ be given. This function determines the weight of nearby points for re-estimation of the mean. Typically a Gaussian kernel on the distance to the current estimate is used: $K(x_i - x) = e^{-c||x_i - x||^2}$. The weighed mean of the density in the window determined by K is:

$$m(x) = \frac{\sum_{x_i \in N(x)} K(x_i - x) x_i}{\sum_{x_i \in N(x)} K(x_i - x)}$$

where $N(x)$ is the neighborhood of x, a set of points for which $K(x_i) \neq 0$, and $m(x)$ is the "center of mass" of the neighborhood of x.

The difference $m(x) - x$ is called the **mean shift**. The mean shift algorithm now sets $x \leftarrow m(x)$ (sets x to be the new center of mass of its neighborhood), and repeats the estimation until $m(x)$ converges ($x = m(x)$).

So, how can we use mean shift for segmentation? Lets first define an important concept - **Attraction basin**: Attraction basin is the region for which all trajectories lead to the same mode. This is a region of points such that if we run mean shift on each point in that region, the final mean (mode) will end up to be the same. A cluster can then be defined as all data points in the attraction basin of a mode. We can run mean shift on each point, and decide that its cluster is the one corresponding to the mode of the point's attraction basin:

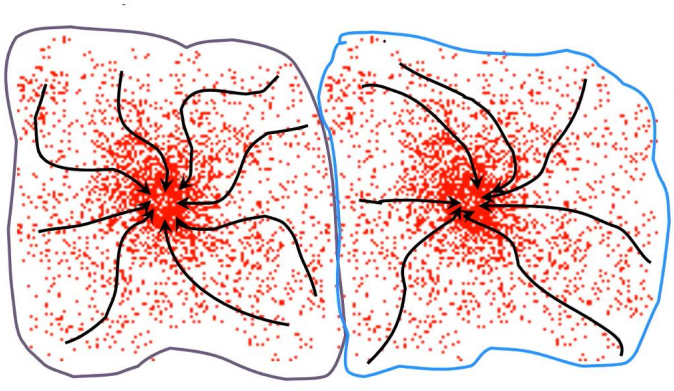


Figure 48: Example of two clusters corresponding to two attraction basins.

- Pros: good general purpose segmentation. Flexible in number and shape of regions. Robust to outliers. General mode-finding algorithm (useful for other problems such as finding most common surface normals).
- Cons: we have to choose the kernel size in advance, and also the type of kernel. Not suitable for high-dimensional features.

- When to use it: oversegmentation, multiple segmentations, tracking, clustering, and filtering applications.

1. Pros: General, and application-independent. Model-free (does not assume spherical clusters). Finds variable number of modes. Robust to outliers.
2. Cons: Output depends on window size. Computationally expensive. Does not scale well with dimension of feature space.

6.2 Over-Segmentation Methods

In over segmentation, our goal is to divide the image into a large number of regions. First, we'll talk about a concept called "superpixels"

A superpixel can be defined as a group of pixels that share common characteristics (like pixel intensity). Superpixels are becoming useful in many Computer Vision and Image processing algorithms like Image Segmentation, Semantic labeling, Object detection and tracking etc because of the following:

- They carry more information than pixels.
- Superpixels have a perceptual meaning since pixels belonging to a given superpixel share similar visual properties.
- They provide a convenient and compact representation of images that can be very useful for computationally demanding problems.

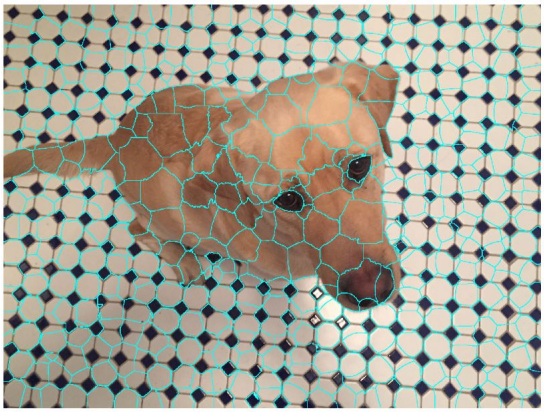


Figure 49: An example of superpixels of an image (the superpixels are bounded in cyan color)

In the next sections, we'll discuss some of the algorithms that segments images to superpixels. Note that in all over-segmentation methods, we can control the "amount" of over segmentation (reduce the number of segments) by blurring the image first (for example, by using a Gaussian or median filter).

6.2.1 Watershed Segmentation

A watershed is a transformation defined on a grayscale image. The name refers metaphorically to a geological watershed, or drainage divide, which separates adjacent drainage basins. The watershed transformation treats the image it operates upon like a topographic map, with the brightness of each point representing its height, and finds the lines that run along the tops of ridges:

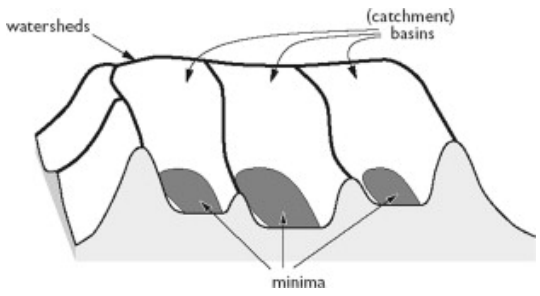


Figure 50: Watershed lines demonstration

There are a bunch of watershed segmentation algorithms, and we'll learn one of them - **Meyer's Flooding Algorithm**:

1. Find all of the local minimum points in the image (using the gradient image). Mark each of these points with a unique label.
2. Add the neighbors of the local minimum points to a shared priority queue, sorted by their pixel values.
3. Extract the top priority pixel from the queue (the pixel with the lowest intensity value).
   - If all of its labeled pixel neighbors are marked with he same label, then assign that label to the pixel.
   - Otherwise, add all of its non-labeled neighbors to the queue (in this case, at least two neighbors of the pixel must have different labels, which means that this pixel is on a boundary. Notice that every pixel extracted from the priority queue must have at least one labeled neighbor).
4. Repeat step 3 until all pixels passed through the queue.
5. The non-labeled pixels are then the watershed lines.

- Pros: fast, and preserves boundaries.
- Cons: only as good as the soft boundaries (which may be slow to compute). Not east to get a variety of regions for multiple segmentations.

Watershed can be used a starting point for hierarchical segmentation algorithm (or in general as a starting point for more advanced segmentation methods).

6.2.2 Felzenszwalb and Huttenlocher's Graph Based Segmentation

This is a graph based over segmentation algorithm, and it works as follows:

1. The input is a graph $G = (V, E)$, with n vertices and m edges. Each vertex is a pixel, and each edge connects adjacent pixels. The weight on each edge is the gradient value between the adjacent pixels. The output is a segmentation of V into components $S = (C_1, \ldots, C_r)$.
2. Sort E into $\pi = (o_1, \ldots, o_m)$, by non-decreasing edge weight.
3. Start with a segmentation S^0, where each vertex v_i is in its own component.
4. Repeat step 5 for $q = 1, \ldots, m$.
5. Construct S^q given S^{q-1} as follows: Let v_i and v_j denote the vertices connected by the q_{th} edge in the ordering, i.e., $o_q = (v_i, v_j)$. If v_i and v_j are in disjoint components of S^{q-1} and $w(o_q)$ is small compared to the internal difference of both of those components, then merge the two components. Otherwise, do nothing. More formally, let C_i^{q-1} be the component of S^{q-1} containing v_i and C_j^{q-1} be the component containing v_j. If $C_i^{q-1} \neq C_j^{q-1}$ and $w(o_q) \leq MInt(C_i^{q-1}, C_j^{q-1})$ then S^q is obtained from S^{q-1} by merging C_i^{q-1} and C_j^{q-1}. Otherwise, $S^q = S^{q-1}$.
6. Return $S = S^m$

- Pros: good for thin regions. Fast. Easy to control coarseness of segmentations. Can include both large and small regions.
- Cons: often creates regions with strange shapes. Sometimes makes very large errors.

6.3 Graph Based Segmentation Methods

We can think of the image as a graph. Each pixel is a vertex in the graph, and two vertices are connected with an edge if their corresponding pixels are neighbors in the image (possibly not only direct neighbors, but maybe also connected to the neighbors of the neighbors, or "sufficiently" close pixels). Each edge is weighed with the affinity or similarity of the two vertices it connects. Now, we are trying to break the graph into connected components. These components will be the segmentation of the image. To segment the image by partitioning the graph, we aim to break the connections between components that have low affinity between them: similar pixels should be in the same segments, and dissimilar pixels should be in different segments.

We have many ways to measure affinity. Some of them are:

- Distance affinity: $aff(x, y) = e^{-\frac{1}{2\sigma_d^2}||x-y||^2}$
- Intensity affinity: $aff(x, y) = e^{-\frac{1}{2\sigma_d^2}||I(x)-I(y)||^2}$

- Color affinity: $aff(x,y) = e^{-\frac{1}{2\sigma_d^2} dist(c(x)-c(y))^2}$ where $dist$ is some suitable color space distance.
- Texture affinity: $aff(x,y) = e^{-\frac{1}{2\sigma_d^2} ||f(x)-f(y)||^2}$ where f results in a vector of filter outputs.

Notice that the scale σ_d affects the affinity:

- Small σ_d: group only nearby points (nearby in affinity).
- Large σ_d: group far-away points.

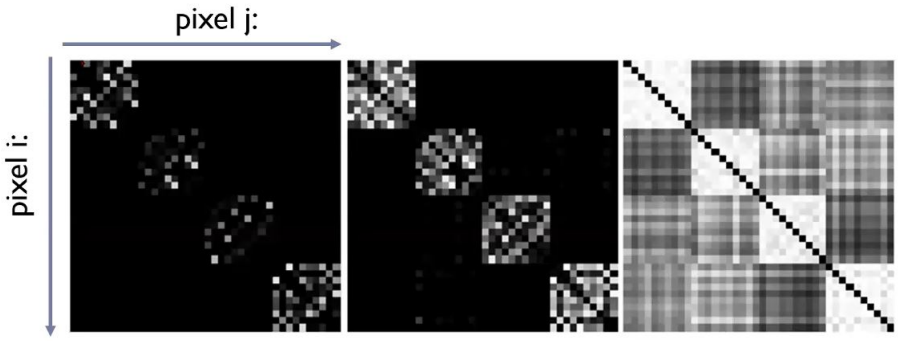


Figure 51: An example of three affinity matrices, for three different choices of σ_d. In each matrix, element (i,j) corresponds pixel i and pixel j, and the value of that element in the matrix is the affinity between these pixels: $aff(i,j)$. The order of the pixels are sorted with the order of the components: the first k_1 pixels are of the first component, the next k_2 pixels are of the second component, etc... Therefore, we can see that "close" pixels have high affinity between them, because they belong to the same component.

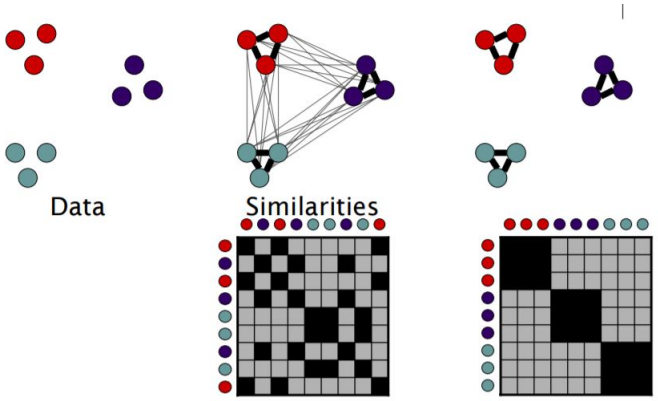


Figure 52: Another example of a simpler affinity matrix. The affinity matrix is the matrix on the right, after we sort the elements component by component. We can see that since we ordered the elements component by component, the resulting matrix is divided to blocks.

6.3.1 Normalized Cuts

To talk about normalized cuts, we'll first define **a graph cut**.

A graph cut is a set of edges whose removal makes the graph disconnected. The cost of a cut is the sum of weights of the cut edges. This means that a graph cut partitions the graph, and gives us a segmentation. So, what's a good graph cut and how do we find one?

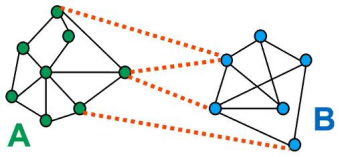
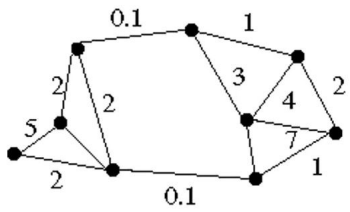
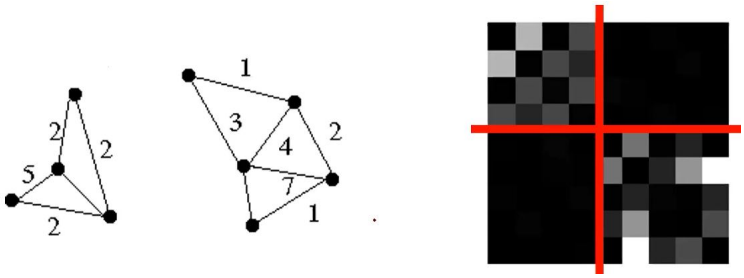


Figure 53: The graph cut is marked in dashed red lines.

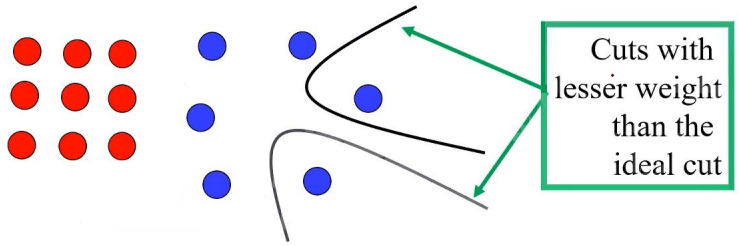
We can segment the graph by finding its minimum cut (there are efficient algorithms that do this, such as Ford Fulkerson's), while we still define the weight of each weight as some affinity between the pixels that correspond to that edge. For example, suppose we have the following graph:



When we remove all the edges in its minimum cut, we get the following components and corresponding affinity matrix (the top left block in the matrix correspond to the left graph component, and to bottom right block in the matrix correspond to the right graph component):



The minimum cut drawback is that it tends to cut off very small, isolated components:



Thus, we finally arrive to the idea of **Normalized Cut**:

Instead of searching for a cut with minimal cost (the cost is the sum of weights of all edges in the cut), we now search for a cut with minimal normalized cost. The normalized cost of a cut is defined as follows:

- $cut(A,B) = sum\ of\ weights\ of\ all\ edges\ between\ the\ components\ A\ and\ B.$
- $assoc(A,V) = sum\ of\ weights\ of\ all\ edges\ in\ A.$
- The normalized cut cost is then $Ncut(A,B) = \frac{cut(A,B)}{assoc(A,V)} + \frac{cut(A,B)}{assoc(B,V)}$

So, we are searching for a cut between A and B such that $Ncut(A,B)$ is minimal.

Searching for a normalized cut is $NP-Complete$, so we don't know of an efficient algorithm that finds it, but we can approximate it.

Normalized Cut As a Generalized Eigenvector Problem:

Let W be the adjacency matrix of the graph, and let D be the diagonal matrix that satisfies the following: $D_{ii} = \sum_j W_{ij}$. Let y be the cluster vector of a component A: $y_p = \begin{cases} 1 & p \in A \\ -1 & otherwise \end{cases}$. Then, the normalized cut cost can be written as:

$$Ncut(A,B) = \frac{y^T(D-W)y}{y^TDy}$$

As we said, this is an $NP-Complete$ problem, so to solve it we can relax y to take on arbitrary (continuous) values (and not only discrete values). Then, this can be solved as a generalized eigenvalue problem:

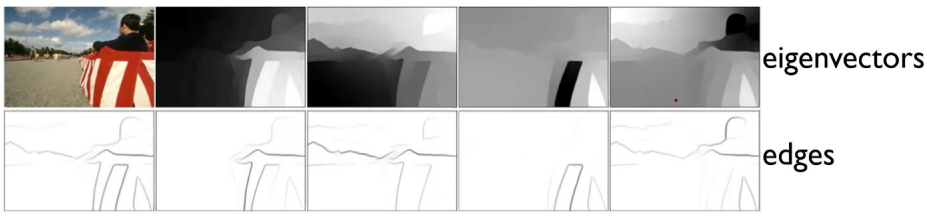
$$(D-W)y = \lambda Dy$$

The solution y to the above problem is given by the eigenvector corresponding to the second smallest eigenvalue. To convert the achieved continuous result to a discrete result (we have to do it, so that we can partition the graph to components), we can simply put a threshold on the values of y (everything under 0 belongs to B, and everything above 0 belongs to A). Thus we arrive to the following **Normalized Cut Algorithm**:

1. Construct a weighted graph $G = (V, E)$ from an image: connect each pair of pixels (i, j) with an edge, and assign weights $w(i, j) = aff(i, j)$.
2. Compute the diagonal matrix D, where $D_{ii} = \sum_j w(i, j)$.
3. Solve $(D - W)y = \lambda Dy$ for the eigenvector with the second smallest eigenvalue.
4. Threshold the eigenvector to get a discrete cut.

- Pros: generic framework, can be used with many different features and affinity formulations. Doesn't require data model or data distribution.
- Cons: high storage requirement and time complexity. Biased towards partitioning into equal segments.

Another usage of the normalized cut algorithm is to use it as an edge detector: we can reshape the found eigenvectors into images. Then, at each pixel, we can compute the probability that it belongs to an edge. Finally, for each pixel we sum the probabilities found in all eigenvectors images, and the result is an image containing the edges of the original image:



6.3.2 Energy Minimization

This method is useful when we want to segment foreground and background, and have priors on how they look. Specifically, we should have some kind of notion of the probability function over labels given a pixel value. We should also have a marginal probability function of the labels of neighboring pixels.

The goal is to minimize the following energy function:

$$E(y;) = \sum_i \Psi_1(y_i;) + \sum_{(i,j)\in edges} \Psi_2(y_i, y_j;)$$

Ψ_1 is usually $-\log P(y_i)$ and Ψ_2 can be $K \cdot XOR(y_1, y_2)$ where $K > 0$. In order to solve this problem, we create a graph where each pixel is a node and every node is connected to two special nodes: the source and the sink. Then, our task is to find a graph cut such that the source is in one set of nodes and the sink is in the remaining set of nodes. All the pixels in the source's set are labeled 0 and all the pixels in the sink's set are labeled 1. The trick is, that the edge weight between pixels is exactly the cost of them having different labels (Ψ_2), and the the weight between a pixel and the source or the sink is according to Ψ_1. Now, the problem of energy minimization can be solved as a min-cut problem.

6.3.3 GrabCut

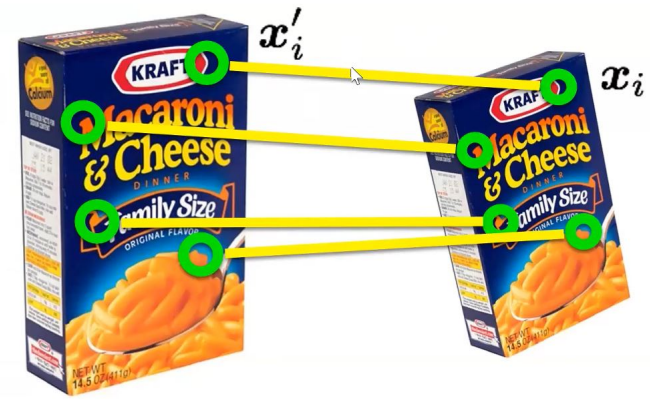
GrabCut is an algorithm that proposes how to define Ψ_1 in the energy minimization problem. First, the use draws a recrangle around the foreground object. Then, a GMM (Gaussian Mixture Model) is constructed for all the pixels in the drawn rectangle, where the GMM is defined over the color space of these pixels. This constructed GMM serves as the data model of the foreground. Similarly, a GMM is also constructed for the pixels outside the drawn rectangle, which serves as the background data model. Then, the min-cut problem is solved in order to find a segmentation. This segmentation is then used to construct a better foreground and background models, and this process continues iteratively.

7 Image Geometry

In this section, we'll mainly talk about image transformations. Recall that we can think of an image (gray scale) as a 2D function $f(x, y)$. We can mainly do two types of image transformations:

- Filtering (blurring, etc...), which we talked about before. This type of transformations change the *range* of the image function.
- Warping (rotation, scale, etc...), which we'll talk about now. This type of transformations change the *domain* of the image function.

So, why do we need image warping? Suppose we have two images of the same object, but the object is pictures from different point of views. How can we match points/pixels/features in both of these images?



Given a set of matched feature points $\{x_i, x_i^{'}\}_{i=1}^{N}$, where x_i are feature points in image 1 and $x_i^{'}$ are the corresponding matched points in image 2, we want to find a transformation:

$$x' = f(x|p)$$

where f is the transformation function, and p is some set of parameters that define f. For a given kind of transformation f, we want to find the best parameters p.

One example in which image transformations are useful is when we want to create panorama images. To create a panorama images, we can either use a very wide lens, or we can find transformations between multiple images from different point of views and then stitch those images. Using a wide lens might be beneficial because everything is done optically in a single image capture. But, wide lens is super expensive and bulky, and there's a lot of distortion. Thus, taking multiple images from different point of views is usually preferred. So, how do we stitch images from different viewpoints? We can use a special kind of image transformations called **homographies**, which we'll talk about later.

In general, there are 6 kinds of image domain transformations:

- Translation.
- Rotation.
- Aspect.
- Affine.
- Perspective.
- Cylindrical.

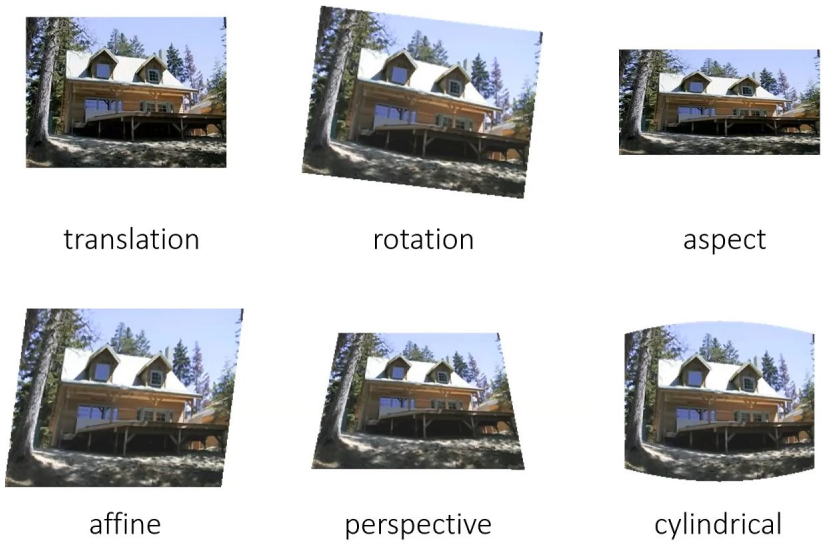


Figure 54: The general 6 kinds of image domain transformations.

Lets start talking about domain transformations.

7.1 Transformations In Heterogeneous Coordinates

7.1.1 Aspect (scale)

To perform aspect transformation, we can simply do the following: suppose we have a pixel at the coordinates (x, y) in the original image. We can change the aspect of that image by performing:

$$x' = ax,\ y' = by$$

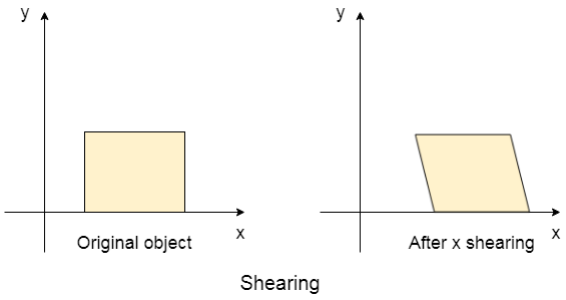
So:

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = S \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} a & 0 \\ 0 & b \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix}$$

where S is the aspect (scale) transformation.

7.1.2 Shear

Shearing transformation is when we want to take a rectangle and transform it to a parallelogram:



We can implement shearing in the following way:

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} 1 & a \\ b & 1 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix}$$

where a is the shearing in x, and b is the shearing in y.

7.1.3 Rotation

Rotation can be implement as follows:

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} cos(\theta) & -sin(\theta) \\ sin(\theta) & cos(\theta) \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix}$$

Thus far we have seen some basic transforamtions defined by a 2×2 matrix M. Other than what we have seen, we can also flip across the origin:

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} -1 & 0 \\ 0 & -1 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix}$$

and also flip across each of the axes (for example, across y):

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} -1 & 0 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix}$$

7.1.4 Translation

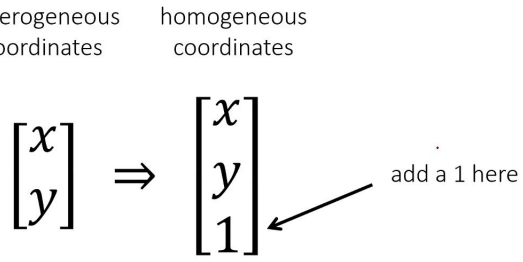
To implement translation, we have to concatenate a constant coordinate to $\begin{pmatrix} x \\ y \end{pmatrix}$ and use a 2×3 transformation matrix:

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

This constant that we concatenated to the vector will be useful to create a more general kind of transformation, which is called a homography. Before we get to homographies, lets first define heterogeneous and homogeneous coordinates.

7.2 Homegeneous Coordinates

In an image, each pixel has a location $\begin{pmatrix} x \\ y \end{pmatrix}$, which defines the **heterogenous coordinates** of the pixel. The **homogenous coordinates** of the pixel is then defined as $\begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$:



This way, we represent a 2D vector by a 3D vector. These homogeneous coordinate will be very useful.

Now, if we work in homogeneous coordinates, translation is defined as follows:

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

In homogeneous coordinates we have 3 coordinates. What if we want to convert a given homogeneous coordinates to their corresponding heterogeneous coordinates? We can simply do the following:

$$\begin{pmatrix} x \\ y \\ w \end{pmatrix} \underbrace{\rightarrow}_{to\ heterogeneous} \begin{pmatrix} x/w \\ y/w \end{pmatrix}$$

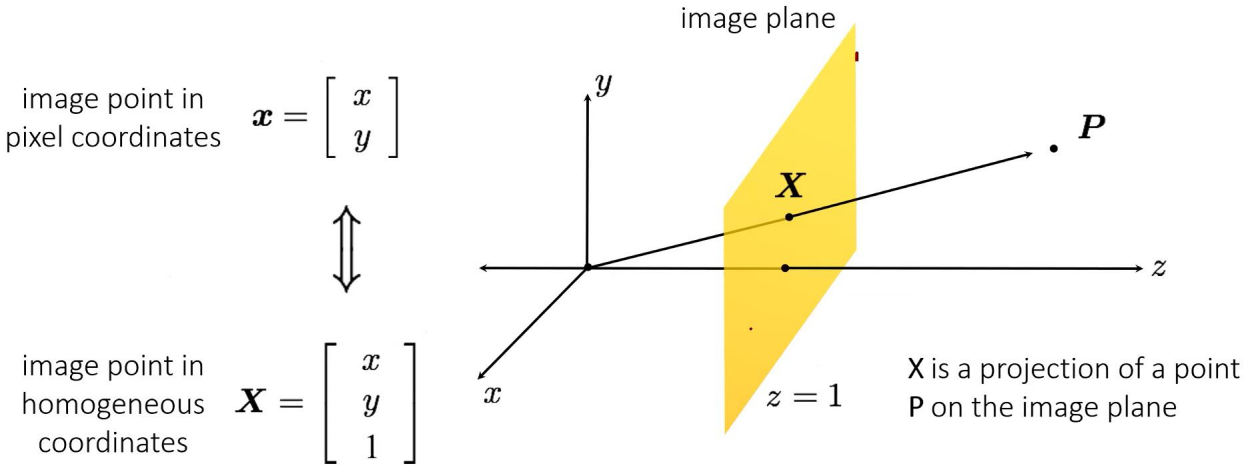


Figure 55: Heterogeneous Coordinates

$$\begin{pmatrix} x \\ y \end{pmatrix} \underbrace{\rightarrow}_{to\ homogeneous} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$$

We defined this conversion from heterogeneous to homogeneous coordinates because w will not always be 1, as we'll see soon.

Notice that heterogeneous coordinates are scale invariant: suppose we have $\begin{pmatrix} x \\ y \\ w \end{pmatrix}$ and $\lambda \begin{pmatrix} x \\ y \\ w \end{pmatrix}$. Thus, both of them correspond to the same homogeneous coordinates $\begin{pmatrix} x/w \\ y/w \end{pmatrix}$.

Also note two special cases: in heterogeneous coordinates, a point at infinity is defined as $\begin{pmatrix} x \\ y \\ 0 \end{pmatrix}$, and an undefined point is $\begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}$

Another way to think about heterogeneous coordinates is the following:

Since heterogeneous coordinates are scale invariant, multiplying X by any scalar λ will result in a point along the line in the direction of P, as we can see in the figure above. Note that all the points with $z = 1$ lie on the plane $z = 1$, which is the plane of the image. Thus, all the points in 3D space that lie on the same line that's connected to the camera's center, will be projected to the same heterogeneous coordinates on the image. After we have defined homogeneous coordinates, lets talk about transformations in their domain.

7.3 Transformations In Homogeneous Coordinates

7.3.1 Aspect (scale)

$$\begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} = \begin{pmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$$

7.3.2 Shear

$$\begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & \beta_x & 0 \\ \beta_y & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$$

7.3.3 Rotation

$$\begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} = \begin{pmatrix} cos(\theta) & -sin(\theta) & 0 \\ sin(\theta) & cos(\theta) & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$$

7.3.4 Translation

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

We can also perform matrix composition (compose multiple transformations) to get a combination of these transformations:

$$\begin{pmatrix} x' \\ y' \\ w \end{pmatrix} = M_{scale} M_{translate} M_{rotate} M_{shear} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$$

and we can define the required order, of course (notice that the order of this composition matters! A different order might result in a different transformation).

7.3.5 Rigid (rotation + translation)

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

This transformation has 3 degrees of freedom (DOF).

7.3.6 Affine

Affine transformation is a combination of translation, rotation, shearing, and scale:

$$\begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} = \begin{pmatrix} a_1 & a_2 & a_3 \\ a_4 & a_5 & a_6 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$$

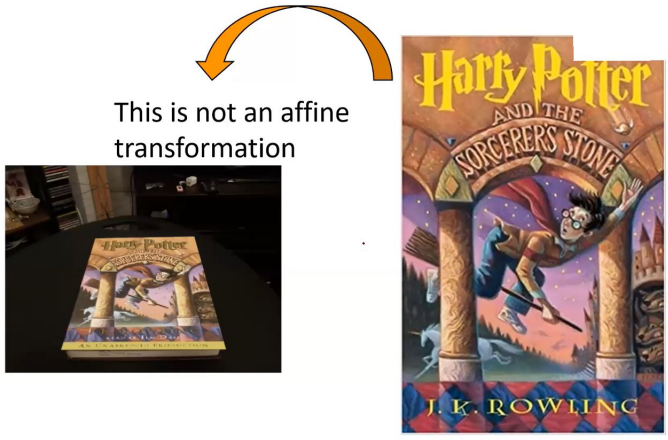
since there are no constrains between $a_1, \ldots, a_6$, this transformation has 6 DOF.

