CSC320 Notes

Visual Computing

Last updated March 7, 2024

Contents

1 Image Transformations

Transformations we can do to images:

- Scaling
- Warping (preserves straight lines)
  - The basic transformation that is used to scan documents; identify the corners (perhaps by hand) which should be enough to do the warp
  - A homography / linear transformation

What are the class of transformations used to perform the operation? First, we need to know more about affine transforms.

Summary

- Two homogeneous coordinates are the same if they're multiples of each other (except 0)

- A point at infinity is where the last element of a homogenous coordinate is 0; can be represented as an angle from 0 to 180 degrees
- We can represent a line by a vector $\begin{bmatrix} a \\ b \\ c \end{bmatrix}$ such that $\begin{bmatrix} a & b & c \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = 0$ or more familiarly $ax + by + c = 0$
- Given points in homogeneous coordinates p_1, p_2, the homogeneous coordinates of the line that passes through them is $p_1 \times p_2$
- Given two lines in homogeneous coordinates, their point of intersection is $l_1 \times l_2$
- Convert a homogeneous point coordinate to a regular coordinate by scaling it so that the last element is 1, then remove the last element
- Affine transformations preserve parallelism, and look like $\begin{bmatrix} a & b & c \\ d & e & f \\ 0 & 0 & g \end{bmatrix}$

1.1 Notation Conventions

- Assume that an image is continuous. This means accessing a point on the image can be done with continuous $\mathbb{R}$ values.
- **Points are represented using column vectors:** $\begin{bmatrix} x \\ y \end{bmatrix}$, bottom 0 top image height
  - Row vectors are matrices that only contain a single row: $\begin{bmatrix} x & y \end{bmatrix}$
  - Transposing: $\begin{bmatrix} x \\ y \end{bmatrix}^T = \begin{bmatrix} x & y \end{bmatrix}$
- Homogenous coordinate representation of any point $p \in \mathbb{R}^2$: Euclidean coordinate to homogeneous coordinates
  - $\begin{bmatrix} x \\ y \end{bmatrix} \rightarrow \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$

  - When we represent a point in homogeneous coordinates, we don't represent that point with that vector only. It's this vector and any scaled version of this vector, all represent the same 2D point. In other words, for any $\lambda \in \mathbb{R} \setminus \{0\}$, $\lambda \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$ represents the same 2D point.
  - $p \cong \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \cong \begin{bmatrix} -2x \\ -2y \\ -2 \end{bmatrix} \cong \begin{bmatrix} 2x \\ 2y \\ 2 \end{bmatrix}$
  - Two vectors of homogeneous coordinates are called equal if they represent the same 2D point.
  - $\begin{bmatrix} x \\ y \\ w \end{bmatrix} \cong \begin{bmatrix} x' \\ y' \\ w' \end{bmatrix} \iff \exists \lambda \neq 0, \begin{bmatrix} x \\ y \\ w \end{bmatrix} = \lambda \begin{bmatrix} x' \\ y' \\ w' \end{bmatrix}$
- Homogeneous coordinates to Euclidean coordinates:
  - $\frac{1}{c} \begin{bmatrix} a \\ b \\ c \end{bmatrix} [0:2]$

1.2 Points at Infinity

$$\begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} \to \begin{bmatrix} \infty \\ 0 \end{bmatrix}, \begin{bmatrix} 1 \\ 1 \\ 0 \end{bmatrix} \to \begin{bmatrix} \infty \\ \infty \end{bmatrix}$$

With homogeneous coordinates, we have a finite representation of a point that is infinitely far away. We can represent a point infinitely away only using $\mathbb{R}$. This leads to very stable geometric computations.

Points at infinity have their last coordinate equal to 0. Points at infinity are also called *ideal points* in textbooks.

What do points at infinity represent? The space described by these homogenous coordinates are called a projected plane: the Euclidean plane and a bit more.

You can encode them as a clock position with arrows on both hands (0-180 degrees)

1.3 Homogeneous 2D Line Coordinates

How do we represent a line?

We have a line l on the plane. Suppose there is a point p that lies on the line: $p \cong \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$.

What's the most general equation for a line?

$$ax + by + c = 0$$

($y = mx + b$ cannot encode vertical lines) In matrix form, the homogeneous coordinates of a line can be represented by:

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

So, a line can also be represented by this. If you multiply this equation by any non-zero scalar, the line remains the same. Meaning for all $\lambda \neq 0$, this represents the same line.

$$\lambda \begin{bmatrix} a & b & c \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = 0$$

The vector $\begin{bmatrix} a \\ b \\ c \end{bmatrix}$ is the vector holding the line coordinate. It can be interpreted in any

way.

1.3.1 Line Coordinates Conversion Examples

What are the homogeneous coordinates of the line $y = x$? They're written in $l^T p = 0$

$$
\begin{aligned}
& y = x \\
& \iff -x + y = 0 \\
& \iff -x + y + 0(1) = 0 \\
& \iff \begin{bmatrix} -1 & 1 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = 0
\end{aligned}
$$

$\begin{bmatrix} -1 \\ 1 \\ 0 \end{bmatrix}$ are the homogeneous coordinates of this line.

1.3.2 Coordinates of the line passing through two points

What are the homogeneous coordinates of the line that connects two points?

The setup, given p_1, p_2:

$$l^T p = 0$$

The line passes through two points, so they must satisfy the line equation: l must satisfy $l^T p_1 = 0$, $l^T p_2 = 0$

The fact that $l^T p_1 = 0$, $l^T p_2 = 0$ implies that l must be perpendicular to p_1 and p_2, so it means that l must be the cross product between the two

So, the general expression is, if I know the homogeneous coordinates of p_1 and p_2, then I can get the homogeneous coordinates of the line that passes through both.

$$l = p_1 \times p_2$$

So, given two image points $\begin{bmatrix} x_1 \\ y_1 \end{bmatrix}, \begin{bmatrix} x_2 \\ y_2 \end{bmatrix}$?

1. Convert to homogeneous coordinates
2. Compute the cross product

This gives us an immediate expression for the equation of the line.

1.3.3 Calculating the cross product

As a matrix-vector product:

$$p_1 \times p_2 = \begin{bmatrix} 0 & -z_1 & y_1 \\ z_1 & 0 & -x_1 \\ -y_1 & x_1 & 0 \end{bmatrix} p_2$$

So, we have an analytical expression for computing the line coordinates from two points.

Alternatively, as a determinant:

$$\begin{aligned} p_1 \times p_2 &= \begin{vmatrix} i & j & k \\ x_1 & y_1 & z_1 \\ x_2 & y_2 & z_2 \end{vmatrix} \\ &= i \begin{vmatrix} y_1 & z_1 \\ y_2 & z_2 \end{vmatrix} - j \begin{vmatrix} x_1 & z_1 \\ x_2 & z_2 \end{vmatrix} + k \begin{vmatrix} x_1 & y_1 \\ x_2 & y_2 \end{vmatrix} \end{aligned}$$

i, j, k are short hands for vectors. $i = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}, j = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}, k = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$.

Feel free to use whatever you want for cross products.

1.4 Coordinates of the Intersection of Two Lines

We have two lines that we know, and we want to find the homogeneous coordinates of their intersection.

We know line $l_1 = \begin{bmatrix} a_1 \\ b_1 \\ c_1 \end{bmatrix}$, $l_2 = \begin{bmatrix} a_2 \\ b_2 \\ c_2 \end{bmatrix}$. Find $p = \begin{bmatrix} x \\ y \\ z \end{bmatrix}$. It must satisfy:

$$l_1^T p = 0,\ l_2^T p = 0$$

So, what is p? It's the cross product.

$$p = l_1 \times l_2$$

We have a very easy way to compute intersections.

1.4.1 In case they're parallel

Now, what happens when the two lines are **parallel?** You'll get a point at infinity, with a 0 at the third coordinate. Here's an example given $y = 1$, $y = 2$:

$$l_1 = \begin{bmatrix} 0 \\ 1 \\ -1 \end{bmatrix},\ l_2 = \begin{bmatrix} 0 \\ 1 \\ -2 \end{bmatrix}$$

We have the homogeneous coordinates of two lines. Their intersection is going to be the cross product, $l_1 \times l_2$. What's the result?

$$l_1 \times l_2 = \begin{bmatrix} 0 & 1 & 1 \\ -1 & 0 & 0 \\ -1 & 0 & 0 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \\ -2 \end{bmatrix} = \begin{bmatrix} -1 \\ 0 \\ 0 \end{bmatrix}$$

This represents negative infinity at the x-axis. Everything is completely finite from the perspective of homogeneous coordinates, but this is a point of infinity. In a different

direction, you would get a different point at infinity. This is how we can check if two lines are parallel. If their intersection computed through the cross product gives us 0 at the last coordinate, they have to be parallel.

The order of the terms in the cross product do not matter due to how homogeneous coordinates work. A point at infinity could be seen as a clock with a handle that points in both directions

1.5 Affine Transformations

A matrix can transform all vectors in a space.

Scaling: Where x is the scale of x

$$\begin{bmatrix} x & 0 \\ 0 & y \end{bmatrix}$$

Shearing:

$$\begin{bmatrix} 1 & \text{horizontal shear} \\ \text{vertical shear} & 1 \end{bmatrix}$$

Rotations: shear horizontally and vertically by the same amount. The cosine function prevents size changes from rotating.

$$\begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix}$$

Translation: requires homogeneous coordinates

$$\begin{bmatrix} 1 & 0 & \Delta x \\ 0 & 1 & \Delta y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

Chaining multiplications **compose transformations**, the order being from right to left. You can encode many transformations in a single matrix. However, regardless of how you multiply, as long as you are multiplying transform matrices, it they will **always** be in this form:

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

The last row will **always be** $\begin{bmatrix} 0 & 0 & 1 \end{bmatrix}$. It may look like $\begin{bmatrix} 0 & 0 & g \end{bmatrix}$ in some cases.

We can multiply this entire matrix by any scalar except 0 and the homogeneous transformation would remain the same.

Most general affine transform matrix:

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

1.5.1 Geometric Properties Preserved by Affine Transformations

PRESERVED

- Parallelism
  - Parallel line pairs stay parallel to each other. Remember, parallel lines intersect at infinity
  - The transformation maps point at infinity to points at infinity (it may not be the same point at infinity)
  - Why? Because check the bottom row $\begin{bmatrix} 0 & 0 & g \end{bmatrix}$, the first two are 0
  - What if they were not? Then, some non-infinity points might be mapped to infinity and vice versa. This implies that parallelism would not be preserved (possibly but not always a 3D rotation could do this).

NOT PRESERVED

- Angles
- Lengths

1.6 Projective Transforms

Any 2D transform of homogeneous coordinates that is **represented by an invertible** 3×3 **matrix**. Known as Homography. It will **not** preserve parallelism.

For example, the way scanner apps distort images is a projective transform.

This is a fundamental distinction between general linear transformations from affine transformations. Affine transformations are much more restrictive; scanner apps can't use affine transforms by themselves.

The scanning procedure depends on figuring out what is the homography the 3×3 matrix that allows us to take a raw image and convert it to a proper, scanned image.

$$\begin{bmatrix} a & b & c \\ d & e & f \\ l & m & g \end{bmatrix}$$

Homographies preserve linearity. Any lines that lie before a homography, will remain on another line after a homography.

1.7 Forward Mapping Algorithm

Suppose that we have the homography transformation matrix H. Now, create an algorithm that creates an image given an old image.

There is a bunch of ways to do it. There's a very easy way that doesn't give great results, and there's a slight tweak that solves it.

Our input: `src_image`, H

Output: `dest_image`.

How does this work? Let's see how the array of pixels can be represented as points on a 2D plane. Every pixel is just a (row, column) coordinate and we need to convert it to an (x, y) point on the plane.

So, the forward mapping algorithm:

```
for c=1 to num_columns
  for r=1 to num_rows:
    x, y = pixel_xy(r, c) # get the source pixel's (x, y)
       coordinates
    p = homogeneous_coords(x, y)
    p_prime = H * p
    x_prime, y_prime = euclidean_coords(p_prime)
    r_prime, c_prime = pixel_rc(x_prime, y_prime) # floor,
       round, ceiling, I don't care for now
    dest_image(r_prime, c_prime) = src_image(r, c)
```

This algorithm already has problems.

If I stretch the image, we could have gaps that will never be filled. A lot of my pixels in the destination image will contain nothing.

If I shrink the image, I can have the opposite problem. Two pixels from the source might map overwrite an existing written pixel in the destination image. We're losing information; not a good thing either.

It's possible to have both happen in the same image.

There's a very simple fix for this: an algorithm that doesn't go forward; it goes backwards.

1.8 Backward Mapping Algorithm

We have a loop that goes over the destination pixels. We go over the (x, y) coordinates in the destination image and use the inverse homography H^{-1} to figure out what pixel to target in the source image.

