Computation & Robot Vision: Lecture 1

Sam Barrett

March 4, 2021

1 Camera and image formation

1.1 The human eye

The ultimate goal of Computer vision is to allow computers to *see* in a similar (or better) way to humans.

We must first start with what we are trying to emulate: the human eye.

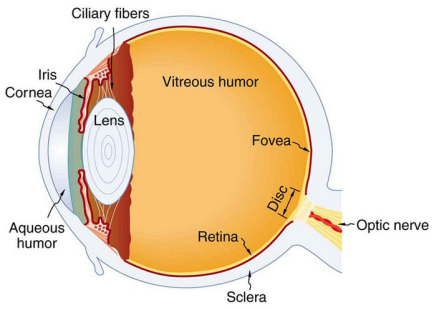


Figure 1: Diagram of the human eye

The human eye works by focusing the light entering via the pupil using the lens. This forms an image on the retina.

The retina is made up of many different types of cells, ultimately terminating in the ganglion cells which feed information to the brain via optic nerve fibres. It's composition can be seen in Figure 2.

The ganglion cells collect information regarding the visual world from bipolar and amacrine cells. The information takes the form of chemicals messages emitted by receptors on the ganglion cell's membrane.

1.2 Evolution of cameras

The evolution of artificial cameras started in the renaissance with the likes of Da Vinci experimenting with camerae obscurae (pinhole image projections) and perspective. Later, photographic film was invented, allowing for the capture of

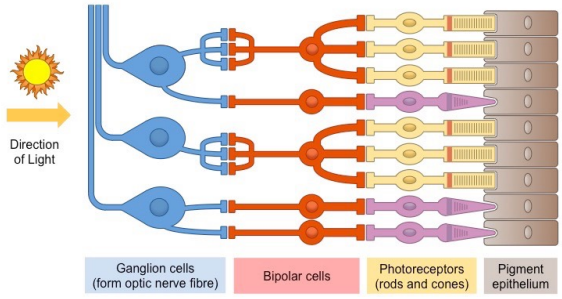


Figure 2: Diagram of the retina

images by exposing the film to different levels of light, focused by a lens the same as we have seen in Figure 1.

We have since gone on to develop entirely digital cameras but the principals remain the same. Here, the photographic film containing photosensitive particles is replaced with a camera *sensor* which are arrays of light-sensitive diodes which convert photons to electrons.

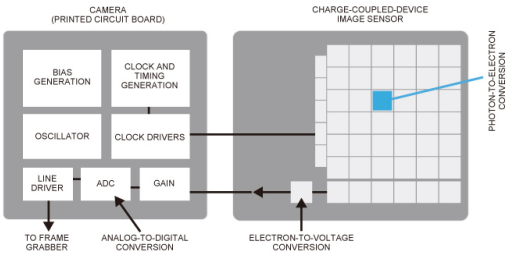


Figure 3: Basic photo-sensor construction

Digital images are a 2D grid (or matrix) of integers which show the continuous fluctuations in levels of light.

1.2.1 Encoding colour

How do we encode colour into this matrix of integers? We first need to decide on a representation of colour that a computer can understand. We do this by decomposing colours into their primary components. I.e. the different levels of red, green and blue. We can represent **any** colour using weighted combinations of these colours.

Unfortunately, each photo-diode is *colour blind* and cannot tell the colour of the light hitting it, only it's intensity. So in order to detect the colour of an image, sensors deploy filters such as the Bayer filter. These are layouts of photo-diodes such that each receptor site is known to only detect light from

a particular part of the spectrum. The Bayer filter can be seen in Figure 4. Using such a filter we can estimate the RGB at each cell using the values of it's neighbours.

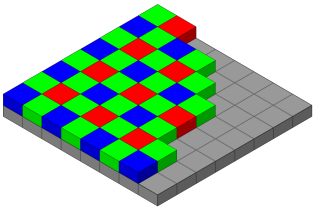


Figure 4: Bayer filter over-top of a sensor

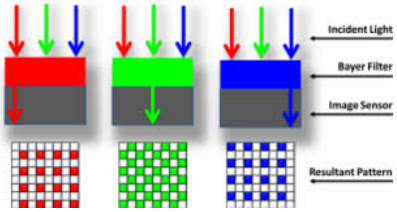


Figure 5: Diagram showing effect of the Bayer filter

Why are there twice as many green as red and blue pixels?

This is done to mimic the higher sensitivity the human eye has towards green light.

1.2.2 Digital colour images

A digital colour image is composed of 3 colour channels. Each channel is a $n \times n$ matrix of intensities representing the intensity of either red, green or blue light at any given position.

We represent each intensity value using an unsigned 8-bit integer (`uint8`). Using this encoding $(0, 0, 0)$ represents black and $(255, 255, 255)$ represents pure white.

1.3 Pinhole cameras

A pinhole camera is an abstract camera model used to explain the principals of modern cameras. They do *work* in practice.

There are several notable effects caused by using this perspective projection:

- it produces an **inverted** image.
- the size of an object is related to it's distance from the camera.

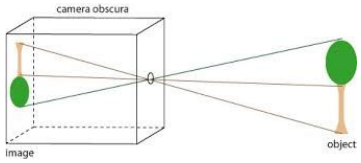


Figure 6: An example of a pinhole camera

In fact, the projections of any 2 parallel lines lying in the same plane, Π appear to converge on a horizon line H which lies at the intersection of the image plane with the plane **parallel** to Π passing through the pinhole. This can be observed in real life such as in Figure 7

Figure 7: Two parallel lines appearing to converge on the horizon

This effect is captured by our cameras meaning:

1. Parallel lines in 3D meet at the same *vanishing point* in the image.
2. The 3D ray passing through the camera centre and the vanishing point is parallel to the lines.
3. There (can) exist multiple vanishing points in the camera plane.

   We say the line connecting multiple vanishing points is the horizon line.

These properties can be proved in a geometric fashion, but it is more convenient to reason instead in terms of reference frames, coordinate and equations.

1.4 The equation of projection

Consider a coordinate system $(O, \mathbf{i}, \mathbf{j}, \mathbf{k})$ is attached to the pinhole camera, whose origin O coincides with the pinhole and the vectors $\mathbf{i}$ and $\mathbf{j}$ form a basis for a vector plane parallel to the image plane Π. Π is located a distance d from the pinhole along vector $\mathbf{k}$. The line perpendicular to Π and passing through the pinhole is called the optical axis, and the point c where it *pierces* Π is called the *image centre*. This point can be used as the origin of an image plane coordinate frame, and is important in camera calibration procedures.

Let P denote a scene point with coordinates (X,Y,Z) and p denote the corresponding image with coordinates (x, y, z)

- Since p lies on the image plane, we know that $z = d$.
- Since the three points P, O and p are co-linear, we know $\vec{Op} = \lambda \vec{OP}$ for some value λ so,

$$\begin{cases} x = \lambda X \\ y = \lambda Y \\ d = \lambda Z \end{cases} \iff \lambda = \frac{x}{X} = \frac{y}{Y} = \frac{d}{Z},$$

  and therefore, by similar triangles

$$\begin{cases} x = d\frac{X}{Z}, \\ y = d\frac{Y}{Z} \end{cases}$$

  Where we ignore the third coordinate

not really sure I follow this in the book or the slides

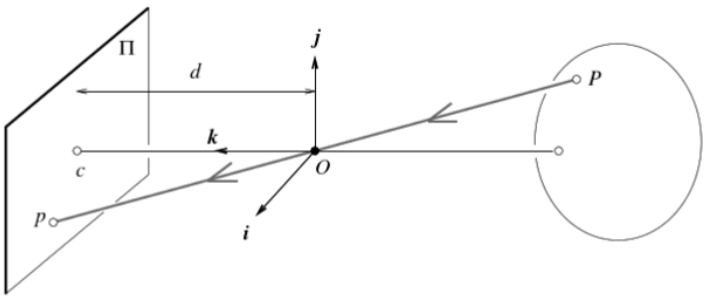


Figure 8: Deriving the perspective projection equations

1.5 Weak Perspective

The pinhole perspective is only an approximation of the geometry of the imaging process. A more accurate model is called *weak* perspective.

Consider the *fronto-parallel* plane Π_0 defined by $Z = Z_0$, for any point P in Π_0 we can reformat our earlier equation to form:

$$\begin{cases} x = -mX, \\ y = -mY \end{cases}$$

where $m = -\frac{d}{Z_0}$

Physical constrains mean that Z_0 must be negative (or simply, the plane must be in front of the pinhole), so the magnification m associated with the plane Π_0 is positive.

Consider two points P and Q in Π_0 and their corresponding projected images p and q, it is obvious that $\vec{PQ}$ and $\vec{pq}$ are parallel, meaning that $\|\vec{pq}\| = m\|\vec{PQ}\|$. This simply demonstrates the dependence of image size on object distance that we observed earlier.

An advantage of weak perspective is that it is relatively easy to use but it isn't particularly accurate. It can be used when the scene depth is small relative to the average distance from the camera, i.e. m can be taken to be constant.

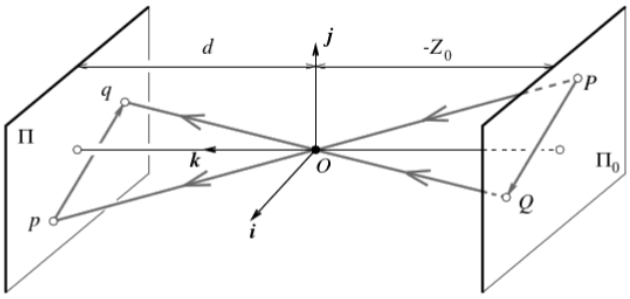


Figure 9: Weak Perspective Projection

1.6 Orthographic Projection

When it is known that the camera always remains at a roughly constant distance from the scene, we can normalise the image coordinates so that $m = -1$. This is the *orthographic projection* and is defined by:

$$\begin{cases} x = X, \\ y = Y \end{cases}$$

with all light rays parallel to the **k** axis and orthogonal to the image plane π. This can be an acceptable model in many conditions but the assumption of pure orthographic projection is usually unrealistic.