In general, affine transformations transform a rectangle to a parallelogram, anywhere on the plane. Properties:

- Straight lines are transformed to straight lines.
- Parallel lines are transformed to parallel lines.
- Any composition of affine transformations is also an affine transformation.
- The origin doesn't necessarily map to the origin.

Take another look at figure 55. What happens to a point on the image plane, if we perform affine transformation on it? The point will still remain on the image plane, at $z = 1$!

Notice the following transformation, which is not an affine transformation:



The reason this transformation is not affine is because parallel lines do **not** transform to parallel lines. Thus, we'll have to further expand our transformation capabilities to allow for these kinds of warpings. The next transformation we'll talk about is called **projective transformation** (as known as homography), which allows general transformations of rectangles.

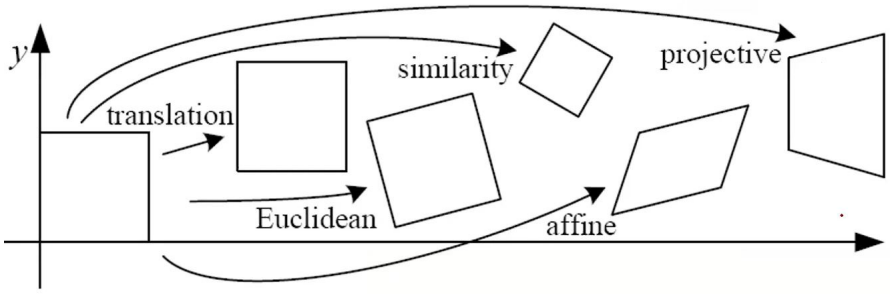


Figure 56: Classification of 2D transformations

7.3.7 Projective (homography)

Projective transformation is a combination of affine transformation and projective warp:

$$\begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} \underbrace{=}_{up\ to\ scale} \begin{pmatrix} h_1 & h_2 & h_3 \\ h_4 & h_5 & h_6 \\ h_7 & h_8 & h_9 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$$

Since this transformation is defined up to scale, we get:

$$\begin{pmatrix} x'_w \\ y'_w \\ w \end{pmatrix} = \begin{pmatrix} h_1 & h_2 & h_3 \\ h_4 & h_5 & h_6 \\ h_7 & h_8 & h_9 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$$

where w is the scale. Notice that $w = h_7x + h_8y + h_9$, and $\begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} = \begin{pmatrix} x'_w/w \\ y'_w/w \\ 1 \end{pmatrix}$. Thus, the following transformation is equivalent:

$$\begin{pmatrix} x_w/h_9 \\ y_w/h_9 \\ w/h_9 \end{pmatrix} = \begin{pmatrix} h_1/h_9 & h_2/h_9 & h_3/h_9 \\ h_4/h_9 & h_5/h_9 & h_6/h_9 \\ h_7/h_9 & h_8/h_9 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$$

because then we still get $\begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} = \begin{pmatrix} x'_w/w \\ y'_w/w \\ 1 \end{pmatrix}$. Thus, projective transformations have 8 DOF (and not 9).

Properties:

- The origin does not necessarily map to the origin.
- Straight lines transform to straight lines.
- Parallel lines don't necessarily map to parallel lines.
- Ratios are not necessarily conserved.
- And composition of projective transformation is also a projective transformation.

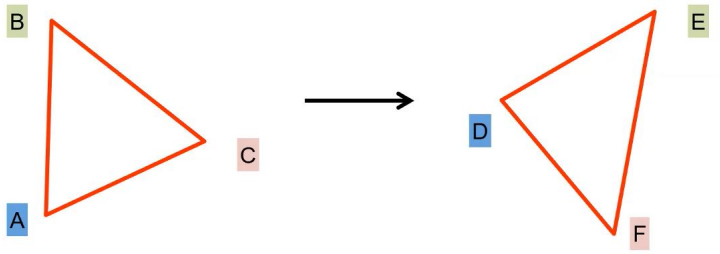
Now, take another look at figure 55. What happens to a point on the image plane, if we perform projective transformation on it? The transformed point can now be on another plane!

Important theorem: every image of a **plane** in camera 1 can be mapped to an image of a plane in camera 2 via a 3×3 projective transformation. This is true only for planes!

8 Solving For Unknown Transformations

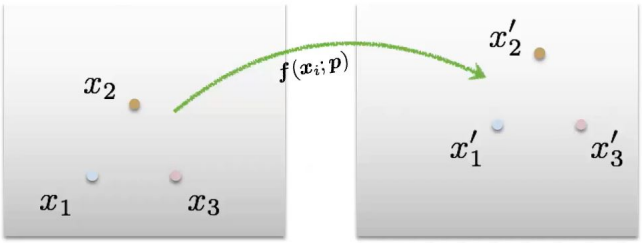
8.1 Solving For Unknown Affine Transformation

Suppose we have two triangles, ABC and DEF, and we want to find a transformation between these two triangles:



In this case, it's sufficient to use an affine transformation. Notice that each point in the triangle provides 2 constraints, because it contains two coordinates (x and y). Since affine transformation has 6 DOF, we need 3 points to solve for the required transformation. So, how do we find the required transformation?

One way to do it is to solve a least squares problem:



We'll refer to these points as p_1, p_2 and p_3 from now on (for convenience. p=pixel).

$$E_{LS} = \sum_i ||Ap_i - p'_i||^2$$

where A is the unknown affine transformation, and $p_i = (x_i, y_i), p'_i = (x'_i, y'_i)$ are known points (pixel coordinates) correspondences between the two images. We solve it by finding A that minimizes E_{LS}. We can also write the same error as follows:

$$E_{LS} = ||Ma - b||^2$$

where:

$$M = \begin{pmatrix} x_1 & y_1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & x_1 & y_1 & 1 \\ x_2 & y_2 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & x_2 & y_2 & 1 \\ x_3 & y_3 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & x_3 & y_3 & 1 \end{pmatrix}$$

$$a = \begin{pmatrix} A_{11} \\ A_{12} \\ A_{13} \\ A_{21} \\ A_{22} \\ A_{23} \end{pmatrix}$$

$$b = \begin{pmatrix} x'_1 \\ y'_1 \\ x'_2 \\ y'_2 \\ x'_3 \\ y'_3 \end{pmatrix}$$

and then solve for a. This method is called the "Direct Linear Transform", or DLT in short. Of course that this can be used on more than 3 points (an arbitrary size of M). But, when we want to solve for affine transformation, using 3 points is sufficient (we can use more point correspondences to account for possible noise).

In this formulation, there's a known closed solution that minimizes E_{LS}:

$$E_{LS} = a^T(M^T M)a - 2a^T(M^T b) + ||b||^2$$

$$\frac{\partial E_{LS}}{\partial a} = 2(M^T M)a - 2M^T b = 0$$

Since E_{LS} is convex, the global minimum is given by:

$$(M^T M)a = M^T b$$

$$\Downarrow$$

$$a = (M^T M)^{-1} M^T b$$

Usually it's a bad idea to inverse a matrix. Instead, we usually find its pseudo-inverse.

8.2 Solving For Unknown Projective Transformation

First of all, to apply a homography H on a point $p_{het} = \begin{pmatrix} x \\ y \end{pmatrix}$, we need to convert p_{het} to homogeneous coordinates: $p_{hom} = \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$. Then, after we apply H, we get: $p'_{hom} = Hp_{hom} = \begin{pmatrix} x' \\ y' \\ w \end{pmatrix}$. Thus, we need to divide by p'_{hom} by w to convert back to heterogeneous coordinates (the image plane coordinates):

$$p'_{het} = \begin{pmatrix} x'/w \\ y'/w \end{pmatrix}$$

Once again, we can solve for H using Direct Linear Transform, given at least 4 point correspondences (since projective transformations have 8 DOF):

- Write out a linear equation for each points correspondence:

$$\begin{pmatrix} x'_w \\ y'_w \\ w \end{pmatrix} = \begin{pmatrix} h_1 & h_2 & h_3 \\ h_4 & h_5 & h_6 \\ h_7 & h_8 & h_9 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$$

$$x' = h_1 x + h_2 y + h_3,\ y' = h_4 x + h_5 y + h_6,\ w = h_7 x + h_8 y + h_9$$

- Divide by w to get the homogeneous coordinates solution:

$$x' = x'_w/w,\ y' = x'_w/w$$

$$\Downarrow$$

$$x'(h_7 x + h_8 y + h_9) = h_1 x + h_2 y + h_3$$
$$y'(h_7 x + h_8 y + h_9) = h_4 x + h_5 y + h_6$$

  where x', y' are in homogeneous coordinates.

- Rearrange the terms:

$$h_7 x x' + h_8 y x' + h_9 x' - h_1 x - h_2 y - h_3 = 0$$
$$h_7 x y' + h_8 y y' + h_9 y' - h_4 x - h_5 y - h_6 = 0$$

- Rewrite it in matrix form:

$$M_i = \begin{pmatrix} -x & -y & -1 & 0 & 0 & 0 & xx' & yx' & x' \\ 0 & 0 & 0 & -x & -y & -1 & xx' & yx' & x' \end{pmatrix}$$

$$h = \begin{pmatrix} h_1 & h_2 & h_3 & h_4 & h_5 & h_6 & h_7 & h_8 & h_9 \end{pmatrix}^T$$

  Notice that we have 2 linear equations for each point correspondence i.

- Arrange $Mh = 0$ and solve for h (assuming we have N point correspondences):

$$Mh = \begin{pmatrix} M_1 \\ M_2 \\ \vdots \\ M_N \end{pmatrix} h = 0$$

This is a homogeneous system of linear equations, and thus it will always have a trivial solution $h = 0$ (but this solution is not relevant). If there exist $h \neq 0$ that solves this set of linear equations, it means that the rank of M is at most 8, since finding this h means that the columns (and rows) of M are linearly dependant. Also, notice that the size of M is $2N \times 9$.

To solve for h, we can't use the same solution as we used to solve for affine transformation (least squares). This time, we want to minimize $||Mh||^2$ subject to $||h||^2 = 1$. The way to do that is to use Singular Value Decomposition:

- We can either solve for the eigenvector corresponding to the smallest eigenvalue of $M^T M$.
- Or, we can solve for the right singular vector corresponding to the smallest singular value of the SVD of M.
- These solutions are equivalent.

Note that if we use $N > 4$ noise free point correspondences, the rank of M is still at most 8 (otherwise, there are point correspondences that don't agree).
Important note: the objective we are minimizing is not our actual objective. In the above formulation, we are trying to find h that minimizes $||Mh||^2$, which is equivalent to minimizing the following::

$$\sum_i (h_7 x_i x_i' + h_8 y_i x_i' + h_9 x_i' - h_1 x_i - h_2 y_i - h_3)^2 + (h_7 x_i y_i' + h_8 y_i y_i' + h_9 y_i' - h_4 x_i - h_5 y_i - h_6)^2$$

But, what we really want to minimize is:

$$\sum_i (\frac{h_1 x_i + h_2 y_i + h_3}{h_7 x_i + h_8 y_i + h_9} - x_i')^2 + (\frac{h_4 x_i + h_5 y_i + h_6}{h_7 x_i + h_8 y_i + h_9} - y_i')^2$$

These minimization problems are not equivalent given localization errors. The latter formulation is not linear, so it's more difficult to solve it. It's a lot easier to solve the former formulation, and that's why we solve it and not the latter.

8.3 Finding Point Correspondences Automatically

Our next goal is to automate the process of finding point correspondences between two images.

Figure 57: An objective we want to accomplish. Our next goal is to find a way to automate this step.

There's a very natural pipeline to do this:

1. **Detect features in both images** using any of the feature detection algorithms we learned, such as Harris corner detector.
2. **Describe all of the detected features** using any of the feature descriptors we learned, such as SIFT.
3. **Match corresponding features** using their descriptors and using some distance metric. For example, you can search for features that are closest in L2 distance.

It's common to find false corresponding points (due to error, feature descriptor variance to image transformations, and more). These false correspondence points are called "outliers", and they represent correspondences that are not correct. Thus, we want to find a way to eliminate those outliers.

8.3.1 Random Sample Consensus (RANSAC)

The RANSAC algorithm is a learning technique to estimate parameters of a model by random sampling of observed data. Given a dataset whose data elements contain both inliers and outliers, RANSAC uses the voting scheme to find the optimal fitting result. Data elements in the dataset are used to vote for one or multiple models. The implementation of this voting scheme is based on two assumptions: that the noisy features will not vote consistently for any single model (few outliers) and there are enough features to agree on a good model (few missing data). The RANSAC algorithm is essentially composed of two steps that are iteratively repeated:

1. In the first step, a sample subset containing minimal data items is randomly selected from the input dataset. A fitting model and the corresponding model parameters are computed using only the elements of this sample subset. The cardinality (number of elements) of the sample subset is the smallest sufficient to determine the model parameters.

2. In the second step, the algorithm checks which elements of the entire dataset are consistent with the model instantiated by the estimated model parameters obtained from the first step. A data element will be considered as an outlier if it does not fit the fitting model instantiated by the set of estimated model parameters within some error threshold that defines the maximum deviation attributable to the effect of noise.

The set of inliers obtained for the fitting model is called the consensus set. The RANSAC algorithm will iteratively repeat the above two steps until the obtained consensus set in certain iteration has enough inliers.

The input to the RANSAC algorithm is a set of observed data values, a way of fitting some kind of model to the observations, and some confidence parameters. RANSAC achieves its goal by repeating the following steps:

1. Select a random subset of the original data. Call this subset the hypothetical inliers.

2. A model is fitted to the set of hypothetical inliers.

3. All other data are then tested against the fitted model. Those points that fit the estimated model well, according to some model-specific loss function, are considered as part of the consensus set.

4. The estimated model is reasonably good if sufficiently many points have been classified as part of the consensus set (each inlier votes that it is indeed an inlier, and the model is good enough if it got enough votes). We can define the best model to be the model achieved the most amount of votes thus far.

5. Afterwards, the model may be improved by re-estimating it using all members of the consensus set.

This procedure is repeated a fixed number of times, each time producing either a model which is rejected because too few points are part of the consensus set, or a refined model together with a corresponding consensus set size. In the latter case, we keep the refined model if its consensus set is larger than the previously saved model.

Talking about our case, in which we want to find the best homography out of automatically detected point correspondences, we can run the following algorithm (RANSAC applied to our case):

1. Randomly select 4 point correspondences.

2. Compute the projective transformation H that agrees with those correspondences.

3. Iterate over all point correspondences (x_i, x_i'), compute Hx_i, and convert the result to homogeneous coordinates. If the result is close enough to x_i', vote in favor of H.

4. If the current H got more votes than the last H, keep the current H.

5. Repeat steps 1-4 for K times.

6. Recompute H that got the most amount of votes, using all of its inliers. Return the final result.

8.4 When Can We Use Homographies?

As we stated before, the following theorem holds: **Every image of a plane in camera 1 can be mapped into an image of a plance in camera 2 via a 3×3 projective transformation.**

Thus, we can use homographies in the following cases:

- When the scene is planar.
- When the scene is very far or has a small relative depth variation (the scene is approximately planar).
- When the point of views of the images differ by pure rotation (the camera purely rotated).

8.5 Solving For Unknown Image Range Warps

Suppose we have two images, and that we computed the domain transformation between those images (such as a projective transformation). Now, under the computed domain transformation, pixels with integer coordinates might transform to non-integer coordinates. So, how do we define the color/intensity value at those non-integer coordinate locations? One way to do that is **forward warping**:

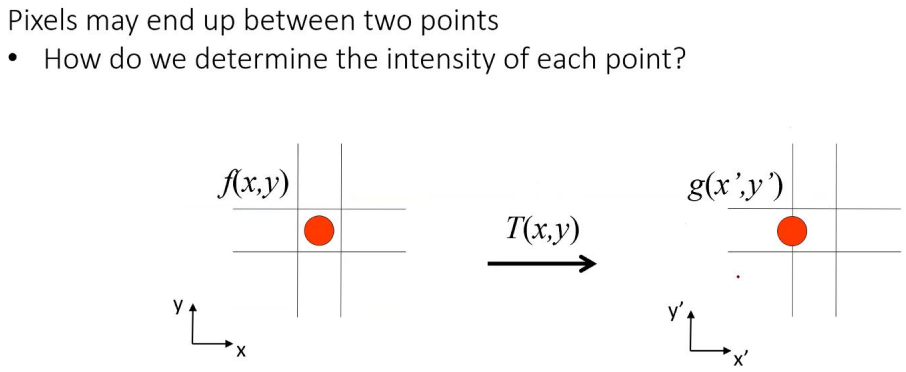


Figure 58: An example of forward warping in which the domain transformation T maps integer coordinates (x, y) to non-integer coordinates (x', y'). In this case, we can't determine a pixel intensity value at (x', y'). Instead, we need to determine the intensity values at its neighboring integer coordinate locations.

To determine the intensity at each point, we can distribute the color of (x', y') among neighboring pixels ("splatting"). If pixel (x', y') receives intensity from more than one pixel (x, y), then we can average those pixels' intensity contributions.

A better way to do that is to perform **inverse warping**:

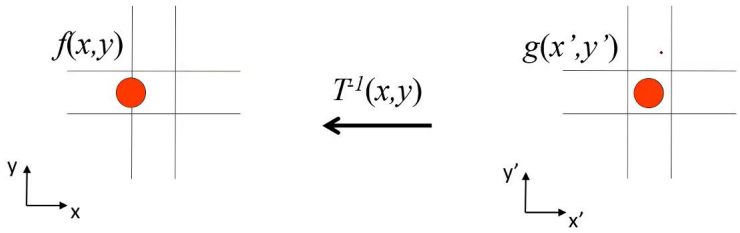


Figure 59: An example of inverse warping. In this case, we use T^{-1}.

To determine the intensity at each point (x', y'), we first calculate its corresponding (x, y) point using the inverse transform T^{-1}. As before, the resulting (x, y) might be a non-integer pixel location. Thus, to determine the intensity at (x', y') we can interpolate the intensity around (x, y) (such as bilinear interpolation).

Usually, inverse warping works a lot better.

9 Geometric Camera Models

9.1 Introduction

The camera is one of the most essential tools in computer vision. It is the mechanism by which we can record the world around us and use its output - photographs - for various applications. Therefore, one question we must ask in introductory computer vision is: how do we model a camera?

9.2 Pinhole cameras

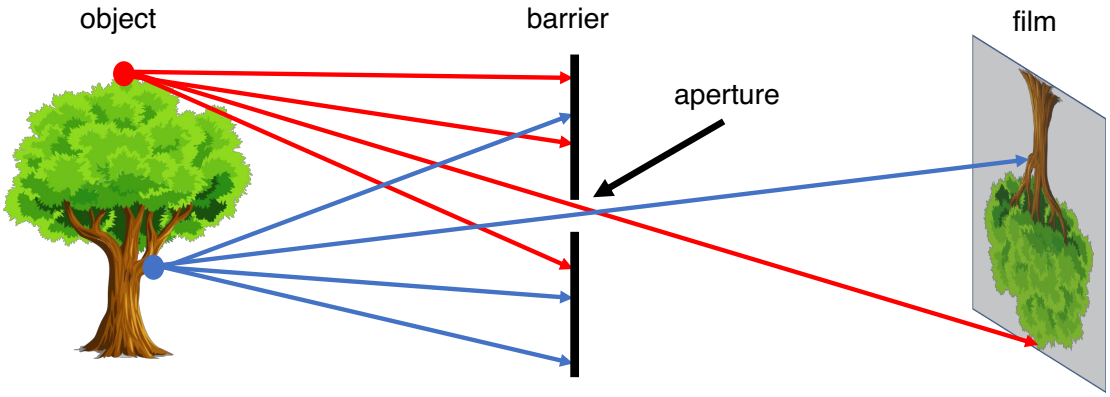


Figure 60: A simple working camera model: the pinhole camera model.

Let's design a simple camera system – a system that can record an image of an object or scene in the 3D world. This camera system can be designed by placing a barrier with a small aperture between the 3D object and a photographic film or sensor. As Figure 60 shows, each point on the 3D object emits multiple rays of light outwards. Without a barrier in place, every point on the film will be influenced by light rays emitted from every point on the 3D object. Due to the barrier, only one (or a few) of these rays of light passes through the aperture and hits the film. Therefore, we can establish a one-to-one mapping between points on the 3D object and the film. The result is that the film gets exposed by an "image" of the 3D object by means of this mapping. This simple model is known as the *pinhole camera model*.

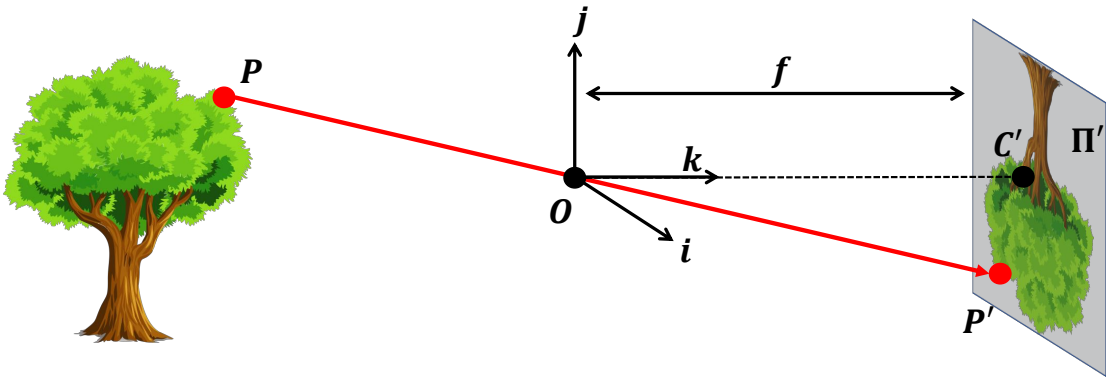


Figure 61: A formal construction of the pinhole camera model.

A more formal construction of the pinhole camera is shown in Figure 61. In this construction, the film is commonly called the *image or retinal plane*. The aperture is referred to as the *pinhole* O or *center of the camera*. The distance between the image plane and the pinhole O is the *focal length* f. Sometimes, the retinal plane is placed between O and the 3D object at a distance f from O. In this case, it is called the *virtual image* or *virtual retinal plane*. Note that the projection of the object in the image plane and the image of the object in the virtual image plane are identical up to a scale (similarity) transformation.

Now, how do we use pinhole cameras? Let $P = \begin{bmatrix} x & y & z \end{bmatrix}^T$ be a point on some 3D object visible to the pinhole camera. P will be mapped or **projected** onto the image plane Π', resulting in point[1] $P' = \begin{bmatrix} x' & y' \end{bmatrix}^T$. Similarly, the pinhole itself can

[1]Throughout the course notes, let the prime superscript (e.g. P') indicate that this point is a projected or complementary point to the non-superscript version. For example, P' is the projected version of P.

be projected onto the image plane, giving a new point C'.

Here, we can define a coordinate system $\begin{bmatrix} i & j & k \end{bmatrix}$ centered at the pinhole O such that the axis k is perpendicular to the image plane and points toward it. This coordinate system is often known as the *camera reference system* or *camera coordinate system*. The line defined by C' and O is called the *optical axis* of the camera system.

Recall that point P' is derived from the projection of 3D point P on the image plane Π'. Therefore, if we derive the relationship between 3D point P and image plane point P', we can understand how the 3D world imprints itself upon the image taken by a pinhole camera. Notice that triangle $P'C'O$ is similar to the triangle formed by P, O and $(0, 0, z)$ (for some z). Therefore, using the law of similar triangles we find that:

$$P' = \begin{bmatrix} x' & y' \end{bmatrix}^T = \begin{bmatrix} f\frac{x}{z} & f\frac{y}{z} \end{bmatrix}^T \tag{1}$$

Notice that one large assumption we make in this pinhole model is that the aperture (the pin hole opening through which light travels) is a single point. In most real world scenarios, however, we cannot assume the aperture can be infinitely small. Thus, what is the effect of varying aperture size?

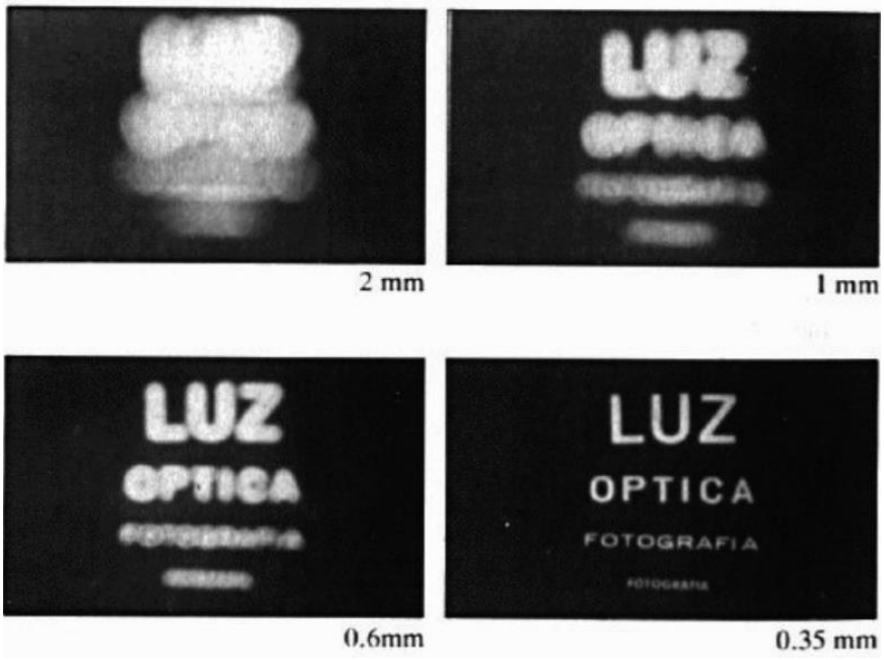


Figure 62: The effects of aperture size on the image. As the aperture size decreases, the image gets sharper, but darker.

As the aperture size increases, the number of light rays that passes through the barrier consequently increases. With more light rays passing through, then each point on the film may be affected by light rays from multiple points in 3D space, blurring the image. Although we may be inclined to try to make the aperture as small as possible, recall that a smaller aperture size causes less light rays to pass through, resulting in crisper but darker images. Therefore, we arrive at the fundamental problem presented by the pinhole formulation: can we develop cameras that take crisp and bright images?

9.3 Cameras and lenses

In modern cameras, the above conflict between crispness and brightness is mitigated by using *lenses*, devices that can focus or disperse light. If we replace the pinhole with a lens that is both properly placed and sized, then it satisfies the following property: all rays of light that are emitted by some point P are refracted by the lens such that they converge to a single point P' in the image plane. Therefore, the problem of the majority of the light rays blocked due to a small aperture is removed (Figure 63). However, please note that this property does not hold for all 3D points, but only for some specific point P. Take another point Q which is closer or further from the image plane than P. The corresponding projection into the image will be blurred or out of focus. Thus, lenses have a specific distance for which objects are "in focus". This property is also related to a photography and computer graphics concept known as depth of field, which is the effective range at which cameras can take clear photos.

Camera lenses have another interesting property: they focus all light rays traveling parallel to the optical axis to one point known as the *focal point* (Figure 64). The distance between the focal point and the center of the lens is commonly referred to as the *focal length* f. Furthermore, light rays passing through the center of the lens are not deviated. We thus can arrive at a similar construction to the pinhole model that relates a point P in 3D space with its corresponding point P' in the image plane.

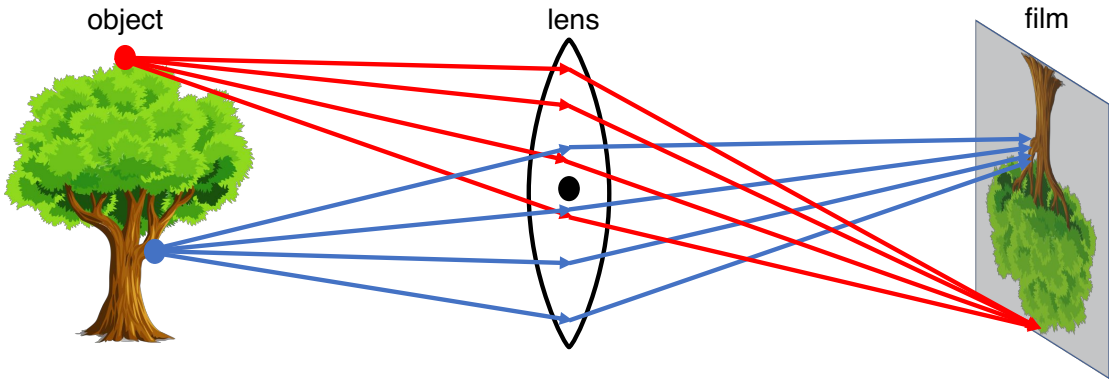


Figure 63: A setup of a simple lens model. Notice how the rays of the top point on the tree converge nicely on the film. However, a point at a different distance away from the lens results in rays not converging perfectly on the film.

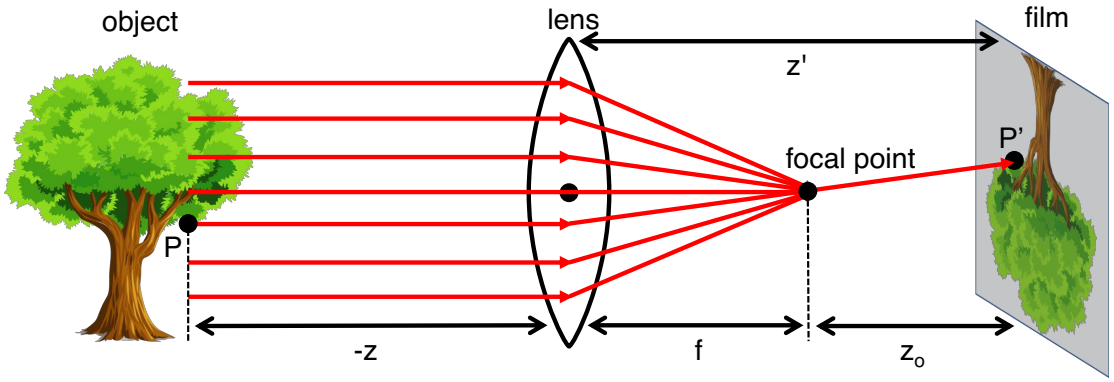


Figure 64: Lenses focus light rays parallel to the optical axis into the focal point. Furthermore, this setup illustrates the paraxial refraction model, which helps us find the relationship between points in the image plane and the 3D world in cameras with lenses.

$$P' = \begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} z' \frac{x}{z} \\ z' \frac{y}{z} \end{bmatrix} \tag{2}$$

The derivation for this model is outside the scope of the class. However, please notice that in the pinhole model $z' = f$, while in this lens-based model, $z' = f + z_0$ (this is called focal distance). Additionally, since this derivation takes advantage of the paraxial or "thin lens" assumption[2], it is called the **paraxial refraction model**.

Because the paraxial refraction model approximates using the thin lens assumption, a number of aberrations can occur. The most common one is referred to as *radial distortion*, which causes the image magnification to decrease or increase as a function of the distance to the optical axis. We classify the radial distortion as *pincushion distortion* when the magnification increases and *barrel distortion*[3] when the magnification decreases. Radial distortion is caused by the fact that different portions of the lens have differing focal lengths. These are also called *perspective distortions*.