Because we are looping over the destination pixels, we can look at every single pixel in the destination image, and we'll get a value for every one of them. The destination image will get filled. There will be no gaps, regardless of whether we're doing magnification, stretching, and so on.

This means that will every single pixel in the destination image be filled. **No**, there could be blank pixels. A pixel in the destination image might map to a pixel **outside** the source image.

```
for c_prime=1 to num_columns
  for r_prime=1 to num_rows:
     x_prime, y_prime = pixel_xy(r_prime, c_prime)
     p_prime = homogeneous(x_prime, y_prime)
     p = inverse(H)*p_prime
     x, y = euclid(p)
     r, c = pixel_rc(x, y)
     des_image(r_prime, c_prime) = source[r, c]
```

2 Image Projection

How do we relate 3D points in the world to 2D pixels in an image?

We'll look at

- The geometry of perspective projection and concepts of center of projection, focal length
- How to represent 3D rays and points in homogeneous 3D coordinates
- Proving that all perspective images of a plane can be stitched via Homographies
- Why is it that homography warping is enough
- How do we estimate the Homographies we need?
- Understanding what 3D information is lost by perspective projection
- Making 3D measurements on a planar surface by warping its photo to a canonical view

- So on

2.1 Camera Aperture

Why do cameras have an aperture? Why not just have the sensor and nothing in part of it? Isn't that good enough?

Can't orthographically ray-trace. You'll get an extremely blurry image. All pixels receive light from all possible points.

So, instead, let's use an aperture. This results in a 1-1 correspondence between world points and image points, ideally. Like what you've been told, you'll get an upside down horizontally reflected image. This way of getting images has been known for hundreds of years; this is called a camera obscura.

2.1.1 Adjusting the Aperture

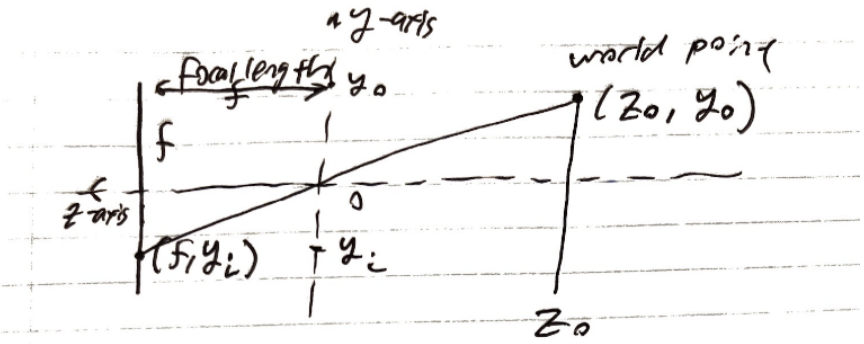
- ↑ Aperture size, ↑ Blurriness

2.2 Geometry of Perspective Projection

The focal length is the distance between the aperture and the image plane. Decreasing the focal length gives us a smaller image. It's a number that describes magnification.

- ↑ Focal length, ↑ image size, ↓ field of view

The lower the focal length, the image is being concentrated in a smaller set of pixels if the sensor's pixel count remains constant. Yet, it also gives us more field of view.



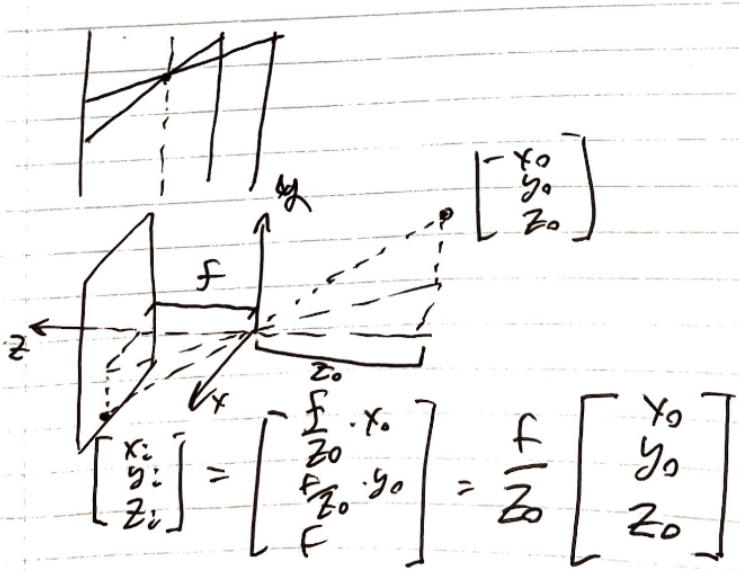
from similar triangles

focal length

$$\frac{y_i}{y_0} = \frac{f}{z_0} \Rightarrow y_i = \frac{f}{z_0} \cdot y_0$$

dist from origin

Image of a point is a scaled version



$$\begin{bmatrix} x_i \\ y_i \\ z_i = f \end{bmatrix} = \frac{f}{z_0} \begin{bmatrix} x_0 \\ y_0 \\ z_0 \end{bmatrix}$$

Where:

- $\begin{bmatrix} x_i \\ y_i \\ z_i \end{bmatrix}$ is the location on the camera sensor
- $\begin{bmatrix} x_0 \\ y_0 \\ z_0 \end{bmatrix}$ is the location of the actual image
- f is the focal length
- For the purposes of axis-alignment, $-z_0$ is the distance from the aperture to the image.
- $\begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$ is the pinhole / apeature

Observation: a lower magnitude of z_0 (but constant x_0, y_0) means the object is closer to the camera, so it appears larger on the sensor (points are more spread out)

If we scale $\begin{bmatrix} x_0 \\ y_0 \\ z_0 \end{bmatrix}$ by a constant factor, how the image is projected **will look the same.**

This is how we can think of 2D homogeneous coordinates (think of superliminal).
Interpretation of homogeneous equality: All 3D points having the same projection:

$$\begin{bmatrix} x_i \\ y_i \\ z_i \end{bmatrix} \cong \begin{bmatrix} x_0 \\ y_0 \\ z_0 \end{bmatrix}$$

2.3 Representing 3D Points in Homogeneous Coordinates

3D coordinates with scale invariance can only represent rays. So, how do we represent 3D points? Introducing homogeneous 3D coordinates, like always, defined up to a scale factor.

$$\underbrace{\begin{bmatrix} x_0 \\ y_0 \\ z_0 \end{bmatrix}}_{\text{euclidean}} \rightarrow \underbrace{\begin{bmatrix} x_0 \\ y_0 \\ z_0 \\ w_0 \end{bmatrix}}_{\text{homogeneous}} \rightarrow \underbrace{\frac{1}{w_0} \begin{bmatrix} x_0 \\ y_0 \\ z_0 \end{bmatrix}}_{\text{euclidean}}$$

And guess what? $\begin{bmatrix} x_0 \\ y_0 \\ z_0 \\ 0 \end{bmatrix}$ represents a point at infinity. Homogeneous coordinates allow us to represent points at infinity in 3D.

Let's look at how homogeneous coordinates help us simplify the expression for perspective projection.

Let's expand our *point in world space to sensor space* transformation using our new homogeneous coordinates system:

$$\begin{bmatrix} x_i \\ y_i \\ 1 \end{bmatrix} = \begin{bmatrix} \frac{f_0}{z_0} x_0 \\ \frac{f_0}{z_0} y_0 \\ 1 \end{bmatrix} = \begin{bmatrix} f & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} x_0 \\ y_0 \\ z_0 \\ 1 \end{bmatrix}$$

(We can change the scaling of the input $\begin{bmatrix} x_0 \\ y_0 \\ z_0 \\ 1 \end{bmatrix}$ and we would still get the "same" sensor space coordinate.)

2.4 Alignment and Stitching

Let's talk about images of specific geometric objects (like lines and planes). A very important property of prospective project is that it preserves linearity. All the lines

that were straight in the original source is straight in the output. It falls from the fact that our transformation has a matrix in between – it's a linear transformation. But it's also possible to reason geometrically why lines in the world map lines to the image.

In order to take two photographs of the same object in two different viewpoints and stitch them together (in order to do a homography), **two conditions must hold for the homography:**

- Lines map to lines
- Each line in one image is transformed to a **unique** line in the other (invertibility) – in other words, we can reverse the transformation (minus loss of quality).

2.4.1 Linearity of Perspective Projection

Why does perspective projection preserve linearity?

Projection goes through a center of projection. Every point of a line gets mapped to the sensor along the center of projection.

2.5 When can I Stitch Together?

- If the image can be aligned
- Viewpoint must remain the same (same lines **in real life**) must map to the same place in the sensor – **preserve the center of projection.** You may rotate your camera, but your camera **must be anchored at the center of projection (usually the pinhole).**
  - Beware, an iPhone panorama is **not** that because you are moving your phone very far. It's not a real photograph anymore. You wouldn't be able to create a camera with a wider field of view and capture the same image.
  - That blue fence in the slide is curved not because the requirements for stitching failed – it's some post processing just for visual intent. Maybe your sensor is not planar – that doesn't matter.

  - Your Minecraft screenshots by moving your character's camera angle can be stitched together (with some assumptions I'm making).
  - **What a 360 camera can do**

- Photos taken from pure camera rotation can be aligned and stitched

Place the camera (or at least the center of projection) in the same place and you can stitch.

Radial distortions break linearity. Images that have non-linear distortions are not stitch-able without first undistorting (correcting) them. This could be an artifact of the actual lens. The image plane (sensor) **is** a flat surface, but because you're trying to get an image with a very wide FOV, you must pack all that information on a flat surface, and it is the lens that creates these distortions.

Because these distortions do not depend on what the lens is taking a picture of, they can be undone. This is called radial distortion correction.

2.6 How do we compute Homographies?

Let's see.

The big picture – as long as you can find 4 points in one image, you can compute H – a 3×3 matrix that maps one image to the other.

A single correspondence means you have some point out there and you can map it somewhere else.

$$\underbrace{\begin{bmatrix} x_i' \\ y_i' \\ 1 \end{bmatrix}}_{\text{known}} \cong \underbrace{\begin{bmatrix} a & b & c \\ d & e & f \\ h & k & 1 \end{bmatrix}}_{\text{unknown : } H} \begin{bmatrix} x_i \\ y_i \\ \underbrace{1}_{\text{known}} \end{bmatrix}$$

Make some conversions:

$$x_i'(hx_i + ky_i + 1) = ax_i + by_i + c$$
$$y_i'(hx_i + ky_i + 1) = dx_i + ey_i + f$$

If 1 point correspondence gives us two equations, a 4-point correspondence gives us **8 equations and 8 unknowns.** That gives us our homography. That is our source and destination.

2.7 What 3D Information is Lost in Perspective Projection?

2D photographs can't tell us too much. There is this relation between the 3D coordinates in the world and the projection in the image. It is a 3D $\rightarrow$ 2D map. This means:

- Depth is lost

We will not be able to look at a photograph and figure out the 3D coordinates projected. Parallelism is not preserved. This means a lot of illusions can be made.

However, if we have some extra information, we can make some inferences:

- Known dimensions or lengths or angles for some objects
- Surfaces known to be planar
- Lines known to be parallel

Homographies can correctly transform planes, but anything that isn't on the plane will transform weirdly.

2.8 Vanishing Points, Vanishing Lines, Parallelism

We can project points at infinity. This is how vanishing points can be drawn and can be computed using a homography.

Vanishing points in 3D are points at infinity, but in a photo they have a concrete location.

Each direction on a plane has its unique vanishing point.

A set of parallel lines in 3D (same or opposite direction) will **always have the same vanishing point.**

The horizon is the vanishing line of the ground plane.

Important to note:

- Parallel lines have a vanishing point, but converse is not true

Just because two lines appear to converge doesn't mean they converge at a point from infinity.

3 Image Filtering

Just a transformation of an image. But a different kind of transformation. We've looked at geometric transformations first.

3.1 A Taxonomy of Image Transforms

We start with image f. We transform it with T to get g.

$$g = T[f]$$

- What is being transformed?
  - Geometric: coordinates only
    - Linear transforms
    - Non-linear transforms
  - Intensity: intensities / colors only
    - Point-wise: for every pixel independently of all others, we map intensities: $g(x, y) = T\left[f(x, y)\right]$. For example, image darkening or brightening

        * Local: transformations where a value at a pixel will depend on the values of the pixels in a neighborhood of the input image. We'll be focusing on linear transforms.
    - Domain: Image representation is being changed (no more pixel coordinates)
- How is it being transformed?

3.2 Linear Filters

A linear filter, for the time being, is a black box that takes the image as an input and outputs another image. Our task is to model mathematically what the black box does.