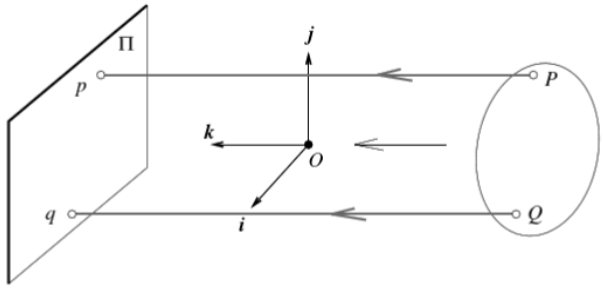


Figure 10: Orthographic Projection

1.6.1 Size of pinhole

- Pinhole too big

  Many directions are averaged, blurring the image

- Pinhole correctly sized

  Dark but sharp image

- Pinhole too small

  Diffraction effects blur the image.

Generally, pinhole cameras produce dark images as very little light is allowed through the pinhole onto the sensor.

1.7 Cameras with lenses

Most cameras are equipped with lenses, this is to circumvent the issues that arise from using a pinhole.

- lenses can help to gather more light
- lenses can help to keep the produced image in sharp focus.

1.7.1 The law of geometric optics

Lenses are governed by the law of geometric optics.

- Light travels in straight lines when travelling in homogeneous media.
- When a ray is reflected from a surface, this ray, its reflection, and the surface normal are co-planar and the angles between them are complimentary
- When a ray passes from one medium to another, it is *refracted*, meaning its direction changes.

Law 1 *(Snell's law)*

$$n_1 \sin \alpha_1 = n_2 \sin \alpha_2$$

Where:

- r_1 *is the incident ray*
- n_1 *and* n_2 *are the refractive indexes of the origin medium and destination medium respectively.*
- r_2 *is the refracted ray*
- r_1 *and* r_2 *are the normal to the incident surface and are co-planar*
- α_1 *and* α_2 *are the angles between the normal and the* r_1 *and* r_2

When the angles between these rays and the refracting surfaces of the lens are assumed to be small, Snell's law becomes $n_1\alpha_1 \approx n_2\alpha_2$. **This is known as the paraxial form of Snell's law.**

If we consider a *thin* lens with two spherical surfaces of radius R and a refractive index of n and we assume the lens is surrounded by a vacuum, rays passing through O are not refracted. This is the case as any ray passing through the right boundary of our lens is refracted but is then refracted precisely the same amount the other way when passing the left boundary.

Consider a point P which is located at $-Z$ from the optical axis (the plane on which the lens sits), and denote a ray PO which passes from this point through the centre of the lens O. It follows from our paraxial form of Snell's law that PO is **not** refracted. It is also the case that all other rays passing through P are focused by our lens to the point p located at a depth of z such that,

$$\frac{1}{z} - \frac{1}{Z} = \frac{1}{f}$$

Where $f = \frac{R}{2(n-1)}$ and is the focal length of the lens.

All of this however, relies on an idealised *thin* lens. In actuality, our lenses do not have these same characteristics.

There are other such aberrations:

- Chromatic aberration
  - Light at different wavelengths follows different paths, meaning some wavelengths are defocussed
  - This can be avoided in machines by coating the lens. Humans cannot avoid it.
- Scattering at the lens surface
  - Some light entering the lens system is reflected iff each surface it encounters (see Fresnel's law)

- This can be circumvented in machines by coating the lens and its interior. Humans must live with it.

These aberrations can also be avoided through the use of compound lenses.

Computation & Robot Vision: Lecture 2, Camera parameters

Sam Barrett

March 4, 2021

Digital images are fundamentally **spatially discrete**, meaning they are divided up into a countable number of subsections. Usually, these subsections take the form of rectangular picture elements or *pixels*.

Note: the perspective equation derived in the previous lecture is only valid when all distances are measured from the camera's reference frame and when image coordinates have their origin at the image centre, where the axis of symmetry of the camera pierces the retina or sensor.

In practice, the world and camera coordinate systems are related by a set of physical parameters including:

- The focal length f of the the lens
- the size of the individual pixels on the sensor.
- the position of the image centre
- the position and orientation of the camera.

There are a number of issues with this system however, including that one unit in the camera's coordinate system may not be the same as one unit in the world coordinate system (resulting in perspective shift or objects appearing relatively larger than they actually are). This is related to what is known as the **intrinsic parameter problem** where intrinsic parameters include focal length, principal point, aspect ratio and the angle between the axis.

Another issue is that a cameras coordinate system will have a different position and rotation in space to that of the world. This is related to the **extrinsic parameter problem**

$$\begin{pmatrix} U \\ V \\ W \end{pmatrix} = \overbrace{\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix}}^{\text{transformation representing intrinsic parameters}} \cdot \underbrace{\begin{pmatrix} X \\ Y \\ Z \\ T \end{pmatrix}}_{\text{transformation representing extrinsic parameters}}$$

1 Single-view geometry

Here we work with a single camera. Imagine there is one point in a world 3D coordinate system (c.s.), P. Our 3D c.s. has some origin O, so we can now represent P in the world with a set of coordinates relative to O. We want now to project P onto the image plane, the image plane is located in the camera 3D c.s., which is separate from the world c.s.. We can have two different kinds of projection:

1. "Extrinsic" projection: the 3D world coordinate system should be projected onto the 3D camera coordinate system
2. "Intrinsic" projection: the 3D camera coordinate system should be projected onto the 2D image plane.

1.1 Intrinsic projection

The steps of an intrinsic projection are as follows:

- Given a point P in the camera 3D coordinate system which is measured in **metres**
- We project P onto the camera image plane, also measured in **metres**
- We then project from our 2D image plane to a discretisesd image which is measured in **pixels**

1.1.1 Homogeneous co ordinates

Euclidean geometry uses the **Cartesian coordinates system**, however, for a projective geometry, **homogeneous coordinates** are more appropriate.

Conversion is simple: add an extra element at the *'end'*:

$$\overbrace{\begin{bmatrix} x \\ y \end{bmatrix}}^{\text{Cartesian form}} \rightarrow \overbrace{\begin{bmatrix} x \\ y \\ 1 \end{bmatrix}}^{\text{Homogeneous form}}$$

This system has the benefit that if a point is multiplied by a non-zero scalar value w, our point does not change:

$$\begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \equiv \begin{bmatrix} wx \\ wy \\ w \end{bmatrix}$$

In order to convert from a homogeneous to a Euclidean system: divide by the last coordinate to make it equal to 1 and ignore it.

1.1.2 Concepts

- Principal axis: this is a line from the camera centre perpendicular to the image plane.
- Principal point: this is a point where the principal axis punctures the image plane
- Normalised camera coordinate system: this is a system with its origin at the principal point

1.1.3 Pinhole camera: revisited

With our new coordinate system, we can now revisit the pinhole camera from the previous lecture.

A 3D point in the world coordinate system can be mapped to a 2D projection in the image plane as follows:

$$\begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix} \mapsto \begin{bmatrix} f\frac{X}{Z} \\ f\frac{Y}{Z} \\ 1 \end{bmatrix}$$

This can be represented as a vector-matrix multiplication:

$$\overbrace{\begin{pmatrix} fX \\ fY \\ Z \end{pmatrix}}^{\mathbf{x}} = \underbrace{\begin{bmatrix} f & & & 0 \\ & f & & 0 \\ & & 1 & 0 \end{bmatrix}}_{\mathbf{P_0}} \overbrace{\begin{pmatrix} X \\ Y \\ Z \\ 1 \end{pmatrix}}^{\mathbf{X}}$$

which can also be written

$$\mathbf{x} = \mathbf{P_0}\mathbf{X}$$

We can re-write the projection matrix $\mathbf{P_0}$ to separate the focal lengths:

$$\mathbf{P_0} = \text{diag}([f, f, 1])[\text{I}|0] = \begin{bmatrix} f & & \\ & f & \\ & & 1 \end{bmatrix} \begin{bmatrix} 1 & & & 0 \\ & 1 & & 0 \\ & & 1 & 0 \end{bmatrix}$$

1.1.4 Image plane to image pixels

- Our normalised camera coordinate system has it's origin at the principal point $p = [p_x, p_y]^{\text{T}}$.
- Our image coordinate system has it's origin in the corner of the image sensor.

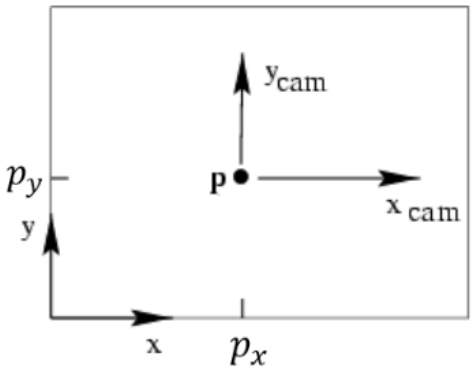


Figure 1: Figure showing camera and image coordinate systems

This can be seen in Figure 1

Moving our camera coordinate system origin to p makes our calculations much easier as we only need consider positive numbers. It allows our transformation seen above to become:

$$(X, Y, Z) \mapsto (f\frac{X}{Z} + p_x, f\frac{Y}{Z} + p_y)$$

Which, in vector-matrix multiplication becomes:

$$\begin{pmatrix} fX + Zp_x \\ fY + Zp_y \\ Z \end{pmatrix} = \begin{bmatrix} f & 0 & p_x & 0 \\ 0 & f & p_y & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{pmatrix} X \\ Y \\ Z \\ 1 \end{pmatrix}$$

1.1.5 From image plane to image pixels

We now want to project onto our sensor of size Ws $\times$ Hs (in metres). We represent pixels in a rectangular $M_x \times M_y$ matrix.