9.4 Going to digital image space

In this section, we will discuss the details of the parameters we must account for when modeling the projection from 3D space to the digital images we know. All the results derived will use the pinhole model, but they also hold for the paraxial refraction model.

As discussed earlier, a point P in 3D space can be mapped (or projected) into a 2D point P' in the image plane Π'. This $\mathbb{R}^3 \to \mathbb{R}^2$ mapping is referred to as a *projective transformation*. This projection of 3D points into the image plane does not directly correspond to what we see in actual digital images for several reasons. First, points in the digital images are, in general, in a different reference system than those in the image plane. Second, digital images are divided into discrete pixels,

[2]For the angle θ that incoming light rays make with the optical axis of the lens, the paraxial assumption substitutes θ for any place $\sin(\theta)$ is used. This approximation of θ for $\sin\theta$ holds as θ approaches 0.

[3]Barrel distortion typically occurs when one uses fish-eye lenses.

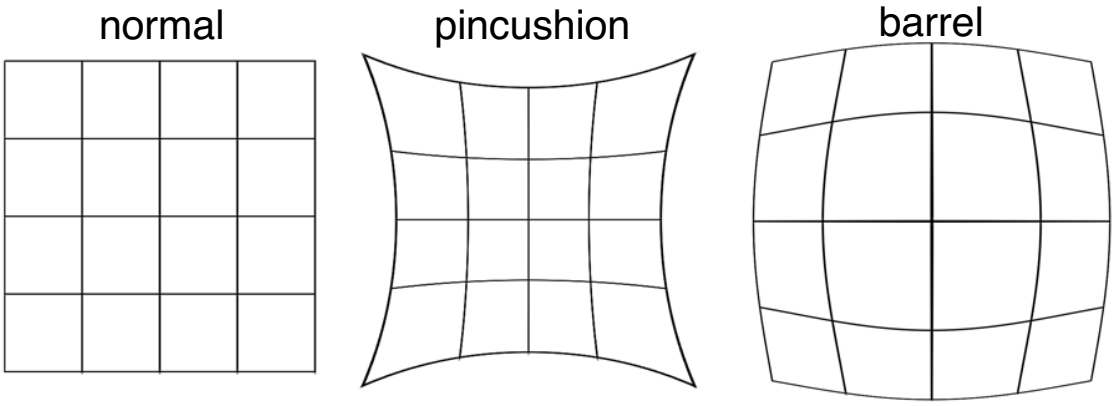


Figure 65: Demonstrating how pincushion and barrel distortions affect images.

whereas points in the image plane are continuous. Finally, the physical sensors can introduce non-linearity such as distortion to the mapping. To account for these differences, we will introduce a number of additional transformations that allow us to map any point from the 3D world to pixel coordinates.

9.4.1 Introduction to the Camera Matrix Model

The camera matrix model describes a set of important parameters that affect how a world point P is mapped to image coordinates P'. As the name suggests, these parameters will be represented in matrix form. First, let's introduce some of those parameters.

The first parameters, c_x and c_y, describe how image plane and digital image coordinates can differ by a translation. Image plane coordinates have their origin C' at the image center where the k axis intersects the image plane. On the other hand, digital image coordinates typically have their origin at the lower-left corner of the image. Thus, 2D points in the image plane and 2D points in the image are offset by a translation vector $\begin{bmatrix} c_x, c_y \end{bmatrix}^T$. To accommodate this change of coordinate systems, the mapping now becomes:

$$P' = \begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} f\frac{x}{z} + c_x \\ f\frac{y}{z} + c_y \end{bmatrix} \tag{3}$$

The second effect we must account for that the points in digital images are expressed in pixels, while points in image plane are represented in physical measurements (e.g. centimeters). In order to accommodate this change of units, we must introduce two new parameters k and l. These parameters, whose units would be something like $\frac{\text{pixels}}{\text{cm}}$, correspond to the change of units in the two axes of the image plane. Note that k and l may be different because the aspect ratio of a pixel is not guaranteed to be one. If $k = l$, we often say that the camera has *square pixels*. We adjust our previous mapping to be

$$P' = \begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} fk\frac{x}{z} + c_x \\ fl\frac{y}{z} + c_y \end{bmatrix} = \begin{bmatrix} \alpha\frac{x}{z} + c_x \\ \beta\frac{y}{z} + c_y \end{bmatrix} \tag{4}$$

Is there a better way to represent this projection from $P \to P'$? If this projection is a linear transformation, then it can be represented as a product of a matrix and the input vector (in this case, it would be P). However, from Equation 4, we see that this projection $P \to P'$ is not linear, as the operation divides one of the input parameters (namely z). Still, representing this projection as a matrix-vector product would be useful for future derivations. Therefore, can we represent our transformation as a matrix-vector product despite its nonlinearity? Homogeneous coordinates are the solution.

9.4.2 Homogeneous Coordinates

One way to solve this problem is to change the coordinate systems. For example, we introduce a new coordinate, such that any point $P' = (x', y')$ becomes $(x', y', 1)$. Similarly, any point $P = (x, y, z)$ becomes $(x, y, z, 1)$. This augmented space is referred to as the *homogeneous coordinate system*. As demonstrated previously, to convert a Euclidean vector $(v_1, ..., v_n)$ to homogeneous coordinates, we simply append a 1 in a new dimension to get $(v_1, ..., v_n, 1)$. Note that the equality between a vector and its homogeneous coordinates only occurs when the final coordinate equals one. Therefore, when converting back from arbitrary homogeneous coordinates $(v_1, ..., v_n, w)$, we get Euclidean coordinates $(\frac{v_1}{w}, ..., \frac{v_n}{w})$. Using homogeneous coordinates, we can formulate

$$P'_h = \begin{bmatrix} \alpha x + c_x z \\ \beta y + c_y z \\ z \end{bmatrix} = \begin{bmatrix} \alpha & 0 & c_x & 0 \\ 0 & \beta & c_y & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} \alpha & 0 & c_x & 0 \\ 0 & \beta & c_y & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} P_h \tag{5}$$

From this point on, assume that we will work in homogeneous coordinates, unless stated otherwise. We will drop the h index, so any point P or P' can be assumed to be in homogeneous coordinates. As seen from Equation 5, we can represent the relationship between a point in 3D space and its image coordinates by a matrix vector relationship:

$$P' = \begin{bmatrix} x' \\ y' \\ z \end{bmatrix} = \begin{bmatrix} \alpha & 0 & c_x & 0 \\ 0 & \beta & c_y & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} \alpha & 0 & c_x & 0 \\ 0 & \beta & c_y & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} P = MP \tag{6}$$

We can decompose this transformation a bit further into

$$P' = MP = \begin{bmatrix} \alpha & 0 & c_x \\ 0 & \beta & c_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} I & 0 \end{bmatrix} P = K \begin{bmatrix} I & 0 \end{bmatrix} P \tag{7}$$

The matrix K is often referred to as the *camera matrix*.

9.4.3 The Complete Camera Matrix Model

The camera matrix K contains some of the critical parameters that describes a camera's characteristics and its model, including the c_x, c_y, k, and l parameters as discussed above. Two parameters are currently missing this formulation: *skewness* and *distortion*. We often say that an image is skewed when the camera coordinate system is skewed, meaning that the angle between the two axes is slightly larger or smaller than 90 degrees. Most cameras have zero-skew, but some degree of skewness may occur because of sensor manufacturing errors. Deriving the new camera matrix accounting for skewness is outside the scope of this class and we give it to you below:

$$K = \begin{bmatrix} x' \\ y' \\ z \end{bmatrix} = \begin{bmatrix} \alpha & -\alpha \cot\theta & c_x \\ 0 & \frac{\beta}{\sin\theta} & c_y \\ 0 & 0 & 1 \end{bmatrix} \tag{8}$$

Most methods that we introduce in this class ignore distortion effects, therefore our class camera matrix K has 5 degrees of freedom: 2 for focal length, 2 for offset, and 1 for skewness. These parameters are collectively known as the *intrinsic parameters*, as they are unique and inherent to a given camera and relate to essential properties of the camera, such as its manufacturing.

9.4.4 Extrinsic Parameters

So far, we have described a mapping between a point P in the 3D camera reference system to a point P' in the 2D image plane using the intrinsic parameters of a camera described in matrix form. But what if the information about the 3D world is available in a different coordinate system? Then, we need to include an additional transformation that relates points from the world reference system to the camera reference system. This transformation is captured by a rotation matrix R and translation vector T. Therefore, given a point in a world reference system P_w, we can compute its camera coordinates as follows:

$$P = \begin{bmatrix} R & T \\ 0 & 1 \end{bmatrix} P_w \tag{9}$$

Note that to transform P_w, we are first rotating it and then translating its rotated version (in heterogeneous coordinates system, this is equivalent to $P_{het} = RP_{w,het} - T$) Substituting this in equation (7) and simplifying gives

$$P' = K \begin{bmatrix} R & T \end{bmatrix} P_w = MP_w \tag{10}$$

These parameters R and T are known as the *extrinsic parameters* because they are external to and do not depend on the camera. Note that M can be written as $M = \begin{bmatrix} A & b \\ v & 1 \end{bmatrix}$ for some A, b, v. R is a 3×3 matrix and T is a 3×1 vector. Thus, M is a 3×4 matrix. In addition, notice that $T = -RO_w$ where O_w is the camera's center in the world's coordinates system. Thus, O_w is in the nullspace of M:

$$K \begin{bmatrix} R & -RO_w \end{bmatrix} O_w = K \begin{bmatrix} R & -RO_w \end{bmatrix} \begin{pmatrix} O_{w,1} \\ O_{w,2} \\ O_{w,3} \\ 1 \end{pmatrix} = K(RO_w - RO_w) = \vec{0} \tag{11}$$

This completes the mapping from a 3D point P in an arbitrary world reference system to the image plane. To reiterate, we see that the full projection matrix M consists of the two types of parameters introduced above: *intrinsic* and *extrinsic*

parameters. All parameters contained in the camera matrix K are the intrinsic parameters, which change as the type of camera changes. The extrinsic paramters include the rotation and translation, which do not depend on the camera's build. Overall, we find that the 3×4 projection matrix M has 11 degrees of freedom: 5 from the intrinsic camera matrix, 3 from extrinsic rotation, and 3 from extrinsic translation.

9.5 Weak Perspective Camera

Note: this section is written based on the lectures, and it's explained again in appendix B. Another camera model is the weak perspective camera model. In this model, we move the camera center by αz_0 to the right, and we also change z' to be $\alpha z'$. Thus, we get:

$$P' = \begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} k\alpha \frac{x}{z+kz_0} \\ k\beta \frac{y}{z+kz_0} \end{bmatrix} \underset{k\to\infty}{\longrightarrow} \begin{bmatrix} \alpha \frac{x}{z_0} \\ \beta \frac{y}{z_0} \end{bmatrix} \tag{12}$$

Thus, for large enough α, the magnification of an object wouldn't depend on its distance from the camera. Now, the relationship between a point in 3D space (in the camera's coordinates system) and its image coordinates is:

$$P' = \begin{bmatrix} x' \\ y' \\ z_0 \end{bmatrix} = \begin{bmatrix} \alpha & 0 & 0 & z_0 c_x \\ 0 & \beta & 0 & z_0 c_y \\ 0 & 0 & 0 & z_0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} \alpha & 0 & 0 & z_0 c_x \\ 0 & \beta & 0 & z_0 c_y \\ 0 & 0 & 0 & z_0 \end{bmatrix} P = M_{weak} P \tag{13}$$

We can use the weak perspective camera estimation in the following cases:

1. When the scene (or parts of it) is very far away.
2. When we use a telecentric lens (microscopes). Place a pinhole at the focal length of the lens, so that only parallel rays pass through.
3. Orthographic camera (all of the intrinsic parameters are 1).

9.6 Camera Calibration

To precisely know the transformation from the real, 3D world into digital images requires prior knowledge of many of the camera's intrinsic parameters. If given an arbitrary camera, we may or may not have access to these parameters. We do, however, have access to the images the camera takes. Therefore, can we find a way to deduce them from images? This problem of estimating the extrinsic and intrinsic camera parameters is known as *camera calibration*.

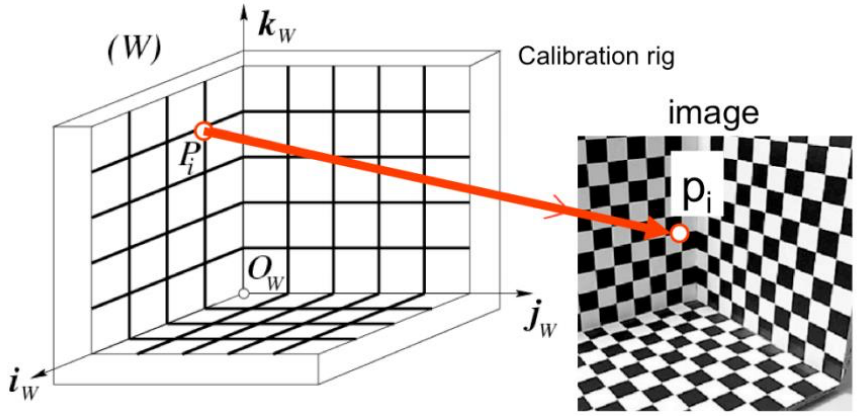


Figure 66: The setup of an example calibration rig.

Specifically, we do this by solving for the intrinsic camera matrix K and the extrinsic parameters R, T from Equation 10. We can describe this problem in the context of a calibration rig, such as the one show in Figure 66. The rig usually consists of a simple pattern (i.e. checkerboard) with known dimensions. Furthermore, the rig defines our world reference frame with origin O_w and axes i_w, j_w, k_w. From the rig's known pattern, we have known points in the world reference frame $P_1, ..., P_n$. Finding these points in the image we take from the camera gives corresponding points in the image $p_1, ..., p_n$.

We set up a linear system of equations from n correspondences such that for each correspondence P_i, p_i and camera matrix M whose rows are m_1, m_2, m_3:

$$p_i = \begin{bmatrix} u_i \\ v_i \end{bmatrix} = MP_i = \begin{bmatrix} \frac{m_1 P_i}{m_3 P_i} \\ \frac{m_2 P_i}{m_3 P_i} \end{bmatrix} \tag{14}$$

As we see from the above equation, each correspondence gives us two equations and, consequently, two constraints for solving the unknown parameters contained in m. From before, we know that the camera matrix has 11 unknown parameters. This means that we need at least 6 correspondences to solve this. However, in the real world, we often use more, as our measurements are often noisy. To explicitly see this, we can derive a pair of equations that relate u_i and v_i with P_i.

$$\begin{aligned} u_i(m_3 P_i) - m_1 P_i = 0 \\ v_i(m_3 P_i) - m_2 P_i = 0 \end{aligned}$$

Given n of these corresponding points, the entire linear system of equations becomes

$$\begin{aligned} u_1(m_3 P_1) - m_1 P_1 = 0 \\ v_1(m_3 P_1) - m_2 P_1 = 0 \\ \vdots \\ u_n(m_3 P_n) - m_1 P_n = 0 \\ v_n(m_3 P_n) - m_2 P_n = 0 \end{aligned}$$

This can be formatted as a matrix-vector product shown below:

$$\begin{bmatrix} P_1^T & 0^T & -u_1 P_1^T \\ 0^T & P_1^T & -v_1 P_1^T \\ & \vdots & \\ P_n^T & 0^T & -u_n P_n^T \\ 0^T & P_n^T & -v_n P_n^T \end{bmatrix} \begin{bmatrix} m_1^T \\ m_2^T \\ m_3^T \end{bmatrix} = \mathbf{P}m = 0 \tag{15}$$

When $2n > 11$, our homogeneous linear system is overdetermined. For such a system $m = 0$ is always a trivial solution. Furthemore, even if there were some other m that were a nonzero solution, then $\forall k \in \mathbb{R}, km$ is also a solution. Therefore, to constrain our solution, we complete the following minimization:

$$\begin{aligned} \underset{m}{\text{minimize}} \quad & \|\mathbf{P}m\|^2 \\ \text{subject to} \quad & \|m\|^2 = 1 \end{aligned} \tag{16}$$

To solve this minimization problem, we simply use singular value decomposition. If we let $P = UDV^T$, then the solution to the above minimization is to set m equal to the last column of V (the right singular vector corresponding to the smallest singular value). The derivation for this solution is outside the scope of this class and you may refer to Section 5.3 of Hartley & Zisserman on pages 592-593 for more details.

After reformatting the vector m into the matrix M, we now want to explicitly solve for the extrinsic and intrinsic parameters. We know our SVD-solved M is known up to scale, which means that the true values of the camera matrix are some scalar multiple of M:

$$\rho M = \begin{bmatrix} \alpha r_1^T - \alpha \cot\theta r_2^T + c_x r_3^T & \alpha t_x - \alpha \cot\theta t_y + c_x t_z \\ \frac{\beta}{\sin\theta} r_2^T + c_y r_3^T & \frac{\beta}{\sin\theta} t_y + c_y t_z \\ r_3^T & t_z \end{bmatrix} \tag{17}$$

Here, r_1^T, r_2^T, and r_3^T are the three rows of R. Dividing by the scaling parameter gives

$$M = \frac{1}{\rho} \begin{bmatrix} \alpha r_1^T - \alpha \cot\theta r_2^T + c_x r_3^T & \alpha t_x - \alpha \cot\theta t_y + c_x t_z \\ \frac{\beta}{\sin\theta} r_2^T + c_y r_3^T & \frac{\beta}{\sin\theta} t_y + c_y t_z \\ r_3^T & t_z \end{bmatrix} = \begin{bmatrix} A & b \end{bmatrix} = \begin{bmatrix} a_1^T \\ a_2^T \\ a_3^T \end{bmatrix} \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix}$$

Solving for the intrinsics gives

$$\begin{aligned} \rho &= \pm \frac{1}{\|a_3\|} \\ c_x &= \rho^2 (a_1 \cdot a_3) \\ c_y &= \rho^2 (a_2 \cdot a_3) \\ \theta &= \cos^{-1}\left(-\frac{(a_1 \times a_3) \cdot (a_2 \times a_3)}{\|a_1 \times a_3\| \cdot \|a_2 \times a_3\|} \right) \\ \alpha &= \rho^2 \|a_1 \times a_3\| \sin\theta \\ \beta &= \rho^2 \|a_2 \times a_3\| \sin\theta \end{aligned} \tag{18}$$

The extrinsics are

$$\begin{aligned} r_1 &= \frac{a_2 \times a_3}{\|a_2 \times a_3\|} \\ r_2 &= r_3 \times r_1 \\ r_3 &= \rho a_3 \\ T &= \rho K^{-1} b \end{aligned} \tag{19}$$

We leave the derivations as a class exercise or you can refer to Section 5.3.1 of the Forsyth & Ponce textbook.

With the calibration procedure complete, we warn against degenerate cases. Not all sets of n correspondences will work. For example, if the points P_i lie on the same plane, then the system will not be able to be solved. These unsolvable configurations of points are known as *degenerate configurations*. More generally, degenerate configurations have points that lie on the intersection curve of two quadric surfaces. Although this outside the scope of the class, you can find more information in Section 1.3 of the Forsyth & Ponce textbook.

1. Advantages of geometric camera calibration: very simple to formulate, and we get an analytical solution.
2. Disadvantages of geometric camera calibration: doesn't model radial distortion, hard to impose constraints (such as focal length), and doesn't minimize the correct error function (as we saw in projective transformations estimation).

Another way to perform calibration is to do **multi-plane calibration**. We take a plane (for example a chess board) and we take multiple pictures of it. From picture to picture, we change the orientation of the plane, effectively doing the same thing as in figure 66.

9.7 Handling Distortion in Camera Calibration

So far, we have been working with ideal lenses which are free from any distortion. However, as seen before, real lenses can deviate from rectilinear projection, which require more advanced methods. This section provides just a brief introduction to handling distortions.

Often, distortions are radially symmetric because of the physical symmetry of the lens. We model the radial distortion with an isotropic transformation:

$$QP_i = \begin{bmatrix} \frac{1}{\lambda} & 0 & 0 \\ 0 & \frac{1}{\lambda} & 0 \\ 0 & 0 & 1 \end{bmatrix} MP_i = \begin{bmatrix} u_i \\ v_i \end{bmatrix} = p_i \tag{20}$$

If we try to rewrite this into a system of equations as before, we get

$$\begin{aligned} u_i q_3 P_i &= q_1 P_i \\ v_i q_3 P_i &= q_2 P_i \end{aligned}$$

This system, however, is no longer linear, and we require the use of nonlinear optimization techniques, which are covered in Section 22.2 of Forsyth & Ponce. We can simplify the nonlinear optimization of the calibration problem if we make certain assumptions. In radial distortion, we note that the ratio between two coordinates u_i and v_i is not affected. We can compute this ratio as

$$\frac{u_i}{v_i} = \frac{\frac{m_1 P_i}{m_3 P_i}}{\frac{m_2 P_i}{m_3 P_i}} = \frac{m_1 P_i}{m_2 P_i} \tag{21}$$

Assuming that n correspondences are available, we can set up the system of linear equations:

$$\begin{aligned} v_1(m_1 P_1) - u_1(m_2 P_1) &= 0 \\ &\vdots \\ v_n(m_1 P_n) - u_n(m_2 P_n) &= 0 \end{aligned}$$

Similar to before, this gives a matrix-vector product that we can solve via SVD:

$$Ln = \begin{bmatrix} v_1 P_1^T & -u_1 P_1^T \\ \vdots & \vdots \\ v_n P_n^T & -u_n P_n^T \end{bmatrix} \begin{bmatrix} m_1^T \\ m_2^T \end{bmatrix} \tag{22}$$

Once m_1 and m_2 are estimated, m_3 can be expressed as a nonlinear function of m_1, m_2, and λ. This requires to solve a nonlinear optimization problem whose complexity is much simpler than the original one.

9.8 Appendix A: Rigid Transformations

The basic rigid transformations are rotation, translation, and scaling. This appendix will cover them for the 3D case, as they are common type in this class.

Rotating a point in 3D space can be represented by rotating around each of the three coordinate axes respectively. When rotating around the coordinate axes, common convention is to rotate in a counter-clockwise direction. One intuitive way to think of rotations is how much we rotate around each degree of freedom, which is often referred to as *Euler angles*. However, this methodology can result in what is known as *singularities*, or *gimbal lock*, in which certain configurations result in a loss of a degree of freedom for the rotation.

One way to prevent this is to use rotation matrices, which are a more general form of representing rotations. Rotation matrices are square, orthogonal matrices with determinant one. Given a rotation matrix R and a vector v, we can compute the resulting vector v' as

$$v' = Rv$$

Since rotation matrices are a very general representation of matrices, we can represent a rotation α, β, γ around each of the respective axes as follows:

$$R_x(\alpha) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\alpha & -\sin\alpha \\ 0 & \sin\alpha & \cos\alpha \end{bmatrix}$$

$$R_y(\beta) = \begin{bmatrix} \cos\beta & 0 & \sin\beta \\ 0 & 1 & 0 \\ -\sin\beta & 0 & \cos\beta \end{bmatrix}$$

$$R_z(\gamma) = \begin{bmatrix} \cos\gamma & -\sin\gamma & 0 \\ \sin\gamma & \cos\gamma & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Due to the convention of matrix multiplication, the rotation achieved by first rotating around the z-axis, then y-axis, then x-axis is given by the matrix product $R_x R_y R_z$.

Translations, or displacements, are used to describe the movement in a certain direction. In 3D space, we define a translation vector t with 3 values: the displacements in each of the 3 axes, often denoted as t_x, t_y, t_z. Thus, given some point P which is translated to some other point P' by t, we can write it as:

$$P' = P + t = \begin{bmatrix} P_x \\ P_y \\ P_z \end{bmatrix} + \begin{bmatrix} t_x \\ t_y \\ t_z \end{bmatrix}$$

In matrix form, translations can be written using homogeneous coordinates. If we construct a translation matrix as

$$T = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

then we see that $P' = TP$ is equivalent to $P' = P + t$.

If we want to combine translation with our rotation matrix multiplication, we can again use homogeneous coordinates to our advantage. If we want to rotate a vector v by R and then translate it by t, we can write the resulting vector v' as:

$$\begin{bmatrix} v' \\ 1 \end{bmatrix} = \begin{bmatrix} R & t \\ 0 & 1 \end{bmatrix} \begin{bmatrix} v \\ 1 \end{bmatrix}$$

Finally, if we want to scale the vector in certain directions by some amount S_x, S_y, S_z, we can construct a scaling matrix

$$S = \begin{bmatrix} S_x & 0 & 0 \\ 0 & S_y & 0 \\ 0 & 0 & S_z \end{bmatrix}$$

Therefore, if we want to scale a vector, then rotate, then translate, our final transformation matrix would be:

$$T = \begin{bmatrix} RS & t \\ 0 & 1 \end{bmatrix}$$

Note that all of these types of transformations would be examples of affine transformations. Recall that projective transformations occur when the final row of T is not $\begin{bmatrix} 0 & 0 & 0 & 1 \end{bmatrix}$.

9.9 Appendix B: Different Camera Models

We will now describe a simple model known as the *weak perspective model*. In the weak perspective model, points are first projected to the reference plane using orthogonal projection and then projected to the image plane using a projective transformation.

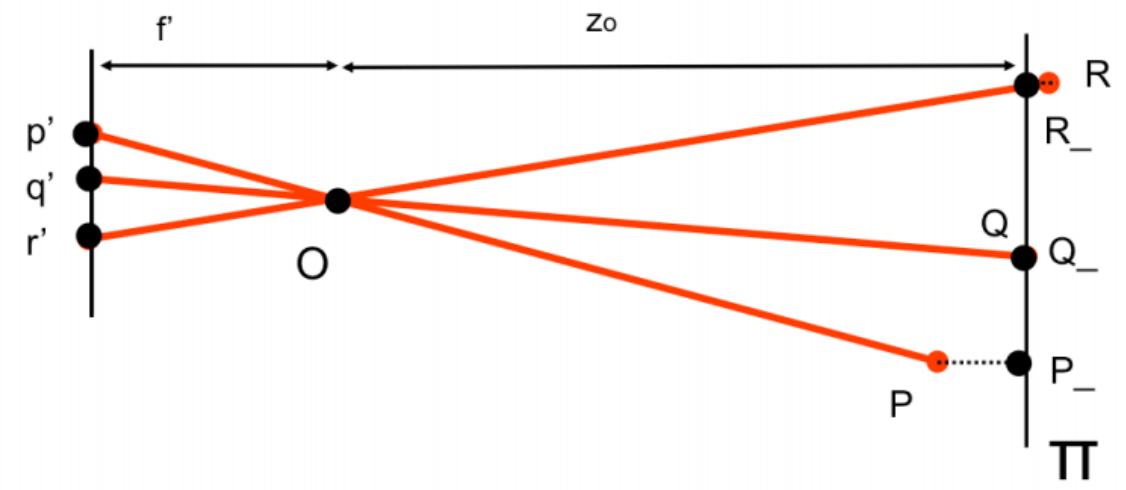


Figure 67: The weak perspective model: orthogonal projection onto reference plane

As Figure 67 shows, given a reference plane Π at a distance z_o from the center of the camera, the points P, Q, R are first projected to the plane Π using an orthogonal projection, generating points P_-, Q_-, R_-. This is a reasonable approximation when deviations in depth from the plane are small compared to the distance of the camera.

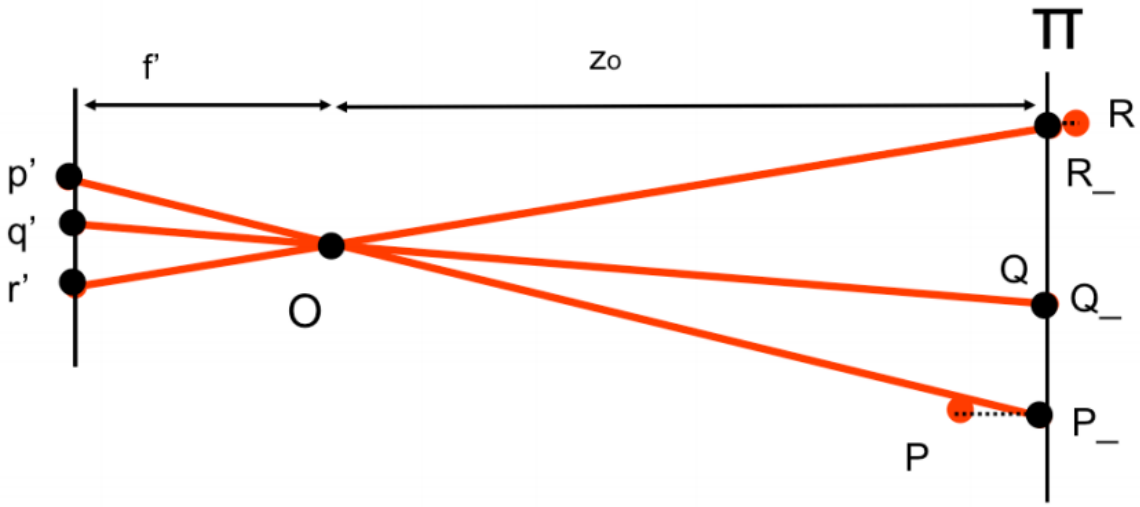


Figure 68: The weak perspective model: projection onto the image plane

Figure 68 illustrates how points P_-, Q_-, R_- are then projected to the image plane using a regular projective transformation to produce the points p', q', r'. Notice, however, that because we have approximated the depth of each point to z_o the projection has been reduced to a simple, constant magnification. The magnification is equal to the focal length f' divided by z_o, leading to

$$x' = \frac{f'}{z_0}x \qquad y' = \frac{f'}{z_0}y$$

This model also simplifies the projection matrix

$$M = \begin{bmatrix} A & b \\ 0 & 1 \end{bmatrix}$$

As we see, the last row of M is $\begin{bmatrix} 0 & 0 & 0 & 1 \end{bmatrix}$ in the weak perspective model, compared to $\begin{bmatrix} v & 1 \end{bmatrix}$ in the normal camera model. We do not prove this result and leave it to you as an exercise. The simplification is clearly demonstrated when mapping the

3D points to the image plane.

$$P' = MP = \begin{bmatrix} m_1 \\ m_2 \\ m_3 \end{bmatrix} P = \begin{bmatrix} m_1 P \\ m_2 P \\ 1 \end{bmatrix} \tag{23}$$

Thus, we see that the image plane point ultimately becomes a magnification of the original 3D point, irrespective of depth. The nonlinearity of the projective transformation disappears, making the weak perspective transformation a mere magnifier.