A linear filter is a transformation that has the following properties:

- Linear scaling of the intensities of the input = that of the output
- Literally the same as what you've learned in linear algebra

A filter transforms one signal into another. Filters are used to describe image formation (lens, blur, etc.), as well as to implement operations on images (edge detection, denoising)

A transformation T is linear if and only if it sastifies

$$T\left[a_1 f_1(x) + a_2 f_2(x)\right] = a_1 T\left[f_1(x)\right] + a_2 T\left[f_2(x)\right]$$

3.3 The Superposition Integral

Any transformation that can be expressed as a linear filter MUST have this property:

- The result g must be writable in the following way: as a weighted sum of the input at t multiplied by a function that calculates the weighted coefficient of it.

$$g(x) = \int_{-\infty}^{\infty} h(x,\ t)\, f(t) dt$$

Where $h(x,\ t)$ is the filter. For h, it asks: how much does this particular part of the input affect the output of that pixel?

Every value of the input will contribute to the output and the way it contributes to it is the function h. T was the black box. Now, T is the function h. Now, as I have that function, I can write out the integral.

h is a function of two variables. It depends on the coordinate of the 1D image, and the coordinate of the image on the input. We only have one constraint: the transformations are linear. Next, what are linear shift-invariant filter?

3.4 Linear Shift Invariant Input

Shifting the image gives me the same output. It doesn't matter where the image is, it is always invariant to the shift.

A transformation is shift-invariant $\iff$ shifted inptus produce identical but shifted outputs: $f^{'}(x) = f(x - x_0)$. The output would be: $g^{'}(x) = g(x - x_0)$. So, the shift-invariance property formally is:

$$T[f(x - x_0)] = g(x - x_0) \ \ \forall x_0$$

3.5 The Box Function

The box function is 0 except for a small interval, where otherwise it is 1. The function is:

$$\text{box}_\varepsilon(\tau) = \begin{cases} 1 & |\tau| \leq \frac{\varepsilon}{2} \\ 0 & \text{else} \end{cases}$$

And the scaled box function, where the area under the curve is always 1

$$\frac{\text{box}_\varepsilon(\tau)}{\varepsilon}$$

As of this time, there is no notation for the scaled box function. Good luck writing the entire expression out.

3.6 The Impulse Function

The impulse function (AKA Dirac's delta function) is:

$$\delta(\tau) = \lim_{\varepsilon \to 0} \frac{\text{box}_\varepsilon(\tau)}{\varepsilon}$$

The property of this function is:

$$\delta(\tau) = 0 \; \forall \tau \neq 0$$
$$\int_{-\infty}^{\infty} f(t)\delta(t)dt = f(0)$$

3.7 Impulse Response of a Linear Filter

Let's send δ through the filter. We'll get the response of the filter to an impulse. Applying the integral to $\delta(\tau)$:

$$g_1(x) = \int_{-\infty}^{\infty} h(x,\ \tau)\delta(\tau)d\tau = h(x,\ 0)$$

The filter's response to the impulse tells us a lot about the filter itself. For an image, we could get the shape of the filter.

The delta function is shift-invariant. If we have $\delta\left(\tau - \tau_0\right)$, applying the superposition integral to it:

$$g_2(x) = \int_{-\infty}^{\infty} h(x,\ \tau)\delta\left(\tau - \tau_0\right) d\tau = h\left(x,\ \tau_0\right)$$

g_1 and g_2 are shifted versions of each other. They will be shifted in the same way.

$$g_2(x) = g_1(x - \tau_0)$$

In other words:

$$h(x, \tau_0) = h(x - \tau_0, 0)$$

So you could treat g_2 as a shifted version of the response.

So rather, the impulse response function is not a 2D function but rather a 1D function. This is why we can discard the second parameter of the impulse response.

A linear filter with impulse response h is shift-invariant if and only if for all shifts $\tau_0 \in \mathbb{R}$, $h(x, \tau_0) = h(x - \tau_0, 0)$. This comes from the shift-invariance property.

So:

$$g(x) = \int_{-\infty}^{\infty} h(x - \tau) f(\tau) d\tau$$

And this happens to be the convolution operation. Linearity gives us the superposition integral, and shift invariance gives us convolution.

$$g = f * h = (x) \Rightarrow \int_{-\infty}^{\infty} h(x - \tau) f(\tau) d\tau$$

The convolution operator takes in a function and outputs a function (I wrote this like a JS arrow function).

What you call a filter and what you call a signal can be completely interchanged. You can do variable substitution and create an equivalent expression.

The convolution operation is commutative, associative, and has distributivity over addition.

So, the convolution filter is what can be done to do an LSI filter.

3.8 Convolution in 2D

$$g(x, y) = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} h(x-u, y-v) f(u,v) du dv$$

3.9 Convolving With An Impulse

Delta is 0 everywhere except where $\tau = x$. This is like the identity filter.

$$\begin{aligned} g(x) &= \int_{-\infty}^{\infty} \delta(x-\tau) f(\tau) d\tau \\ &= f(x) \end{aligned}$$

Shifted impulse? Suppose that δ was shifted to the right? Then:

$$\begin{aligned} g(x) &= \int_{-\infty}^{\infty} \delta\left(x-\tau-x_0\right) f(\tau) d\tau \\ &= f(x-x_0) \end{aligned}$$

The product of two shifted impulses? It's like overlaying, as shift invariant linear filters are shift invariant and sum invariant. You would get two copies of the image in different places.

An infinite sum of identically shifted impulses – the impulse train:

$$III_{\Delta}(x) = h(x) = \sum_{k=-\infty}^{\infty} \delta(x, -k\Delta)$$

If your images are Δ tall and wide, what you get is a tile that repeats the image, given k is greater than the side-length of the image.

3.10 Types of Filters

3.10.1 Box Filter

The box filter: $h(x) = \frac{1}{\varepsilon}\text{box}_\varepsilon(x)$. Forces integral to be under 1. What happens when I convolve a 1D function with the box filter?

$$g(x) = \int_{-\infty}^{\infty} f(u)\text{box}_{30}(x-u)du$$

This is the standard definition for convolution. Given that you already know that the function is non-zero in a small neighborhood of zero, we can change that to avoid redundant calculations.

$$\begin{aligned} g(x) &= \int_{x-15}^{x+15} f(u)\text{box}_{30}\left(\underset{\text{nonzero between } [x-15,\, x+15]}{x-u}\right)du \\ &= \frac{1}{30}\int_{x-15}^{x+15} f(u)du \end{aligned}$$

So, this box filter is really an averaging filter. We should expect the convolution to smooth out the signal. Convolving an image using the box filter blurs the image, as every result of the pixel is the result of averaging the neighborhood around that pixel.

3.10.2 The Pillbox Filter

Like the box filter, but it's the disk variant.

$$h(x,\, y) = \begin{cases} \frac{1}{\pi r^2} & \sqrt{x^2+y^2} \leq r \\ 0 & \text{otherwise} \end{cases}$$

$$\iint_{\mathbb{R}^2} h(x,\, y)dA = 1$$

Why? Most apertures are circular. If we want to model the blur that comes from an aperture, we can use this filter.

If I were to take my original image and pass it through the filter, vs. take the image, rotate it by 45 degrees, and pass it through the filter, I would get the same blur.

Visually, **you will not see a huge difference compared to the box filter.**

3.10.3 The Gaussian Filter

It averages pixels in a neighborhood, but it's a weighted average. Not all pixels will be weighted the same.

$$G_\sigma(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{x^2}{2\sigma^2}}$$

We will call σ the scale parameter. It controls the width of the gaussian. The higher the σ, the more spread out the gaussian will be.

In 2D, this is just the product of two 1D gaussians:

$$G_\sigma(x, y) = G_\sigma(x) \cdot G_\sigma(y)$$

It's a circular symmetric function. Convolving an image with a Gaussian will just give you a blurry version of the image, but for every pixel, if you move over 3σ pixels away, you might as well not count it as the Gaussian function is essentially zero.

Since I can control σ now if I increase σ it blurs the image even more.

So far, we've talked about smoothing. But there's more. We can compute image derivatives.

3.11 Computing Derivatives by Filtering

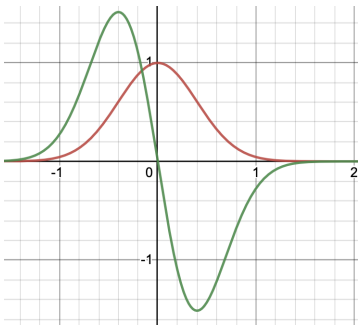
Because of the linearity of convolution, I can bring the derivative inside the integral.

$$\begin{aligned}\frac{d}{dx}(f * h)(x) &= \frac{d}{dx}\int_{-\infty}^{\infty} h(x-\tau)f(\tau)d\tau \\ &= \int_{-\infty}^{\infty} \frac{d}{dx}h(x-\tau)f(\tau)d\tau \\ &= \int_{-\infty}^{\infty} \left(\frac{d}{dx}h(x-\tau)\right)f(\tau)d\tau \\ &= \left(\frac{d}{dx}h\right) * f\end{aligned}$$

Let's do this for a Gaussian. Here, we've taken our function, smoothed it with our Gaussian, and then we take the derivative of the result. We could do the exact same thing by convolving f with the derivative of the gaussian.

$$\frac{d}{dx}(f * G_\sigma) = f * \left(\frac{d}{dx}G_\sigma\right)$$

How does the derivative of the Gaussian look like?



Now, how do we represent negative pixels? We're just going to color code it now and assume that pixels are greyscale. Our heatmap would be

- Red > 0
- White = 0
- Blue < 0

If I convolve the filter with the image, I will not expect the resultant image to be positive, as the convolution would also contain negative components. What would we expect

the result to be?

$$f * \frac{\partial}{\partial x} G_3(x, y)$$

What would I expect the output to look like? Well, if an image goes from dark → bright from left to right, the derivative will be positive. However, at a sharp edge:

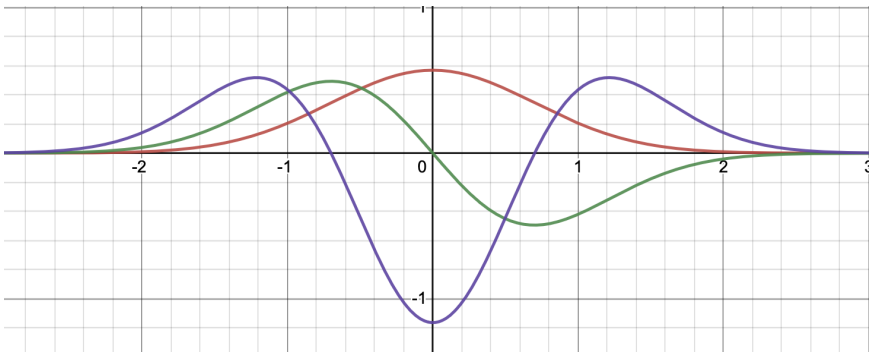
- An abrupt change from a bright left to a dark right would have a negative derivative.

Our resultant image would be zero nearly everywhere expect for an abrupt change in luminance somewhere in the image.

3.11.1 The Gaussian Second Derivative Function

$$\frac{d^n}{dx^n}(f * G_\sigma) = f * \left(\frac{d^n}{dx^n} G_\sigma\right)$$

The second derivative of the Gaussian function is:



The second derivative function is purple in the figure above. This function is symmetric.

The point is, we can compute second derivatives just as easily as computing the first derivative, just as easily as smoothing the image. Everything applies to either direction.

3.12 The DoG Filter

- Take our image
- Convolve it with the Gaussian σ
- Convolve it with a slightly larger σ
- Subtract the two images

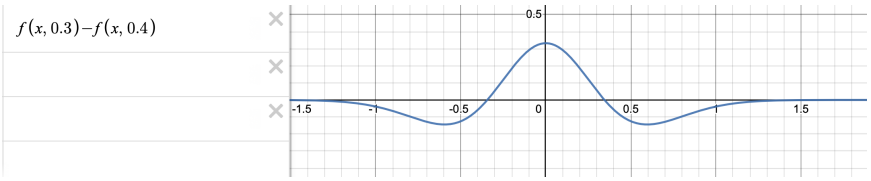
What would I get?

- Areas that are already smooth would just get 0
- Areas where smoothing by a slightly larger Gaussian would smooth more, well, I get another edge detector.

$$G_3 - G_4$$

Why is this called the DoG filter? Because it's called the difference of gaussians.

$$f * G_{\sigma_1} - f * G_{\sigma_2} = f * \left(G_{\sigma_1} - G_{\sigma_2}\right)$$



(This would be rotationally symmetric in 3D)

3.13 Sharpening

$$f + \left(f * \left(G_{\sigma_1} - G_{\sigma_2}\right)\right)$$

Gives you something that is slightly sharper. You could put a coefficient:

$$f + \underbrace{s}_{\text{adjustable sharpening parameter}} \left(f * \left(G_{\sigma_1} - G_{\sigma_2}\right)\right)$$

This would control the sharpness of the image. It **will** reduce the contrast of the images – if we push the edges to be stronger, the camera will have to squint (by squashing intensities above 255 to 255).

Now, how would I compact this statement?

$$f * \delta + \left(f * \left(G_{\sigma_1} - G_{\sigma_2}\right)\right) s$$
$$= f * \left(\delta + \left(G_{\sigma_1} - G_{\sigma_2}\right) s\right)$$

So, now we have an expression for the filter that sharpens the image.