Let $m_x = \frac{M_x}{W_s}$ and $m_y = \frac{M_y}{H_s}$

We now construct the following projection in vector-matrix multiplication form:

$$\begin{pmatrix} x \\ y \\ z \end{pmatrix} = \underbrace{\begin{bmatrix} m_x & 0 & 0 \\ 0 & m_y & 0 \\ 0 & 0 & 1 \end{bmatrix}}_{\text{pixel /m}} \underbrace{\begin{bmatrix} f & 0 & p_x & 0 \\ 0 & f & p_y & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}}_{\text{m}} \begin{pmatrix} X \\ Y \\ Z \\ 1 \end{pmatrix}$$

Which can also be written:

$$\begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{bmatrix} \alpha_x & 0 & x_0 & 0 \\ 0 & \alpha_y & y_0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{pmatrix} X \\ Y \\ Z \\ 1 \end{pmatrix}$$

It is often difficult to guarantee a perfectly rectangular sensor, so we also have a case for a skewed sensor, here we simply add a single value to the projection matrix $\mathbf{P_0}$ to form:

$$\begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{bmatrix} \alpha_x & s & x_0 & 0 \\ 0 & \alpha_y & y_0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{pmatrix} X \\ Y \\ Z \\ 1 \end{pmatrix}$$

We can decompose $\mathbf{P_0}$ into two separate matrices to allow for easier computation and reasoning, we can construct $\mathbf{P_0}$ from the product of a square matrix K and a concatination of the 3×3 identity matrix and a 3D 0-vector:

$$\mathbf{P_0} = \mathbf{K}[\mathbf{I}|0] = \begin{bmatrix} \alpha_x & s & x_0 \\ 0 & \alpha_y & y_0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

We refer to $\mathbf{K}$ as our **projection matrix** which prescribes the projection of any 3D point in the camera coordinate system onto our pixels.

Where:

- $\alpha_x = m_x \cdot f$
- $\alpha_y = m_y \cdot f$
- $x_0 = p_x \cdot m_x$
- $y_0 = p_y \cdot m_y$
- s is our skewness factor

1.2 Extrinsic Projection

Here we are concerned with how to project the world coordinate system onto the 3D camera coordinate system.

The 3D camera coordinate system is related to the 3D world coordinate system by a rotation matrix $\mathbf{R}$ and translation $\tilde{\mathbf{t}} = \tilde{\mathbf{C}}$

In Euclidean terms we can write this process of translation followed by rotation as:

$$\tilde{X}_{\text{cam}} = \mathbf{R}(\tilde{X} - \tilde{\mathbf{C}})$$

Our camera is specified by a calibration matrix $\mathbf{K}$, the projection centre in the world coordinate system is given by $\tilde{\mathbf{C}}$ and a rotation matrix $\mathbf{R}$. A 3D point

given in (homogeneous) world coordinates $\mathbf{X}$ is projected onto pixels $\mathbf{x}$ by the following relation:

$$\mathbf{x} = \mathbf{K}[\mathrm{I}|0]\tilde{X}_{\mathrm{cam}} = \mathbf{K}[\mathbf{R}|t]\mathbf{X} = P\mathbf{X}$$

Where:

- $P = \mathbf{K}[\mathbf{R}|t]$
- $t = -\mathbf{R}\tilde{\mathbf{C}}$

Lenses add more complexity through non-linearity, previously straight lines are no longer straight, leading to image distortion.

Lens distortion assumes radially symmetric lenses. We radially expand an image to un-distort it, here only the point changes, not the angle.

Our extrinsic parameters include camera rotation and camera translation.

1.3 Vanishing Points

We can now give a more robust definition of vanishing points,

If we consider a point on one of two parallel lines l_1 and l_2, $\mathbf{A} = \begin{bmatrix} X_A \\ Y_A \\ Z_A \end{bmatrix}$ and a vector $\mathbf{D}$ from $\mathbf{A}$, $\mathbf{D} = \begin{bmatrix} X_D \\ Y_D \\ Z_D \end{bmatrix}$.

A point on a the line $\mathbf{X}(\lambda) = \mathbf{A} + \lambda\mathbf{D}$ is projected to a point $x(\lambda)$ in the image plane by:

$$x(\lambda) = \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} f\frac{X}{Z} \\ f\frac{Y}{Z} \end{bmatrix} = \begin{bmatrix} \frac{f(X_A+\lambda X_D)}{(Z_A+\lambda Z_D)} \\ \frac{f(Y_A+\lambda Y_D)}{(Z_A+\lambda Z_D)} \end{bmatrix}$$

Here we can see that as $\mathbf{X}(\lambda) \to \infty$, $x(\lambda)$ tends towards the vanishing point, $\mathbf{v}$.

$$\mathbf{v} = \lim_{\lambda\to\infty} x(\lambda) = \lim_{\lambda\to\infty} \begin{bmatrix} f\frac{X_A+\lambda X_D}{Z_A+\lambda Z_D} \\ f\frac{Y_A+\lambda Z_D}{Z_A+\lambda Z_D} \end{bmatrix}$$

Giving us a vanishing point of:

$$\mathbf{v} = \begin{bmatrix} f\frac{X_D}{Z_D} \\ f\frac{Y_D}{Z_D} \end{bmatrix}$$

Our vanishing point depends on the direction $\mathbf{D}$ and not on the point $\mathbf{A}$, meaning that a different set of parallel lines even if they share a point, have a different vanishing point.

1.4 Homography

Homography is the process of projecting points or images from image plane to image plane. It has many uses from Image stitching to augmented reality.

In order to estimate the homography we must find all corresponding points between the two image planes.

Assuming we know all corresponding points between $\mathbf{x}$ and $\mathbf{x}'$ we can represent the translation as:

$$w\mathbf{x}' = \mathbf{H}\mathbf{x}$$

Or,

$$w \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} H_{11} & H_{12} & H_{13} \\ H_{21} & H_{22} & H_{23} \\ H_{31} & H_{32} & H_{33} \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

Where the elements of $\mathbf{H}$ can be estimated by applying a direct linear transform or **DLT**.

For each corresponding point i we have:

$$w\mathbf{x}'_\mathbf{i} = \mathbf{H}\mathbf{x}_\mathbf{i}$$

$$w \begin{bmatrix} x'_i \\ y'_i \\ 1 \end{bmatrix} = \begin{bmatrix} H_{11} & H_{12} & H_{13} \\ H_{21} & H_{22} & H_{23} \\ H_{31} & H_{32} & H_{33} \end{bmatrix} \begin{bmatrix} x_i \\ y_i \\ 1 \end{bmatrix} = \begin{bmatrix} h_1^T \\ h_2^T \\ h_3^T \end{bmatrix} X_i = \begin{bmatrix} h_1^T X_i \\ h_2^T X_i \\ h_3^T X_i \end{bmatrix}$$

As these are equal, by the definition of cross-product:

$$\mathbf{x}'_\mathbf{i} \times \mathbf{H}\mathbf{x}_\mathbf{i} = 0$$

We can rearrange this vector product into a a vector-matrix product:

$$\mathbf{x}'_\mathbf{i} \times \begin{bmatrix} h_1^T x_i \\ h_2^T x_i \\ h_3^T x_i \end{bmatrix} = \begin{bmatrix} \mathbf{x}'_i \times \end{bmatrix} \begin{bmatrix} x_i^T h_1 \\ x_i^T h_2 \\ x_i^T h_3 \end{bmatrix} = \begin{bmatrix} 0 & -1 & y'_i \\ 1 & 0 & x'_i \\ -y'_i & x'_i & 0 \end{bmatrix} \begin{bmatrix} x_i^T h_1 \\ x_i^T h_2 \\ x_i^T h_3 \end{bmatrix}$$

We can then multiply these matrix terms and expose the homography terms h_1, h_2, h_3 into a single vector:

$$\mathbf{x}'_\mathbf{i} \times \mathbf{H}\mathbf{x}_\mathbf{i} = \begin{bmatrix} 0^T & -\mathbf{x}_\mathbf{i}^\mathbf{T} & y'_i \mathbf{x}_i^T \\ \mathbf{x}_i^T & 0^T & -\mathbf{x}'_i \mathbf{x}_i^T \\ -y'_i \mathbf{x} & & \end{bmatrix}$$

I fucking give up, fuck this

Computer Robot Vision: Lecture 3

Sam Barrett

March 4, 2021

1 Image Gradients & Edges

1.1 Edge Detection

The goal of edge detection is to take an image and map it from a 2D array of pixel values to a set of curves, line segments or contours. In so doing we filter out less relevant information whilst preserving important structural properties.

The main idea is to look for strong gradients in the pixel values. This allows us to loosly define an edge as a *place of rapid change in the image intensity function*. The first derivative of the intensity function over a horizontal section of pillar should show two spikes corresponding to each edge of the object.

We can express the partial derivative $f(x, y)$ for a 2D function as:

$$\frac{\delta f(x,y)}{\delta x} = \lim_{\varepsilon \to 0} \frac{f(x+\varepsilon, y) - f(x,y)}{\varepsilon}$$

For discrete data, such as images, we can approximate this using *finite differences*:

$$\frac{\delta f(x,y)}{\delta x} = \frac{f(x+1, y) - f(x,y)}{1}$$

We can also efficiently perform this operation using matrix convolution. We can think of an image gradient as being separate in the x and y direction, as such we must calculate the image gradient of both, $\frac{\delta f(x,y)}{\delta x}$ and $\frac{\delta f(x,y)}{\delta y}$

The filter for constructing the x derivative is simply:

-1	1

The filter for the y direction is: **check**

-1
1

The image gradient can be obtained as the partial derivative of the intensity values of an image in the x and y direction and can be denoted as:

$$\nabla f = \left[\frac{\delta f}{\delta x}, \frac{\delta f}{\delta y}\right]$$

With this value we can define:

- The **gradient direction**, or orientation of edge normal. It is given by:

$$\theta = \tan^{-1}(\frac{\delta f}{\delta y} / \frac{\delta f}{\delta x})$$

- The edge or gradient **strength** is given by the gradient magnitude:

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

1.1.1 Effects of noise

For any real-life image, if we consider a single row or column of the image and plot its intensity as a function we see a noisy signal. This noise effects our derivative as this noise is amplified by derivation.

Different filters have different sensitivity to noise. What can we do to tackle it?