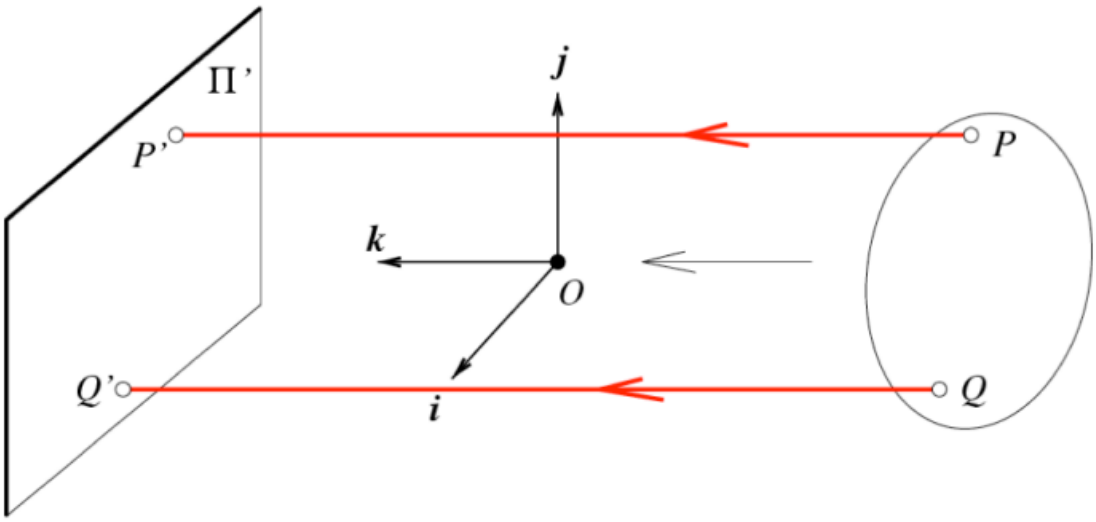


Figure 69: The orthographic projection model

Further simplification leads to the *orthographic (or affine) projection model*. In this case, the optical center is located at infinity. The projection rays are now perpendicular to the retinal plane. As a result, this model ignores depth altogether. Therefore,

$$x' = x$$
$$y' = y$$

Orthographic projection models are often used for architecture and industrial design.

Overall, weak perspective models result in much simpler math, at the cost of being somewhat imprecise. However, it often yields results that are very accurate when the object is small and distant from the camera.

10 Epipolar Geometry

10.1 Introduction

Previously, we have seen how to compute the intrinsic and extrinsic parameters of a camera using one or more views using a typical camera calibration procedure or single view metrology. This process culminated in deriving properties about the 3D world from one image. However, in general, it is not possible to recover the entire structure of the 3D world from just one image. This is due to the intrinsic ambiguity of the 3D to the 2D mapping: some information is simply lost.

Figure 70: A single picture such as this picture of a man holding up the Leaning Tower of Pisa can result in ambiguous scenarios. Multiple views of the same scene help us resolve these potential ambiguities.

For example, in Figure 70, we may be initially fooled to believe that the man is holding up the Leaning Tower of Pisa. Only by careful inspection can we tell that this is not the case and merely an illusion based on the projection of different depths onto the image plane. However, if we were able to view this scene from a completely different angle, this illusion immediately disappears and we would instantly figure out the correct scene layout.

The focus of these lecture notes is to show how having knowledge of geometry when multiple cameras are present can be extremely helpful. Specifically, we will first focus on defining the geometry involved in two viewpoints and then present how this geometry can aid in further understanding the world around us.

10.2 Epipolar Geometry

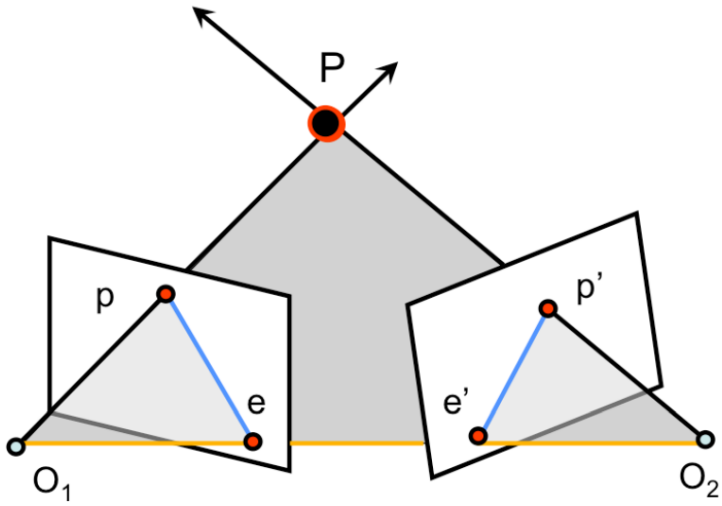


Figure 71: The general setup of epipolar geometry. The gray region is the epipolar plane. The orange line is the baseline, while the two blue lines are the epipolar lines.

Often in multiple view geometry, there are interesting relationships between the multiple cameras, a 3D point, and that point's projections in each of the camera's image plane. The geometry that relates the cameras, points in 3D, and the the corresponding observations is referred to as the *epipolar geometry* of a stereo pair.

As illustrated in Figure 71, the standard epipolar geometry setup involves two cameras observing the same 3D point P, whose projection in each of the image planes is located at p and p' respectively. The camera centers are located at O_1 and

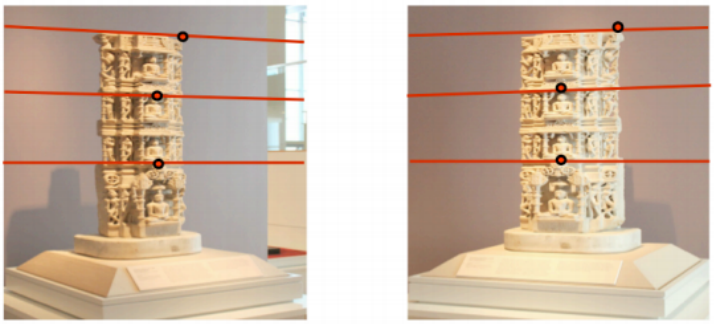


Figure 72: An example of epipolar lines and their corresponding points drawn on an image pair.

O_2, and the line between them is referred to as the *baseline*. We call the plane defined by the two camera centers and P the *epipolar plane*. The locations of where the baseline intersects the two image planes are known as the the *epipoles* e and e'. Finally, the lines defined by the intersection of the epipolar plane and the two image planes are known as the *epipolar lines*. The epipolar lines have the property that they intersect the baseline at the respective epipoles in the image plane (all possible epipolar lines hold this property).

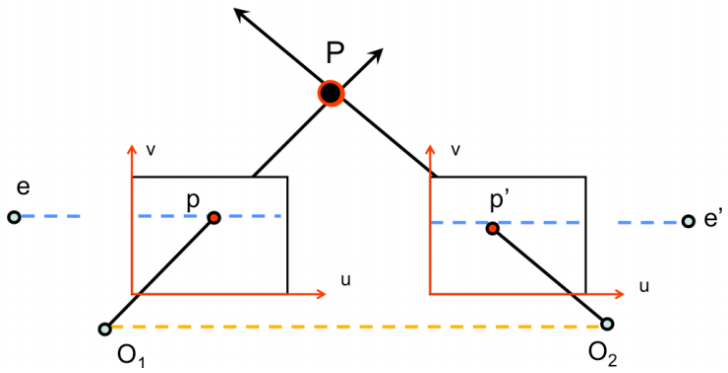


Figure 73: When the two image planes are parallel, then the epipoles e and e' are located at infinity. Notice that the epipolar lines are parallel to the u axis of each image plane.

An interesting case of epipolar geometry is shown in Figure 73, which occurs when the image planes are parallel to each other. When the image planes are parallel to each other, then the epipoles e and e' will be located at infinity since the baseline joining the centers O_1, O_2 is parallel to the image planes. Another important byproduct of this case is that the epipolar lines are parallel to an axis of each image plane. This case is especially useful and will be covered in greater detail in the subsequent section on image rectification.

In real world situations, however, we are not given the exact location of the 3D location P, but can determine its projection in one of the image planes p. We also should be able to know the cameras locations, orientations, and camera matrices. What can we do with this knowledge? With the knowledge of camera locations O_1, O_2 and the image point p, we can define the epipolar plane. With this epipolar plane, we can then determine the epipolar lines[4]. By definition, P's projection into the second image p' must be located on the epipolar line of the second image, as can be seen in Figure 74. Thus, a basic understanding of epipolar geometry allows us to create a strong constraint between image pairs without knowing the 3D structure of the scene.

[4]This means that epipolar lines can be determined by just knowing the camera centers O_1, O_2 and a point in one of the images p

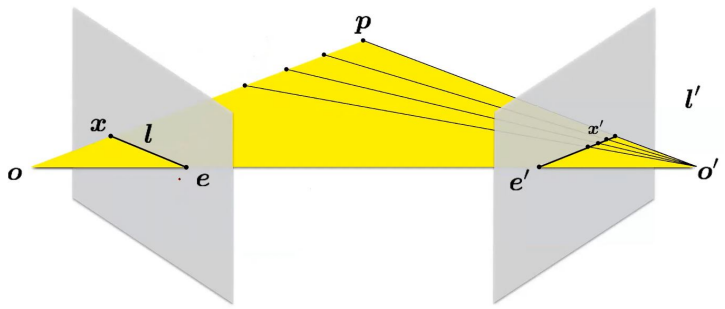


Figure 74: The epipolar constraint. Given an image point x in the left image, we know that it corresponds to some image point x' on the right image, which lies somewhere along the epipolar line ℓ' (this is a strong constraint on the location of x' in the right image)

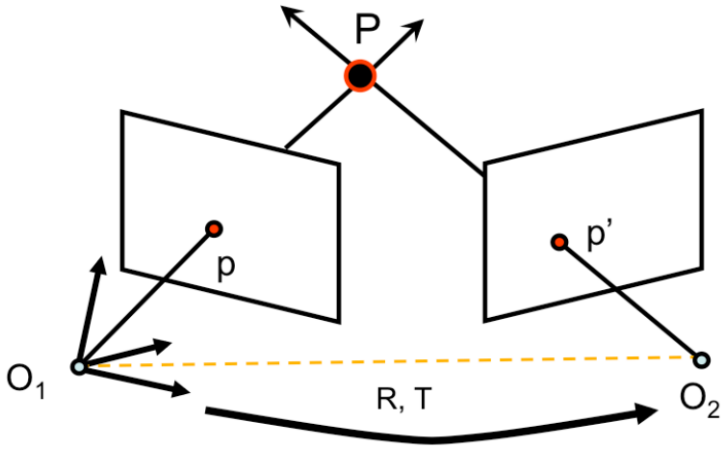


Figure 75: The setup for determining the essential and fundamental matrices, which help map points and epipolar lines across views.

We will now try to develop seamless ways to map points and epipolar lines across views. If we take the setup given in the original epipolar geometry framework (Figure 75), then we shall further define M and M' to be the camera projection matrices that map 3D points into their respective 2D image plane locations. Let us assume that the world reference system is associated to the first camera with the second camera offset first by a rotation R and then by a translation T. This specifies the camera projection matrices to be:

$$M = K \begin{bmatrix} I & 0 \end{bmatrix} \qquad M' = K' \begin{bmatrix} R & T \end{bmatrix} \tag{24}$$

10.3 The Essential Matrix

Our next goal is to build a matrix E, called the essential matrix, such that if we take a point p in the image plane of the 1st camera, then $Ep = l'$ will be the epipolar line in the image plane of the 2nd camera.

In the simplest case, let us assume that we have *canonical cameras*, in which $K = K' = I$. This reduces Equation 24 to

$$M = \begin{bmatrix} I & 0 \end{bmatrix} \qquad M' = \begin{bmatrix} R & T \end{bmatrix} \tag{25}$$

Furthermore, this means that the location of p' (in homogeneous coordinates, of course) in the first camera's reference system is $R^{-1}(p' - T) = R^T p' - R^T T$. Since the vectors $R^T p' - R^T T$ and $R^T T$ lie in the epipolar plane, then if we take the cross product of $R^T T \times (R^T p' - R^T T) = R^T T \times R^T p' = R^T (T \times p')$, we will get a vector normal to the epipolar plane. This also means that p, which lies in the epipolar plane, is normal to $R^T (T \times p')$, giving us the constraint that their dot product is zero:

$$\begin{aligned} (R^T (T \times p'))^T p &= 0 \\ (T \times p')^T R p &= 0 \end{aligned} \tag{26}$$

From linear algebra, we can introduce a different and compact expression for the cross product: we can represent the cross product between any two vectors a and b as a matrix-vector multiplication:

$$\mathbf{a} \times \mathbf{b} = \begin{bmatrix} 0 & -\mathbf{a}_z & \mathbf{a}_y \\ \mathbf{a}_z & 0 & -\mathbf{a}_x \\ -\mathbf{a}_y & \mathbf{a}_x & 0 \end{bmatrix} \begin{bmatrix} \mathbf{b}_x \\ \mathbf{b}_y \\ \mathbf{b}_z \end{bmatrix} = [\mathbf{a}_\times]\mathbf{b} \tag{27}$$

Combining this expression with Equation 26, we can convert the cross product term into matrix multiplication, giving

$$\begin{aligned} ([T_\times]p')^T Rp &= 0 \\ p'^T [T_\times]^T Rp &= 0 \\ p'^T [T_\times] Rp &= 0 \end{aligned} \tag{28}$$

The matrix $E = [T_\times]R$ is known as the *Essential Matrix*, creating a compact expression for the epipolar constraint:

$$p'^T Ep = 0 \tag{29}$$

The Essential matrix is a 3×3 matrix that contains 5 degrees of freedom. It has rank 2 and is singular.

The Essential matrix is useful for computing the epipolar lines associated with p and p'. For instance, since any point p' on the line ℓ' holds $p'^T \ell' = 0$, we get that $\ell' = Ep$ gives the epipolar line in the image plane of camera 2. Similarly $\ell = E^T p'$ gives the epipolar line in the image plane of camera 1. Other interesting properties of the essential matrix is that its dot product with the epipoles equate to zero: $Ee = E^T e' = 0$: for any point x (other than e) in the image of camera 1, the corresponding epipolar line in the image of camera 2, $l' = Ex$, contains the epipole e'. Thus e' satisfies $e'^T(Ex) = (e'^T E)x = 0$ for all the x, so $e'^T E = 0$. Similarly $Ee = 0$.

10.4 The Fundamental Matrix

Although we derived a relationship between p and p' when we have canonical cameras, we should be able to find a more general expression when the cameras are no longer canonical. Recall that gives us the projection matrices:

$$M = K \begin{bmatrix} I & 0 \end{bmatrix} \qquad M' = K' \begin{bmatrix} R & T \end{bmatrix} \tag{30}$$

First, let p and p' be the projections of a point P in 3D to the image planes of the two cameras, respectively. Thus, we know that $p = MP = K \begin{bmatrix} I & 0 \end{bmatrix} P = Kp_c$ and $p' = M'P = K' \begin{bmatrix} R & T \end{bmatrix} P = K'p'_c$, where p_c and p'_c are the canonical projections of P in the reference camera's coordinate system. We then get:

$$\begin{aligned} p_c &= K^{-1} p \\ p'_c &= K'^{-1} p' \end{aligned} \tag{31}$$

which are the projections of P to the corresponding camera images if the cameras were canonical. Recall that in the canonical case:

$$p'^T_c [T_\times] R p_c = 0 \tag{32}$$

By substituting in the values of p_c and p'_c, we get

$$p'^T K'^{-T} [T_\times] R K^{-1} p = 0 \tag{33}$$

The matrix $F = K'^{-T}[T_\times]RK^{-1}$ is known as the *Fundamental Matrix*, which acts similar to the Essential matrix from the previous section but also encodes information about the camera matrices K, K' and the relative translation T and rotation R between the cameras. Therefore, it is also useful in computing the epipolar lines associated with p and p', even when the camera matrices K, K' and the transformation R, T are unknown. Similar to the Essential matrix, we can compute the epipolar lines $\ell' = Fp$ and $\ell = F^T p'$ from just the Fundamental matrix and the corresponding points. One main difference between the Fundamental matrix and the Essential matrix is that the Fundamental matrix contains 7 degrees of freedom, compared to the Essential matrix's 5 degrees of freedom.

But how is the Fundamental matrix useful? Like the Essential matrix, if we know the Fundamental matrix, then simply knowing a point in an image gives us an easy constraint (the epipolar line) of the corresponding point in the other image. Therefore, without knowing the actual position of P in 3D space, or any of the extrinsic or intrinsic characteristics of the cameras, we can establish a relationship between any p and p'.

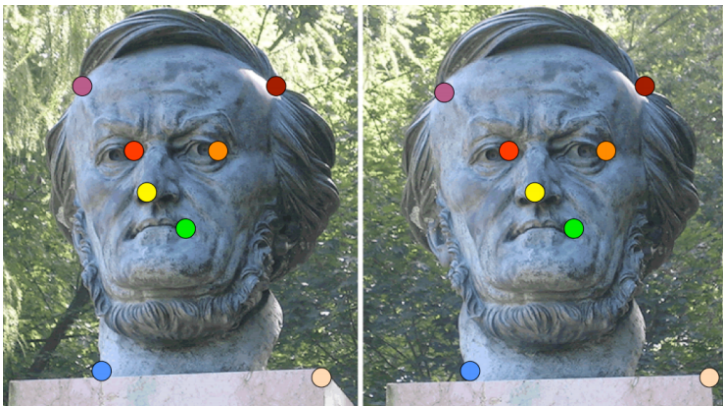


Figure 76: Corresponding points are drawn in the same color on each of the respective images.

10.4.1 The Eight-Point Algorithm

Still, the assumption that we can have the Fundamental matrix, which is defined by a matrix product of the camera parameters, seems rather large. However, it is possible to estimate the Fundamental matrix given two images of the same scene and without knowing the extrinsic or intrinsic parameters of the camera. The method we discuss for doing so is known as the *Eight-Point Algorithm*, which was proposed by Longuet-Higgins in 1981 and extended by Hartley in 1995. As the title suggests, the Eight-Point Algorithm assumes that a set of at least 8 pairs of corresponding points between two images is available.

Each correspondence $p_i = (u_i, v_i, 1)$ and $p'_i = (u'_i, v'_i, 1)$ gives us the epipolar constraint $p'^T_i F p_i = 0$. We can reformulate the constraint as follows:

$$\begin{bmatrix} u_i u'_i & v_i u'_i & u'_i & u_i v'_i & v_i v'_i & v'_i & u_i & v_i & 1 \end{bmatrix} \begin{bmatrix} F_{11} \\ F_{12} \\ F_{13} \\ F_{21} \\ F_{22} \\ F_{23} \\ F_{31} \\ F_{32} \\ F_{33} \end{bmatrix} = 0 \tag{34}$$

Since this constraint is a scalar equation, it only constrains one degree of freedom. Since we can only know the Fundamental matrix up to scale, we require eight of these constraints to determine the Fundamental matrix:

$$\begin{bmatrix} u_1 u'_1 & v_1 u'_1 & u'_1 & u_1 v'_1 & v_1 v'_1 & v'_1 & u_1 & v_1 & 1 \\ u_2 u'_2 & v_2 u'_2 & u'_2 & u_2 v'_2 & v_2 v'_2 & v'_2 & u_2 & v_2 & 1 \\ u_3 u'_3 & v_3 u'_3 & u'_3 & u_3 v'_3 & v_3 v'_3 & v'_3 & u_3 & v_3 & 1 \\ u_4 u'_4 & v_4 u'_4 & u'_4 & u_4 v'_4 & v_4 v'_4 & v'_4 & u_4 & v_4 & 1 \\ u_5 u'_5 & v_5 u'_5 & u'_5 & u_5 v'_5 & v_5 v'_5 & v'_5 & u_5 & v_5 & 1 \\ u_6 u'_6 & v_6 u'_6 & u'_6 & u_6 v'_6 & v_6 v'_6 & v'_6 & u_6 & v_6 & 1 \\ u_7 u'_7 & v_7 u'_7 & u'_7 & u_7 v'_7 & v_7 v'_7 & v'_7 & u_7 & v_7 & 1 \\ u_8 u'_8 & v_8 u'_8 & u'_8 & u_8 v'_8 & v_8 v'_8 & v'_8 & u_8 & v_8 & 1 \end{bmatrix} \begin{bmatrix} F_{11} \\ F_{12} \\ F_{13} \\ F_{21} \\ F_{22} \\ F_{23} \\ F_{31} \\ F_{32} \\ F_{33} \end{bmatrix} = 0 \tag{35}$$

This can be compactly written as

$$W\mathbf{f} = 0 \tag{36}$$

where W is an $N \times 9$ matrix derived from $N \geq 8$ correspondences and $\mathbf{f}$ is the values of the Fundamental matrix we desire.

In practice, it often is better to use more than eight correspondences and create a larger W matrix because it reduces the effects of noisy measurements. The solution to this system of homogeneous equations can be found in the least-squares sense by Singular Value Decomposition (SVD), as W is rank-deficient. SVD will give us a estimate of the Fundamental matrix $\hat{F}$, which may have full rank. However, we know that the true Fundamental matrix has rank 2 (since $Fe = 0$, where e is the epipole). Therefore, we should look for a solution that is the best rank-2 approximation of $\hat{F}$. To do so, we solve the following optimization problem:

$$\begin{aligned} \underset{F}{\text{minimize}} \quad & \|F - \hat{F}\|_{\text{F}} \\ \text{subject to} \quad & \det F = 0 \end{aligned} \tag{37}$$

This problem is solved again by SVD, where $\hat{F} = U\Sigma V^T$, then the best rank-2 approximation is found by

$$F = U \begin{bmatrix} \Sigma_1 & 0 & 0 \\ 0 & \Sigma_2 & 0 \\ 0 & 0 & 0 \end{bmatrix} V^T \tag{38}$$

10.4.2 The Normalized Eight-Point Algorithm

In practice, the standard least-squares approach to the Eight-Point Algorithm is not precise. Often, the distance between a point p_i and its corresponding epipolar line $\ell_i = F^T p'$ will be very large, usually on the scale of 10+ pixels. To reduce this error, we can consider a modified version of the Eight-Point Algorithm called the *Normalized Eight-Point Algorithm*.

The main problem of the standard Eight-Point Algorithm stems from the fact that W is ill-conditioned for SVD. For SVD to work properly, W should have one singular value equal to (or near) zero, with the other singular values being nonzero. However, the correspondences $p_i = (u_i, v_i, 1)$ will often have extremely large values in the first and second coordinates due to the pixel range of a modern camera (i.e. $p_i = (1832, 1023, 1)$). If the image points used to construct W are in a relatively small region of the image, then each of the vectors for p_i and p'_i will generally be very similar. Consequently, the constructed W matrix will have one very large singular value, with the rest relatively small.

To solve this problem, we will normalize the points in the image before constructing W. This means we *pre-condition* W by applying both a translation and scaling on the image coordinates such that two requirements are satisfied. First, the origin of the new coordinate system should be located at the centroid of the image points (translation). Second, the mean square distance of the transformed image points from the origin should be 2 pixels (scaling). We can compactly represent this process by a transformation matrices T, T' that translate by the centroid and scale by the scaling factor $\frac{2}{\text{mean distance}}$ for each respective image.

Afterwards, we normalize the coordinates:

$$q_i = Tp_i \qquad q'_i = T'p'_i \tag{39}$$

Using the new, normalized coordinates, we can compute the new F_q using the regular least-squares Eight Point Algorithm. However, the matrix F_q is the fundamental matrix for the normalized coordinates. For it to be usable on regular coordinate space, we need to de-normalize it, giving

$$F = T'^T F_q T \tag{40}$$

Ultimately, this new Fundamental matrix F gives good results in real-world applications.

11 Triangulation and Structure From Motion

11.1 Introduction

In the previous notes, we covered how adding additional viewpoints of a scene can greatly enhance our knowledge of the said scene. We focused on the epipolar geometry setup in order to relate points of one image plane to points in the other without extracting any information about the 3D scene. In these lecture notes, we will discuss how to recover information about the 3D scene from multiple 2D images.

11.2 Triangulation

One of the most fundamental problems in multiple view geometry is the problem of *triangulation*, the process of determining the location of a 3D point given its projections into two or more images. w

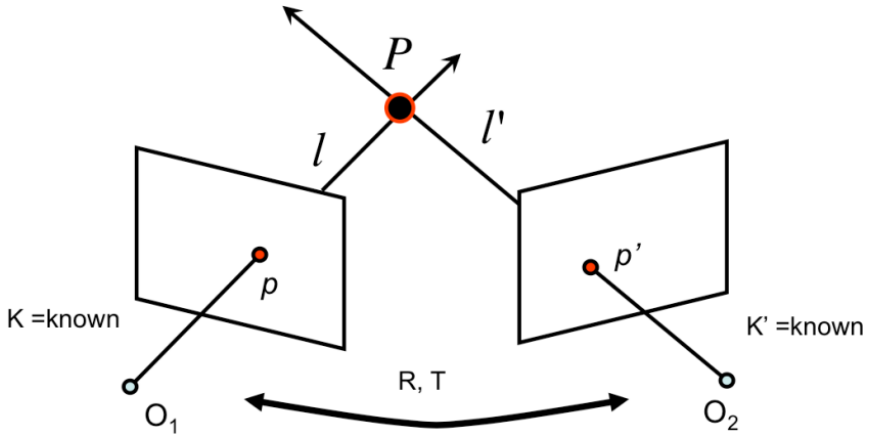


Figure 77: The setup of the triangulation problem when given two views.

In the triangulation problem with two views, we have two cameras with known camera intrinsic parameters K and K' respectively. We also know the relative orientations and offsets R, T of these cameras with respect to each other. Suppose that we have a point P in 3D, which can be found in the images of the two cameras at p and p' respectively. Although the location of P is currently unknown, we can measure the exact locations of p and p' in the image. Because K, K', R, T are known, we can compute the two lines of sight ℓ and ℓ', which are defined by the camera centers O_1, O_2 and the image locations p, p'. Therefore, P can be computed as the intersection of ℓ and ℓ'.

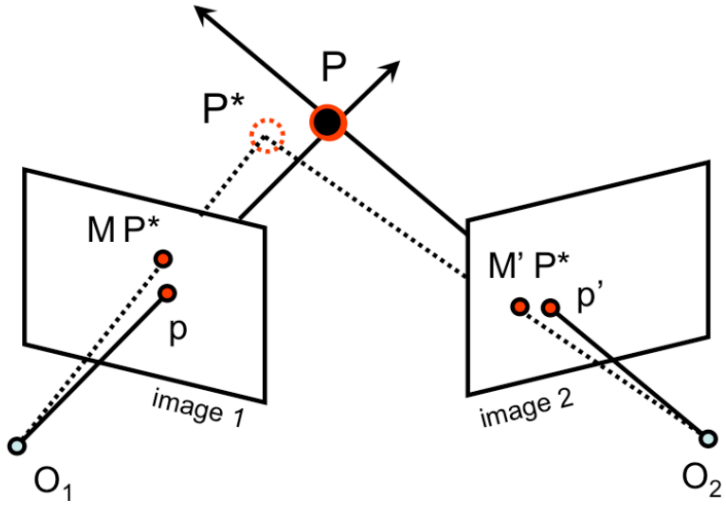


Figure 78: The triangulation problem in real-world scenarios often involves minimizing the reprojection error.

Although this process appears both straightforward and mathematically sound, it does not work very well in practice. In the real world, because the observations p and p' are noisy and the camera calibration parameters are not precise, finding the intersection point of ℓ and ℓ' may be problematic. In most cases, it will not exist at all, as the two lines may never intersect.

11.2.1 A linear method for triangulation

In this section, we describe a simple linear triangulation method that solves the lack of an intersection point between rays. We are given two points in the images that correspond to each other $p = MP = (x, y, 1)$ and $p' = M'P = (x', y', 1)$. By the definition of the cross product ($a \times b = \begin{pmatrix} a_2b_3 - a_3b_2 \\ a_3b_1 - a_1b_3 \\ a_1b_2 - a_2b_1 \end{pmatrix}$), we get $p \times (MP) = 0$. We can explicitly use the equalities generated by the cross product to form three constraints:

$$\begin{aligned} x(M_3P) - (M_1P) &= 0 \\ y(M_3P) - (M_2P) &= 0 \\ x(M_2P) - y(M_1P) &= 0 \end{aligned} \tag{41}$$

where M_i is the i-th row of the matrix M. These are actually 2 constraints, since the 3rd constraint is linearly dependent on the 1st and 2nd constraints (so we'll use only the 1st and 2nd constraints). Similar constraints can be formulated for p' and M'. Using the constraints from both images, we can formulate a linear equation of the form $AP = 0$ where

$$A = \begin{bmatrix} xM_3 - M_1 \\ yM_3 - M_2 \\ x'M_3' - M_1' \\ y'M_3' - M_2' \end{bmatrix} \tag{42}$$

This equation can be solved using SVD to find the best linear estimate of the point P. Another interesting aspect of this method is that it can actually handle triangulating from multiple views as well. To do so, one simply appends additional rows to A corresponding to the added constraints by the new views.

This method, however is not suitable for projective reconstruction, as it is not projective-invariant. For example, suppose we replace the camera matrices M, M' with ones affected by a projective transformation $MH^{-1}, M'H^{-1}$. The matrix of linear equations A then becomes AH^{-1}. Therefore, a solution P to the previous estimation of $AP = 0$ will correspond to a solution HP for the transformed problem $(AH^{-1})(HP) = 0$. Recall that SVD solves for the constraint that $\|P\| = 1$, which is not invariant under a projective transformation H. Therefore, this method, although simple, is often not the optimal solution to the triangulation problem. -

11.2.2 A nonlinear method for triangulation

Instead, the triangulation problem for real-world scenarios is often mathematically characterized as solving a minimization problem:

$$\min_{\hat{P}} \|M\hat{P} - p\|^2 + \|M'\hat{P} - p'\|^2 \tag{43}$$

In the above equation, we seek to find a $\hat{P}$ in 3D that best approximates P by finding the best least-squares estimate of the *reprojection error* of $\hat{P}$ in both images. The reprojection error for a 3D point in an image is the distance between the projection of that point in the image and the corresponding observed point in the image plane. In the case of our example in Figure 78, since M is the projective transformation from 3D space to image 1, the projected point of $\hat{P}$ in image 1 is $M\hat{P}$. The matching observation of $\hat{P}$ in image 1 is p. Thus, the reprojection error for point P in image 1 is the distance $\|M\hat{P} - p\|$. The overall reprojection error found in Equation 43 is the sum of the reprojection errors across all the points in the image. For cases with more than two images, we would simply add more distance terms to the objective function.

$$\min_{\hat{P}} \sum_i \|M\hat{P}_i - p_i\|^2 \tag{44}$$

In practice, there exists a variety of very sophisticated optimization techniques that result in good approximations to the problem. However, for the scope of the class, we will focus on only one of these techniques, which is the Gauss-Newton algorithm for nonlinear least squares. The general nonlinear least squares problem is to find an $x \in \mathbb{R}^n$ that minimizes

$$\|r(x)\|^2 = \sum_{i=1}^{m} r_i(x)^2 \tag{45}$$

where r is any residual function $r : \mathbb{R}^n \to \mathbb{R}^m$ such that $r(x) = f(x) - y$ for some function f, input x, and observation y. The nonlinear least squares problem reduces to the regular, linear least squares problem when the function f is linear. However, recall that, in general, our camera matrices are not affine. Because the projection into the image plane often involves a division by the homogeneous coordinate, the projection into the image is generally nonlinear.

Notice that if we set e_i to be a 2×1 vector $e_i = M\hat{P}_i - p_i$, then we can reformulate our optimization problem to be:

$$\min_{\hat{P}} \sum_i e_i(\hat{P})^2 \tag{46}$$

which can be perfectly represented as a nonlinear least squares problem.

In these notes, we will cover how we can use the popular Gauss-Newton algorithm to find an approximate solution to this nonlinear least squares problem. First, let us assume that we have a somewhat reasonable estimate of the 3D point $\hat{P}$, which we can compute by the previous linear method. The key insight of the Gauss-Newton algorithm is to update our estimate by correcting it towards an even better estimate that minimizes the reprojection error. At each step we want to update our estimate $\hat{P}$ by some δ_P: $\hat{P} = \hat{P} + \delta_P$.