Problem? Sharpening images causes a halo effect. Sharpening an image too much will cause it to no longer look natural.

3.14 Bounded Domain and Out of Bounds Filtering

Images are typically defined over a bounded domain. We need some convention to define the concept of convolution on a boundary.

There are two ways we can handle integrals when the image is not defined over the entire range.

3.14.1 0-Padding

Assume $f(x, y) = 0$ at the image border.

One way: for pixels out of bounds, assume that the image value is 0. What this really means, is that I've defined my image to be a function that is zero everywhere except in the actual image. If I were to convolve the image with an edge enhancement filter, I would get a very large brightness change in the borders, which will cause the edge-enhancement filters to respond.

3.14.2 Tiling / Wrap Around

The other approach would be to wrap around the pixel values:

Assume that $f(x, y) = f(x\%W, y\%H)$. Another way to think about this, is the image is not the image that you're seeing, but an infinitely tiled version of the image. If you're trying to convolve an image using this convention, you are convolving your filter with an image that extends infinitely in both dimensions and has this tiling corresponding to the tiled version of the image. It is a periodic function where the same image repeats – but this is the version of the image I'm applying my filter to.

If I have some change of brightness between edges, if I apply a filter, I will get strong responses around that area.

This particular way to think about an image is very common. We'll come back to this when we talk about Fourier transforms – which assumes that are images are tiled like this.

3.15 Edge-Degrading Behavior of Smoothing LSI Filters

How do we improve the quality of a noisy photo? What can I do to get a less noisy image back?

I can say, if I were to apply a blurring filter that will take all pixels in the neighborhood and average them together (discrete or gaussian), it will make an image that looks smoother, but you start losing details from the edges.

At any given position, when we're trying to compute the value of the image at this position, it takes the weighted sum of all pixels around it. If I'm at an edge, the same weighted sum will apply.

In other words, LSI filters will always act the same way no matter where I am in the image. Can we do better than that? Totally, but we can't maintain the shift invariance property. By definition, shift invariance is agnostic to the image content.

3.16 The Bilateral Filter

It can denoise and preserve sharpness.

This tells us how better we can get when we remove shift-invariance. It is a transformation of the intensity of the image, it is local, but it is a non-linear operation.

Now, the bilateral filter does not behave the same way in all parts of the image. It behaves almost like a gaussian filter when the image is smooth. However, when we are in a detailed area, the filter behaves different. Depending on where I am in the image, the weighting changes. But how is the weighting computed?

It's done by revising the expression for the output image. We write that as:

$$g(x,y) = \iint \text{Weight}(u,\ v,\ x,\ y) \cdot f(u,\ v) du dv$$

We will be assigning a different weight depending on where the pixels are in the image. All that matters is figuring out what the weight is.

This is a 4D function. It depends on the destination pixel, and which pixel in the source image we are looking at. This weight function will depend on two factors – a product of two factors:

- Like the original gaussian filter
- Something else

Our weight expression can be decomposed into:

$$\text{Weight}(u,\ v, x, y) = w_{\text{spatial}}(x,y,u,v) \cdot w_{\text{intensity}}(x,y,u,v)$$

With the following decomposition being:

- w_{spatial}: weight goes down the further the pixel is, like the gaussian filter
  - $= G_{\sigma_s}(||(x,y) - (u,\ v)||)$
  - If we only have this, then our filter would be LSI. But we don't.
- $w_{\text{intensity}}$: gives more weights to pixels with similar intensities.

- This is a very special term. Two pixels that are roughly similar in brightness, then it will contribute. If its brightness is very different, it won't contribute. Note that the higher the input passed into G_{σ_r}, the closer it will be to zero.
- $= G_{\sigma_r}(|f(x, y) - f(u, v)|)$

So, our final expression is:

$$G(x,y) = \iint_{\mathbb{R}^2} G_{\sigma_s}(||(x,y) - (u, v)||) \cdot G_{\sigma_r}(|f(x, y) - f(u,v)|) \cdot f(u,v) du dv$$

So, it's based on intensity and proximity.

Closer intensity and proximity ⇒ higher of the value of the filter at that point

This is a very expensive operation. For every pair of pixels, I have a weight. If I try to do this in a naïve way, with an $n \times n$ image, computing the weight function has an n^4 time complexity. This is not a super useful filter unless there is a way to compute it efficiently. It took a couple of years after this filter was introduced for people to come up with efficient solutions, and at least it exists.

What this allows us to do, is that once we've processed the image, our filter is edge preserves. It does not blur edges. Very important filter has some nice properties, but we lose the clean mathematical properties of convolution.

4 2D Digital Images

How can we represent images that are captured with a sensor? These are discrete, but it's important to understand how these discrete images we capture relate to the underlying continuous functions. When we want to display an image, we may need to think about it in a continuous fashion first. Suppose I have an image that is 10000 × 10000 pixels and display it with my phone, with a smaller screen pixel count. The expressions we've talked about 2 lectures ago introduce artifacts and issues.

4.1 Aliasing

If you downscale an image, you are changing the image that is being displayed while maintaining the fixed sampling (pixel grid). What happens is that you're spacing out a high-frequency sinusoidal wave which could then be represented by a completely different wave with just some matching points. That is the underlying problem of aliasing. This is caused by inherently sampling a signal.

4.2 In a Camera

Behind all cameras, there exists a sensor. If you zoom into the sensor, it looks like an array of individual sensing elements. Each of these, it corresponds to a pixel. Notice in front of the pixel, there's a color filter. The sensor records the intensity of the light after it goes through a red, blue, or green filter. It will return an array of scalar values – it will not send you color images. All you have is the brightness of the light that fell onto the pixel, passed through the red filter, and was passed through the sensor.

We'll use twice as many green filters as red and blue as the human eye is more sensitive to the color green.

4.2.1 The Micro-Lens Array

There is a lens in the front and redirects the light to the part of the sensor that can turn down input light down into a discrete value. What's really important, is that what gets recorded is **the integral of all the light that fell onto the surface.** There is some continuous representation of light that falls onto the sensor, but we just get the integral of it with, through the pixel's footprint. What the sensor gives you is the intensity, and they have to be turned into RGB values. There is processing involved that takes the scalar values passed in by the camera into RGB values. All cameras do this. No camera would record just the red, green, and blue.

The point here, is that all we end up with is a discrete array of digital numbers. You can think these numbers of being computed from a continuous function. Whatever

happened in a pixel footprint was integrated (or averaged), and what we end up getting is one value.

There is a connection between the continuous function and the discrete representation. The connection is that there is integration involved.

4.3 Digital Images Expressed as Convolution and Sampling

We start with a continuous function of brightness that is defined on the plane of the sensor. It is subdivided into a grid of footprints of the original pixels. For each of the footprints, we get the values of it. What's the value? The average of the rightness.

What did we capture? One measurement per pixel. Mathematically, they are represented as a set of δ functions. The height of the function is the value of the pixel. This is a more accurate way of picturing what the sensor captured. A pixel's footprint you see in a pixelated image is **not constant,** conceptually.

So, what goes on?

- I take my original image
- I convolve it with the box filter

Inside a single pixel, we are computing one value: the **average** of the intensity of the image within that footprint. What's the expression?

$$\text{pixel}(r,\ c) = \text{pixel center}$$
$$\Delta x,\ \Delta y = \text{width/height of a pixel}$$

$$f_{rc} = \int_{c\Delta x - \frac{\Delta x}{2}}^{c\Delta x + \frac{\Delta x}{2}} \int_{r\Delta y - \frac{\Delta y}{2}}^{r\Delta y + \frac{\Delta y}{2}} f(x,y)dxdy$$

So, this whole integration can be written as the convolution of the original image with a box filter whose dimension is the same as the footprint of a pixel. But the convolved image is not captured, as we don't have the continuous convolved image. All we have

is the values at the pixel center. It is not f_{rc} because the sensor only measures the averages at the pixel centers.

What we get is:

$$\widetilde{f}(x,\, y) = \left(f(x,y) \ast \left(\text{box}_{\Delta x}(x) \cdot \text{box}_{\Delta y}(y) \cdot \frac{1}{\Delta x \Delta y} \right) \right) \cdot \left(\text{III}_{\Delta x}(x) \cdot \text{III}_{\Delta y}(y) \right)$$

Note:

- $f(x,y) \ast \left(\text{box}_{\Delta x}(x) \cdot \text{box}_{\Delta y}(y) \cdot \frac{1}{\Delta x \Delta y} \right)$ is what the camera "sees"
- $\widetilde{f}(x,\, y)$ is what the camera captures to us. This is the sampled image, the collection of images that is being acquired. $\widetilde{f}$ is expressed as a function over a continuous domain but it is non-zero only at a discrete set of points.

It may look like a complicated expression, but it's not too bad. **But why do we care?**

In this convention, $\delta(t) = \begin{cases} 1 & t = 0 \\ 0 & \text{otherwise} \end{cases}$

4.4 The Image Resampling Problem

What we ultimately need to do is resampling. Somehow, I want to display an image on a screen with a completely different resolution. How do I up-sample (super sample) an image, on a screen with more pixels than I started with?

Sub-sampling is the opposite – how do I display a higher-res image on a screen with fewer pixels?

This is something that has to happen when I display an image on **any** device. When I zoom in or zoom out, you end up having to take that image and display it on some *finite* neighborhood of pixels. We need to be able to perform this image resampling problem. Want to print an image? This depends on the resolution of your printer. Want to rotate the image? Suddenly, your pixels, which was originally nice, is now rotated and it's no longer on a regular grid anymore. In assignment 1, your homography did not put your pixels right on the center.

So, how do you sample?

4.4.1 How to sample

- Start with a discrete representation
- Interpolate it: $\widetilde{f}(x,y) \rightarrow \widetilde{f}_{\text{interpolated}}(x,\ y) \rightarrow \widetilde{f}_{\text{resampled}}(x,\ y)$
  - Resampled uses another grid. $\widetilde{f}_{\text{interpolated}}$ is continuous.
  - Before you resample, you may need to re-filter the image to prevent aliasing artifacts. It's a very important step.

We're going to use filtering to do this. All the tools we've learned so far are going to be very helpful.

4.5 Function Interpolation

Given a discrete set of samples $f_k = f\ (x_k)$ of a function $f(x)$ at $x_1,\ x_2,\ \ldots,\ x_k$, construct a new function $g(x)$ that satisfies $g\ (x_k) = f\ (x_k)\ \forall i \in 1 \ldots k$, and we can evaluated for any x

There are two forms of sampling:

- Uniform sampling: the samples we have are spaced Δ apart (this is regular sample). We start with δ functions where the height (**multiplier** of δ) is the value of f at that location.
- Non-uniform interpolation: where the spaces between the samples are not the same. We're not covering this.

This interpolation should be shifting invariant.

Whether I shift first then interpolate, it would make no difference than if I did them the other way around.

4.6 The Expression of Function Interpolation

Given a uniformly sampled function f with period Δ, an infinite number of it so the samples extend to infinity.

We want to compute the interpolated function g. The expression is going to be as follows:

$$g(x) = \sum_{k=-\infty}^{\infty} f_k \cdot h(x - k\Delta)$$

Where f_k is f sampled at x_k, read the definition of it above in the previous section.

Where h is the interpolation filter (kernel). Treat it as a weighted combination of the samples.

Say $h(x)$ is my interpolation filter. How would our interpolation filter look like? It'll probably look like a bell curve, or a bell curve with humps below its first descent to 0. Normally, the higher the absolute value of what is passed into h, the closer to zero h returns (with some exceptions). However, regardless, $h(x)$ **must be zero at the sample locations except for the center one** so if we're sampling at an existing defined pixel, it won't mess up the image.

4.7 Derivation of the Interpolation Equation

Let $\widetilde{f}(x) = \sum_{k=-\infty}^{\infty} f_k \delta(x - k\Delta)$

We want a shift-invariant transformation $\widetilde{f}(x) \rightarrow g(x)$ such that $g(x) = f * h$. If we plug in this expression for f, we end up with:

$$\begin{aligned}
&\left(\sum_{k=-\infty}^{k} f_k \delta(x - \Delta k)\right) * h \\
&= \sum_{k=-\infty}^{\infty} f_k \cdot (\delta(x - k\Delta) * h) \\
&= \sum_{k=-\infty}^{\infty} f_k \cdot h(x - k\Delta)
\end{aligned}$$

g is written in this way as shift invariance requires the transformation for f to g to be a convolution, so it has to be written this way. Linear shift invariance implies convolution which implies the existence of this particular expression.