One solution is to *smooth* our image first. One way of doing this is using the Gaussian kernel which can be seen in Figure 1

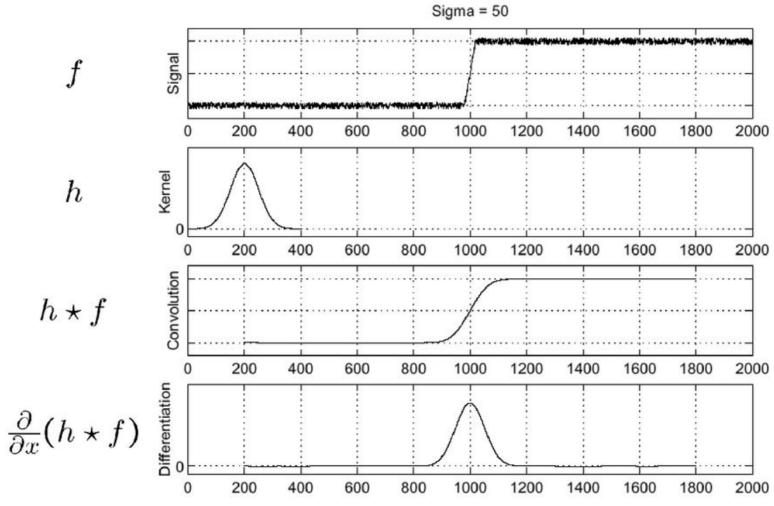


Figure 1: Effect of Gaussian smoothing on noisy function

In order to find edges in such an example, simply look for peaks in $\frac{\delta}{\delta x}(h \star f)$, where $h \star f$ is f convolved with the kernel h.

Convolution has many nice properties including being transitive and composable, allowing us to say that $\frac{\delta}{\delta x}(h \star f) \equiv (\frac{\delta}{\delta x}h) \star f$, this allows us to store the Gaussian derivative as a filter and perform one less step when finding the edges in a noisy function f.

There are other popular kernels including Prewitt, Sobel and Roberts.

Every smoothing mask has the following properties:

- Positive values
- all values sum to 1, as they take averages from surrounding pixels this makes sure the overall intensity of the image stays the same in areas of constant intensity.
- Amount of smoothing is proportional to the size of the mask
- They remove *high frequency* components, meaning the filter also acts as a *low-pass* filter.

We sometimes want to go further, we can take the second derivative by finding the derivative of our Gaussian derivative filter, this is known as the **Laplacian of Gaussian** filter $\nabla^2 h_\sigma = \frac{\delta^2}{\delta x^2}h$. When this filter is applied the edge is where the resulting graph crosses the x axis. These results are often more precise than the 1st derivative.

The size, σ of our kernel effects the sharpness and the structures found, choosing σ depends on the task at hand.

1.2 Laplacian Pyramid

This operation can be performed recursively and works by taking an image L_i and convolving it with our Gaussian filter t produce G_i, this resulting smoothed image is then up-scaled by injecting 0s in between each row and column before applying the Laplacian operator to produce an image L_i. As $i \to n$, the edges in L_i get thicker.

As i increases, only the thicker edges are present.

2 Edge detection and matching

We have seen the first step in edge detection in the previous section, smoothing. We will now look at the subsequent steps of **Edge enhancement** whereby we increase the contrast between edges and the background (via differentiation) and **Edge localisation** where we determine which local maxima are edges and which are noise.

2.1 Thresholding

The process of thresholding is as follows:

- Choose a threshold t
- set any pixels less than t to 0
- Set any pixels greater than t to 1

The standard thresholding procedure can also be defined formally as:

$$E(x,y) = \begin{cases} 1 & \text{if } \|\nabla f(x,y)\| > t \text{ for some threshold } t \\ 0 & \text{otherwise} \end{cases}$$

This process can only select *strong* edges and does not guarantee *continuity*.

2.2 Hysteresis Thresholding

An alternative method of thresholding is known as **Hysteresis thresholding**.
It uses 2 thresholds t_l, t_h, to represent a high and low boundary.

Note: usually $t_h = 2t_l$.

$\|\nabla f(x,y)\| \geq t_h$	definitely an edge
$t \geq \|\nabla f(x,y)\| < t_h$	maybe an edge, depends on the neighbouring pixel classifications
$\|\nabla f(x,y)\| < t_l$	definitely not an edge

Results of hysteresis thresholding often have better contour connectivity, i.e. more likely for entirely of edge to be preserved.

foreach *position* (x,y) **do**
  if *if* $\|\nabla f(x,y)\| < t_h$ **then**
    discard pixel (x,y)
  else
    keep pixel (x,y) as it is **strong**, and definitely an edge
  end
end
foreach *kept pixel* **do**
  foreach *connected pixel* (x,y) *(different notions of connectedness see Section 3.5)* **do**
    If $\|\nabla f(x,y)\| \geq t_l$ re-add the pixel, it is a part of an edge
  end
end

Algorithm 1: Hysteresis Thresholding

2.3 Non-maximum suppression

This is a thinning operator, and transforms wide *ridges* to a single pixel wide.

It does this by removing non-maximal pixels whilst preserving edge connectivity.

It does this by:

- Checking if a pixel is a local maxima along the gradient direction
- Selecting a single maximum across the width of the edge

  This sometimes requires interpolation if the gradient direction points towards the middle of 2 pixels.

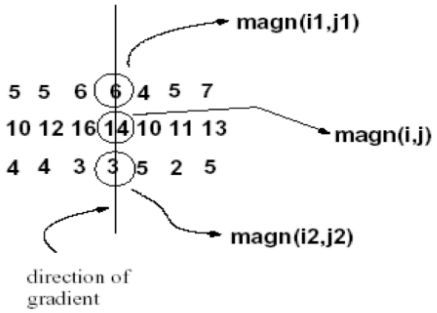


Figure 2: Non-maximum suppression

for *each pixel* (x, y) **do**
  if $magn(i,j) < magn(i_1,j_1)$ *or* $magn(i,j) < magn(i_2,j_2)$ **then**
    $I_N(i,j) = 0$
  else
    $I_N(i,j) = magn(i,j)$
  end
end

Algorithm 2: Non-Maximum Suppression

2.4 Designing an Edge Detector

We have the following criteria for an *optimal* edge detector:

1. **Good detection**

   An optimal detector will minimise the probability of false positives i.e. *"edges"* caused by noise. They will also minimise false negatives, the case in which true edges are missed

2. **Good localisation**

   Detected edges should be close to or at the same position as the true edges

3. **Specificity**

   A detector should return only a single point per true edge, i.e. edges should be sharp and not blurred over many pixels.

2.5 Canny Edge Detector

Canny came up with it, it isn't a particularly shrewd model.

This is the most popular edge detector in computer vision.

Canny showed that the first derivative of a Gaussian well approximates the operator that optimises a trade-off between signal-to-noise ratio and localisation.

The process of the Canny edge detector is as follows:

- Filter image using the derivative of Gaussian
- Calculate the gradient magnitude and direction
- Perform non-maximum suppression (See Section 2.3)
- Linking and thresholding using hysteresis thresholding

  We apply the high threshold to initialise contours, we then trace these contours until the magnitude falls below the low threshold.

2.5.1 Canny Characteristics

The Canny operator gives images with edges which are single pixel wide with good continuation between adjacent pixels. It is one of the most widely used edge detection operators and there are many sub-types. It is very sensitive to parameters, requiring extensive tuning for different domains.

3 Binary Image Analysis

Binary images are black and white, silhouette images.

Advantages:

- They are easy to acquire/take
- They require low amount of space to store
- They are (relatively) simple to process

Disadvantages:

- They have limited utility
- They cannot generally capture 3D scenes

- Specialised equipment is required to capture silhouettes.

The basic steps of binary image analysis are as follows:

1. Convert the image into a binary format

   This can be done by thresholding

2. Clean up the thresholded image

   This is performed using **morphological operators**

3. Extact separate *blobs*

4. Describe the blobs with region properties

3.1 Converting to binary form

We do this by thresholding, this is where we take a greyscale image and convert it to a binary mask.

We can use standard thresholding, hysteresis thresholding or a mask:

Where we have a set of values Z which define the intensity values which we are interested in:

$$E(x,y) = \begin{cases} 1 & \text{if } \|\nabla f(x,y)\| \in Z \\ 0 & \text{otherwise} \end{cases}$$

3.2 Morphological operators

These are operators used to change the shape of the foreground regions via intersection/ union operations between a scanning structuring element and a binary image.

This is a useful operation to *clean up* the results from thresholding.

The basic operations used are **Dilation** and **Erosion**.

3.2.1 Dilation

This operation expands connected components, grows features and fills in holes in the image. An example of what this operation does can be seen in Figure 3

At each position, if the current pixel is *on* i.e. is 1, then set all the output pixels corresponding to structuring element to 1.

This operation can be seen in a stream of binary pixels in Figure 4. You can see that the *objects* have become larger as gaps have been filled in. This procedure can scale to 2D images through the use of 2D mask.

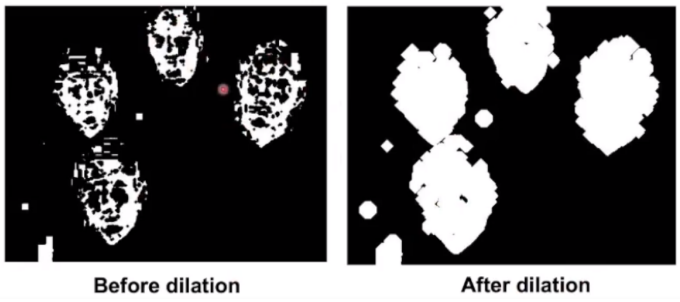


Figure 3: Before and after Dilation operation

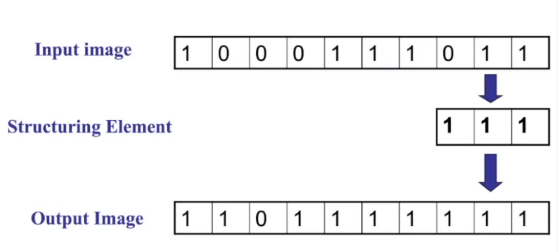


Figure 4: Dilation on a stream of binary pixels

3.2.2 Erosion

This operation *erodes* connected components, shrinks features and removes *bridges, branches* and noise.