But how do we choose the update parameter δ_P? The key insight of the Gauss-Newton algorithm is to linearize the residual function near the current estimate $\hat{P}$. In the case of our problem, this means that the residual error e of a point P can be thought of as:

$$e(\hat{P} + \delta_P) \approx e(\hat{P}) + \frac{\partial e}{\partial P}\delta_P \tag{47}$$

Subsequently, the minimization problem transforms into

$$\min_{\delta_P} \| \frac{\partial e}{\partial P}\delta_P - (-e(\hat{P})) \|^2 \tag{48}$$

When we formulate the residual like this, we can see that it takes the format of the standard linear least squares problem. For the triangulation problem with N images, the linear least squares solution is

$$\delta_P = -(J^T J)^{-1} J^T e \tag{49}$$

where

$$e = \begin{bmatrix} e_1 \\ \vdots \\ e_N \end{bmatrix} = \begin{bmatrix} p_1 - M_1\hat{P} \\ \vdots \\ p_n - M_n\hat{P} \end{bmatrix} \tag{50}$$

and

$$J = \begin{bmatrix} \frac{\partial e_1}{\partial \hat{P}_1} & \frac{\partial e_1}{\partial \hat{P}_2} & \frac{\partial e_1}{\partial \hat{P}_3} \\ \vdots & \vdots & \vdots \\ \frac{\partial e_N}{\partial \hat{P}_1} & \frac{\partial e_N}{\partial \hat{P}_2} & \frac{\partial e_N}{\partial \hat{P}_3} \end{bmatrix} \tag{51}$$

Recall that the residual error vector of a particular image e_i is a 2×1 vector because there are two dimensions in the image plane. Consequently, in the simplest two camera case ($N = 2$) of triangulation, this results in the residual vector e being a $2N \times 1 = 4 \times 1$ vector and the Jacobian J being a $2N \times 3 = 4 \times 3$ matrix. Notice how this method handles multiple views seamlessly, as additional images are accounted for by adding the corresponding rows to the e vector and J matrix. After computing the update δ_P, we can simply repeat the process for a fixed number of steps or until it numerically converges. One important property of the Gauss-Newton algorithm is that our assumption that the residual function is linear near our estimate gives us no guarantee of convergence. Thus, it is always useful in practice to put an upper bound on the number of updates made to the estimate.

11.3 Affine structure from motion (not in the syllabus)

This section is not in the syllabus of the course, but is a good starting point for the next section: perspective structure from motion.

At the end of the previous section, we hinted how we can go beyond two views of a scene to gain information about the 3D scene. We will now explore the extension of the geometry of two cameras to multiple cameras. By combining observations of points from multiple views, we will be able to simultaneously determine both the 3D structure of the scene and the parameters of the camera in what is known as *structure from motion*.

Here, we formally introduce the structure from motion problem. Suppose we have m cameras with camera transformations M_i encoding both the intrinsic and extrinsic parameters for the cameras. Let X_j be one of the n 3D points in the scene. Each 3D point may be visible in multiple cameras at the location x_{ij}, which is the projection of X_j to the image of the camera i using the projective transformation M_i. The aim of structure from motion is to recover both the structure of the scene (the n 3D points X_j) and the motion of the cameras (the m projection matrices M_i) from all the observations x_{ij}.

11.3.1 The affine structure from motion problem

Before tackling the general structure from motion problem, we will first start with a simpler problem, which assumes the cameras are affine or weak perspective. Ultimately, the lack of the perspective scaling operation makes the mathematical derivation easier for this problem.

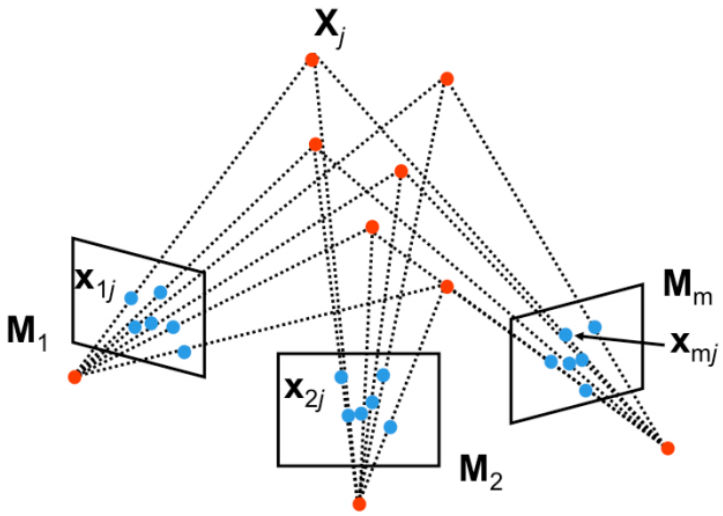


Figure 79: The setup of the general structure from motion problem.

Previously, we derived the above equations for perspective and weak perspective cases. Remember that in the full perspective model, the projection matrix is defined as

$$M = \begin{bmatrix} A & b \\ v & 1 \end{bmatrix} \tag{52}$$

where v is some non-zero 1×3 vector. On the other hand, for the weak perspective model, $v = 0$. We find that this property makes the homogeneous coordinate of MX equal to 1:

$$x = MX = \begin{bmatrix} & m_1 & \\ & m_2 & \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X_1 \\ X_2 \\ X_3 \\ 1 \end{bmatrix} = \begin{bmatrix} m_1 X \\ m_2 X \\ 1 \end{bmatrix} \tag{53}$$

Consequently, the nonlinearity of the projective transformation disappears as we move from homogeneous to Euclidean coordinates, and the weak perspective transformation acts as a mere magnifier. We can more compactly represent the projection as:

$$\begin{bmatrix} m_1 X \\ m_2 X \end{bmatrix} = \begin{bmatrix} A & b \end{bmatrix} X = A X_{het} + b \tag{54}$$

and represent any camera matrix in the format $M_{\text{affine}} = \begin{bmatrix} A & b \end{bmatrix}$. Thus, we now use the affine camera model to express the relationship from a point X_j in 3D and the corresponding observations in each affine camera (for instance, x_{ij} in camera i).

Returning to the structure from motion problem, we need to estimate m matrices M_i, and the n world coordinate vectors X_j, for a total of $8m + 3n$ unknowns, from mn observations. Each observation creates 2 constraints per camera (see the section about triangulation to remember why), so there are $2mn$ equations in $8m + 3n$ unknowns. We can use this equation to know the lower bound on the number of corresponding observations in each of the images that we need to have. For example, if we have $m = 2$ cameras, then we need to have at least $n = 16$ points in 3D. However, once we do have enough corresponding points labeled in each image, how do we solve this problem?

11.3.2 The Tomasi and Kanade factorization method

In this part, we outline Tomasi and Kanade's *factorization method* for solving the affine structure from motion problem. This method consists of two major steps: the data centering step and the actual factorization step.

Let's begin with the data centering step. In this step, the main idea is center the data at the origin. To do so, for each image i, we redefine new coordinates $\hat{x}_{ij}$ for each image point x_{ij} by subtracting out their centroid $\bar{x}_i$:

$$\hat{x}_{ij} = x_{ij} - \bar{x}_i = x_{ij} - \frac{1}{n} \sum_{j=1}^{n} x_{ij} \tag{55}$$

Recall that the affine structure from motion problem allows us to define the relationship between image points x_{ij}, the camera matrix variables A_i and b_i, and the 3D points X_j as:

$$x_{ij} = A_i X_j + b_i \tag{56}$$

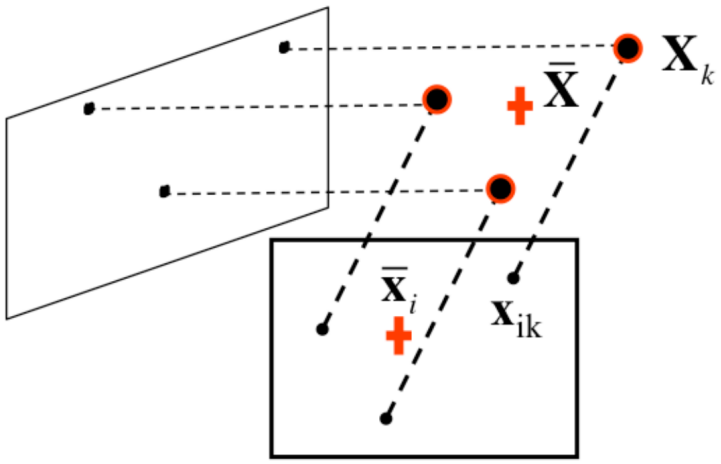


Figure 80: When applying the centering step, we translate all of the image points such that their centroid (denoted as the lower left red cross) is located at the origin in the image plane. Similarly, we place the world coordinate system such that the origin is at the centroid of the 3D points (denoted as the upper right red cross).

After this centering step, we can combine definition of the centered image points $\hat{x}_{ij}$ in Equation 55 and the affine expression in Equation 56:

$$\begin{aligned}
\hat{x}_{ij} &= x_{ij} - \frac{1}{n}\sum_{k=1}^{n} x_{ik} \\
&= A_i X_j - \frac{1}{n}\sum_{k=1}^{n} A_i X_k \\
&= A_i (X_j - \frac{1}{n}\sum_{k=1}^{n} X_k) \\
&= A_i (X_j - \bar{X}) \\
&= A_i \hat{X}_j
\end{aligned} \tag{57}$$

As we see from Equation 57, if we translate the origin of the world reference system to the centroid $\bar{X}$, then the centered coordinates of the image points $\hat{x}_{ij}$ and centered coordinates of the 3D points $\hat{X}_{ij}$ are related only by a single 2×3 matrix A_i. Ultimately, the centering step of the factorization method allows us to create a compact matrix product representation to relate the 3D structure with their observed points in multiple images.

However, notice that in the matrix product $\hat{x}_{ij} = A_i \hat{X}_j$, we only have access to the values on the left hand side of the equation. Thus, we must somehow factor out the motion matrices A_i and structure X_j. Using all the observations for all the cameras, we can build a measurement matrix D, made up of n observations in the m cameras (remember that each $\hat{x}_{ij}$ entry is a 2x1 vector):

$$D = \begin{bmatrix} \hat{x}_{11} & \hat{x}_{12} & \dots & \hat{x}_{1n} \\ \hat{x}_{21} & \hat{x}_{22} & \dots & \hat{x}_{2n} \\ & & \ddots & \\ \hat{x}_{m1} & \hat{x}_{m2} & \dots & \hat{x}_{mn} \end{bmatrix} \tag{58}$$

Now recall that because of our affine assumption, D can be expressed as the product of the $2m \times 3$ motion matrix M (which comprises the camera matrices $A_1, \dots A_m$) and the $3 \times n$ structure matrix S (which comprises the 3D points $X_1, \dots X_n$). An important fact that we will use is that $\text{rank}(D) = 3$ since D is the product of two matrices whose max dimension is 3.

To factorize D into M and S, we will use the singular value decomposition, $D = U\Sigma V^T$. Since we know the $\text{rank}(D) = 3$, so there will only be 3 non-zero singular values σ_1, σ_2, and σ_3 in Σ. Thus, we can further reduce the expression and obtain

the following decomposition:

$$
\begin{aligned}
D &= U\Sigma V^T \\
&= \begin{bmatrix} u_1 & \dots & u_n \end{bmatrix} \begin{bmatrix} \sigma_1 & 0 & 0 & 0 & \dots & 0 \\ 0 & \sigma_2 & 0 & 0 & \dots & 0 \\ 0 & 0 & \sigma_3 & 0 & \dots & 0 \\ 0 & 0 & 0 & 0 & \dots & 0 \\ & & & & \ddots & \\ 0 & 0 & 0 & 0 & \dots & 0 \end{bmatrix} \begin{bmatrix} v_1^T \\ \vdots \\ v_n^T \end{bmatrix} \\
&= \begin{bmatrix} u_1 & u_2 & u_3 \end{bmatrix} \begin{bmatrix} \sigma_1 & 0 & 0 \\ 0 & \sigma_2 & 0 \\ 0 & 0 & \sigma_3 \end{bmatrix} \begin{bmatrix} v_1^T \\ v_2^T \\ v_3^T \end{bmatrix} \\
&= U_3\Sigma_3 V_3^T
\end{aligned}
\tag{59}
$$

In this decomposition, Σ_3 is defined as the diagonal matrix formed by the non-zero singular values, while U_3 and V_3^T are obtained by taking the corresponding three columns of U and rows of V^T respectively. Unfortunately, in practice, $\text{rank}(D) > 3$ because of measurement noise and the affine camera approximation. However, recall that when $\text{rank}(D) > 3$, $U_3W_3V_3^T$ is still the best possible rank-3 approximation of MS in the sense of the Frobenius norm.

Upon close inspection, we see that the matrix product $\Sigma_3V_3^T$ forms a $3 \times n$ matrix, which exactly the same size as the structure matrix S. Similarly, U_3 is a $2m \times 3$ matrix, which is the same size as the motion matrix M. While this way of associating the components of the SVD decomposition to M and S leads to a physically and geometrical plausible solution of the affine structure from motion problem, this choice is not a unique solution. For example, we could also set the motion matrix to $M = U_3\Sigma_3$ and the structure matrix to $S = V_3^T$, since in either cases the observation matrix D is the same. So what factorization do we choose? In their paper, Tomasi and Kanade concluded that a robust choice of the factorization is $M = U_3\sqrt{\Sigma_3}$ and $S = \sqrt{\Sigma_3}V_3^T$.

11.3.3 Ambiguity in reconstruction

Nevertheless, we find inherent ambiguity in any choice of the factorization $D = MS$, as any arbitrary, invertible 3×3 matrix A may be inserted into the decomposition:

$$D = MAA^{-1}S = (MA)(A^{-1}S) \tag{60}$$

This means that the camera matrices obtained from motion M and the 3D points obtained from structure S are determined up to a multiplication by a common matrix A. Therefore, our solution is underdetermined, and requires extra constraints to resolve this affine ambiguity. When a reconstruction has affine ambiguity, it means that parallelism is preserved, but the metric scale is unknown.

Another important class of ambiguities for reconstruction is the similarity ambiguity, which occurs when a reconstruction is correct up to a similarity transform (rotation, translation and scaling). A reconstruction with only similarity ambiguity is known as a metric reconstruction. This ambiguity exists even when the camera are intrinsically calibrated. The good news is that for calibrated cameras, the similarity ambiguity is the only ambiguity[5].

The fact that there is no way to recover the absolute scale of a scene from images is fairly intuitive. An object's scale, absolute position and canonical orientation will always be unknown unless we make further assumptions (e.g, we know the height of the house in the figure) or incorporate more data. This is because some attributes may compensate for others. For instance, to get the same image, we can simply move the object backwards and scale it accordingly. One such example of removing similarity ambiguity occurred during the camera calibration procedure, where we made the assumption that we know the location of the calibration points with respect to the world reference system. This enabled us to know the size of the squares of the checkerboard to learn a metric scale of the 3D structure.

11.4 Perspective structure from motion

After studying the simplified affine structure from motion problem, let us now consider the general case for projective cameras M_i. In the general case with projective cameras, each camera matrix M_i contains 11 degrees of freedom, as it is defined up to scale:

$$M_i = \begin{bmatrix} a_{11} & a_{12} & a_{13} & b_1 \\ a_{21} & a_{22} & a_{23} & b_2 \\ a_{31} & a_{32} & a_{33} & 1 \end{bmatrix} \tag{61}$$

Moreover, similar to the affine case where the solution can be found up to an affine transformation, solutions for structure and motion can be determined up a projective transformation in the general case: we can always arbitrarily apply a $4 \times$

[5]See [Longuet-Higgins '81] for more details.

4 projective transformation H to the motion matrix, as long as we also transform the structure matrix by the inverse transformation H^{-1}. The resulting observations in the image plane will still be the same.

Similar to the affine case, we can set up the general structure from motion problem as estimating both the m motion matrices M_i and n 3D points X_j from mn observations x_{ij}. Because cameras and points can only be recovered up to a 4×4 projective transformation up to scale (15 parameters), we have $11m + 3n - 15$ unknowns in $2mn$ equations. From these facts, we can determine the number of views and observations that are required to solve for the unknowns.

11.4.1 The algebraic approach

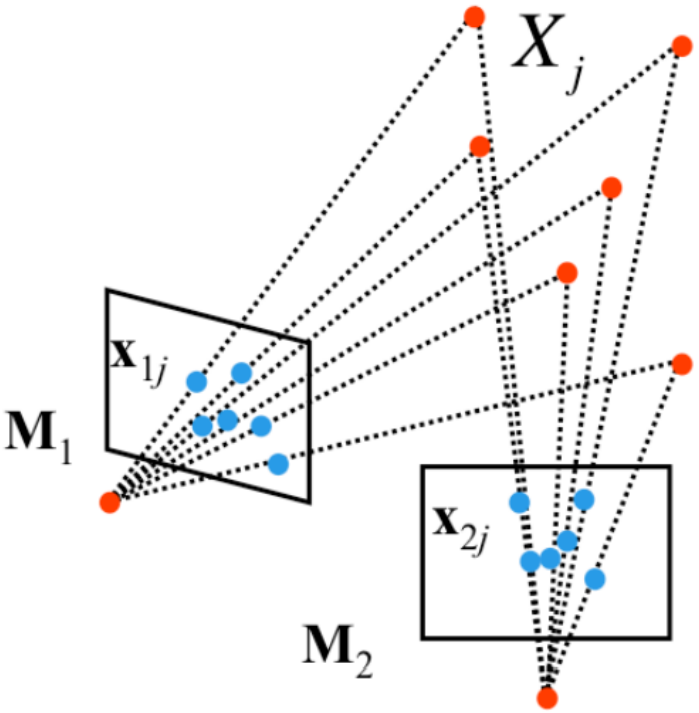


Figure 81: In the algebraic approach, we consider sequential, camera pairs to determine camera matrices M_1 and M_2 up to a perspective transformation. We then find a perspective transformation H such that $M_1 H = [I \;\; 0]$ and $M_2 H = [A \;\; B]$

.

We will now cover the *algebraic approach*, which leverages the concept of fundamental matrix F for solving the structure from motion problem for two cameras. As shown in Figure 81, the main idea of the algebraic approach is to compute two camera matrices M_1 and M_2, which can only be computed up to a perspective transformation H. Since each M_i can only be computed up a perspective transformation H, we can always consider a H such that the first camera projection matrix $M_1 H^{-1}$ is canonical. Of course, the same transformation must also be applied to the second camera which lead to the form shown:

$$M_1 H^{-1} = [I \;\; 0] \qquad M_2 H^{-1} = [A \;\; b] \tag{62}$$

In order to accomplish this task, we must first compute the fundamental matrix F using the eight point algorithm covered in the previous course notes. We now will use F to estimate the projective camera matrices M_1 and M_2. In order to do this estimation, we define P to be the corresponding 3D point for the corresponding observations in the images p and p'. Since we have applied H^{-1} to both camera projection matrices, we must also apply H to the structure, giving us $\widetilde{P} = HP$. Therefore, we can relate the pixel coordinates p and p' to the transformed structure as follows:

$$\begin{aligned} p &= M_1 P = M_1 H^{-1} H P = [I \mid 0]\widetilde{P} \\ p' &= M_2 P = M_2 H^{-1} H P = [A \mid b]\widetilde{P} \end{aligned} \tag{63}$$

An interesting property between the two image correspondences p and p' occur by some creative substitutions:

$$\begin{aligned} p' &= [A|b]\widetilde{P} \\ &= A[I|0]\widetilde{P} + b \\ &= Ap + b \end{aligned} \tag{64}$$

Using Equation 64, we can write the cross product between p' and b as:

$$p' \times b = (Ap + b) \times b = Ap \times b \tag{65}$$

By the definition of cross product, $p' \times b$ is perpendicular to p'. Therefore, we can write:

$$\begin{aligned} 0 &= p'^T (p' \times b) \\ &= p'^T (Ap \times b) \\ &= p'^T \cdot (b \times Ap) \\ &= p'^T [b]_\times Ap \end{aligned} \tag{66}$$

Looking at this constraint, it should remind you of the general definition of the Fundamental matrix $p'^T Fp = 0$. If we set $F = [b]_\times A$, then extracting A and b simply breaks down to a decomposition problem.

Let us begin by determining b. Again, by the definition of cross product, we can simply write Fb as

$$Fb = [b]_\times Ab = (b \times A)b = 0 \tag{67}$$

. Since F is singular, b can be computed as a least square solution of $Fb = 0$, with $\|b\| = 1$, using SVD.

Once b is known, we can now compute A. If we set $A = -[b]_\times F$, then we can verify that this definition satisfies $F = [b]_\times A$:

$$\begin{aligned} [b_\times]A' &= -[b_\times][b_\times]F \\ &= (bb^T - |b|^2 I)F \\ &= bb^T F + |b|^2 F \\ &= 0 + 1 \cdot F \\ &= F \end{aligned} \tag{68}$$

Consequently, we determine the two expressions for our camera matrices $M_1 H^{-1}$ and $M_2 H^{-1}$:

$$\tilde{M}_1 = [I \;\; 0] \qquad \tilde{M}_2 = [-[b_\times]F \;\; b] \tag{69}$$

Before we conclude this section, we want give a geometrical interpretation for b. We know b satisfies $Fb = 0$. Remember the epipolar constraints we derived in the previous course notes, which found that the epipoles in an image are the points that map to zero when transformed by the Fundamental matrix (i.e. $Fe_2 = 0$ and $F^T e_1 = 0$). We can see, therefore, that b is an epipole. This provides a new set of equations for the camera projection matrices (Eqs. 70).

$$\tilde{M}_1 = [I \;\; 0] \qquad \tilde{M}_2 = [-[e_\times]F \;\; e] \tag{70}$$

11.4.2 Determining motion from the Essential matrix

One useful way of improving the reconstruction obtained by the algebraic approach is to use calibrated cameras. We can extract a more accurate, initial estimate of camera matrices by using the Essential matrix, which is a special case of the Fundamental matrix for normalized coordinates. Recall that, by using the Essential matrix E, we make an assumption that we have calibrated the camera and thus know the intrinsic camera matrix K. We can either compute the Essential matrix E either from the normalized image coordinates directly or from its relationship with the Fundamental matrix F and intrinsic matrix K:

$$E = K^T F K \tag{71}$$

Because the Essential matrix assumes that we have calibrated cameras, we should remember that it only has five degrees of freedom, as it only encodes the extrinsic parameters: the rotation R and translation t between the cameras. Luckily, this is exactly the information that we want to extract to create our motion matrix. First, recall that the Essential matrix E can be represented as

$$E = [t]_\times R \tag{72}$$

As such, perhaps we can find a strategy to factor E into its two components. First, we should notice that the cross product matrix $[t]_\times$ is skew-symmetric. We define two matrices that we will use in the decomposition:

$$W = \begin{bmatrix} 0 & -1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}, \quad Z = \begin{bmatrix} 0 & 1 & 0 \\ -1 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \tag{73}$$

One important property we will use later is that $Z = \text{diag}(1, 1, 0)W$ up to a sign. Similarly, we will also use the fact that $ZW = ZW^T = \text{diag}(1, 1, 0)$ up to a sign.

As a result of eigenvalue decomposition, we can create a block decomposition of a general skew-symmetric matrix known up to scale. Thus, we can write $[t]_\times$ as

$$[t]_\times = UZU^T \tag{74}$$

where U is some orthogonal matrix. Therefore, we can rewrite the decomposition as:

$$E = U\text{diag}(1, 1, 0)(WU^T R) \tag{75}$$

Looking at this expression carefully, we see that it closely resembles the singular value decomposition $E = U\Sigma V^T$, where Σ contains two equal singular values. If we know E up to scale and we assume that it takes the form $E = U\text{diag}(1, 1, 0)V^T$, then we arrive at the following factorizations of E:

$$[t]_\times = UZU^T, \quad R = UWV^T \text{ or } UW^TV^T \tag{76}$$

We can prove that the given factorizations are valid by inspection. We can also prove that there are no other factorizations. The form of $[t]_\times$ is determined by the fact that its left null space must be the same as the null space of E. Given unitary matrices U and V, any rotation R can be decomposed into UXV^T where X is another rotation matrix. After substituting these values in, we get $ZX = \text{diag}(1, 1, 0)$ up to scale. Thus, X must be equal to W or W^T.

Note that this factorization of E only guarantees that the matrices UWV^T or UW^TV^T is orthogonal. To ensure that R is a valid rotations, we simply make sure that the determinant of R is positive:

$$R = (\det UWV^T)UWV^T \text{ or } (\det UW^TV^T)UW^TV^T \tag{77}$$

Similar to how the rotation R can take on two potential values, the translation vector t can also take on several values. From the definition of cross product, we know that

$$t \times t = [t]_\times t = UZU^T t = 0 \tag{78}$$

Knowing that U is unitary, we can find that the $\|[t]_\times\|_F = \sqrt{2}$. Therefore, our estimate of t from this factorization will come from the above equation and the fact that E is known up to scale. This means that

$$t = \pm U \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} = \pm u_3 \tag{79}$$

where u_3 is the third column of U. By inspection, we can also verify that we get the same results by reformatting $[t]_\times = UZU^T$ into the vector t known up to a sign.

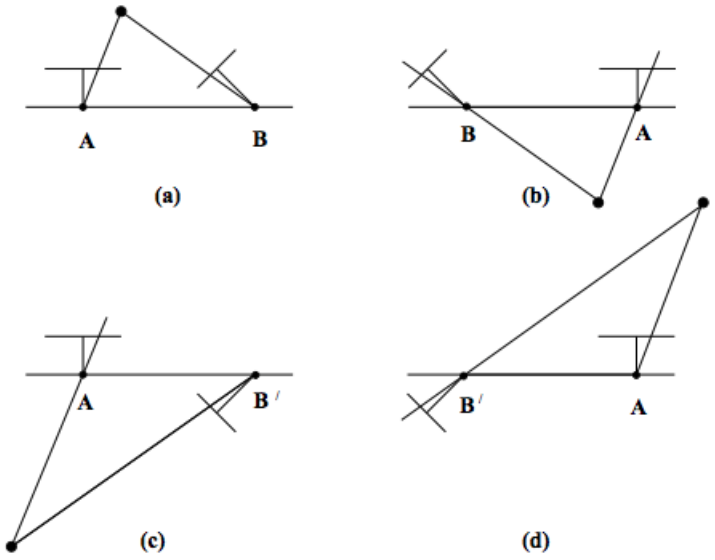


Figure 82: There are four possible solutions for extracting the relative camera rotation R and translation t from the Essential matrix. However, only in (a) is the reconstructed point in front of both of the cameras. (Figure taken from Hartley and Zisserman textbook page 260)

As illustrated in Figure 82, there are four potential R, t pairings since there exists two options for both R and t. Intuitively, the four pairings include all possible pairings of rotating a camera in a certain direction or rotating the camera in the opposite direction combined with the option of translating it in a certain direction or the opposite direction. Therefore, under ideal conditions, we would only need to triangulate one point to determine the correct R, t pair. For the correct R, t pair, the triangulated point $\hat{P}$ exists in front of both cameras, which means that it has a positive z-coordinate with respect to both camera reference systems. Due to measurement noise, we often do not rely on triangulating only one point, but will instead triangulate many points and determine the correct R, t pair as the one that contains the most of these points in front of both cameras.

11.5 An example structure from motion pipeline

After finding the relative motion matrices M_i, we can use them to determine the world coordinates of the points X_j. In the case of the algebraic method, the estimate of such points will be correct up to the perspective transformation H. In extracting the camera matrices from the Essential matrix, the estimates can be known up to scale. In both cases, the 3D points can be computed from the estimated camera matrices via the triangulation methods described earlier.

The extension to the multi-view case can be done by chaining pairwise cameras. We can use the algebraic approach or the Essential matrix to obtain solutions for the camera matrices and the 3D points for any pair of cameras, provided that there are enough point correspondences. The reconstructed 3D points are associated to the point correspondences available between the camera pair. Those pairwise solutions may be combined together (optimized) in a approach called bundle adjustment as we will see next.

11.5.1 Bundle adjustment

There are major limitations related to the previous methods for solving the structure from motion problem that we have discussed so far. The factorization method assumes that all points are visible in every image. This is very unlikely to happen because of occlusions and failures to find correspondences when we either have many images or some of the images were taken far apart. Finally the algebraic approach produces pairwise solutions that can be combined into a camera chain, but does not solve for a coherent optimized reconstruction using all the cameras and 3D points.

To address these limitations, we introduce *bundle adjustment*, which is a nonlinear method for solving the structure from motion problem. In the optimization, we aim to minimize the reprojection error, which is the pixel distance between the projection of a reconstructed point into the estimated cameras and its corresponding observations for all the cameras and for all the points. Previously, when discussing nonlinear optimization methods for triangulation, we focused primarily on the two camera case, in which we naturally assumed that each camera saw all the correspondences between the two. However, since bundle adjustment handles several cameras, it only calculates the reprojection error for only the observations that can be seen by each camera. Ultimately though, this optimization problem is very similar to the one we introduced when talking about nonlinear methods for triangulation.

Two common approaches for solving bundle adjustment's nonlinear optimization include the Gauss-Newton algorithm and the Levenberg-Marquardt algorithm. You can refer to the previous section about details on the Gauss-Newton algorithm and refer to the Hartley and Zisserman textbook for more details on the Levenberg-Marquardt algorithm.

In conclusion, bundle adjustment has some important advantages and limitations when compared to the other methods we have surveyed. It is particularly useful because it can handle a large number of views smoothly and also handle cases when particular points are not observable by every image. However, the main limitation is that it is a particularly large minimization problem, as the parameters grow with the number of views. Additionally, it requires a good initial condition since it relies on nonlinear optimization techniques. For this reason, bundle adjustment is often used as final step of most structure from motion implementations (i.e., after the factorization or algebraic approach), as a factorization or algebraic approach may provide a good initial solution for the optimization problem.