4.8 Conditions of the interpolation filter

There are THREE conditions:

- $h(0) = 1$
- $h(k) = 0 \; \forall k \in \mathbb{Z}, k \neq 0$
- (May not apply always) $h(x) = 0$ for $|x| \gg 0$

Legacy text

What can this h be? It cannot be arbitrary. The very important properties:

$$\begin{aligned}
h(0) &= 1 \\
h(k\Delta) &= 0 \; \forall k \neq 0
\end{aligned}$$

This ensures that when I evaluate g at some discrete sample (original pixel), only that original pixel contributes. It applies everywhere that has a pixel defined at it.

- Ideally, we want h to be a smooth function (continuous derivatives) everywhere. It is not a must.

- And we want local support: $h(x) = 0$ outside a small neighborhood of 0. Computationally, this makes a difference and ensures that $g(x, y)$ depends only on a few samples.

4.9 Types of Interpolation Filters

Okay. Let's do this.

4.9.1 Nearest Neighbors

Use the nearest sample. For any x, look at the closest sample and get the value of it. Well, our function becomes piecewise constant.

$$h(x) = \text{box}_\Delta(x) = \begin{cases} 1 & |x| \leq \frac{\Delta}{2} \\ 0 & \text{else} \end{cases}$$

It's the box filter. We're taking shifted copies of h.

Our image will just look like a mosaic. The visualization of a discrete image is simply its nearest neighbor interpolation.

4.9.2 Linear Interpolation

We compute the in-between values of two samples. Between two samples, the values blend. **Connect neighboring samples with a straight line.**

We can write an expression for this for any given point:

$$g(x) = \left(1 - \frac{x}{\Delta}\right) f_0 + \frac{x}{\Delta} f_1 \text{ for } 0 \leq x < \Delta$$

If we wanted to treat interpolation as a filtering operation, what filter gives us this linear interpolation result when we convolve with the original function? It all comes down to this expression:

$$g(x) = \sum_{k=-\infty}^{\infty} f_k h(x - k\Delta)$$

Where:

$$h(x) = \begin{cases} 1 - \frac{|x|}{\Delta} & |x| < \Delta \\ 0 & |x| \geq \Delta \end{cases}$$

Notice how a maximum of two sampled points can contribute to $g(x)$.

The actual filter looks like a triangle: $h(x) = \text{box}_\Delta(x) * \text{box}_\Delta(x) \cdot \frac{1}{\Delta}$. In 2D, this is going to be a pyramid:

$$g(x,y) = \widetilde{f}(x,y) * ((\text{box}_\Delta(x) * \text{box}_\Delta(x)) \cdot (\text{box}_\Delta(y) * \text{box}_\Delta(y)))$$

This is the most common form of 2D interpolation. It is very fast, results are okay.

4.9.3 Cubic Interpolation

How do I find a function that is a cubic polynomial and has the properties we need (1 at 0, 0 at all interval multiples of delta)?

$$h(x) = \begin{cases} \frac{3}{2}|s|^3 - \frac{5}{2}|s|^2 + 1 & 0 \leq |s| < 1 \\ -\frac{1}{2}|s|^3 + \frac{5}{2}|s|^2 - 4|s| + 2 & 1 \leq |s| < 2 \\ 0 & |s| > 2 \end{cases}$$

With $s = \frac{x}{\Delta}$

This looks *like* the DoG filter but analytically they're not the same. Why is there a dip? Please don't ask. It can cause "ringing" due to the negative dips.

Why isn't there a quadratic? We don't have much control over a quadratic function. You can move the vertex, but you can't move anything else. You can use a piecewise function but expect discontinuities at a boundary.

4.10 Could you compare?

Bilinear? Bicubic? Bicubic is smoother. The larger region of support, you are getting smoother results.

5 Image Morphing

- I am going to become Switzerland
- That one effect in kaput
- Except deep learning tends to outperform all of this, so most of this is useless anyways.

We have two different images, and we would like to perform a warp between the two. Morphing involves two steps:

1. Pre-warp the two images
    a. Into a common space, so at the half-way point, they look structurally the same. The cheek size, the face composition is roughly in the same place
    b. This prevents ghosting
2. Cross-dissolve their colors
    a. Simple cross-dissolve

We note

- An image pre-wrap is a re-positioning of all pixels in an image to avoid the double-image effects much as possible.
- The field morphing algorithm is a pre-warping algorithm that offers intuitive warp control.

And the process goes like this when you see it:

- Pre-warp the first image

- Cross dissolve
- Then undo the pre-warp on the second image

5.1 Cross-Dissolving two images

A weighted combination of two images, pixel by pixel

$$morph(t) = (1 - t)warp_{1(t)} + t\ warp_2(t)$$

5.2 Warping Images by Backward Mapping

The coordinate map: two functions $R(r,\ c)$, $C(r,\ c)$ that map each pixel $(r,\ c)$ in a destination image to its (potential fractional) location in the source image. Coordinate maps are essential for specifying the image warp. We are backward mapping to prevent holes in the picture.

Warp: $(r,\ c) \longrightarrow$ source $(R(r,c),\ C(r,c))$

5.3 The Backward Mapping Algorithm With Footprints

Given a map $R,\ C$

- For r in `rmin` to `rmax`
  - For c in `cmin` to `cmax`
    * $r' = R(r,c)$
    * $c' = C(r,c)$
    * Sample source image at $\left(r',\ c'\right)$
    * Copy that sample to the destination pixel

Some areas in the warped image are going to be smaller than other areas, and some areas are going to have more information. What we often do, is use super-sampling. So, let's do this again:

- For r in `rmin` to `rmax`
  - For c in `cmin` to `cmax`
    * $r' = R(r, c)$
    * $c' = C(r, c)$
    * Define $K \times K$ grid of samples within footprint of destination pixel $(r,\ c)$
    * Use grid to super-sample the source image by interpolation
    * Copy the average of these samples to the destination pixel $(r,\ c)$

5.4 The Beier-Neely field-warping Algorithm

We want to map the two coordinate systems: the original grid to the warped image. It is an algorithm for specifying the coordinate maps of a warp interactively.

Its input is a set of corresponding line segments drawn on the source images. Yes, there is some work you'll have to do here. It's not all automatic.

For every t, the endpoints of corresponding segments are linearly blended to define their position in $\text{warp}_i(t)$

$$\vec{P}(t) = (1 - t)\vec{P} + t\vec{P}'$$
$$\vec{Q}(t) = (1 - t)\vec{Q} + t\vec{Q}'$$

For $warp_1(t)$ and $warp_2(t)$, the segments should be identical.

The field warping algorithm computes $R_i(r,\ c)$ and $C_i(r,\ c)$ for $i = 1, 2$ from the line segment positions at t

For a single segment:

1. Destination pixel $(r,\ c)$ assigned coordinates $(u,\ v)$ relative to the segment (that is, $\vec{P}(t)$ to $\vec{Q}(t)$ and we make an orthogonal line).
2. These $(u,\ v)$ coordinates and line segments are converted to $\left(r^{'},\ c^{'}\right)$ in the source image

The destination $X = (r, c)$ expressed in a local coordinate system defined by $\vec{P}(t)\vec{Q}(t)$

$$\begin{aligned} & X \\ & = \underset{\text{origin}}{\vec{P}(t)} + \underset{\text{The } u \text{ axis is along this segment}}{u\left(\vec{Q}(T) - \vec{P}(t)\right)} \\ & + \underset{v\text{–axis defined by this unit length vector}}{v\frac{\text{Perpendicular}\left(Q(t) - P(t)\right)}{||Q(t) - P(t)||}} \end{aligned}$$

Which, then we can show that $u,\ v$ can be given by these simple formulas, where $\vec{X} = (r,\ c)$ in our destination image:

$$u = \frac{\left(\vec{X} - \vec{P}(t)\right)\left(\vec{Q}(t) - \vec{P}(t)\right)}{\left|\left|\vec{Q}(t) - \vec{P}(t)\right|\right|^2}$$
$$v = \frac{\left(\vec{X} - \vec{P}(t)\right)\left(\text{Perp}\left(\vec{Q}(t) - \vec{P}(t)\right)\right)}{\left|\left|\vec{Q}(t) - \vec{P}(t)\right|\right|}$$

Change of basis detected

5.5 Dealing with Multiple Lines

Apply the single-segment algorithm to each segment separately to obtain N coordinates $\left(r_1^{'},\ c_1^{'}\right) \cdots \left(r_N^{'},\ c_N^{'}\right)$

Compute $r^{'}, c'$ as:

$$\left(r^{'}, c^{'}\right) = \sum_{n=1}^{N} w_n \cdot \left(r_n^{'},\ c_n^{'}\right)$$

Where $w_n = \frac{\widetilde{w}_n}{\sum_{n=1}^{N} \widetilde{w}_n}$ with $\widetilde{w}_n = \left(\frac{(\text{length of segment}^{??})}{a + \text{dist of } (r,c) \text{from segment}} \right)^{??}$

a: controls influence of segment for pixels near it. Just view the assignment page and view the slides. I'm not repeating the code again. You can just translate pseudocode into actual code.

6 Fourier Transforms

Involve representing images in a completely different way. Down with functions of x, y, we're using functions of spatial frequencies.

The idea is that any function can be expressed as an (infinite) weighted sum of sinusoids. We have to look at sinusoids of many different frequencies. We'll see the function as a collection of weights where we have one weight per frequency. Depending on how many frequencies you choose to use, you get a representation that gets increasingly more accurate. Get out the high frequencies, and the function isn't exactly what you see.

When you Fourier-transform an image, you'll get two images: magnitude and phase. The center is a constant, and moving away gives you images of sinusoids with increasing frequencies.

If we remove high frequency components, you'll start blurring the image.

6.1 Normalizing a Sinusoid

Why does $\sin(2\pi x)$ have a period of 1? The period of $\sin(x)$ is 2π, and by instead passing $\sin(2\pi x)$ into the argument, we are compressing it by a factor of $2\pi x$ which reduces the period down to 1.

$\sin\left(\frac{2\pi x}{c}\right)$ stretches the sinusoid to have a period of c.

And then:

$$\sin\left(\omega\frac{2\pi x}{c}\right)$$

Here, we get ω phases through a distance of c, and the phase will always reset when we hit c (given ω is an integer).

6.2 The 2D Fourier Domain

We'll think of an image as a function of frequencies. We have two spatial frequencies ωx and ωy.

- Frequency $(\omega_x, \omega_y) = (0,\ 0)$ represents a constant-intensity image (the "DC component")
- The image corresponding to $(5,\ 0)$, this is an image that is a sinusoid in the x-direction and constant in the y-direction. It has 5 periods from the beginning from the start to the end of the image (again, assume we're dealing with cosine waves).

Suppose on the Fourier domain, we have $(\omega_x,\ \omega_y)$

$$I(x,y) = \cos\left(\omega_x\left(\frac{2\pi}{c} + \phi_x\right)\right) + j\sin\left(\omega_x\left(\frac{2\pi}{c} + \phi_x\right)\right)$$

This gives us two images: the real-valued image and the complex-valued image.

On $(-\omega_x,\ 0)$, using the properties of odd and even functions:

$$I(x,\ y) = \cos\left(\omega_x\left(\frac{2\pi}{c} + \phi_x\right)\right) - j\sin\left(\omega_x\left(\frac{2\pi}{c} + \phi_x\right)\right)$$

So, on the Fourier domain, if we have $(-\omega_x,\ 0)$ and $(\omega_x,\ 0)$, then our image ends up being:

$$I(x,\ y) = \cos\left(\omega_x\left(\frac{2\pi}{c}x + \phi_x\right)\right)$$

So, if we just want a real-valued image we make the fourier domain symmetric on the y-axis.

Along the y-axis, we're just dealing with, assuming we have $(0,\ \omega_y)$, $(0,\ -\omega_y)$:

$$I(x,y) = \cos\left(\omega_y\left(\frac{2\pi}{R}y + \phi_y\right)\right)$$

And for $\pm\ (\omega_x, \omega_y)$:

$$I(x,\ y) = \cos\left(\left(\omega_x\left(\frac{2\pi}{c}x + \phi_x\right)\right) + \omega_y\left(\frac{2\pi}{R}y + \phi_y\right)\right)$$

6.3 Fourier Domain Image

It represents a weighted sum of each component. The image in the fourier domain is the magnitude of the fourier transform.

6.4 What is a Fourier Transform?

It takes an image and decomposes it into a sum of sinusoids. The fourier representation shows the magnitude and the phase of every component. The inverse fourier transform reconstructs the image from its component cosines.

6.5 The Convolution Theorem

A convolution in the spatial domain is equivalent to pointwise multiplication in the fourier domain.

Convolving an image with a filter is the same as multiplying the fourier transform of that image with the fourier transform of that filter.

Also, the opposite holds true: Multiplication in the spatial domain is equivalent to convolution in the fourier domain.

$$f * h = FT^{-1}\left(FT(f) \cdot FT(h)\right)$$

6.6 Low-Pass Filtering

We can see the process of blurring as low-pass filtering. The frequencies around the center are low frequencies. Low-pass filtering preserves the low frequencies and attenuates (reduces) the high frequencies.

A high pass filter does the opposite.