An example of what it does to the result of our Dilation operation can be seen in Figure 5

To perform the erosion operation:

If every pixel under the structuring element is 1, then set the output pixel corresponding to the current pixel to 1, see Figure 6. Notice that the *object* get smaller.

3.3 Opening

Opening is the process of performing erosion and then dilation to a binary image. This process removes small objects but keep the original shape.

3.4 Closing

Closing is the process of performing dilation followed by erosion to a binary image. This fills in holes and keeps the original shape.

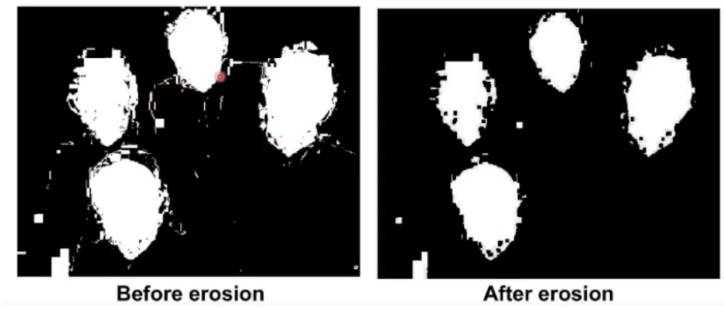


Figure 5: Erosion Example

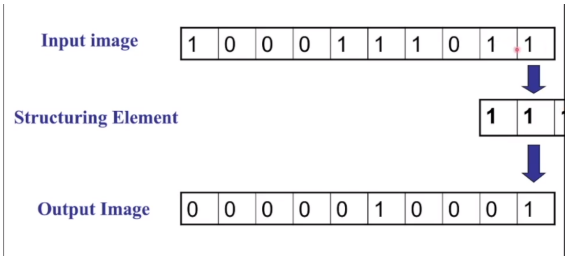


Figure 6: Erosion applied to binary pixel stream

3.5 Connectedness

How do we determine what pixels are *connected* to eachother when attempting to filter out noise without removing object detail?

Connectedness is the process of defining which pixels are considered neighbours.

3.5.1 4-Connected

This is the notion that a pixel is connected to others if it shares *northerly, easterly, southerly* or *westerly* border with them.

3.5.2 8-Connected

This is an extension of 4-Connectedness, here a pixel is deemed connected to all 8 of the pixels surrounding it, adding the diagonally connected pixels to the 4-Connected notion.

3.6 Region Properties

Given connected components, we can compute many simple *features* per blob including:

- Area, defined as the number of pixels in the region
- The centroid, the average x and y position of pixels in the region
- The bounding box, the minimum and maximum coordinates in both directions.

Computer Robot Vision: Lecture 4

Sam Barrett

March 4, 2021

1 Image Texture

Texture is a very important factor in understanding properties of an object however they can be difficult to define due to potential near random patterns.

We can use texture to:

- estimate surface orientation or shape.
- classify/ segment an image. Allowing us to group image regions with consistent texture.
- synthesise new texture patches or entire images given some example set.

By analysing texture we can often identify properties of the material it is made from. Texture acts as a *feature* that is one step abstracted from the basics of filters and edges.

We can think of textures as **repeated local patterns**. This reduces the first stage of our task of analysing texture to finding patterns. Patterns can include: spots, bars, raw patches etc. We describe textures with statistics related to their local window including mean, standard deviation etc.

We can use our x and y derivative filters to find the texture *responses* in each section of an image. We can then find local statistics by moving a arbitrarily large *window* across the image.

Having calculated this information we can identify all windows in the image with high vertical or horizontal (or both) activity, these areas are likely to have *texture* following these directions. Colour-coding these 4 sets (no intensity, high x, high y, high x, y) we can roughly visualise textures present in the image.

On a plot of mean $\frac{\delta}{\delta x}$ against mean $\frac{\delta}{\delta y}$ the further apart 2 windows are, the more dissimilar their textures. We measure the distance between two points on this plot simply as Euclidean distance ($D(a, b) = \sqrt{\sum_{i=1}^{2}(a_i - b_i)^2}$)

When performing this form of analysis, the window size we use is very important. If our window is close ins size to an object of interest, made up of multiple textures, we will lose a lot of information.

So far, we have discussed using 1st order derivative filters to find texture information. We can more generally say that we aapply a collection of multiple,

d, filters called a *filter bank*. This results in feature vectors in d dimensions. We can still consider *nearness* as the Euclidean distance but now we consider it in d dimensional space. This is a simple extension to the previous equation:

$$D(a,b) = \sqrt{\sum_{i=1}^{d}(a_i - b_i)^2}$$

1.1 Filter Banks

We design filter banks so that they cover most bases of possible textures. They contain filters specific to edges, bars, spots or any other textural feature that we are interested in.

Many of these different filters are **multivariate Gaussian** filters. Multivariate Gaussian filters can be stretched and contorted to respond to different shapes through **manual** tuning of its parameters. Multivariate Gaussian filters have the form:

$$p(x;\mu,\Sigma) = \frac{1}{(2\pi)^{n/2}|\Sigma|^{1/2}} \exp\left(-\frac{1}{2}(x-\mu)^T\Sigma^{-1}(x-\mu)\right)$$

By changing the parameter Σ we can change the direction and distribution of the filter.

More recent work has been done to *learn* filter banks using Convolutional Neural Networks. Here highly specialised filters are learnt from a training set in order to closely match the domain in which the model will be deployed.

CNNs use different levels of filter banks, low level features include basic edges and textures, mid-level filters detect more precise but still abstract features and high-level filters are more concrete still, detecting exact patterns or even objects.

1.2 Texture Synthesis

There are many applications for being able create new samples of a given texture:

- Virtual environments (games, VR, etc.)
- Hole-filling on images
- Texturing surfaces

The challenge with this sort of task the spectrum of textures we are trying to replicate, ideally with a single general solution. We want to model both repeated and stochastic (random) textures.

1.2.1 Markvov Chains

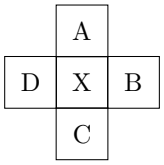
A Markov chain is a sequence of random variables $x_1, x_2, \ldots, x_n$ where x_t is the state of the model at a given time t.

They have been used for text synthesis, with the most basic implementations building a probability histogram or table featuring the probabilities of a word being present given the previous words seen ($p(x_t|t_{t-1}, \ldots, x_{t-(n-1)})$)

More modern approaches use machine learning models such Long-short-term-memory networks or LSTMs. The results of these models are better but still often nonsensical.

We can extend the approach of Markov chains to help us to predict and synthesise textures in images. We do this using **Markov random fields** (MRF), these are generalisations of Markov chains to be in 2 or more dimensions.

A first order MRF (works on 2 dimensions) computes the probability that pixel X takes a certain value **given** the values of neighbours A, B, C, D, $P(X|A, B, C, D)$



1.2.2 Single pixel synthesis

In order to synthesise a value for a single pixel x we need to calculate $P(x|\text{neighbourhood of pixels around } x)$.

To do this we:

- first find all windows **elsewhere in the image** that match the neighbourhood of x.
- Pick 1 matching window at random, W
- assign x the centre pixel of W.

Often an exact neighbourhood match will not exist, so we find the best matches using sum of squared differences (SSD) error and choose between them with a weighted probability, preferring *better* matches.

The size of our neighbourhood window effects the predictions we make. It must be set according to the texture being synthesised. Brickwork, for example, should not be synthesised using a window bigger than a single brick etc.

1.2.3 Efros & Leung algorithm

This is the algorithm for synthesising texture pixel-by-pixel, it is **simple**, yields **good results** but is **very slow**.

1.2.4 Efros & Freeman algorithm

This is an improved version of the Efros and Leung algorithm, it is based on the observation that neighbourhood pixels are highly correlated. The main idea is to use a *block* as the base unit of synthesis rather than a single pixel.

This changes our conditional probability calculation to $P(\mathbf{B}|N(\mathbf{B}))$, this is much faster as we now synthesise all pixels in $\mathbf{B}$ at once.

The layout of blocks used greatly effects performance of this algorithm.

- Placing blocks randomly that do not overlap results in sharp, unnatural edges in the synthesised texture
- Making neighbouring blocks overlap produces better but still unnatural results.
- The minimal error boundary cut approach produces the best result

  This approach is done by overlapping 2 blocks. The edge between them is clear so we take the two overlapping sections of each block and calculate the overlap error. This error gives us the *minimal error boundary*, cutting each block along this boundary allows us to accurately stitch these two blocks together.

  This approach can be seen in Figure 1

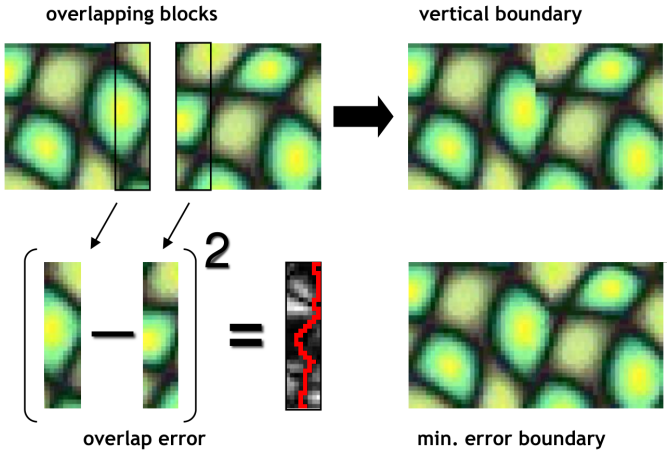


Figure 1: Minimal error boundary calculation

1.3 Texture Transfer

Texture transfer is the process of mapping a texture from one object to another. This requires being able to separate texture from the shape of an object.

This is very difficult to do, so we cheat. We assume we can capture shape through boundaries and rough shading, then we just add similarity to underlying image at the same point as a constraint when sampling.

2 Image Segmentation

2.1 Grouping in vision

The goal of grouping in computer vision is to:

- Collect features that belong together

  This can include determining image regions by colour or texture, or grouping video frames into *shots* to help us extact people or objects from a video.