12 Stereo

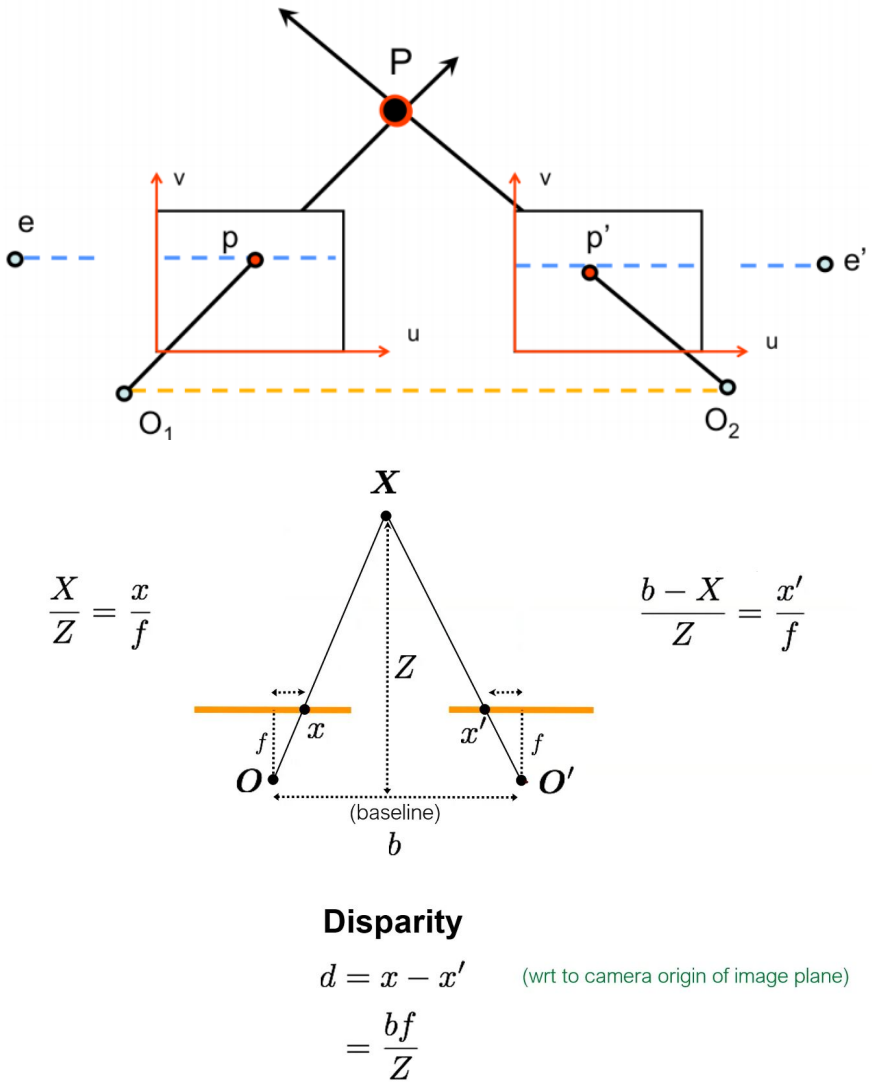
12.1 introduction

Computer stereo vision takes two or more images with known relative camera positions that show an object from different viewpoints. For each pixel in the first image it then determines its corresponding scene point's depth (i.e. distance from the camera) by first finding matching pixels (i.e. pixels showing the same scene point) in the other image(s) and then applying triangulation to the found matches to determine their depth. Finding matches in stereo vision is restricted by epipolar geometry: Each pixel's match in another image can only be found on the epipolar line. If two images are coplanar, i.e. they were taken such that the right camera is only offset horizontally compared to the left camera (not being moved towards the object or rotated), then each pixel's epipolar line is horizontal and at the same vertical position as that pixel. However, in general settings (the camera did move towards the object or rotate) the epipolar lines are slanted. Image rectification warps both images (using projective transformations) such that they appear as if they have been taken with only a horizontal displacement and as a consequence all epipolar lines are horizontal, which slightly simplifies the stereo matching process. Note however, that rectification does not fundamentally change the stereo matching process: It searches on lines, slanted ones before and horizontal ones after rectification. In addition, the triangulation (determining the depth of the scene points) is a lot simpler in stereo vision, and requires only the displacement between the matching points.

12.2 3D Reconstruction Using Stereo Camera System

Recall that in triangulation, we determine the location of a 3D point given its projections into two or more images, and given the cameras' intrinsic parameters. Thus, to reconstruct a 3D object (to its entirety), we need to find point correspondences for all the pixels in the images, and perform triangulation on each one of them (which in the general case requires solving homogeneous linear equations by performing SVD). However, if we constraint our system to be a stereo camera system, where the cameras are offset by pure translation, the 3D reconstruction process is a lot simpler.

Suppose we have two cameras with image planes that differ by pure translation. This cameras system is called the *stereo camera system*:



$$\frac{X}{Z} = \frac{x}{f} \qquad \frac{b - X}{Z} = \frac{x'}{f}$$

$$d = x - x' = \frac{bf}{Z}$$

Figure 83: The stereo camera system setup. O (O_1) and O' (O_2) are the origins of the cameras' coordinate systems, and X (P) is a point in 3D. We can see that its projection on the image planes of the cameras differs by $\frac{bf}{Z}$ (this is called the disparity), and this disparity depends only on Z.

In this case, triangulation is a lot simpler. Assuming we have f (we can find it using camera calibration) and b, if we measure the disparity d between two matching points x and x', we can easily calculate Z, the depth of their corresponding 3D point (no need to solve linear equations for the triangulation!). As we can see, this time we don't need the cameras' projection matrices, neither do we need to solve a linear system of equations to find X. All we need is to find the disparity between all point correspondences, which is a much simpler task. Each corresponding pair of points x and x', in image 1 and image 2 respectively, must lie on their corresponding epipolar lines ℓ_x and $\ell'_{x'}$. Since the cameras differ by pure translation, we know that $\ell_x = \ell'_{x'}$. Determining ℓ_x is easy: it's the line in the direction of the translation of the cameras (no need to calculate F to get ℓ_x!). Thus, for each point x in image 1, we can search along ℓ_x in image 2 and use some feature descriptor to find its best corresponding point x' (we don't need a multi-scale feature descriptor, as the scale of the images is the same). Then, we can calculate the disparity between x and x' (the distance between the points) and easily reconstruct X.

Thus, it's very useful to have a stereo camera setup. Given the disparities between all points in the images' planes (that lie on the same epipolar line), we can easily build a 3D model of the image scene for each point in the images.

As a result, we get the following 3D reconstruction algorithm for the stereo camera system:

1. Rectify images (make their epipolar lines horizontal).
2. For each pixel x in image 1:
   (a) Find its epipolar line.
   (b) Scan image 2 along this line for a point x' that matches x.

(c) Compute the depth of X from the disparity: $Z = \frac{bf}{d}$

12.2.1 Difficulties

Notice that in step 2.c we scan image 2 along the epipolar line for a point x' that matches x. To do that, we can slide a window along the epipolar line in image 2 and compare it using some similarity measure with a "reference" window that surrounds x in image 1. Let I_1 be the reference window surrounding x in image 1, and let I_2 be a window in image 2, surrounding some point along the epipolar line of x. Some of the similarity measures we can use are:

- Sum of Absolute Differences (SAD): $\sum_{(i,j)\in W} |I_1(i,j) - I_2(i,j)|$
- Sum of Squared Differences (SSD): $\sum_{(i,j)\in W} (I_1(i,j) - I_2(i,j))^2$
- Zero-mean SAD: $\sum_{(i,j)\in W} |I_1(i,j) - \bar{I}_1(i,j) - I_2(i,j) + \bar{I}_2(i,j)|$

We also need to choose the window size. Choosing smaller window size means that the matched points are more noisy: matches are less likely to be correct because there are not enough pixels to compare. Choosing larger window size means that there's higher chance that different parts of the window might correspond to different depths in the image, so matching windows might also be incorrect because the matches could be of points in different depth than x (for example if we want to match a depth of a tree, but because the window is large, the match was actually based on some other object in the window).

Another major difficulty that arises when we try to match points is when we have repeated patterns in the images, have textureless regions (such as flat white background), and when we have specularities.

12.2.2 Improving Stereo Matching - Energy Minimization

We have too many discontinuities in the depths of objects, and this make point matching difficult. But, we expect disparity values to change slowly. Thus, lets make an assumption: depth should change smoothly in the image.

What defines a good stereo correspondence?

- Match quality: we want a good match x' in image 2 for each point x in image 1.
- Smoothness: if two pixels are adjacent, they should move about the same amount.

We can define the following energy minimization problem:

$$E(d) = \underbrace{E_d(d)}_{data\ term} + \lambda \underbrace{E_s(d)}_{smoothness\ term}$$

Where d is some disparity between x and x'. The data term means for the pixel x, its matched point x' should be a good match. The smoothness term means that adjacent pixels should move about the same amount (the disparity is smooth). For example:

- $E_d(d) = \sum_{(x,y)\in I} C(x,y,d(x,y))$, where C is the SSD distance between the windows centered at $I_1(x,y)$ and $I_2(x + d(x,y), y)$.
- $E_s(d) = \sum_{(p,q)\in\epsilon} V(d_p, d_q)$. We want the disparity of neighboring pixels to be about the same.

12.3 Image Rectification (Stereo Rectification)

Recall that an interesting case for epipolar geometry occurs when two images are parallel to each other. Let us first compute the Essential matrix E in the case of parallel image planes. We can assume that the two cameras have the same K and that there is no relative rotation between the cameras ($R = I$). In this case, let us assume that there is only a translation along the x axis, giving $T = (T_x, 0, 0)$. This gives

$$E = [T_\times]R = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & -T_x \\ 0 & T_x & 0 \end{bmatrix} \tag{80}$$

Once E is known, we can find the directions of the epipolar lines associated with points in the image planes. Let us compute the direction of the epipolar line ℓ associated with point p':

$$\ell = E^T p' = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & T_x \\ 0 & -T_x & 0 \end{bmatrix} \begin{bmatrix} u' \\ v' \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ T_x \\ -T_x v' \end{bmatrix} \tag{81}$$

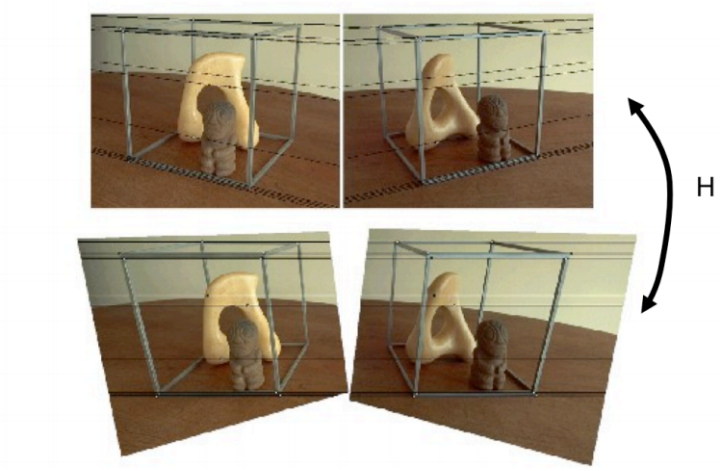


Figure 84: The process of image rectification involves computing two homographies that we can apply to a pair of images to make them parallel.

We can see that the direction of ℓ is horizontal, as is the direction of ℓ', which is computed in a similar manner.

If we use the epipolar constraint $p'^T Ep = 0$, then we arrive at the fact that $v = v'$, demonstrating that p and p' share the same v-coordinate. Consequently, there exists a very straightforward relationship between the corresponding points. Therefore, *rectification*, or the process of making any two given images parallel, becomes useful when discerning the relationships between corresponding points in images.

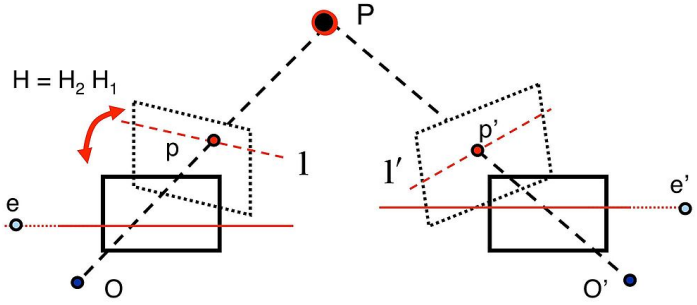


Figure 85: The rectification problem setup: we compute two homographies that we can apply to the image planes to make the resulting planes parallel.

Rectifying a pair of images does not require knowledge of the two camera matrices K, K' or the relative transformation R, T between them. Instead, we can use the Fundamental matrix estimated by the Normalized Eight Point Algorithm. Upon getting the Fundamental matrix, we can compute the epipolar lines ℓ_i and ℓ'_i for each correspondence p_i and p'_i.

From the set of epipolar lines, we can then estimate the epipoles e and e' of each image. This is because we know that the epipole lies in the intersection of all the epipolar lines. In the real world, due to noisy measurements, all the epipolar lines will not intersect in a single point. Therefore, computing the epipole can be found by minimizing the least squared error of fitting a point to all the epipolar lines. Recall that each epipolar line can be represented as a vector ℓ such that all points on the line (represented in homogeneous coordinates) are in the set $\{x | \ell^T x = 0\}$. If we define each epipolar line as $\ell_i = \begin{bmatrix} \ell_{i,1} & \ell_{i,2} & \ell_{i,3} \end{bmatrix}^T$, then we can we formulate a linear system of equations and solve using SVD to find the epipole e:

$$\begin{bmatrix} \ell_1^T \\ \vdots \\ \ell_n^T \end{bmatrix} e = 0 \tag{82}$$

After finding the epipoles e and e', we will most likely notice that they are not points at infinity along the horizontal axis. If they were, then, by definition, the images would already be parallel. Thus, we gain some insight into how to make the images parallel: can we find a homography to map an epipole e to infinity along the horizontal axis? Specifically, this means that we want to find a pair of homographies H_1, H_2 that we can apply to the images to map the epipoles to infinity. Let us start by finding a homography H_2 that maps the second epipole e' to a point on the horizontal axis at infinity $(f, 0, 0)$. Since there are many possible choices for this homography, we should try to choose something reasonable. One condition

that leads to good results in practice is to insist that the homography acts like a transformation that applies a translation and rotation on points near the center of the image.

The first step in achieving such a transformation is to translate the second image such that the center is at $(0, 0, 1)$ in homogeneous coordinates. We can do so by applying the translation matrix

$$T = \begin{bmatrix} 1 & 0 & -\frac{\text{width}}{2} \\ 0 & 1 & -\frac{\text{height}}{2} \\ 0 & 0 & 1 \end{bmatrix} \tag{83}$$

After applying the translation, we apply a rotation to place the epipole on the horizontal axis at some point $(f, 0, 1)$. If the translated epipole Te' is located at homogeneous coordinates $(e'_1, e'_2, 1)$, then the rotation applied is

$$R = \begin{bmatrix} \alpha \frac{e'_1}{\sqrt{e'^2_1 + e'^2_2}} & \alpha \frac{e'_2}{\sqrt{e'^2_1 + e'^2_2}} & 0 \\ -\alpha \frac{e'_2}{\sqrt{e'^2_1 + e'^2_2}} & \alpha \frac{e'_1}{\sqrt{e'^2_1 + e'^2_2}} & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{84}$$

where $\alpha = 1$ if $e'_1 \geq 0$ and $\alpha = -1$ otherwise. After applying this rotation, notice that given any point at $(f, 0, 1)$, bringing it to a point at infinity on the horizontal axis $(f, 0, 0)$ only requires applying a transformation

$$G = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -\frac{1}{f} & 0 & 1 \end{bmatrix} \tag{85}$$

After applying this transformation, we finally have an epipole at infinity, so we can translate back to the regular image space. Thus, the homography H_2 that we apply on the second image to rectify it is

$$H_2 = T^{-1}GRT \tag{86}$$

Now that a valid H_2 is found, we need to find a matching homography H_1 for the first image. We do so by finding a transformation H_1 that minimizes the sum of square distances between the corresponding points of the images

$$\arg\min_{H_1} \sum_i \|H_1 p_i - H_2 p'_i\|^2 \tag{87}$$

Although the derivation[6] is outside the scope of this class, we can actually prove that the matching H_1 is of the form:

$$H_1 = H_A H_2 M \tag{88}$$

where $F = [e]_\times M$ and

$$H_A = \begin{bmatrix} a_1 & a_2 & a_3 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{89}$$

with (a_1, a_2, a_3) composing the elements of a certain vector $\mathbf{a}$ that will be computed later.

First, we need to know what M is. An interesting property of any 3×3 skew-symmetric matrix A is $A = A^3$ up to scale. Because any cross product matrix $[e]_\times$ is skew-symmetric and that we can only know the Fundamental matrix F up to scale, then

$$F = [e]_\times M = [e]_\times [e]_\times [e]_\times M = [e]_\times [e]_\times F \tag{90}$$

By grouping the right terms, we can find that

$$M = [e]_\times F \tag{91}$$

Notice that if the columns of M were added by any scalar multiple of e, then the $F = [e]_\times M$ still holds up to scale. Therefore, the more general case of defining M is

$$M = [e]_\times F + ev^T \tag{92}$$

for some vector v. In practice, defining M by setting $v^T = \begin{bmatrix} 1 & 1 & 1 \end{bmatrix}$ works very well.

To finally solve for H_1, we need to compute the $\mathbf{a}$ values of H_A. Recall that we want to find a H_1, H_2 to minimize the problem posed in Equation 87. Since we already know the value of H_2 and M, then we can substitute $\hat{p}_i = H_2 M p_i$ and $\hat{p}'_i = H_2 p'_i$ and the minimization problem becomes

$$\arg\min_{H_A} \sum_i \|H_A \hat{p}_i - \hat{p}'_i\|^2 \tag{93}$$

[6] If you are interested in the details, please see Chapter 11 of Hartley & Zisserman's textbook *Multiple View Geometry*

In particular, if we let $\hat{p}_i = (\hat{x}_i, \hat{y}_i, 1)$ and $\hat{p}'_i = (\hat{x}'_i, \hat{y}'_i, 1)$, then the minimization problem can be replaced by:

$$\arg\min_{\mathbf{a}} \sum_i (a_1\hat{x}_i + a_2\hat{y}_i + a_3 - \hat{x}'_i)^2 + (\hat{y}_i - \hat{y}'_i)^2 \tag{94}$$

Since $\hat{y}_i - \hat{y}'_i$ is a constant value, the minimization problem further reduces to

$$\arg\min_{\mathbf{a}} \sum_i (a_1\hat{x}_i + a_2\hat{y}_i + a_3 - \hat{x}'_i)^2 \tag{95}$$

Ultimately, this breaks down into solving a least-squares problem $W\mathbf{a} = b$ for $\mathbf{a}$ where

$$W = \begin{bmatrix} \hat{x}_1 & \hat{y}_1 & 1 \\ & \vdots & \\ \hat{x}_n & \hat{y}_n & 1 \end{bmatrix} \qquad b = \begin{bmatrix} \hat{x}'_1 \\ \vdots \\ \hat{x}'_n \end{bmatrix} \tag{96}$$

After computing $\mathbf{a}$, we can compute H_A and finally H_1. Thus, we generated the homographies H_1, H_2 to rectify any image pair given a few correspondences.

13 Active and Volumetric Stereo

In traditional stereo, the main idea is to use corresponding points p and p' to estimate the location of a 3D point P by triangulation. A key challenge here, is to solve the correspondence problem: how do we know whether a point p actually corresponds to a point p' in another image? This problem is further accentuated by the fact that we need to handle the many 3D points that are present in the scene. The focus of these notes will discuss alternative techniques that work well in reconstructing the 3D structure.

13.1 Active stereo

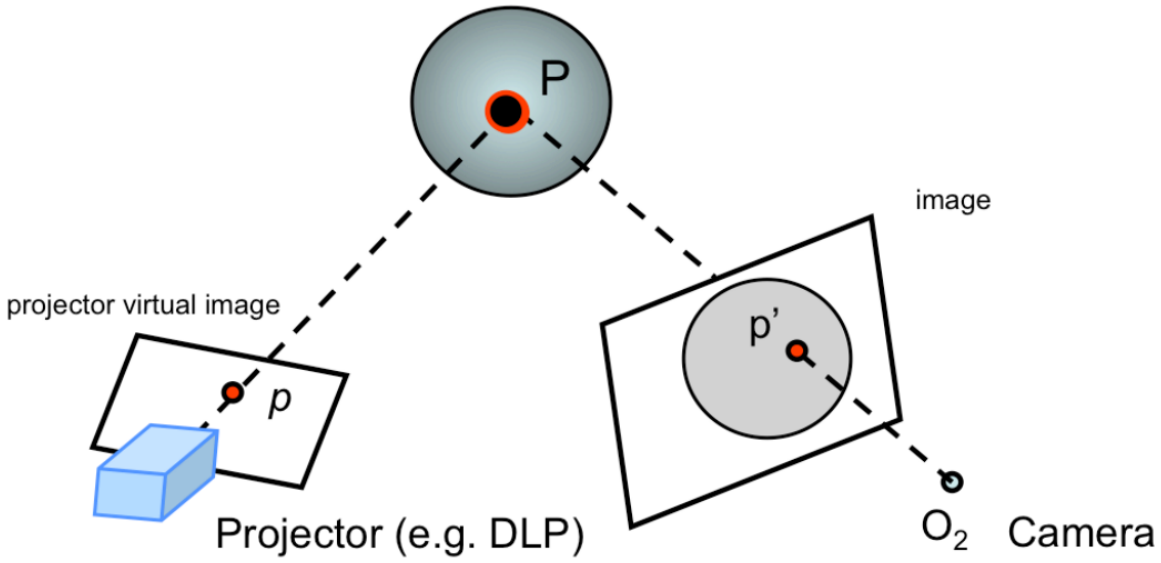


Figure 86: The active stereo setup that projects a point into 3D space.

First, we will an introduce a technique known as *active stereo* that helps mitigate the correspondence problem in traditional stereo. The main idea of active stereo is to replace one of the two cameras with a device that interacts with the 3D environment, usually by projecting a pattern onto the object that is easily identifiable from the second camera. This new projector-camera pair defines the same epipolar geometry that we introduced for camera pairs, whereby the image plane of the replaced camera is replaced with a *projector virtual plane*. In Figure 86, the projector is used to project a point p in the virtual plane onto the object in 3D space, producing a point in 3D space P. This 3D point P should be observed in the second camera as a point p'. Because we know what we are projecting (e.g. the position of p in the virtual plane, the color and intensity of the projection, etc.), we can easily discover the corresponding observation in the second camera p'.

A common strategy in active stereo is to project from the virtual plane a vertical stripe s instead of a single point. This case is very similar to the point case, where the line s is projected to a stripe in 3D space S and observed as a line in the

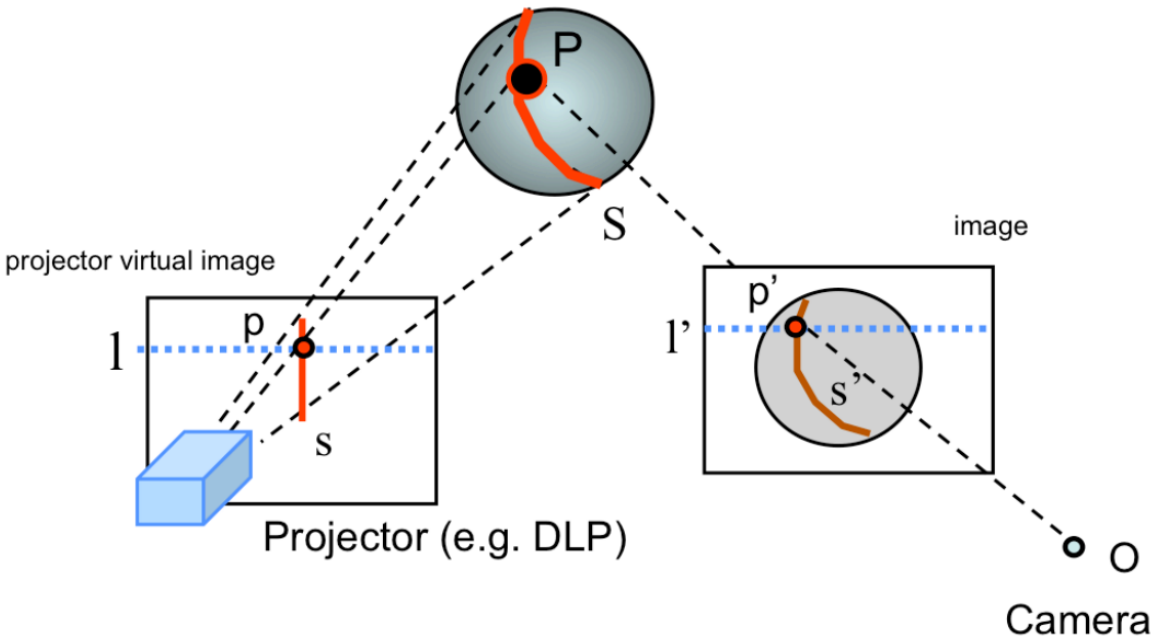


Figure 87: The active stereo setup that projects a line into 3D space.

camera as s'. If the projector and camera are parallel or rectified, then we can discover the corresponding points easily by simply intersecting s' with the horizontal epipolar lines. From the correspondences, we can use the triangulation methods introduced in the previous course notes to reconstruct all the 3D points on the stripe S. By swiping the line across the scene and repeating the process, we can recover the entire shape of all visible objects in the scene.

Notice that one requirement for this algorithm to work is that the projector and the camera need to be calibrated. An active stereo system can be calibrated using similar techniques as described in previous notes. We can first calibrate the camera using a calibration rig. Then, by projecting known stripes onto the calibration rig, and using the corresponding observations in the newly calibrated camera, we can set up constraints for estimating the projector intrinsic and extrinsic parameters. Once calibrated, this active stereo setup can produce very accurate results. In 2000, Marc Levoy and his students at Stanford demonstrated that by using a finely tuned laser scanner, they could recover the shape of Michaelangelo's Pieta with sub-millimeter accuracy.

However, in some cases, having a finely tuned projector may be too expensive or cumbersome. An alternative approach that uses a much cheaper setup leverages shadows to produce active patterns to the object we want to recover. By placing a stick between the object and a light source at a known position, we can effectively project a stripe onto the object as before. Moving the stick allows us to project different shadow stripes onto the object and recover the object in a similar manner as before. This method, although much cheaper, tends to produce less accurate results because it requires very good calibration between the stick, camera, and light source, while needing to maintain a tradeoff between the length and thinness of the stick's shadow.

One limitation of projecting a single stripe onto objects is that it is rather slow, as the projector needs to swipe across the entire object. Furthermore, this means that this method cannot capture deformations in real time. A natural extension is to instead attempt to reconstruct the object from projecting a single frame or image. The idea is to project a known pattern of different stripes to the entire visible of the object, instead of a single stripe. The colors of these stripes are designed in such a way that the stripes can be uniquely identified from the image. Figure 88 illustrates this multiple color-coded stripes method. This concept powered many versions of modern depth sensors, such as the original version of the Microsoft Kinect. In practice, these sensors use infrared laser projectors , which allow it to capture video data in 3D under any ambient light conditions.

13.2 Volumetric stereo (not in the syllabus)

An alternative to both the traditional stereo and active stereo approach is *volumetric stereo*, which inverts the problem of using correspondences to find 3D structure. In volumetric stereo, we assume that the 3D point we are trying to estimate is within some contained, known volume. We then project the hypothesized 3D point back into the calibrated cameras and validate whether these projections are consistent across the multiple views. Figure 89 illustrates the general setup of the volumetric stereo problem. Because these techniques assume that the points we want to reconstruct are contained by a limited volume, these techniques are mostly used for recovering the 3D models of specific objects as opposed to recovering models of a scene, which may be unbounded.

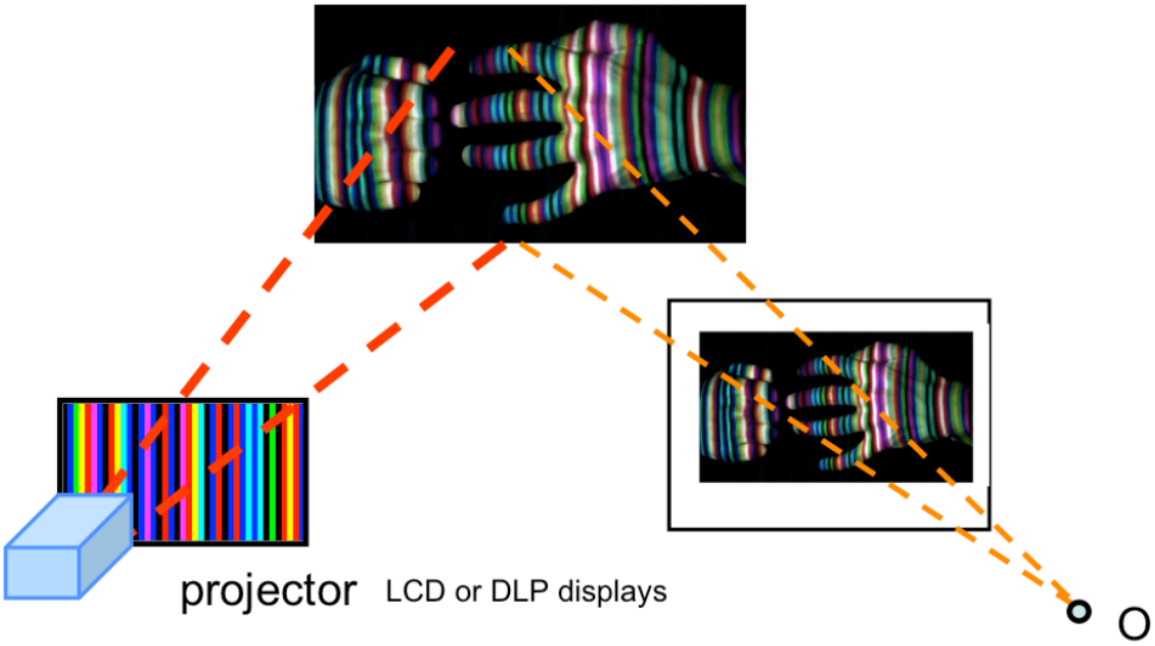


Figure 88: The active stereo setup that uses multiple colored lines to reconstruct an object from a single projection.

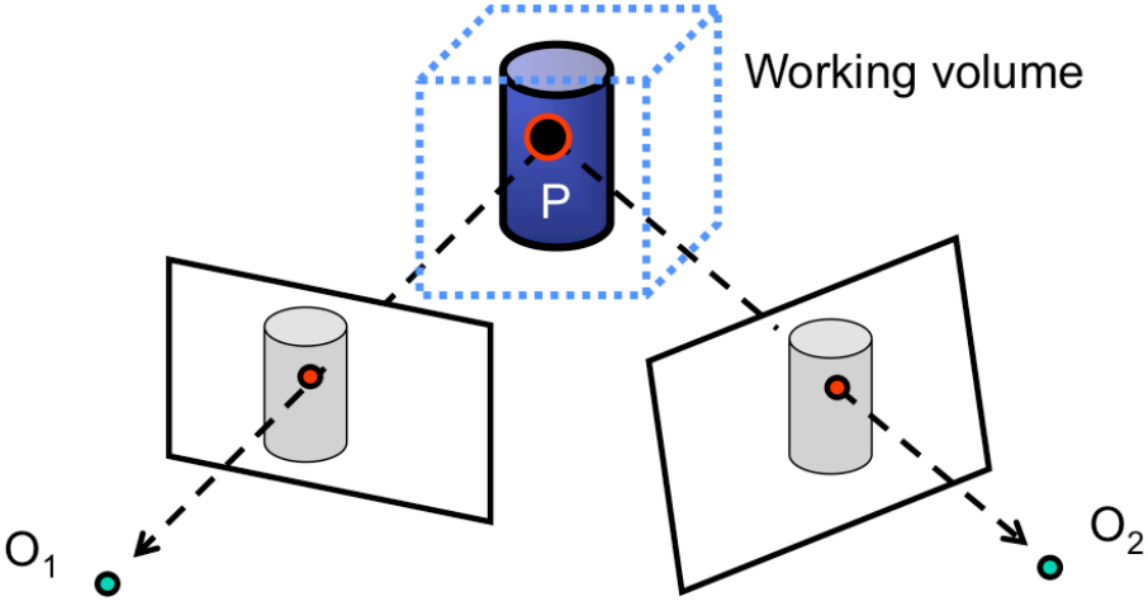


Figure 89: The setup of volumetric stereo, which takes points from a limited, working volume and performs consistency checks to determine 3D shape.

The main tenet of any volumetric stereo method is to first define what it means to be "consistent" when we reproject a 3D point in the contained volume back into the multiple image views. Thus, depending on the definition of the concept of consistent observations, different techniques can be introduced. In these notes, we will briefly outline three major techniques, which are known as space carving, shadow carving, and voxel coloring.