This makes the image average 0, as the DC component ("constant image") is removed.

Band pass: Cut out something from the middle but keep in the low and high frequencies.

You can recreate the albert Einstein image by:

- Applying a high pass filter on the image you want visible when looking at it closely
- Applying a low pass filter on the image you want visible when looking at it far
- Add them

6.7 Fourier-Domain Representation of Basic Signals

- The fourier transform of a Gaussian is a Gaussian.
- A Gaussian in the spatial domain that is narrow becomes wide in the fourier domain, and vice versa.

The larger the σ in the spatial domain, the smaller it is in the fourier domain as less frequencies are preserved. Here:

In the spatial domain, if σ is used then in the fourier domain, it is $\frac{C}{\sigma}$ in ω_x and $\frac{R}{\sigma}$ in ω_y, where C and R are the width and height of the image, respectively.

6.7.1 Transforming Deltas

So, what is the fourier transform of a δ function? **Since convolving an image with that function does not change it, in the fourier domain, it is constant.** $FT(\delta) = 1$ everywhere

6.7.2 Transforming Impulse Trains

The FT of an impulse train is also an impulse train. The spacing will be different:

In the **spatial domain,** if the spacing is Δ

In the fourier domain, we have Δ impulses across the width of the image. In other words, the spacing is $\frac{C}{\Delta}$ along the x-axis, $\frac{R}{\Delta}$ along the y-axis.

So, the inverse relation applies here.

6.7.3 Transforming the box function

We get the `sinc` function. This looks very similar to some of the interpolation filters we have.

$$box_1(x) \xrightarrow{FT} \frac{\sin(\pi x)}{\pi x}$$

The other way around applies:

$$\frac{\sin(\pi x)}{\pi x} \xrightarrow{FT} box_1(x)$$

Notice how applying the box filter in the fourier domain (applying `sinc` in the spatial domain) causes ringing, or visible waves in the image. That is due to a sharp cut-off from the box filter, which causes ringing.

6.8 Image Sampling in the Fourier Domain

Consider what happens when we have a continuous image represented in the fourier domain. Well, we don't have access to that. We have access to a sampled version of that image.

A sampled version of an image is nothing other than the product of a continuous image and an impulse train. Well, now we know what the fourier transform of an impulse train is.

$$f \cdot III_{10}(x) \cdot III_{10}(y) = FT^{-1}\left(FT(f) * III_{\frac{C}{10}}(x) * III_{\frac{R}{10}}(y)\right)$$

So, you only have access to multiple shifted copies of the fourier transform, not the original fourier transform. If the spacings become smaller and I sample the image more densely, the copies are further apart (good time to check the slides). If I sample eery pixel, then I eliminate nearly all copies. The spacing between each copy is $\frac{C}{\Delta}$ (for 1 dimension).

So, my task would be to isolate the copies, perhaps by multiplying our new image in the fourier domain by the box filter of the appropriate width (you see the relation with `sinc` in the spatial domain?).

If I can do this, and the signal is bounded, I can reconstruct the original continuous function perfectly. I start with a discrete representation of my signal, end up with a continuous one – it is a very special interpolation. We're getting an exact reconstruction of the continuous signal.

In a nutshell:

Given a sampled image $\widetilde{f}$: Compute $FT^{-1}\left(FT\left(\widetilde{f}\right) \cdot box_{\frac{C}{2},\frac{R}{2}}\right)$.

So, how does this algorithm relate to interpolation? It's the same as convolving $\widetilde{f}$ with a `sinc`.

Reconstruction of a continuous signal $\Leftrightarrow$ Convolution with a `sinc`.

This is an interpolation filter. `Sinc` itself is an interpolation filter. In fact, this is the optimal interpolation filter – it gives us a perfect reconstruction of the signal given to

us (but nothing between the signal) mainly because it's 1 at 0, 0 at the regularly spaced positions (multiples of Δ). But why not use `sinc`? It extends to infinity, so we can't evaluate it properly as it lacks local support ($h(x) = 0$ for a neighborhood away from 0).

6.9 Aliasing and Nyquist Sampling

When does this type of reconstruction fail? The problem is that when you sample a high-frequency image very sparsely (large spacing between the samples), and when you get the reconstruction, you get a lot of artifacts that have nothing to do with the high frequencies in your existing image.

Aliasing occurs when different copies of the FT overlap. If the spacing is dense enough, the copies of the FT are far apart. But as the spacing between the samples become larger, the copies become closer (remember *III* in spatial and fourier). When they overlap, it's game over. Once you have this overlap, you are not getting contribution from the only copy, you are getting contributions from the shifted copies. The sum of these copies will look nothing like the original, so the box filter can't chop them off. The problems become more severe the further the samples are apart.

So, when you interpolate your image, the high frequencies are aliased by lower frequencies.

We can work out the condition that allows perfect reconstruction. Suppose the spacing between the copies is $\frac{C}{\Delta}$. Suppose that the frequency in our signal zeros out at ω_{max} on the right side and $-\omega_{\text{max}}$ on the left in the fourier transform.

The Nyquist condition is a necessary and sufficient condition for avoiding overlaps:

$$\begin{aligned}
\omega_{\text{max}} &< \frac{c}{\Delta} - \omega_{\text{max}} \\
&\iff 2\omega_{\text{max}} < \frac{c}{\Delta} \\
&\iff \Delta < \frac{C}{2\omega_{\text{max}}} = \frac{1}{2} \cdot \frac{C}{\omega_{\text{max}}} = \frac{1}{2} \cdot \text{lowest discernable period}
\end{aligned}$$

The spacing between samples must be at most half the period of the highest frequency sine and cosine in f.

The maximum frequency of a sampled image on an axis is how many pixels are there on that axis divided by 2.

6.10 Avoiding Aliasing

My image better not contain frequencies higher than twice the spacing of the pixels. So, I have to pre-filter the image through a low pass filter to remove the high frequencies. It may be blurrier, but that's the price I have to pay. That's pre-filtering: **before resampling an image, apply a low pass filter to remove all frequencies above Nyquist.**

Super-sampling: in the case where our sampling is in our control (we have a camera): sample our image densely enough so we don't get aliasing.

For example: If we have a 5000×4000-pixel photo, our maximum image frequency is $\omega_x \leq 2500$ and $\omega_y \leq 2000$, to put it in a 2500×2000 pixel photo, the maximum image frequency may only be $\omega_x = 1250$ and $\omega_y \leq 1000$. So, we must pre-filter out all frequencies above $(1250, 1000)$.

6.11 Anti-Aliasing Filters in Cameras

Cameras have anti-aliasing filters to remove high frequencies above the pixel resolution of the camera.

6.12 Super-Sampling and Averaging

To put something in a destination pixel, if a destination pixel would need to sample multiple source pixels, we can interpolate the many source pixels.

6.13 The Fourier Series

Consider a bounded, continuous, possibly complex periodic signal with $s(x) = s(x + 2\pi)$. When that happens, we can express it as a sum of a discrete collection of a discrete collection of sines and cosines.

Any such function can be expressed as the following series:

$$s(x) = \frac{a_0}{2} + a_1 \cos(x) + b_1 \sin(x) + a_2 \cos(2x) + \\ b_2 \sin(2x) + \cdots + a_n \cos(mx) + b_m \sin(mx) + \cdots$$

This is how we end up with a fourier transform. We start with a general continuous periodic signal, and we can express it as an infinite sum of sines and cosines of different frequencies.

$a_n \cos(nx) + b_n \sin(nx)$: the contributions of the nth harmonic

The fourier transform represents these coefficients: $\vec{a}$ and $\vec{b}$. How do we compute these coefficients?

The coefficients $a_i,\ b_0$ are called the Fourier series coefficients of $s(x)$. They are easy to express because $\cos(nx), \sin(nx)$ are orthogonal over $[0, 2\pi]$ and $[-\pi,\ \pi]$.

They are orthogonal as:

$$\int_{-\pi}^{\pi} \sin(nx) \sin(mx) dx = \begin{cases} 0 & \text{if } n \neq m \\ \pi & \text{if } n = m > 0 \end{cases}$$

So, for $n \neq m$ there will always be as many positive segments as negative segments so integral cancels them out.

So, how do we figure out the expression for coefficient b_n? Suppose that I take $s(x)$ and multiply it with $\sin(nx)$:

$$\int_{-\pi}^{\pi} s(x)\sin(nx)dx$$
$$= \int_{-\pi}^{\pi} \frac{a_0}{2}\sin(nx)dx + \int_{-\pi}^{\pi} a_1\cos(x)\sin(nx)dx$$
$$+ \int_{-\pi}^{\pi} b_1\sin(x)\sin(nx)dx + \cdots +$$
$$\int_{-\pi}^{\pi} b_n\sin(nx)\sin(mx)dx$$

When we look at this sum, what happens? Well, with an integral from $-\pi$ to π, all terms cancel except for $b_n \sin(nx)$.

So, we are really left with

$$\int_{-\pi}^{\pi} s(x)\sin(nx)dx = \pi b_n$$

So:

$$b_n = \frac{1}{\pi}\int_{-\pi}^{\pi} \sin(nx)s(x)dx$$

The inverse fourier transform is evaluating $s(x)$ given the fourier coefficients, while the forward forier transform is just figuring out the coefficients.

$$a_n = \frac{1}{\pi}\int_{-\pi}^{\pi} \cos(nx)s(x)dx$$

This is really all there is for the fourier transform. This is how you get the coefficients and reconstruct the function.

6.14 Encoding It

But there is a more mathematically concise way to write everything. Instead of figuring out that we have all the a and b coefficients, let's represent them with a single, complex-valued number. All it does is that it stores the two coefficients ($j = \sqrt{-1}$).

$$F_n = a_n + jb_n$$

Instead of writing this coefficient in this form, we're going to write it in the Euler notation for complex numbers:

$$e^{j\theta} = \cos\theta + j\sin(\theta)$$

Why do we do this? We now get a much more compact way to write $s(x)$:

$$s(x) = \sum_{n=-\infty}^{\infty} F_n e^{jnx}$$

So this is a very compact way to write it. We just have a sum of complex exponentials which we can use to reconstruct the original signal.

Why do we start from $-\infty$? Because our frequencies can be negative infinity.

The function s can still be complex-valued and this would work. Oh, and:

$$F_n = \int_{-\pi}^{\pi} s(x)e^{-jnx}dx$$

Conversely, to compute one of these fourier coefficients is to multiply the signal with the complex exponential, which says that I'm just going to multiply the s with the sine and the cosine.

The fourier series defines the inverse fourier transform, given coefficients F_n given the coefficients, which can reconstruct the signal s. This is all done for functions which

are periodic between $-\pi$ and π, or from $0 \rightarrow 2\pi$. If I want to look at functions that are periodic over another interval, I simply need to convert them.

If our signals are periodic over $[0,\ T]$:

$$s(x) = \sum_{n=-\infty}^{\infty} F_n e^{\frac{jn}{T}}$$

If your function isn't periodic, just make the period ∞. Then, you don't have a discrete set of frequencies – the frequencies itself become continuous and it becomes:

$$\int_{-\infty}^{\infty} F_w e^{jwx} dw$$

6.15 Magnitude and Phase of a Fourier Coefficient

We can write the fourier coefficients in this way:

$$F_w = a_w + jb_w$$

But it's more convenient to think of this as a 2D vector, whose x is a and y is b. Another way we can write it is by its length and its angle, having this representation:

$$F_w = \rho_w e^{j\phi_w}$$

Where ρ_w is the magnitude and ϕ_w is the phase.

If I want to change the phase of an image (shift it by ϕ), I just multiply all the coefficients by $e^{j\phi}$:

$$e^{j\phi} F_w = \rho_w e^{j(\phi+\phi_w)}$$

6.16 The 2D Fourier Transformation

$$F_{w_x,w_y} = \int_0^C \int_0^R f(x,y) e^{-j\left(w_x \frac{x}{C} + w_y \frac{y}{R}\right)} dxdy$$

$$f(x,y) = \sum_{w_x=-\infty}^{\infty} \sum_{w_y=-\infty}^{\infty} F_{w_x,\, w_y} e^{j\left(w_x \frac{x}{C} + w_y \frac{y}{R}\right)}$$

If you want to derive this, such as the Fourier transform of a delta train and a gaussian, of course you will have to use these expressions. But this is the big picture.

7 Color

The concept of color does not have a basis in physics. *What color is this* is not something that is dictated by any physical property. It is really an artifact of our own color perception, how our brain responds to incident light.

7.1 Human-Color Perception

Color is a perception artifact and is not related to any physical property. How do we end up having a color perception in the first place?

The human perception system is sensitive to a small range of EM wavelengths, from 400 to 700 nanometers. If you look at these wavelengths, you will see colors.