- Obtain an intermediate representation that compactly describes key image or video components

There are two main approaches to grouping: **Top-down** and **bottom-up** segmentation.

Top down segmentation means that pixels belong together as they are from the same object. This approach requires prior knowledge about objects in the image, we then infer that similar pixels are together.

Bottom up segmentation means that pixels belong together as they *look* similar.

2.2 Bottom up segmentation

Algorithms include k-means clustering and mean-shift clustering. There are also graphical approaches.

The goals of segmentation are to:

- separate images into coherent *objects*.

- group together similar looking pixels.

  This is for efficiency of later processing. These blocks of similar pixels are known as *super-pixels* and are very useful for higher level processing.

2.2.1 k-means clustering

This algorithm will **always converge** to some solution. However, this can be a local minimum. I.e. it does nto always find the global minimum of the objective function $\mathcal{F} = \sum_{i=0}^{k} \sum_{j=0}^{n} \|p_j - c_i\|^2$.

It is also very fast and simple to compute. However, setting k is task specific (or inefficient to determine), the process is sensitive to the initial cluster centres as well as to outliers.

We also assume that the means of the our points can be computed.

Randomly initialise the cluster centres $c_1, \ldots, c_k$
while *cluster assignments change* **do**
  foreach *each point* p **do**
    find the closest centre c_i, place p into cluster i
  end
  foreach *cluster,* c_i **do**
    Calculate new centroid (mean point in cluster)
  end
end

Algorithm 1: k-means clusering

2.2.2 Segmentation as clustering

We can group pixels in different ways based on what we select to be our feature space.

Our feature space can be many different things including (but not limited to):

- Intensity
- Colour similarity (3D feature space)
- Intensity and position
- Texture similarity
- n-dimensional feature space

2.2.3 Mean shift algorithm

The mean shift algorithm looks for *modes* or local maxima of density in the feature space.

Initialise random seed and window W
while *not converged* **do**
  calculate the centre of gravity (mean) of W
  shift W to the mean
end

Algorithm 2: Mean shift algorithm

This can be applied to clustering/ segmentation by:

1. Finding features based on some feature space (colour, gradients, texture, etc.)
2. Initialising windows at each individual feature point
3. Performing mean-shift for each window to convergence

4. Merge windows that end up near the same *peak* or mode

Advantages of mean-shift

- Does not assume shape of clusters (k-means assumes spherical clusters)
- Single parameter to tune (window size)
- Generic technique
- Can find multiple modal values

Disadvantages of mean-shift:

- Manual selection of window size
- Does not scale well with dimensionality of feature space (texture etc.)

2.2.4 Graphical approaches

We can also view images as fully-connected graphs. Each node is a pixel and there is a edge between every pair of pixels.

We can assign an *affinity weight* to each edge $w_{p,q}$ measures similarity.

One definition for affinity is:

$$\mathit{aff}(x, y) = \exp\left\{-\left(\frac{1}{2\sigma_d^2}\right)(\|x - y\|^2)\right\}$$

Where small values of σ will group only nearby points and large values of σ will group distant points.

With a image graph and this notion of affinity, we can break graph into segments. We want to remove edges that cross between segments. The easiest method breaks the links that have low similarity. This means that similar pixels should be in the same segments.

MinCut We can construct a notion of **MinCut**, the set of links in a graph G with two subgraphs representing segments A, B whose removal makes G disconnected. The cost of a cut is given as:

$$\mathit{cut}(A, B) = \sum_{p \in A, q \in B} w_{p,q}$$

Finding **MinCut** gives us a segmentation of the image represented by G, there exist efficient algorithms to find this.

The issue with this is that the weight of a cut is proportional to the number of edges in the cut. This means it tends to produce small, isolated components.

Normalised cut Normalised cut is a MinCut alternative. Here we fix the bias of **MinCut** by normalising the size of the segments.

$$Ncut(A,B) = \frac{cut(A,B)}{assoc(A,V)} + \frac{cut(A,B)}{assoc(B,V)}$$

Where $assoc(A,V)$ is the sum of weights of all edges that touch A.

The value of $Ncut$ is small when we get two clusters with many edges with high weights and few edges of low weight between them. The a solution to $\min(Ncut)$ is approximately the same as the generalised eigenvalue problem.

Advantages of normalised cuts

- Generic framework, flexible choice of definition for *affinity*, which assigns weight to edges.
- Does not require model of data distribution

Disadvantages:

- Can have high time complexity, dense graphs require many affinity calculations. Also solving the eigenvalue problem is non-trivial
- It has a preference for balanced partitions.

There is an efficient graph-based segmentation approach, it runs in time linear in the number of edges, it is easy to control coarseness of segmentation however, the results can be unstable.

3 Fitting

In fitting we aim to select a parametric model that best represents a set of features.

We know that membership of a model cannot be determined on a point-local basis, we must consider surrounding pixel values also.

We can reduce fitting to three main questions:

1. What model represents this set of features best?
2. Which of several model instances get which feature?
3. How many model instances are there?

When considering fitting we must be mindful of time complexity, it is not computationally feasible to examine every possible set of parameters and every possible combination of features.

3.1 Line fitting

The simplest type of feature we often wish to fit are straight lines. Many objects are characterised by the presence of straight lines.

But why can this not be achieved simply through edge detection?

There are often multiple extra edge points in an image which can lead to clutter in our models as we ask the question *which points go with which lines, if any?*

Sometimes only sections of a line are detected, how do we find lines that span sections of an image with imperfect information?

There is often noise in measured edge points and orientations, how do we remove noise to detect the true underlying parameters?

3.2 Voting

We have said that it is not feasible to check all combinations of features by fitting a model to each possible subset. **Voting** is a general technique whereby we let each feature *vote* for all models that are compatible with it.

It works by cycling through features and casting votes for model parameters, we then look for the model parameters that receive the most votes.

Features created due to noise and clutter will still vote in this system but generally their votes should be outweighed by the larger set of *"good"* features.

3.3 Hough Transforms

Hough Transforms are a voting technique used to determine the following information:

- Given some point that belong to *a* line, what is the line?
- How many lines are there in a given image?
- Which points belong to which line?

The main idea behind this technique is:

- each straight line in an image can be described by a (unique) equation
- Each point on this line, when considered in isolation, could lie on an infinite number of other straight lines
- In the Hough Transform, each point votes for every line it could be on; the lines with the most votes *win*

How do we represent lines? Any line can be represented using 2 numbers, w and ϕ. We say a line is a line from an agreed origin of length w at an angle ϕ to the horizontal.

Since we can use (w, ϕ) to represent any line in the image space, we can represent any line in the image space as a point in the plane defined by (w, ϕ), we call this plane the **Hough Space**.

How does a point in the image space vote? We can transform a point in the image space to a sinusoidal curve in the Hough Space through the equation:

$$w = x\cos(\phi) + y\sin(\phi)$$

Therefore, two points in the image space correspond to two curves in the Hough space. The intersection of those two curves has two *votes* . This intersection represents the straight line in the image space that passes through both points.

The basic Hough Transform algorithm is:

Create an array `A` indexed by ϕ and w
foreach *point* (x, y) *in the image space* **do**
  foreach *angle* $\phi \in [\phi_{min}, \phi_{max}]$ **do**
    $w = x\cos(\phi) + y\sin(\phi)$
    $\texttt{A}[\phi, w] = \texttt{A}[\phi, w] + 1$
  end
end
Find the value(s) of w, ϕ where $\texttt{A}[w, \phi]$ is maximum
The detected line in the image is given by $w = x\cos(\phi) + y\sin(\phi)$

Algorithm 3: Hough Transform

When we have a noisy image however, we still see votes being cast in the Hough space. This can lead to false positives.

We can extend Hough transform to include the gradient direction, this is a piece of information that we calculate during the edge detection phase.

We can alter our algorithm to form:

Create an array `A` indexed by ϕ and w
foreach *point* (x, y) *in the image space* **do**
  ϕ = gradient at (x, y)
  $w = x\cos(\phi) + y\sin(\phi)$
  $\texttt{A}[\phi, w] = \texttt{A}[\phi, w] + 1$
end
Find the value(s) of w, ϕ where $\texttt{A}[w, \phi]$ is maximum
The detected line in the image is given by $w = x\cos(\phi) + y\sin(\phi)$

Algorithm 4: Hough Transform

This reduces the degrees of freedom.
Other extensions that we can make are:

- Give more votes to stronger edges by incorporating the magnitude of the gradient
- Change the sampling of w, ϕ to increase or reduce the resolution
- We can extend this procedure for use on circle, square or any other regular shape.