13.2.1 Space carving

The idea of space carving is mainly derived from the observation that the contours of an object provide a rich source of geometric information about the object. In the context of multiple views, let us first set up the problem illustrated in Figure 90. Each camera observes some visible portion of an object, from which a contour can be determined. When projected into the image plane, this contour encloses a set of pixels known as the *silhouette* of the object in the image plane. Space carving ultimately uses the silhouettes of objects from multiple views to enforce consistency.

However, if we do not have the information of the 3D object and only images, then how can we obtain silhouette information? Luckily, one practical advantage of working with silhouettes is that they can be easily detected in images if we

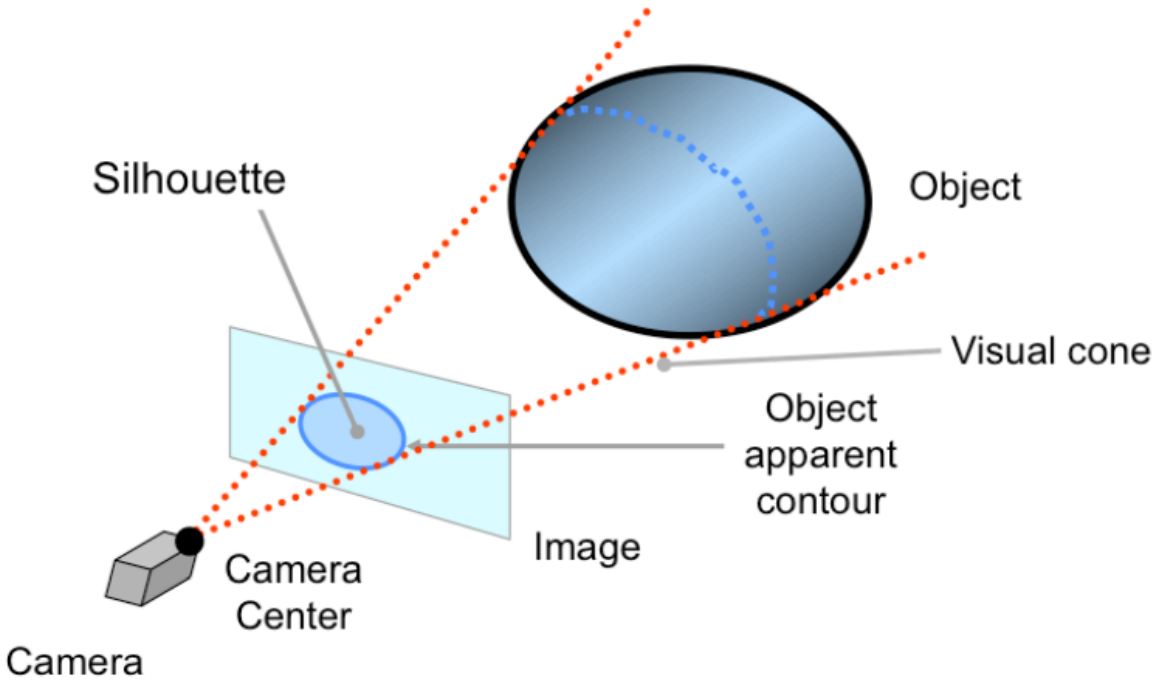


Figure 90: The silhouette of an object we want to reconstruct contains all pixels of the visible portion of the object in the image. The visual cone is the set of all possible points that can project into the silhouette of the object in the image.

have control of the background behind the object that we want to reconstruct. For example, we can use a "green screen" behind the object to easily segment the object from its background.

Now that we have the silhouettes, how can we actually use them? Recall that in volumetric stereo, we have an estimate of some volume that we guarantee that the object can reside within. We now introduce the concept of a *visual cone*, which is the enveloping surface defined by the camera center and the object contour in the image plane. By construction, it is guaranteed that the object will lie completely in both the initial volume and the visual cone.

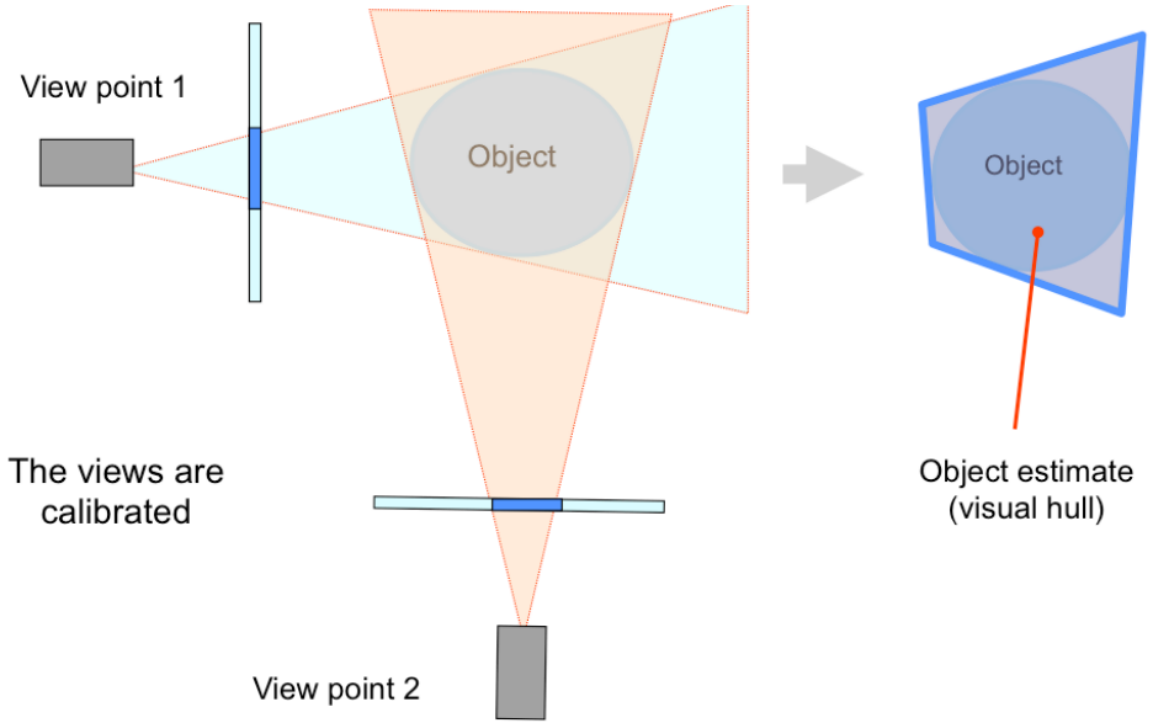


Figure 91: The process of estimating the object from multiple views involves recovering the visual hull, which is the intersection of visual cones from each camera.

Therefore, if we have multiple views, then we can compute visual cones for each view. Since, by definition, the object

resides in each of these visual cones, then it must lie in the intersection of these visual cones, as illustrated in Figure 91. Such an intersection is often called a *visual hull*.

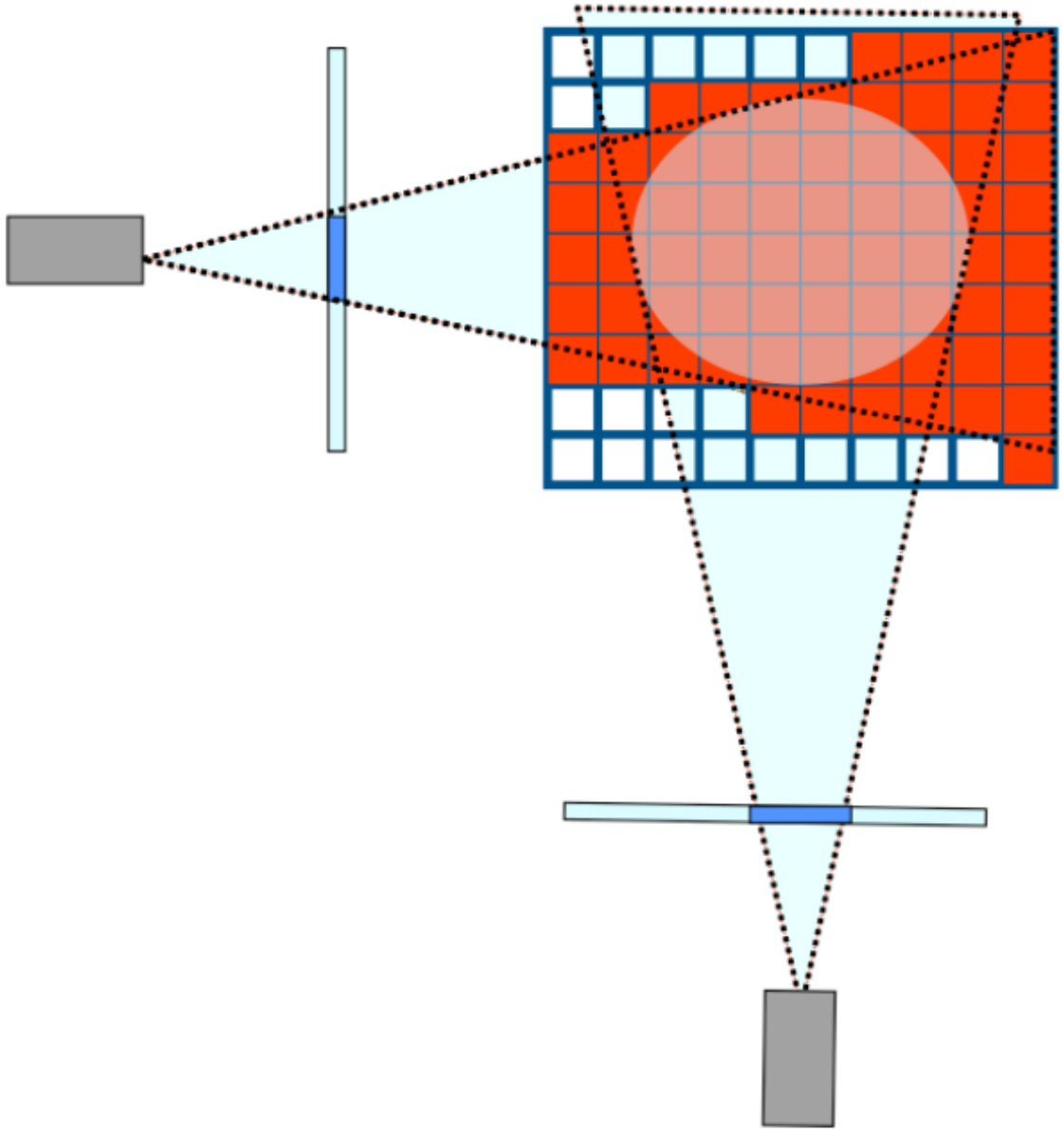


Figure 92: The result of space carving when done on a voxel grid. The region is the reconstructed object after carving from two views, while the shaded part on the inside is the actual object. Notice that the reconstruction is always conservative.

In practice, we first begin by defining a working volume that we know the object is contained within. For example, if our cameras encircle the object, then we can simply say that the working volume is the entire interior of the space enclosed by the cameras. We divide this volume into small units known as *voxels*, defining what is known as a voxel grid. We take each voxel in the voxel grid and project it into each of the views. If the voxel is not contained by the silhouette in a view, then it is discarded. Consequently, at the end of the space carving algorithm, we are left with the voxels that are contained within the visual hull.

Although the space carving method avoids the correspondence problem and is relatively straightforward, it still has many limitations. One limitation of space carving is that it scales linearly with the number of voxels in the grid. As we reduce the size of each voxel, the number of voxels required by the grid increases cubically. Therefore, to get finer reconstruction results in much larger run time. However, some methods such as using octrees can be used mitigate this problem. Related, but simpler methods include doing iterative carvings to reduce the size of the initial voxel grid.

Another limitation is that the efficacy of space carving is dependent on the number of views, the preciseness of the

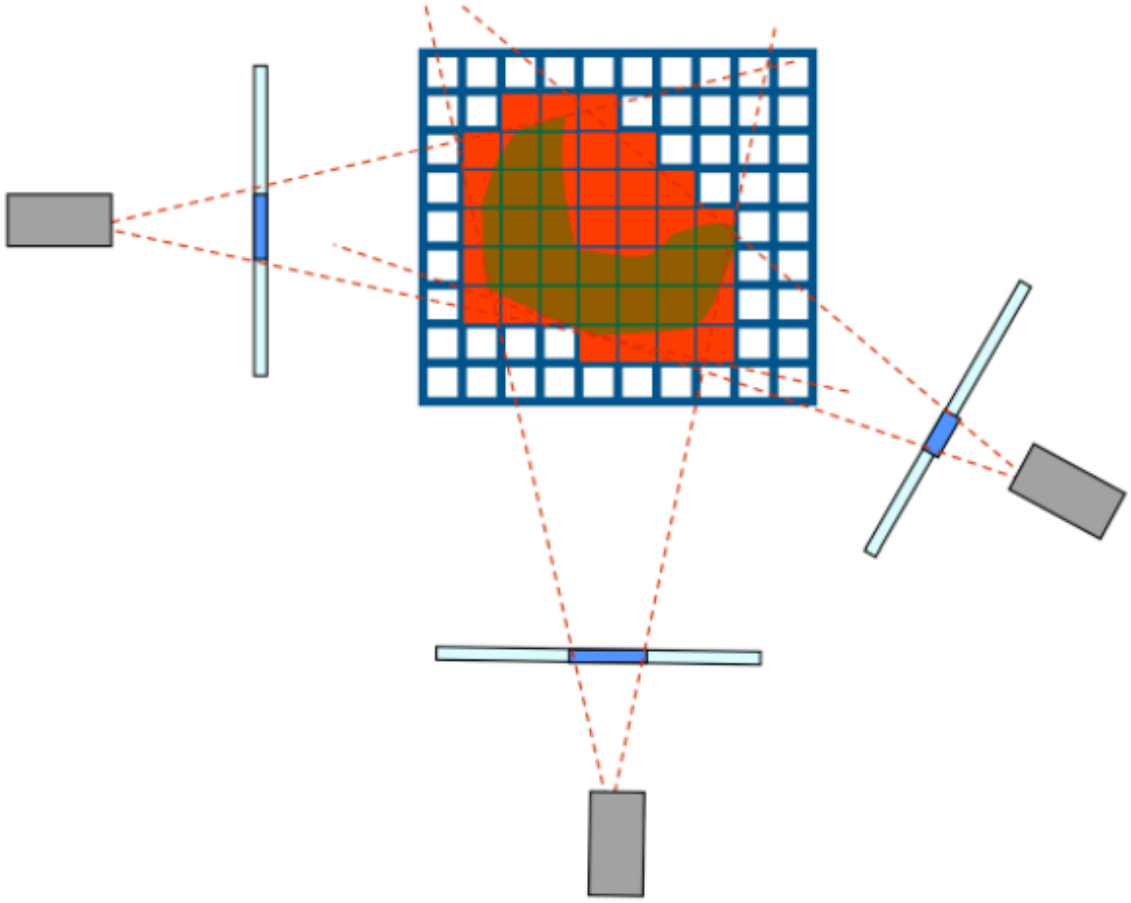


Figure 93: Space carving cannot handle some concavities, as demonstrated here, because it cannot carve into that region, as doing so will carve through the object. Note this means that generally the only concavities that space carving can handle are holes through an object.

silhouette, and even the shape of the object we are trying to reconstruct. If the number of views is too low, then we end of up with a very loose estimate of the visual hull of the object. As the number of views increases, the more extraneous voxels can be removed by the consistency check. Furthermore, the validity of the consistency check is solely upheld by the fact that we believe that the silhouettes are correct. If the silhouette is too conservative and contains more pixels than necessary, then our carving may not be precise. In a potentially even worse case, the silhouette misses portions of the actual object, resulting in a reconstruction that is overly carved. Finally, a major drawback of space carving is that it is incapable of modeling certain concavities of an object, as shown in Figure 93.

13.2.2 Shadow carving

To circumvent the concavity problem posed by space carving, we need to look to other forms of consistency checks. One important cue for determining the 3D shape of an object that we can use is the presence of *self-shadows*. Self-shadows are the shadows that an object projects on itself. For the case of concave objects, an object will often cast self-shadows in the concave region.

Shadow carving at its core augments space carving with the idea of using self-shadows to better estimate the concavities. As shown in Figure 94, the general setup of shadow carving is very similar to space carving. An object is placed in a turntable that is viewed by a calibrated camera. However, there is an array of lights in known positions around the camera whose states can be appropriately turned on and off. These lights will be used to make the object cast self-shadows.

As shown in Figure 95, the shadow carving process begins with an initial voxel grid, which is trimmed down by using the same approach as in space carving. However, in each view, we can turn on and off each light in the array surrounding the camera. Each light will produce a different self-shadow on the object. Upon identifying the shadow in the image plane, we can then find the voxels on the surface of our trimmed voxel grid that are in the visual cone of the shadow. These surface voxels allow us to then make a new visual cone with the image source. We then leverage the useful fact that a voxel that is part of both visual cones cannot be part of the object to eliminate voxels in the concavity.

Like space carving, the runtime of shadow carving is dependent on the resolution of the voxel grid. The runtime scales

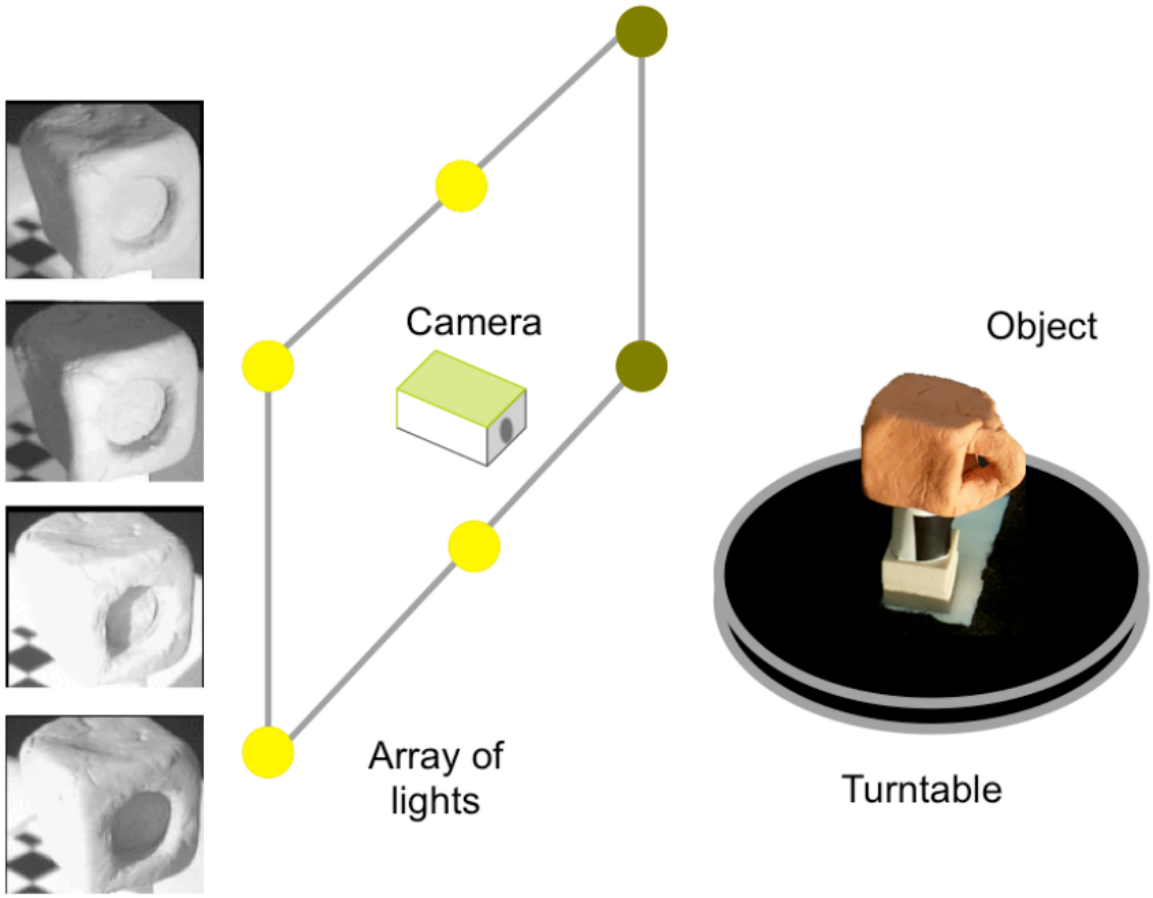


Figure 94: The setup of shadow carving, which augments space carving by adding a new consistency check from an array of lights surrounding the camera.

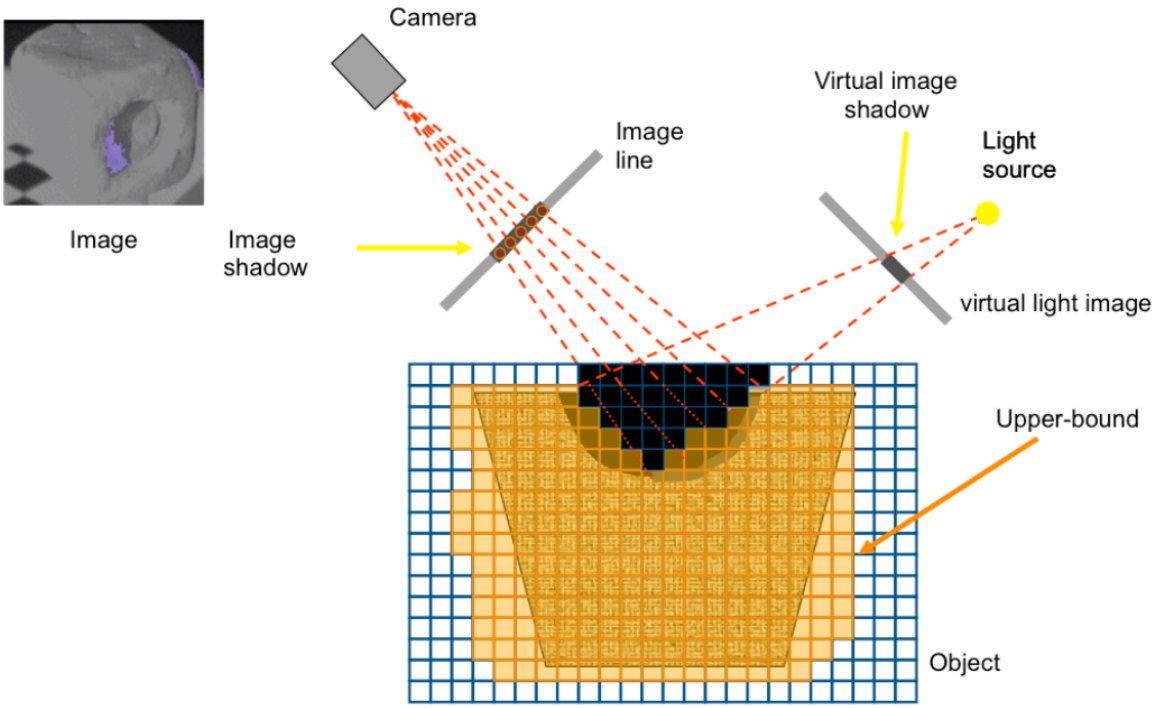


Figure 95: Shadow carving relies on a new consistency check that removes voxels that are in the self-shadow visual cone of the camera and the visual cone of the light.

cubically with the resolution of the voxel grid. However, if there are N lights, then shadow carving takes approximately $N + 1$ times longer than space carving, as each voxel needs to be projected into the camera and each of the N lights can be turned on and off.

In summary, shadow carving always produces a conservative volume estimate that better reconstructs 3D shapes with concavities. The quality of the results depends on both the number of views and the number of light sources. Some disadvantages of this approach are that it cannot handle cases where the object contains reflective or low albedo regions. This is because shadows cannot be detected accurately in such conditions.

13.2.3 Voxel coloring

The last technique we cover in volumetric stereo is *voxel coloring*, which uses color consistency instead of contour consistency in space carving.

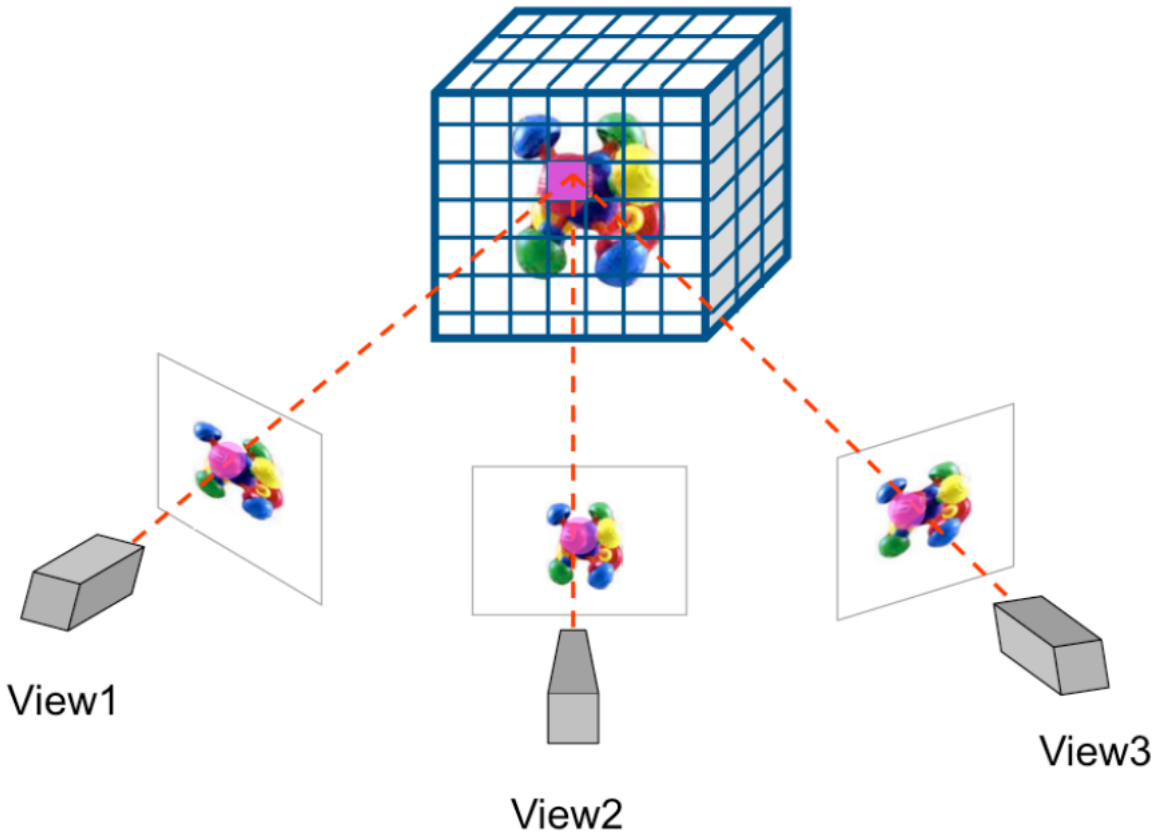


Figure 96: The setup of voxel coloring, which makes a consistency check of the color of all projections of a voxel.

As illustrated in Figure 96, suppose that we are given images from multiple views of an object that we want to reconstruct. For each voxel, we look at its corresponding projections in each of the images and compare the color of each of these projections. If the colors of these projections sufficiently match, then we mark the voxel as part of the object. One benefit of voxel coloring not present in space carving is that color associated with the projections can be transferred to the voxel, giving a colored reconstruction.

Overall, there are many methods that one could use for the color consistency check. One example would be to set a threshold between the color similarity between the projections. However, there exists a critical assumption for any color consistency check used: the object being reconstructed must be *Lambertian*, which means that the perceived luminance of any part of the object does not change with viewpoint location or pose. For non-Lambertian objects, such as those made of highly reflective material, it is easy to conceive that the color consistency check would fail on voxels that are actually part of the object.

One drawback of vanilla voxel coloring is that it produces a solution that is not necessarily unique, as shown in Figure 97. Finding the true, unique solution complicates the problem of reconstruction by voxel coloring. It is possible to remove the ambiguity in the reconstruction by introducing a visibility constraint on the voxel, which requires that the voxels be traversed in a particular order.

In particular, we want to traverse the voxels layer by layer, starting with voxels closer to the cameras and then progress to further away voxels. When using this order, we perform the color consistency check. Then, we check if the voxel is viewable by at least two of the cameras, which constructs our visibility constraint. If the voxel was not viewable by at least two

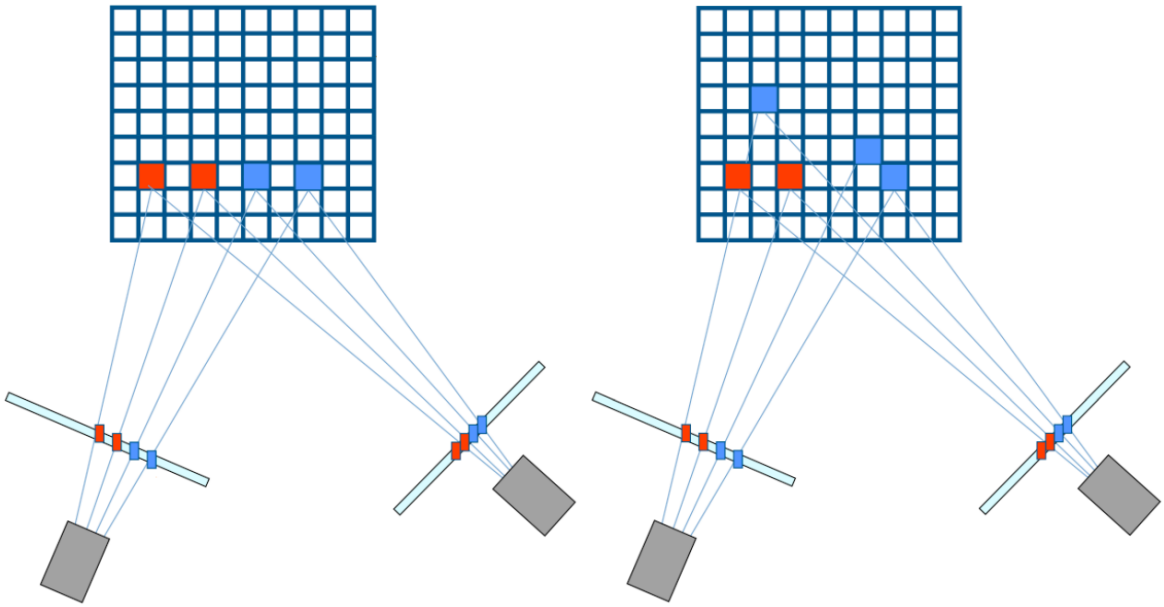


Figure 97: An example of an ambiguous case of vanilla voxel coloring.

cameras, then it must be occluded and thus not part of the object. Notice that our order of processing the closer voxels allows us to make sure that we keep the voxels that can occlude later processed voxels to enforce this visibility constraint.

To conclude, voxel coloring has the advantage of simultaneously capturing the shape and texture of an object. Some of the drawbacks include that the object is assumed to be Lambertian and that the cameras cannot be in certain locations, as the voxels need to be processed in a certain order due to the visibility constraint.

14 Radiometry And Reflectence

14.1 Introduction

Radiometry is a set of techniques for measuring electromagnetic radiation, including visible light. Radiometric techniques in optics characterize the distribution of the radiation's power in space, as opposed to photometric techniques, which characterize the light's interaction with the human eye. Radiometry is distinct from quantum techniques such as photon counting.

We use radiometry techniques as an attempt to understand the details in images. The radiometry results depend on the shape of the objects we examine, the characteristics of the materials they are made of, and the illumination of the scene.