The reason why we have these color perceptions is as our retinas are sensitive to three portions of the EM spectrum. There are:

- Blue cones (S)
- Green cones (M)
- Red cones (L)

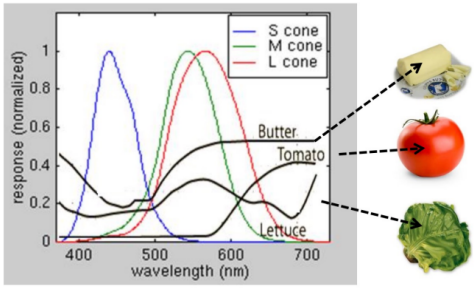
There's an overlap between these cells and do not respond to a single wavelength. In the end, it doesn't matter the wavelengths the lights have, it is how the cells in our retinas respond. It is those responses that dictate our perception of color.

7.2 Spectral Power Distribution

A function (or rather a distribution) of wavelengths that outputs a response. Red objects have high responses in wavelengths 600-700nm.

The point here is that light that reaches our eyes is not just one wavelength. It spans many wavelengths. All that matters is the relative strength of these wavelengths.

The point is this is something that can be measured.



7.3 Grassman's Law

The perception of color does not have a 1:1 correspondence to SPDs. What the S, M, L (color perception/ "RGB" as seen in a camera, cell) responses gives us cannot reconstruct the SPD.

Metamers are colors that have very different SPDs but have identical color perceptions.

7.3.1 Tristimulus Color Theory

Grassman's law states that a source color can be matched by a linear combination of three independent primaries. Combining red, green, and blue.

Summing two colors is the same as summing their RGB components.

7.3.2 Quantifying Color

The fact that this occur makes it possible to treat color in a more quantitive way.

Initially, this is done with a human observer, where experimenters are trying to figure out the RGB of a specific color and the human observer tries to tell them if the colors generated by RGB is the same color they are exposed to otherwise.

The idea is that different observers would have similar responses, and by averaging out their responses we would get the standard perception.

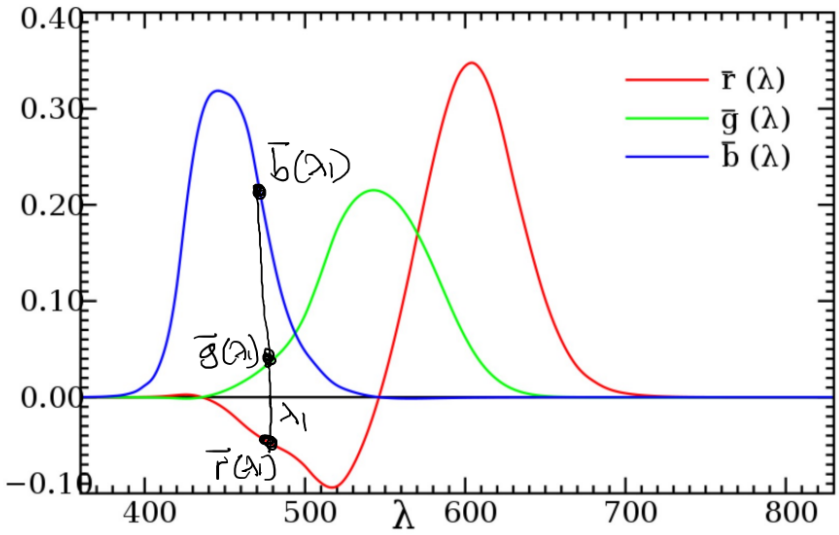
7.4 What is Color

For any given wavelength, you can get three numbers:

$$\lambda_1 = \begin{bmatrix} \overline{r}(\lambda_1) \\ \overline{g}(\lambda_1) \\ \overline{b}(\lambda_1) \end{bmatrix}$$

By scanning the entire range of visible light, you are getting the corresponding response of the observer. The color sensation produced by any given wavelength λ can give a 3D vector.

If you plot this out…



We have a mathematical description of the response from an observer to colors of specific wavelengths.

7.5 The CIE XYZ Space

A new convention. XYZ ⟺ RGB, and we don't want negative values. Properties:

- White is $X, Y, Z = \frac{1}{3}$. White is not a wavelength, it's just a color perception.

And we can convert RGB to XYZ is by using a linear transformation:

$$\begin{bmatrix} x(\lambda) \\ y(\lambda) \\ z(\lambda) \end{bmatrix} = \begin{bmatrix} 0.48 & 0.31 & 0.20 \\ 0.17 & 0.81 & 0.01 \\ 0 & 0.01 & 0.98 \end{bmatrix} \begin{bmatrix} r(\lambda) \\ g(\lambda) \\ b(\lambda) \end{bmatrix}$$

When we talk about color perception, we're talking about the perception from the standard observer.

Now, we have a way to predict, for any given SPD, what a standard observer would perceive. Note that a single wavelength can't give us white, as the eye can perceive more than what a single wavelength can give us. Perceptions give us more than what a single wavelength can do.

7.6 SPD to XYZ

Given an SPD $I(\lambda)$, how can we predict the color perception?

$$X = \int_{380}^{780} I(\lambda)\bar{x}(\lambda)d\lambda$$
$$Y = \int_{380}^{780} I(\lambda)\bar{y}(\lambda)d\lambda$$
$$Z = \int_{380}^{780} I(\lambda)\bar{z}(\lambda)d\lambda$$

Assume $\bar{x} \ldots \bar{z}$ are given because they already are, from the experiments I stated in the experiments above.

If two SPDs have the same CIE XYZ values, then the perceived color is the same. The standard observers can't distinguish these colors.

7.6.1 The CIE XYZ 3D Plot

The entire space of colors we can perceive is a 3D space. But if we only have a single wavelength, we have a perception over a 1D curve in the 3D space.

7.7 Color Image Sensors

The way that cameras capture colors is by having each pixel having a certain color filter. One for R, one for G, and one for B. This forms a color filter array (CFA) called a Bayer pattern.

What the sensor measures is the light coming in from a certain range of wavelengths: some pixels measure red, green, or blue, but not two or more at the same time. Why are there more green sensors? The human visual system is more sensitive to green, and assigning more to green because it helps us perceive sharpness.

Sensors only give us intensities: some refer to red, green, and blue. To get a color image where we need values for all colors, what can we do? Interpolate the samples. If we have samples and we would like to find something in between, interpolation is our choice. Modern sensors don't do that as interpolation doesn't care about the content, but in reality, if you have some object, and at the boundaries of the object there is a different color, you do not want interpolation to let colors bleed across edges. So, you'll need a smarter way to do interpolation.

7.8 Device Color Spaces

They use a ton of color filters. And yes, filters can be different. If you have two cameras and you have them take a picture of the same thing, their RGB values will be different. Not all cameras perceive light the same way. The green sensor on an iPhone may be different than the green sensor on another device, so it detects things differently.

Your camera sensor RGB filter is sensitive to different regions of the incoming SPD. This can be incredibly problematic.

When you display an image on a screen, something has to be assumed about the color. Some images are captured and represented in a specific color space for a sensor. If you take a raw image and display it without pre-processing, the cameras would be quite off.

And to pre-process it, you would have to convert it to a known color-space: CIE XYZ or sRGB. The SPDs of these are well known.

7.9 Device-Independent Color Mapping

Transform from device-specific color space to CIE XYZ. This is a standard.

That process is just multiplication of colors with a 3x3 matrix.

$$\underbrace{\begin{bmatrix} - & - & - \\ - & 3\times 3 & - \\ - & \text{provided by camera} & - \end{bmatrix}}_{} \underbrace{\begin{bmatrix} R(\lambda) \\ G(\lambda) \\ B(\lambda) \end{bmatrix}}_{\text{device specific}} = \underbrace{\begin{bmatrix} \bar{x}(\lambda) \\ \bar{y}(\lambda) \\ \bar{z}(\lambda) \end{bmatrix}}_{\text{device-independent}}$$

It moves from a device-dependent color space to an observer color space.

The CIE XYZ space covers everything, but individual cameras and displays cannot cover everything, so screens only cover a subset of XYZ spaces. That's why when a screen is said to have a wide gamut, it covers a wider subspace.

7.10 Other Color Spaces

CIE XYZ is the most foundational color space. Most other things are derived from them.

7.11 Color Constancy

In a real scene, an object's SPD is a combination of its reflectance properties and scene illumination. If you're in a room of just blue light, your tomato will look purple.

But regardless, the observe perceives these lights to be the same. This is possible as our brain does more than determines sensitivity. It takes in account the environment and factors that out. Key word: context

To understand color constancy, we have to consider SPDs of different illuminants.

One way to standardize the illumination condition (or what the perception of illumination does) is to use the concept of color temperature. That's why lightbulbs have color

temperatures. This number says something about our perception and does not say anything about the physical processes that generate that light.

The light source gives us some SPD. We don't have control over that, but the designers of the lightbulb do have some control over that.

We can pass that through the standard observer, and that will give us a specific X, Y, Z value which is how the standard observer would perceive the color of that lightbulb. This is a mathematical function, and we can do it with no problem regardless of the complexity of the SPD.

7.12 Determining Color Temperature

So, what is color temperature? It is a convenient way to produce another SPD.

The light bulb is perceived to have the same color as the radiation of a black body as temperature T. Guess what, we already know the SPD of all black bodies at any temperature T, we have a formula for that.

- Start with a light source, get its SPD, then its XYZ.
- Start with a black body, adjust its T such that the XYZ emitted from it matches the light source.

There is no guarantee you can find a matching T value for this. In this case, just find the closest match. A green-tinted lightbulb may have no correspondence. Regardless, the lightbulbs we use try to stay as close to neutral colors.

Color temperature: Low – red, high – blue. That's why the sky is blue, and the sunset is yellow to orange. And more noticeably, no green.

7.13 White Point Compensation / Illuminant Mapping

A white point is CIE XYZ or white reference $XYZ = \frac{1}{3}$

But our goal is to render color under very specific illumination. We need to specify, what light source is used to light up the environment. What is the ambient lighting

that the human brain perceives, so that when it tries to correct the color, it does the right thing?

This process tries to color-correct an image, under the assumption that it was lit from light source A, and make it look like it was lit from light source B. Obviously, this becomes hard when an image has been lit up by two very different light sources.

We want to color-correct the photo. This type of mapping is an illuminant-to-illuminant mapping. How do we do it? It's just another transformation.

The idea is, that we start with a color from illuminant 1 $(LMS)_1$ and illuminant 2 $(LMS)_2$, and we find a way to map white from illuminant 1 to white in illuminant 2.

- $(LMS)_1$ under an illuminant with white response $(L_{1w},\ M_{1w},\ S_{1w})$
- $(LMS)_2$ under an illuminant with white response $(L_{2w},\ M_{2w},\ S_{2w})$

The mapping is:

$$\begin{bmatrix} L_2 \\ M_2 \\ S_2 \end{bmatrix} = \begin{bmatrix} \frac{L_{2w}}{L_{1w}} & 0 & 0 \\ 0 & \frac{M_{2w}}{M_{1w}} & 0 \\ 0 & 0 & \frac{S_{2w}}{S_{1w}} \end{bmatrix} \begin{bmatrix} L_1 \\ M_1 \\ S_1 \end{bmatrix}$$

7.14 CIE Standard Illuminant SPDs

We have SPDs for some *standard* light sources.

How do we transform colors?

1. Specify the SPD of the illuminant
2. Convert, after the picture is captured, from device RGB to CIE XYZ. This takes us from device specific color to the color of the standard observer.
3. Perform illuminant-to-illuminant mapping.

7.15 The sRGB Color Space

Corresponds to a white point viewed under a specific illuminant.

When you display an image on a display, that image, when you save it onto your disk, the RGB values that you see are the CIE XYZ values that you would observe under a D65 illuminant. The illumination is baked into that image representation. This is when you're saving or representing images in the sRGB standard. When you capture an image on your phone or create some image and give it RGB values, there is nothing attached that says what that RGB value means.

It's like a standard. Most computers know about it. But your computer probably won't know the color space that your obscure smartphone takes.

7.16 Displaying Color

Each display device has its own SPDs. When you specify RGB values for a display, it will create a color space that is specific to a display.

What do we need to do to create a color perception? Say we have some value in the CIE XYZ color space. How do we display it in a way that it would be properly displayed? What is the SPDs of my actual computer display?

The display driver of my computer needs to know how to convert CIE XYZ colors to device-specific colors. This is done in your settings, and you specify what it is that you're displaying it on. That effectively is a 3x3 matrix that does that conversion for you. Of course, your phone does that automatically as it knows what your display is. But when you connect your computer to another display, you need to make sure you choose the right calibration settings.

How do we make sure two displays show the same colors?

$$[3 \times 3] \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} = \begin{bmatrix} R \\ G \\ B \end{bmatrix}$$

Different displays will have different $[3 \times 3]$ matrices.

How are the matrices made? By the manufacturer.

7.17 Printers

This gets more complicated for printers. Displays emit light. But for printers, you're going to see that page in some lighting. The ink has to be printed on a certain paper, otherwise it won't have the SPD that it has expected. When printing a real photograph nicely, you have to specify what the paper is going to be, what the inks are, and through that information, it decides what the SPD is and how the colors have to be rendered through ink.