3.3.1 Hough Transform for circles

Given a circle of centre (a, b) and radius r we define it parametrically as:

$$(x_i - a)^2 + (y_i - b)^2 = r^2$$

Known r For a known radius r the parameter space is reduced to 2D, each point represents the centre of a circle. For each point (x, y) on the original circle, there is a corresponding circle centred at (x, y) with a radius r in the Hough space. The intersection point of all such circles in the Hough space would be corresponding to the centre point of the original circle. See Figure 2

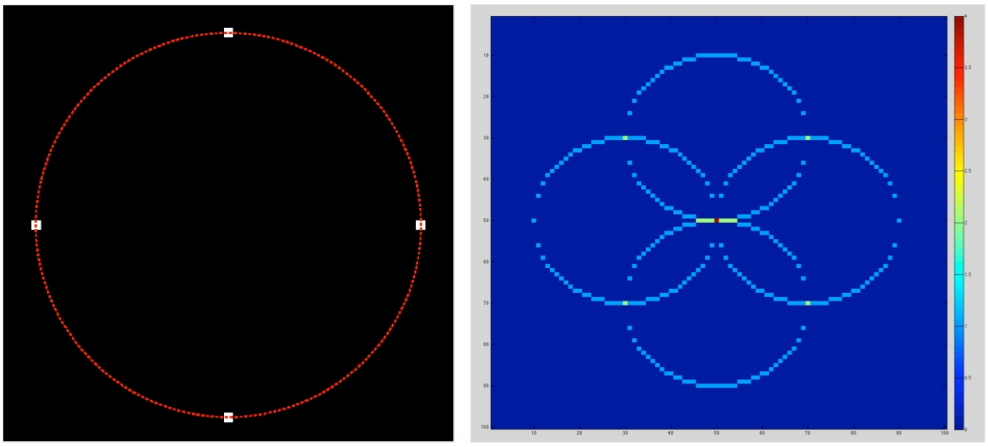


Figure 2: Circle Hough Transform

Unknown r In the case where the radius, as well as the centre, is unknown, the parameter space (Hough space) is 3 dimensional. Each point in the image space corresponds to a cone in the Hough space. The point of the cone is at $r = 0, x, y$. We can still find the set of parameters that gain the most votes in the Hough space by finding intersections between these cones. See Figure 3

Unknown radius, known gradient direction This is a subset of the previous case. By knowing the gradient direction, we make our calculation much simpler. Every point in our image space now becomes a single straight line in the Hough space. Votes are cast by finding intersections between these lines. See Figure 4

In this case, we can re-arrange our circle equation into the form:

$$x = a + r\cos\theta$$
$$y = b + r\sin\theta$$

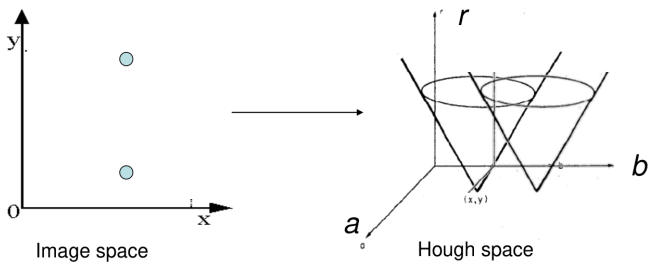


Figure 3: Circle Hough Transform for an unknown r

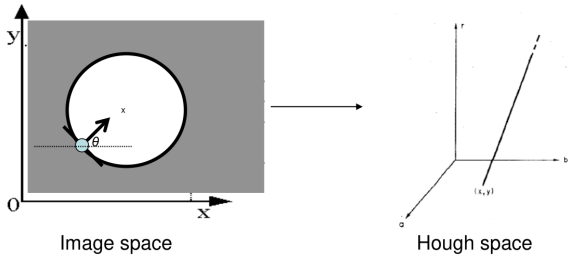


Figure 4: Known gradient direction Hough Transform

Here, every point (x, y) in the image space will be equivalent to a circle in the (a, b) Hough space. By rearranging these equations we get:

$$a = x_i - r\cos\theta$$
$$b = y_i - r\sin\theta$$

If the radius is know then the centre of the circle can be calculated.
UNFINISHED

Computer Robot Vision: Lecture 5

Sam Barrett

March 4, 2021

1 Feature-Based alignment

Feature based alignment can be used to match known objects in a cluttered scene. It can also be used in enhancing medical imaging by combining multiple images into more detailed, accurate images.

We have previously seen how to fit a model to image evidence in the form of matching a line to edge points. Here, we will fit the parameters of some transformation according to a set of matching feature pairs, i.e. we will be able to work out what transformation an image has undergone to produce another image with matching features.

1.1 Parametric (global) warping

Parametric warping is a form of warping and can be caused by:

- translation
- rotation
- aspect shift
- affine shift
- perspective shift

These are all examples of transformations and so can be thought as a black-box operator which takes an image and returns a altered image.

Mathematically, this can be thought of as:

$$\mathbf{p}'_{x',y'} = T(\mathbf{p}_{x,y})$$

For some transformation or *coordinate-changing machine* T.

T being **global** tells us:

- It is the same for any point p
- It can be described parametrically

We can represent T as a matrix $\mathbf{M}$, and we can then rewrite our transformation of p to p' as:

$$p' = \mathbf{M}p$$

or

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \mathbf{M} \begin{bmatrix} x \\ y \end{bmatrix}$$

Scaling a coordinate means multiplying each of its components by a scalar.

Uniform scaling means this scalar is the same for all components

Non-uniform scaling means that different scalars apply to different components (where components are dimensions).

Scaling can be represented as:

$$x' = ax$$
$$y' = by$$

or in matrix form:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \underbrace{\begin{bmatrix} a & 0 \\ 0 & b \end{bmatrix}}_{\text{Scaling matrix } S} \begin{bmatrix} x \\ y \end{bmatrix}$$

Using a 2x2 matrix as seen above, we can represent:

- 2D scaling

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

- 2D rotation around $(0, 0)$

$$x' = \cos(\Theta)x - \sin(\Theta)y$$
$$y' = \sin(\Theta)x + \cos(\Theta)y$$
$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos\Theta & -\sin\Theta \\ \sin\Theta & \cos\Theta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

- 2D shear

$$x' = x + sh_x \times y$$
$$y' = sh_y \times x + y$$
$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} 1 & sh_x \\ sh_y & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

- 2D mirror about Y axis

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

- 2D Mirror about $(0, 0)$

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

- 2D translation

$$x' = x + t_x$$
$$y' = y + t_y$$

NOTE: 2D translation cannot be represented in matrix form

From this we can see that 2D linear transformations can be represented using a 2×2 matrix.

Linear transformations are combinations of:

- scale
- rotation
- shear
- mirror

1.2 Homogeneous coordinates

These are a convenient coordinate system to use to represent transformations.

To convert to this system we add an extra dimension to our points, defaulted at 1.

$$(x, y) \rightarrow \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

To convert **from** homogeneous coordinates:

$$\begin{bmatrix} x \\ y \\ w \end{bmatrix} \rightarrow \left(\frac{x}{w}, \frac{y}{w}\right)$$

Using this system we can represent 2D translations in matrix form, using 3×3 matrices.

$$x' = x + t_x$$
$$y' = y + t_y$$

This translation matrix can now be represented as:

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

Leading to translations of the form:

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

We can re-format our basic 2D transformations seen earlier into transformations in the homogeneous coordinate space:

- Scale

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

- Rotation

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

- Shear

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & sh_x & 0 \\ sh_y & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

1.3 2D Affine Transformations

An affine transformation is a combination of both Linear transformations and translations and have the general form of:

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

When using these transformations, **parallel lines are preserved.**

2 Affine Fit

Image alignment approaches fall into two broad categories:

1. Direct (pixel-based) alignment

   Here we search for the alignment where most pixels *agree*

2. Feature-based alignment

   Here we search for alignment based on where extracted features *agree*. Our result can then be verified by direct alignment.

2.1 Fitting an Affine transformation

Assume that we know the pixel correspondences between two images. How do we work out the transformation T which maps the first image to the second?

In

$$\begin{bmatrix} x_i' \\ y_i' \end{bmatrix} = \begin{bmatrix} m_1 & m_2 \\ m_3 & m_4 \end{bmatrix} \begin{bmatrix} x_i \\ y_i \end{bmatrix} + \begin{bmatrix} t_1 \\ t_2 \end{bmatrix}$$

We want to find $m_{1..4}$ and $t_{1..2}$

We can start by rewriting our equation to the form:

$$\begin{bmatrix} & & \cdots & & & \\ x_i & y_i & 0 & 0 & 1 & 0 \\ 0 & 0 & x_i & y_i & 0 & 1 \\ & & \cdots & & & \end{bmatrix} \begin{bmatrix} m_1 \\ m_2 \\ m_3 \\ m_4 \\ t_1 \\ t_2 \end{bmatrix} = \begin{bmatrix} \cdots \\ x_i' \\ x_i' \\ \cdots \end{bmatrix}$$

How many pixel correspondences do we need to solve this for the transformation parameters (the values of the 1x6 matrix)? We require 3 corresponding pairs.

Once we have these values, we can easily calculate the corresponding point of a point in either image. Say we want to find the corresponding point of (x_{new}, y_{new}) we would calculate the following:

$$\begin{bmatrix} & & \dots & & & \\ x_{new} & y_{new} & 0 & 0 & 1 & 0 \\ 0 & 0 & x_{new} & y_{new} & 0 & 1 \\ & & \dots & & & \end{bmatrix} \begin{bmatrix} m_1 \\ m_2 \\ m_3 \\ m_4 \\ t_1 \\ t_2 \end{bmatrix} = \begin{bmatrix} \dots \\ x'_{new} \\ x'_{new} \\ \dots \end{bmatrix}$$

But where do we get the initial 3 corresponding pairs?

One method is to scan the entire second image for a match for a single point in the first. Similarity between two patches can be calculated as the *Sum of Squared Differences (SSD)*. However, this is clearly very computationally expensive. Another alternative is called **SIFT**, we will not go into detail.

3 RANSAC

We require a more robust method for dealing with outliers. Large disagreements in a small subset of points can cause failure in least-squares based approaches. This can happen often as a lot of data in computer vision suffers from noise, in some cases over half of the data is expected to be outliers!

Outliers can result in the erroneous pairing of points in two images.

RANSAC stands for: **RAN**dom **S**ample **C**onsensus. It is very popular as it is generalisable and simple to implement. It is robust against large proportions of outliers.

The general approach of RANSAC is to ignore outliers. Intuitively we say that if an outlier is chosen to compute the current fit, then the resulting line won't have much *support* from the rest of the points.

The algorithm has the general form:

while *iterations* < *maxIterations* **do**
  Randomly select minimal subset of points as a *seed* group
  Compute a model using these points
  for *each point* **do**
    calculate error wrt. this model
    Select all points *consistent* with this model, i.e. points that fit with the model within some tolerance.
  end
end
return model with largest number of inliers, largest number of supporting points

We can apply this to our transformation fitting example.

1. Randomly select the smallest group of point correspondences from which we can estimate the parameters of our model
2. Fit the parametric mode to the selected correspondences
3. Count how many of all correspondences are in agreement, this is the number of inliers

We repeat the above for a set number of iterations and return the model with the highest number of correspondences supporting it.