In the previous chapters we talked about different algorithms that are based on given images in 2D, but we always neglected difference kinds of deflections such as shade, reflections, and refractions:

Figure 98: From left to right: shading, reflection, and refraction. All of these can confuse our classification algorithms.

14.2 Scattering

When a beam of light hits the surface of an object, the light scatters to all directions, including into the material. This phenomena entails various effects, like the following:

Figure 99: An example of one effect caused by scattering. In the left simulated image we have only one source of light and no scattering, and we can see a lot more details. In the right simulated image, scattering causes similar effect as if we had multiple sources of light, which blurs the image.

14.3 Material Reflectance

The reflectance of the surface of a material is its effectiveness in reflecting radiant energy. It's the fraction of incident electromagnetic power that is reflected at its boundary. Reflectance is a component of the response of the electronic structure of the material to the electromagnetic field of light, and is in general a function of the frequency, or wavelength, of the light, its polarization, and the angle of incidence. The dependence of reflectance on the wavelength is called a *reflectance spectrum* or spectral reflectance curve.

Each material has its spectrum of reflectence, as we can see in the following figure:

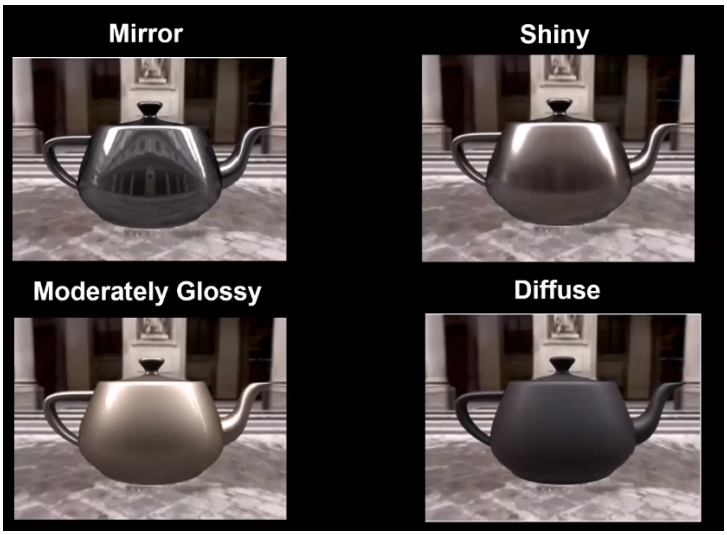


Figure 100: Four examples of reflectence from materials.

14.4 Solid Angle

In geometry, a solid angle (symbol: Ω) is a measure of the amount of the field of view from some particular point that a given object covers. That is, it is a measure of how large the object appears to an observer looking from that point. The point from which the object is viewed is called the apex of the solid angle.

An object's solid angle measurement units is called steradians, and is equal to the area of the segment of a unit sphere, centered at the apex, that the object covers, divided by the distance to the object squared. Therefore, just like a planar angle in radians is the ratio of the length of a circular arc to its radius, a solid angle in steradians is the following ratio:

$$\Omega = \frac{A}{r^2}$$

where A is the spherical surface area and r is the radius of the considered sphere.

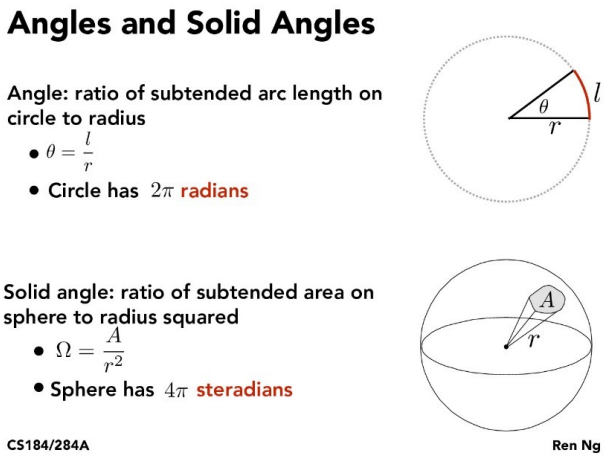


Figure 101: The definition of solid angle, and its relation to the definition of radians in 2D.

In general, the solid angle of an object depends on two things:

- The orientation of the object, which subsequently affects the surface area perceived from the apex (the point of view).
- The distance to the surface of the object.

Notice that the surface of the object might not be spherical. But, when we observe an object we approximate that the surface we see is indeed aligned with the sphere.

We can also talk about the differential solid angle of an object, which depends on an infitesimal change in the surface area:

$$d\Omega = \frac{dA}{r^2}[sr]$$

We can further generalize it to objects in any orientation, and add the orientation parameter to the definition, relative to the normal of the sphere:

$$d\Omega = \frac{dAcos\theta}{r^2}[sr]$$

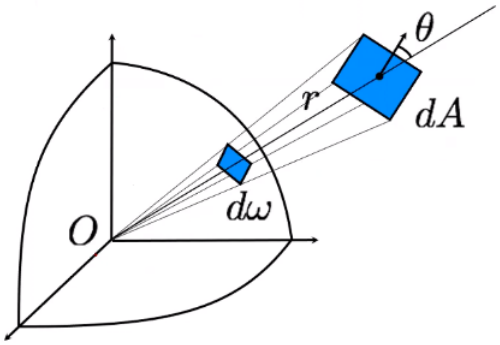


Figure 102: Illustration of the differential solid angle.

This differential definition will help us avoid the assumption that the surface of the object aligns with the sphere.

14.5 Radiant Power

Radiant flux or radiant power is the radiant energy emitted, reflected, transmitted or received, per unit time. Its formal definition is:

$$\Phi_e = \frac{\partial Q_e}{\partial t}$$

where e stands for "energy", Q_e is the radiant energy emmited, reflected, transmitted, or received, and t is time.

14.6 Irradiance

Irradiance is the radiant flux (power) received by a surface per unit area. Irradiance is often called intensity, but this term is avoided in radiometry where such usage leads to confusion with radiant intensity. Its formal definition is:

$$E_e = \frac{\partial \Phi_e}{\partial A}$$

where Φ_e is the radiant flux received, and A is the surface area.

14.7 Radiance

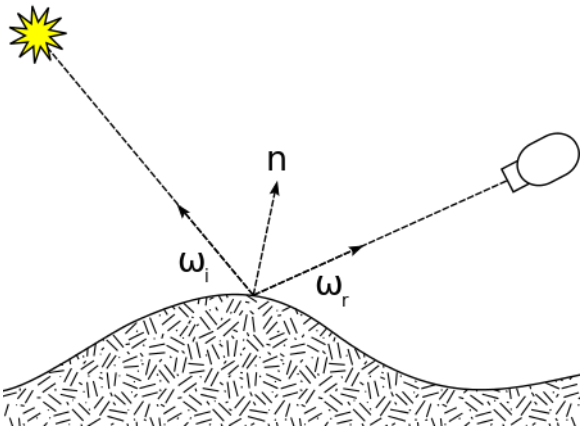
Radiance is the radiant flux emitted, reflected, transmitted or received by a given surface, per unit solid angle per unit projected area. Its formal definition is:

$$L_{e,\Omega} = \frac{\partial^2 \Phi_e}{\partial \Omega \partial Acos\theta}$$

where Φ_e is the radiant flux emmited, reflected, transmitted or received, Ω is the solid angle, and $Acos\theta$ is the projected area.

14.8 Bidirectional Reflectance Distribution Functions (BRDF)

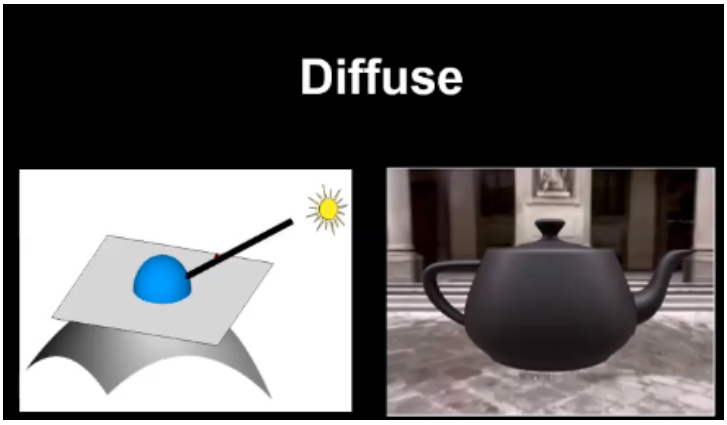
The bidirectional reflectance distribution function $f_r(w_i, w_r)$ is a function of four real variables that defines how light is reflected at an opaque surface. It's employed in the optics of real-world light, in computer graphics algorithms, and in computer vision algorithms. The function takes an incoming light direction w_i and outgoing direction w_r (taken in a coordiante system where the surface normal n lies along the $z - axis$), and returns the ratio of the reflected radiance existing along w_r to the irradiance incident on the surface from direction w_i. Each direction w is itself parameterized by azimuth angle ϕ and zenith angle θ, therefore the BRDF as a whole is a function of 4 variables.



14.8.1 Examples

Lets take a look at some examples of the BRDF function:

1. Diffuse Reflection (lambertian): in a diffusive material, the BRDF function is constant. In this case, for any light direction w_i, the intensity of the reflected light in the w_r direction is the same:

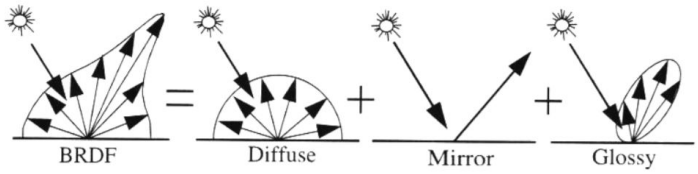


   in this case, $f_r(w_i, w_r) = \rho_d$

2. Mirror Reflection: in a mirror-like material, the BRDF is a delta function - all of the light is reflected to one direction only.

3. Glossy Reflection (we won't get into it).

4. Any other kind of reflection is something in between diffusion and mirroring reflections (combination of a Gaussian directed into the reflected direction, added to a constant).



14.8.2 Definition

In reality, a beam of light can't be a delta function. Light comes from multiple directions, and reflected into their corresponding reflected directions (depends on the material). Thus, our defintion of BRDF is:

$$f_r(w_i, w_r) = \frac{dL_r(w_r)}{dE_i(w_i)} = \frac{dL_r(w_r)}{L_i(w_i)cos(\theta_i)dw_i}$$

where L is radiance, E is irradiance, and θ_i is the angle between w_i and the surface normal n. The index i indicates incident light, whereas the index r indicates reflected light.

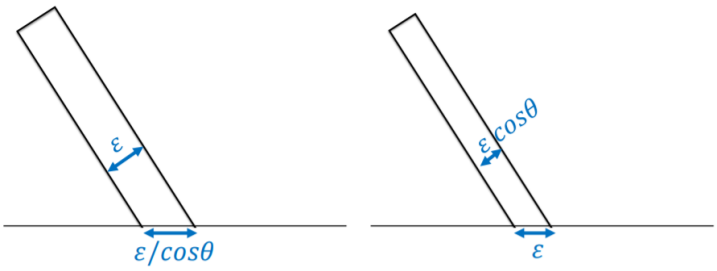
14.8.3 Properties

The BRDF function holds the following properties:

- Positivity: $f_r(w_i, w_r) \geq 0$.
- Obeying Helmholtz Reciprocity: $f_r(w_i, w_r) = f_r(w_r, w_i)$.
- Conserving energy: $\forall w_i : \int_\Omega f_r(w_i, w_r) cos(\theta_r) dw_r \leq 1$.

14.9 Beam Foreshortening

When a beam of light hits a surface with some angle (not perpendicular to the surface), such that the width of the beam is less than the effective width of the beam we observe on the surface, is called foreshortening:



14.10 Albedo

Albedo is a measure of how much light that hits a surface is reflected without being absorbed. Something that appears white reflects most of the light that hits it and has a high albedo, while something that looks dark absorbs most of the light that hits it, indicating a low albedo.

14.11 Calibrated Photometric Stereo

Photometric stereo is a technique in computer vision for estimating the surface normals of objects by observing that object under different lighting conditions. It is based on the fact that the amount of light reflected by a surface is dependent on the orientation of the surface in relation to the light source and the observer. By measuring the amount of light reflected into a camera, the space of possible surface orientations is limited. Given enough light sources from different angles, the surface orientation may be constrained to a single orientation or even overconstrained.

Under Woodham's original assumptions — Lambertian reflectance, known point-like distant light sources, and uniform albedo — the problem can be solved by inverting the linear equation $I = L \cdot n$ where I is a (known) vector of m observed intensities, n is the (unknown) surface normal (a vector in 3D), and L is a (known) $m \times 3$ matrix of normalized light directions.

This model can be easily extended to surfaces with non-uniform albedo, while keeping the problem linear. Taking an (unknown) albedo reflectivity of k, the formula for the reflected light intensity becomes:

$$I = k(L \cdot n)$$

If L is square (there are exactly 3 lights) and non-singular, it can be inverted, giving:

$$L^{-1} I = kn$$

Since the normal vector is known to have length 1, k must be the length of the vector kn, and n is the normalized direction of that vector. If L is not square (there are more than 3 lights, which is preferable to account for possible noisy measurements), a generalization of the inverse can be obtained using pseudoinverse, by simply multiplying both sides with L^T, giving:

$$L^T I = L^T k(L \cdot n) \Rightarrow (L^T L)^{-1} L^T I = kn$$

After which the normal vector (of a specific point on the surface) and albedo can be solved as described above.

14.11.1 Limitations

Big problems:

- Doesn't work for shiny objects, semi-translucent things.

- Shadows, inter-reflections.

Smaller problems:

- Camera and lights have to be distant.
- Calibration requirements: measure light source direction and intensities. We also assumed that the camera is orthographic.

14.11.2 3D reconstruction from calculated surface normals

Recall that a normal $\vec{n}$ of a 3D surface $f(x, y)$ holds:

$$\hat{\vec{n}} = (\frac{df}{dx}, \frac{df}{dy}, -k)$$

$$\vec{n} = \frac{\hat{\vec{n}}}{||\hat{\vec{n}}||} = \frac{\hat{\vec{n}}}{k}$$

where in our case, k is the albedo. So, if we know $\vec{n}$ we know the gradient of $f(x, y)$. Thus, to solve for $f(x, y)$ we can integrate the gradients we find for multiple normals.

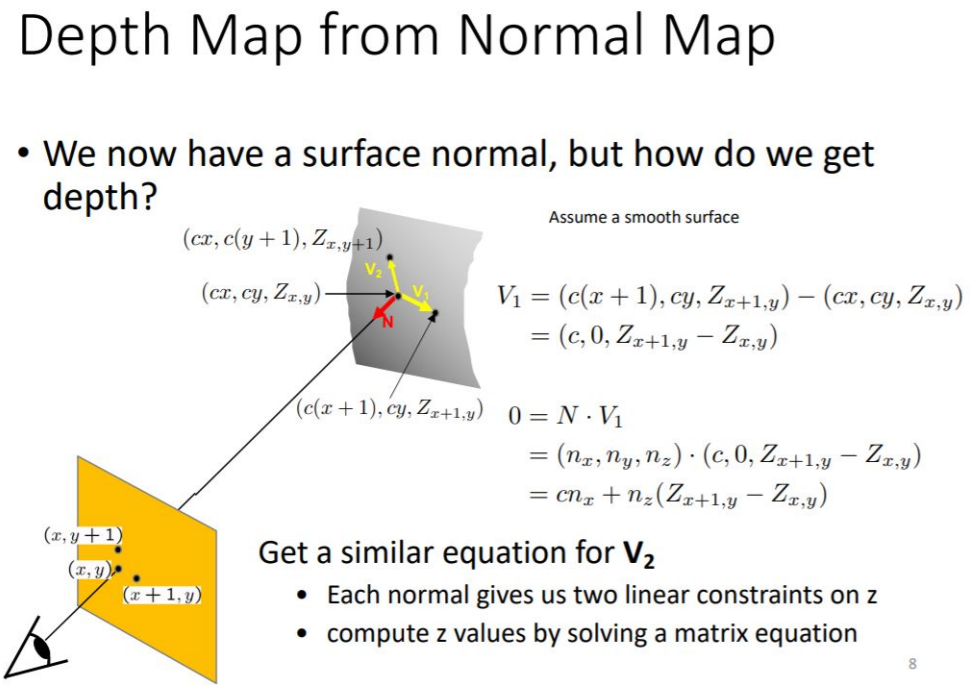


Figure 103: How we can determine the depth of a surface from a map of its normals.

Algorithm to render an object with different light than the light it was originally captured with:

1. Estimate light source directions.
2. Compute surface normals and albedo values.
3. Estimate depth from surface normals.
4. Relight the object (with original texture and uniform albedo).

Notice that photometric stereo **does not** work for convex objects (it does work for concave objects) since in convex objects we can't assume that there's one light source - there are interreflections: reflections within the concaveness of the object (imagine a picture of a shiny bowl).

14.12 Uncalibrated Photometric Stereo

When the light directions are unknown, we only have intensity measurements. Suppose we have n light directions, and that for each light direction we have an intensity measurement for each pixel out of a total of m pixels. So, suppose I is a $n \times m$ matrix of intensities, where $I_{i,j}$ is the ith intensity measurement of pixel j, L is still a $n \times 3$ matrix of unknown light directions, and b is a $3 \times m$ vector of normals (we have a total of m normals since we have m measured pixels). We thus have $3N + 3M$ unknown parameters and $N \times M$ measurements. We want the rank of I to be 3, so we can perform SVD on I to get:

$$I = UWV^T$$

and then choose L to be the first 3 columns of U, and b to be the product of the top left 3×3 block of W with the first 3 rows of V^T. This result is not a unique solution!

15 Optical Flow

Optical flow is the pattern of apparent motion of objects, surfaces, and edges in a visual scene caused by the relative motion between an observer and a scene. The optical flow methods try to calculate the motion between two image frames which are taken at times t and $t + \Delta t$ at every voxel position. These methods are called differential since they are based on local Taylor series approximations of the image signal; that is, they use partial derivatives with respect to the spatial and temporal coordinates.

For a 2D+t (space and time) dimensional case, a voxel at location (x, y, t) with intensity $I(x, y, t)$ will have moved by $\Delta x, \Delta y$ and Δt between the two image frames, and the following **brightness constancy constraint** can be given:

$$I(x, y, t) = I(x + \Delta x, y + \Delta y, t + \Delta t)$$

Assuming the movement is small, the image constraint at $I(x, y, t)$ with Taylor series can be developed to get:

$$I(x + \Delta x, y + \Delta y, t + \Delta t) = I(x, y, t) + \frac{\partial I}{\partial x}\Delta x + \frac{\partial I}{\partial y}\Delta y + \frac{\partial I}{\partial t}\Delta t + ...$$

By truncating the higher order terms, a linearization, it follows that:

$$\frac{\partial I}{\partial x}\Delta x + \frac{\partial I}{\partial y}\Delta y + \frac{\partial I}{\partial t}\Delta t = 0$$

Or, dividing by Δt

$$\frac{\partial I}{\partial x}\frac{\Delta x}{\Delta t} + \frac{\partial I}{\partial y}\frac{\Delta y}{\Delta t} + \frac{\partial I}{\partial t}\frac{\Delta t}{\Delta t} = 0$$

Which results in

$$\frac{\partial I}{\partial x}v_x + \frac{\partial I}{\partial y}v_y + \frac{\partial I}{\partial t} = 0$$

where v_x, v_y are the x and y components of the velocity, or optical flow, of $I(x, y, t)$, and $\frac{\partial I}{\partial x} = I_x, \frac{\partial I}{\partial y} = I_y, \frac{\partial I}{\partial t} = I_t$ are the derivatives of the image at (x, y, t) in the corresponding directions. Thus:

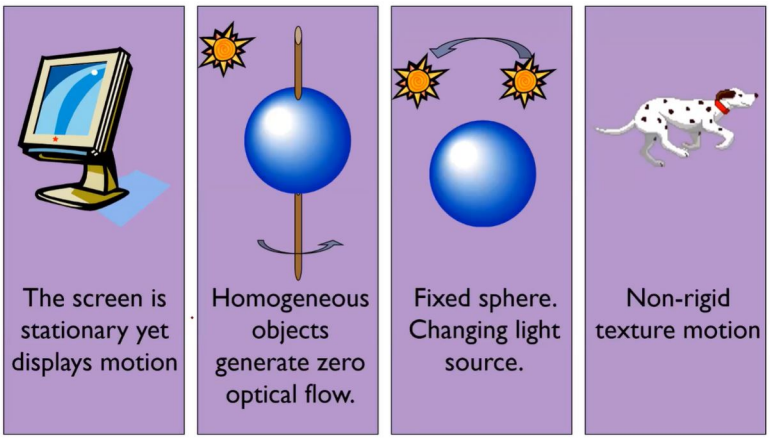
$$I_x v_x + I_y v_y = -I_t$$

or

$$\nabla I^T \cdot \vec{v} = -I_t$$

We already know how to solve for I_x, I_y and I_t (for example, we can solve for I_x by convolving with a $(-1, 1)$ filter). But, how do we compute $\vec{v}$? This is an equation in two unknowns and cannot be solves as such (we need two constraints to solve for $\vec{v}$, and we only have one). This is known as the aperture problem of the optical flow algorithm (the aperture problem refers to the fact that the motion of a one-dimensional spatial structure, such as a bar or edge, cannot be determined unambiguously if it is viewed through a small aperture such that the ends of the stimulus are not visible). To find the optical flow, another set of equations is needed, given by some additional constraint. All optical flow methods introduce additional conditions for estimating the actual flow, which we'll talk about in the next sub sections.

On another note, there are situations in which optical flow is particularly hard to determined:



Another interesting subject is Feature Tracking which somewhat relates to optical flow, but notice the difference:

- Feature Tracking: extracting visual features and track them over multiple frames.
- Optical Flow: compute image motion at each and every pixel.

15.1 Spatial Coherence

Neighboring points in the scene typically belong to the same surface and hence typically have similar motions. Since they also project to nearby points in the image, we expect spatial coherence in image flow.

15.2 Horn-Schunck Optical Flow

The Horn-Schunck algorithm assumes smoothness in the flow over the whole image: it is based on the belief that most of the objects in a series of frames are rigid and thus move together, leading to similar flow velocities in nearby pixels (similar, but not identical as in Lucas-Kanade). Thus, it tries to minimize distortions in flow and prefers solutions which show more smoothness. This solution is a global method (dense). The flow is formulated as a global energy functional which is then sought to be minimized. This function is given for two-dimensional image streams as:

$$E = \int\int [\underbrace{(I_x v_x + I_y v_y + I_t)}_{Brightness\ Constancy\ Constraint} + \alpha^2 \underbrace{(||\nabla v_x||^2 + ||\nabla v_y||^2)}_{Smoothness\ Constraint}] dxdy$$

where α is a regularization constant. Larger values of α lead to a smoother flow. This function can be minimized by solving the associated multi-dimensional Euler-Lagrange equations. These give:

$$I_x(I_x v_x + I_y v_y + I_t) - \alpha^2 \nabla^2 v_x = 0$$
$$I_y(I_x v_x + I_y v_y + I_t) - \alpha^2 \nabla^2 v_y = 0$$

where $\nabla^2 = \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2}$ is the Laplace operator (for example, $\nabla^2 v_x = \frac{\partial^2 v_x}{\partial x^2}$). In practice, the laplacian is approximated numerically using finite differences, and may be written as $\nabla^2 u(x,y) = 4(\bar{u}(x,y) - u(x,y))$ where $\bar{u}(x,y)$ is a weighted average of u calculated in a neighborhood around the pixel at location (x,y). Using this notation, the above equation system may be written as

$$(I_x^2 + 4\alpha^2)v_x + I_x I_y v_y = 4\alpha^2 \bar{v}_x - I_x I_t$$
$$(I_y^2 + 4\alpha^2)v_y + I_x I_y v_x = 4\alpha^2 \bar{v}_y - I_y I_t$$

We can solve this by formulation this problem as $A\vec{v} = \vec{b}$ and get the following iterative scheme:

$$v_x^{k+1} = \bar{v}_x^k - I_x \frac{I_x \bar{v}_x^k + I_y \bar{v}_y^k + I_t}{4\alpha^2 + I_x^2 + I_y^2}$$

$$v_y^{k+1} = \bar{v}_y^k - I_y \frac{I_x \bar{v}_x^k + I_y \bar{v}_y^k + I_t}{4\alpha^2 + I_x^2 + I_y^2}$$

where the subscript $k+1$ denotes the next iteration to be calculated, and k is the last calculated result. We iteratively update the velocity values until convergence.

In conclusion, we get the following algorithm:

1. Precompute the image gradients I_x, I_y.
2. Precompute the temporal gradient I_t.
3. Initialize flow fields to $v_x^{k=1} = 0, v_y^{k=1} = 0$ at all pixel locations.
4. While not converged, update v_x^{k+1} and v_y^{k+1} based on the iterative equation above, at each pixel location.

This method breaks when the pixel movement is not smooth compared to its neighbors.

15.3 Lucas–Kanade Optical Flow

The Lucas–Kanade method assumes "spatial coherence", that the displacement of the image contents between two nearby instants (frames) is small and approximately constant within a neighborhood of the point p under consideration (the flow is assumed to be constant in a surrounding patch of p). This solution is a local method (sparse). Thus, the optical flow equation can be assumed to hold for all pixels within a window centered at p. Namely, the local image flow (velocity) vector $\vec{v} = (v_x, v_y)$ must satisfy:

$$I_x(q_1)v_x + I_y(q_1)v_y = -I_t(q_1)$$
$$I_x(q_2)v_x + I_y(q_2)v_y = -I_t(q_2)$$
$$\vdots$$
$$I_x(q_n)v_x + I_y(q_n)v_y = -I_t(q_n)$$

where $q_1, \ldots, q_n$ are the pixels inside the window surrounding p, and $I_x(q_i), I_y(q_i), I_t(q_i)$ are the partial derivatives of the image I with respect to position x, y and time t, evaluated at the point q_i and at the current time. These equations can be written in matrix form $Av = b$:

$$A = \begin{bmatrix} I_x(q_1) & I_y(q_1) \\ I_x(q_2) & I_y(q_2) \\ \vdots & \vdots \\ I_x(q_n) & I_y(q_n) \end{bmatrix}, \vec{v} = \begin{bmatrix} v_x \\ v_y \end{bmatrix}, \vec{b} = \begin{bmatrix} -I_t(q_1) \\ -I_t(q_2) \\ \vdots \\ -I_t(q_n) \end{bmatrix}$$

This system has more equations than unknowns and thus it's usually over-determined (for example, a 5×5 patch gives us 25 equations). The Lucas-Kanade method obtains a compromise solution by the least squares principle. Namely, it solves the 2×2 system by choosing $\vec{v} = (A^T A)^{-1} A^T b$.

Notice that $A^T A$, the structure tensor, holds:

$$A^T A = \begin{bmatrix} \sum_{i=1}^n I_x(q_i) I_x(q_i) & \sum_{i=1}^n I_x(q_i) I_y(q_i) \\ \sum_{i=1}^n I_y(q_i) I_x(q_i) & \sum_{i=1}^n I_y(q_i) I_y(q_i) \end{bmatrix}$$

and $A^T b$ holds:

$$A^T b = - \begin{bmatrix} \sum_{i=1}^n I_x(q_i) I_t(q_i) \\ \sum_{i=1}^n I_y(q_i) I_t(q_i) \end{bmatrix}$$

Thus, $A^T A$ should be invertible to solve for $\vec{v}$. Using the same intuition from Harris corner detector, we can calculate the inverse of $A^T A$ only for corner-line windows (otherwise, one of the eigenvalues is small and the matrix is not invertible. $\lambda_{max}/\lambda_{min}$ should not be too large). For patches that contain an edge (and not a corner), we don't have a unique solution for the optical flow problem!

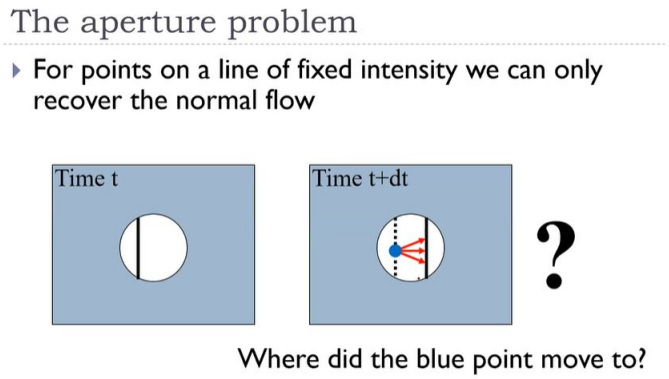


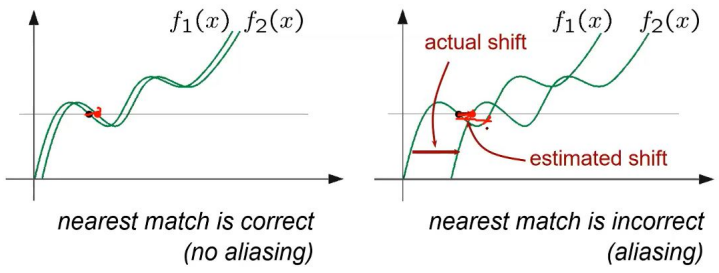
Figure 104: If the patch contains an edge, we can't uniquely determine the movement direction between time t and times $t + \Delta t$.

This method breaks when a point doesn't really moves like the points in its surrounding window.

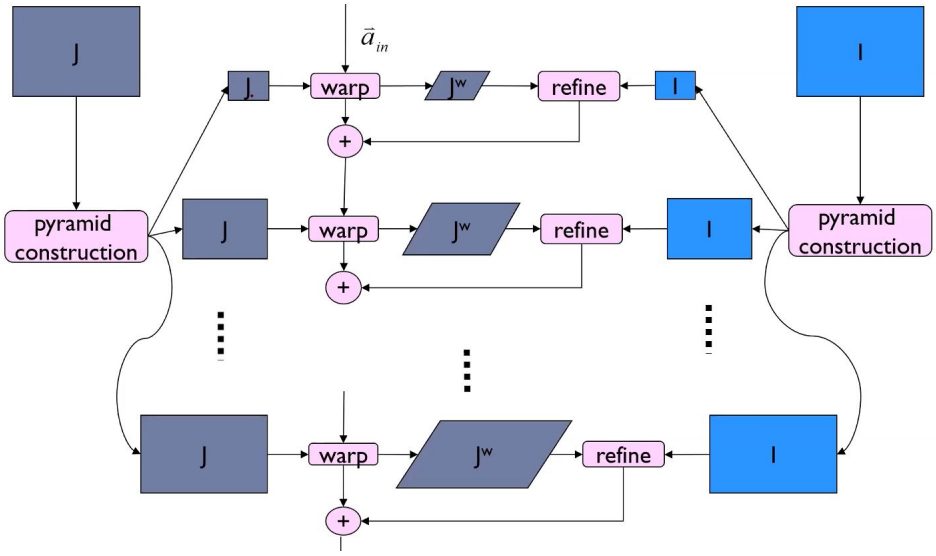
So, in general, optical flow solutions break when:

15.4 Aliasing

Temporal aliasing causes ambiguities in optical flow because images can have many pixels with the same intensity:



Solution:



15.5 Conclusion

When brightness constancy assumption doesn't hold, use correlation based methods like Lucas-Kanade's. When a point doesn't move like its neighbors, use regularization based methods like Horn-Shunck's. When the motion is not small, use multi-scale estimation methods.