4 Panoramic Images

The basic process for creating a panorama (or mosaic) is as follows:

- Take a sequence of images from the same position
- compute transformation between second and first image
- transform the second image to overlap with the first
- blend the two images
- repeat for all images

Abstractly, when creating a panoramic image, we are moving the plane on which the real camera location exist to a synthetic plane on which the images line up. This can be done as long as all images have the same centre of projection. (See Figure 1)

We can think of our mosaic as a synthetic wide-angle camera. We re-project our images onto a common plane on which the mosaic is formed.

In order to map a pixel from one image to another from the same image centre, we can cast a ray through each pixel in the first image and overlay it where it passes through the second image. This can be seen more clearly in Figure 2

We can consider our re-projection as a 2D image warp from one image to another instead of as a 3D re-projection.

4.1 Image re–projection: Homography

A projective transform is a **mapping between any two perspective projections with the same centre of projection**

Any rectangle should map to an arbitrary quadrilateral, parallel lines are not preserved but straight lines are. This process is called **homography** and has the general mathematical form:

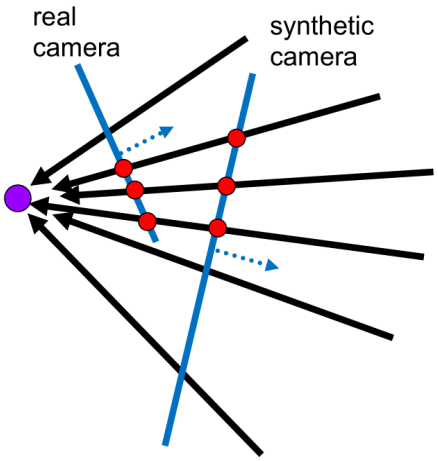


Figure 1: Generating synthetic views

$$\underbrace{\begin{bmatrix} wx' \\ wy' \\ w \end{bmatrix}}_{\mathbf{p}'} = \underbrace{\begin{bmatrix} * & * & * \\ * & * & * \\ * & * & * \end{bmatrix}}_{\mathbf{H}} \underbrace{\begin{bmatrix} x \\ y \\ 1 \end{bmatrix}}_{\mathbf{p}}$$

4.2 The projective plane

We again employ homogeneous coordinates. These allow us to represent points at infinity, homographies, perspective projection and multi-view relationships.

Geometrically, we can say that a point in the image corresponds to a ray in the projective space. (See Figure 3)

We can say that each point in the image (x, y) is represented by a ray (sx, sy, s) in the projective plane. **All points on the ray are equivalent** $(x, y, 1) \equiv (sx, sy, s)$

4.3 Solving for Homographies

We now want to attempt to solve the equation outlined earlier:

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

$$\begin{bmatrix} wx' \\ wy' \\ w \end{bmatrix} = \begin{bmatrix} h_{00} & h_{01} & h_{02} \\ h_{10} & h_{11} & h_{12} \\ h_{20} & h_{21} & h_{22} \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

up to some scale factor determined by the **Frobenius norm** of $\mathbf{H}$ being 1.

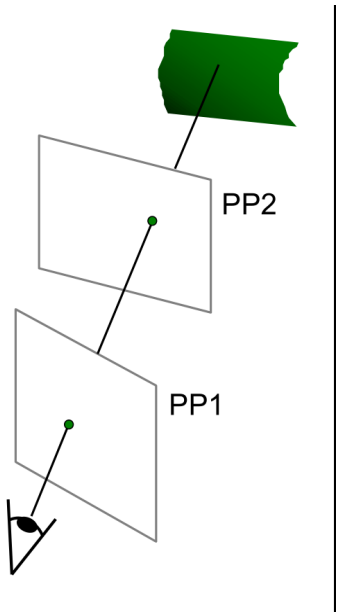


Figure 2:

We define the Frobenius norm as follows:

$$\|A\|_F \equiv \sqrt{\sum_{i=1}^{m}\sum_{j=1}^{n}|a_{ij}|^2} = \sqrt{\mathrm{Tr}(\mathbf{A}\mathbf{A}^H)}$$

Where $\mathbf{A}^H$ is known as the conjugate transpose and $\mathrm{Tr}(\mathbf{A}) = \sum_{i=1}^{n} a_{ii}$

The conjugate transpose is a matrix $\mathbf{A}$ is found by taking the transpose of $\mathbf{A}$ and then taking the complex conjugate of each element in $\mathbf{A}^T$. The complex conjugate being, for $a + ib$, $a - ib$

We therefore, formulate the problem:

$$\min \|Ah - b\|^2$$

s.t. $\|h\|^2 = 1$

Where h is a vector of unknowns $h = \begin{bmatrix} h_{00} & h_{01} & h_{02} & h_{10} & h_{11} & h_{12} & h_{20} & h_{21} & h_{22} \end{bmatrix}^T$

5 2D Image warping

Given a source image f and a transform T, how do we compute the transformed image $g = T(f)$? We can send each pixel $f(x, y)$ to the corresponding location $(x', y') = T(x, y)$ in the second image. But sometimes $T(x, y)$ produces a location *between* pixel values.

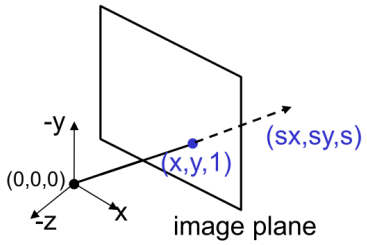


Figure 3: Image point in the projective plane

In this case we distribute the colour of $f(x, y)$ between the neighbouring pixels, this is known as **splatting**.

5.1 Inverse warping

When performing the reverse procedure, if a pixel value comes from between two pixels, we interpolate its colour from surrounding pixels. There are different types of interpolation: nearest neighbour, bilinear, etc.

6 Changing camera centre

We have stated previously that our transformations work when we have a common camera centre for all the images we are attempting to stitch together.

However, we can still perform these operations if our camera centre were to change but only if the (3rd) projection plane is sufficiently far away from all camera centres. This is how large-scale aerial photographs are taken.

7 RANSAC for estimating homography

We can use RANSAC for estimating H as defined in our earlier equations. The general approach for this is as follows:

1. Select feature pairs at random
2. Compute exact homography, H
3. Compute inliers, pairs where $SSD(p_i', \mathbf{H}p_i) < \varepsilon$
4. Keep largest set of inliers
5. Re-compute least-squares H estimate on all inliers

8 Local Features

The main components of local features are as follows:

1. Detection, being able to identify the points of interest in an image
2. Description, being able to extract a feature vector (descriptor) describing the surrounding of each point of interest.
3. Matching, being able to determine the correspondence between two descriptors in two views.

With these components we desire the following properties:

- Repeatability: We want the same features to be found in several images despite geometric and photometric transformations.

  This would likely not happen if we were to simply employ random sampling. We require this to be the case at least partially as we must run detection separately on each image.

- Saliency: Each feature should have a distinctive description

  We want to be able to reliably determine which point goes with which. To achieve this, we must provide some invariance to geometric and photometric differences between the two views

- Compactness and efficiency: We want to find far fewer features than there are image pixels

- Locality: A feature occupies a relatively small area of the image and is robust against clutter and occlusion.

Image features have found uses in image alignment, 3D image reconstruction, motion tracking and robot navigation to name a few.

8.1 Key Point Detection

There are a number of keypoint detectors, including *Hessian & Harris*, *Laplacian, DoG* and many others.

Corners are widely considered to make good keypoints, they are repetitive and distinctive and can be easily recognised using a relatively small kernel. Windows around a corner should see large changes in gradient and gradient direction.

8.1.1 Harris corner detector

This detector is based on the intensity change for a shift $[u, v]$

$$E(u, v) = \sum_{x,y} w(x, y)[I(x + u, y + v) - I(x, y)]^2$$

Where:

- w is the weight function and $w(x, y)$ is the weight at a point (x, y)

  The weight function can be anything but a common one is a Gaussian kernel

- $I(x + u, y + v)$ is the intensity after the shift

- $I(x, y)$ is the intensity before the shift.

For small shifts, E can be linearly approximated as:

$$E(u, v) \approx \begin{bmatrix} u & v \end{bmatrix} M \begin{bmatrix} u \\ v \end{bmatrix}$$

With M being a 2×2 matrix of image derivatives:

$$M = \sum_{x,y} w(x, y) \begin{bmatrix} I_x(x, y)I_x(x, y) & I_x(x, y)I_y(x, y) \\ I_x(x, y)I_y(x, y) & I_y(x, y)I_y(x, y) \end{bmatrix}$$

This can be further optimised by replacing the summation with a convolution by a Gaussian kernel:

$$M = \begin{bmatrix} G(\sigma) \star I_x^2 & G(\sigma) \star I_xI)_y \\ G(\sigma) \star I_xI_y & G(\sigma) \star I_y^2 \end{bmatrix} = G(\sigma) \star \begin{bmatrix} I_x^2 & I_xI_y \\ I_xI_y & I_y^2 \end{bmatrix}$$

8.1.2 Eigenvectors & Eigenvalues

The general form of eigenvalues and eigenvectors is:

$$M\mathbf{e} = \lambda\mathbf{e}$$

Where $\mathbf{e}$ is an eigenvector and λ is an eigenvalue.

Most vectors change direction when multiplied by a matrix M, however, certain exceptions exist. These are known as eigenvectors. In these cases multiplying our vector $\mathbf{e}$ by M is the same as multiplying $\mathbf{e}$ by a scalar λ, this is known as a eigenvalue. This tells us whether (and how) $\mathbf{e}$ is changed when multiplied by M, it can be any change other than a change in direction.

We can rewrite this as:

$$(M - \lambda I)\mathbf{e} = 0$$

Where I is the identity matrix.

To calculate the eigenvector, we must:

1. Compute the determinant of $M - \lambda I$, the result of which is a polynomial
2. Find the roots of the polynomial, i.e. solve $\det(M - \lambda I) = 0$, this returns our eigenvalues
3. For each eigenvalue, solve $(M - \lambda I)\mathbf{e} = 0$ for $\mathbf{e}$

8.1.3 Covariance matrix analysis

We can decompose our covariance matrix M into eigenvectors and eigenvalues:

$$M = R \begin{bmatrix} \lambda_{max} & 0 \\ 0 & \lambda_{min} \end{bmatrix} R^T$$