If you really want to be accurate, you have to specify the lighting conditions in which that photo will be shown on paper, if you want to be perfectly accurate about what the perception is going to be.

7.18 Gamma Encoding / Gamma Correction

A function applied to the raw image data. We don't see light in a linear way. We want computer displays to behave in a more natural way. The reason why we made displays in the first place is to replicate images in the way we see the world.

Human sensation do not follow a strictly linear relation with intensity.

In the case of brightness, perceived brightness follows a cubic root law compared to radiometric power.

If something is not very bright, you can make out small changes easily in brightness. When someone is much bright, you become much less sensitive to individual changes.

$$S = kI^a$$

Where S = human sensation, I = intensity

So, before an intensity is converted to a byte value, it is mapped in the linear space using a cubic law. That's what we call a gamma-corrected image.

By transforming intensity non-linearly before 8-bit conversion, more bits are allocated to perceptually significant brightness values.

This is standard. If you have a JPEG image, that conversion has been baked in. If you display an sRGB image on a monitor, the monitor assumes that your image was encoded in that way. Your monitor has to undo this known transformation before displaying it.

$$V_S \propto \sqrt[3]{I}$$

Display assumes gamma-encoded images, not linear ones.

8 Edge Detection

How can we go from a full image to an image where the corresponding pixels represent the edges? We've seen that derivative filters can enhance edges but how do we actually detect them?

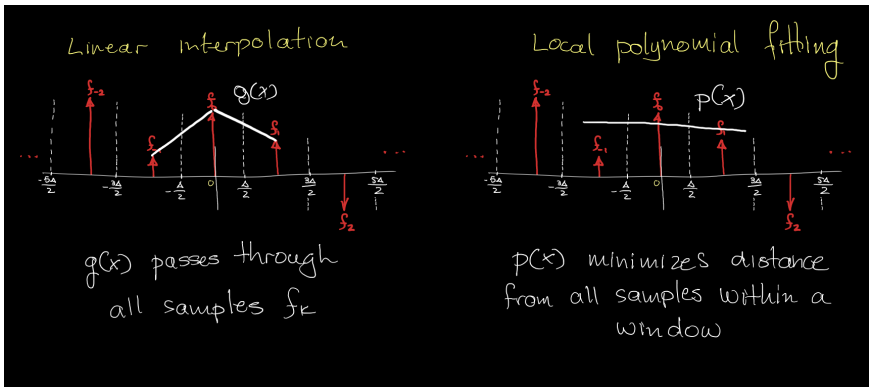
Edge enhancement / detection: at the intensity level

Boundary detection: account for visual perception of what is or is not an object

8.1 Fitting Polynomials to Data / Least Squares Polynomial Fitting

We talked about interpolation previously. The goal there, is given a set of values $\vec{x}$, we are given samples of those functions at those specific values, and we define interpolation as a continuous curve that passes through all the specified points ($g(x)$ that satisfies $g(x_k) = f(x_k) \ \forall k$.

Polynomial fitting is a bit more relaxed – we do not require the polynomial to pass through every point. Yet, we'd like the polynomial to be as close to the points as possible. We will formalize this requirement.



With linear interpolation, function passes exactly through the point. However, for polynomial fitting, $p(x)$ minimizes distances from all samples within a window. It can approximate the data, but it will not pass through all points.

You've heard of linear regression? We can improve it by increasing the degree of the polynomial.

8.2 Taylor Series Expansion

What is the Taylor series? We are given a function f, a value of x that is *close* to 0 (in the neighborhood of 0), and what the approximation does, is that it is a series. No matter what function f is, as long as it is smooth, we can approximate the function **locally** within a neighborhood of 0 as a polynomial. We get to choose the number of degrees (such as n degrees).

WLOG, assume that the window's central sample is equal to 0.

$$f(x) = f(0) + x\frac{df}{dx}(0) + \frac{1}{2}x^2\frac{d^2f}{dx^2}(0) + \cdots + \frac{1}{n!}x^n\frac{d^nf}{dx^n}(0) + R_{n+1}(x)$$

The residual $R_{n+1}(x)$ satisfies $\lim_{x\to 0} R_{n+1}(x) = 0$

We are getting a better approximation if we get close to 0 in the approximated function. As the degree of the polynomial is higher, we get a better approximation.

Typically, we have our function, and we are given an approximation to a certain order. Say we want to approximate the function with a 10^{th} degree polynomial. That is an

approximation, so there will be something missing. Whatever that is, it's going to be very small, and that error will go to 0 as x goes to 0.

For a specific value of x, we can rewrite this Taylor series of n degrees as a product of 2 vectors. One that holds all the x values and the others that hold the constant terms and the derivatives.

$$f(x) = \begin{bmatrix} 1x & \frac{1}{2}x^2 & \frac{1}{6}x^3 & \cdots & \frac{1}{n!}x^n \end{bmatrix} \begin{bmatrix} f(0) \\ \frac{df}{dx}(0) \\ \frac{d^2f}{dx^2}(0) \\ \vdots \\ \frac{d^nf}{dx^n}(0) \end{bmatrix}$$

When doing polynomial fitting, IF we know $f(0)$, then we know $f(x)$ at 0 and this vector $\begin{bmatrix} 1x & \frac{1}{2}x^2 & \frac{1}{6}x^3 & \cdots & \frac{1}{n!}x^n \end{bmatrix}$. We do not know $\begin{bmatrix} f(0) \\ \frac{df}{dx}(0) \\ \frac{d^2f}{dx^2}(0) \\ \vdots \\ \frac{d^nf}{dx^n}(0) \end{bmatrix}$.

If we know $f(x_1) = f_1,\ f(x_2) = f_2,\ f(x_3) = f_3$, then we can express this as a matrix:

$$\begin{bmatrix} f_1 \\ f_2 \\ f_3 \\ \vdots \\ \vdots \end{bmatrix} = \begin{bmatrix} 1x_1 & \frac{1}{2}x_1^2 & \frac{1}{6}x_1^3 & \cdots & \frac{1}{n!}x_1^n \\ 1x_2 & \frac{1}{2}x_2^2 & \frac{1}{6}x_2^3 & \cdots & \frac{1}{n!}x_2^n \\ 1x_3 & \frac{1}{2}x_3^2 & \frac{1}{6}x_3^3 & \cdots & \frac{1}{n!}x_3^n \\ & & \vdots & & \\ 1n & \frac{1}{2}x_n^2 & \frac{1}{6}x_n^3 & \cdots & \frac{1}{n!}x_n^n \end{bmatrix} \begin{bmatrix} f(0) \\ \frac{df}{dx}(0) \\ \frac{d^2f}{d(x)^2}(0) \\ \vdots \\ \frac{d^nf}{dx^n}(0) \end{bmatrix}$$

No, we do not have enough to solve the system.

This system is:

For $2w+1$ samples and an n-degree poynominal, system has $2w+1$ equations and $n+1$ unknowns.

This is our system:

$$b = Ad$$

And we want to minimize the error:

$$||b - Ad||^2 = \sum_{k=1}^{2w+1} ||f_k - p(x_k)||^2$$

And once we have solved for this, we can evaluate the polynomial for any values of x we want.

According to the Taylor approximation, the solution of the linear system provides estimates of the n derivatives of $f(x)$ at the window's central sample.

We don't want our approximated polynomial degree to be too high – those results in overfitting, where the polynomial becomes sensitive the noise in the data. Also, incredibly large-degree polynomials require very large windows (the window must be as large as the degree of the polynomial).

8.3 Moving Least Squares

You have a bunch of data, and you use a window to do a local fit to estimate a quantity somewhere in a window, and then you slide that window over.

Algorithm:

1. Set $k = 1$
2. Define sample window $x_k, \ldots, x_{2w+k}$ centered at sample x_{w+k}
3. Fit an n degree polynomial $p(x)$ to datapoints $(x_x, f_k), \ldots, (x_{2w+k}, f_{2w+k})$
4. Use $p(x)$ to evaluate $f(x)$ and its derivatives at $x = x_{w+k}$
5. Set $k = k + 1$ and repeat steps 2-4 until last sample is reached

Oh, and you don't compute this beforehand. Have you heard of "lazy evaluation"? You have to compute this every time you want an x value.

8.4 Parametric 2D Curves

A continuous mapping $r: (a, b) \rightarrow \mathbb{R}^2$, mapping $t \rightarrow (x(t), y(t))$ Where t is the curve parameter and $(x(t), y(t))$ is the coordinate function.

Example: a boundary curve

- t = pixel no. along boundary
- $x(t) = x$ coordinate of tth pixel
- $y(t) = y$ coordinate of tth pixel

What we care about is smooth curves. The mapping is smooth, as the derivatives are well defined.

A curve is smooth when all derivatives of the coordinate function exist, i.e. $x(t) \in C^\infty$ and $y(t) \in C^\infty$.

Every pixel on the boundary is a *sample* on the curve.

If we're looking at a **closed curve,** our mapping has this additional constraint: $r(a) = r(b)$.

8.5 Coordinate Function Derivatives

The first and second derivatives of $x(t), y(t)$ are highly informative about the curve's shape.

A curve is nothing other than a vector-valued function. You give it a parameter, and it gives you a vector. For vector-valued functions, the derivative generalizes:

$$\vec{f}(t) = (f_1(t), \ldots, f_n(t))$$
$$\frac{d\vec{f}(t)}{dt} = \left(\frac{df_1}{dt}(t), \ldots, \frac{df_n}{dt}(t)\right)$$

So, we have a vector of derivatives. This is just the definition.

8.6 Curve Tangent

A way to get a first-order approximate of the curve itself. Consider a point $\gamma(0) = (x(0), y(0))$ and we want an approximation of the curve at 0.

At the linear case, $\gamma(t) = (x(t), y(t))$

If we want to get an approximate of $\gamma(t)$ near 0, what we would do is apply Taylor series approximation to the individual coordinate functions.

$$\gamma(t) = \begin{bmatrix} x(0) + t\frac{dx}{dt}(0) \\ y(0) + t\frac{dy}{dt}(0) \end{bmatrix}$$

Is there a better way to write this?

$$\gamma(t) = \begin{bmatrix} x(0) \\ y(0) \end{bmatrix} + t \begin{bmatrix} \frac{dx}{dt}(0) \\ \frac{dy}{dt}(0) \end{bmatrix}$$

Guess what? $\begin{bmatrix} x(0) \\ y(0) \end{bmatrix}$ is the original point. And the second term is t multiplied by a vector. The vector will have some orientation. As we vary t, we are moving on the **tangent** of that curve. And the direction? $\begin{bmatrix} \frac{dx}{dt}(0) \\ \frac{dy}{dt}(0) \end{bmatrix}$ is the tangent vector.

So, the tangent vector at $\gamma(t)$ is $\begin{bmatrix} \frac{dx}{dt}(t) \\ \frac{dy}{dt}(t) \end{bmatrix}$ and gives us a way to approximate a curve by a line.

As we look at smaller and smaller neighborhoods near t, the linear approximation is closer and closer to the curve itself.

8.6.1 Invariance to Curve Parameterization

One property of the tangent is, it does not depend on how we define the curve parameter (t).

If two functions $\beta(s)$, $\gamma(t)$ map to the same points (draw the same shape), the tangent vector remains the same.

So, the tangent property is invariant to how we represent the curve itself. It depends only on the points.

What this means, is that if we normalize the vector to unit length, then that is completely invariant to the parameterization.

8.6.2 Unit Tangent

Normalized unit tangent vector.

$$T(t) = \frac{\begin{bmatrix} \frac{dx}{dt}(t) \\ \frac{dy}{dt}(t) \end{bmatrix}}{\left\| \begin{bmatrix} \frac{dx}{dt}(t) \\ \frac{dy}{dt}(t) \end{bmatrix} \right\|}$$

The unit tangent vector does not depend on the choice of the parameter t.

8.7 Unit Normal

Remember how to find a perpendicular slope?

The unit normal is

$$N(t) = \frac{\begin{bmatrix} -\frac{dy}{dt}(t) \\ \frac{dx}{dt}(t) \end{bmatrix}}{\left\| \begin{bmatrix} -\frac{dy}{dt}(t) \\ \frac{dx}{dt}(t) \end{bmatrix} \right\|}$$

The normal is the result of a CCW rotation of the tangent vector for this particular equation. It's a convention for the normal vector to be inward facing.

8.8 The Moving Frame

The moving frame is the pair of orthogonal vectors:

$$(T(t),\ N(t))$$

As we change the parameter t, the moving frame rotates. The faster the frame rotates, the more curved the curve is. That's curvature. If it rotates one way, it corresponds to a bend in one direction, and if it rotates in another way it corresponds to a bend in another direction.

8.8.1 Moving Frame of a Circle

$$\gamma(\theta) = r\begin{bmatrix}\cos(\theta)\\ \sin(\theta)\end{bmatrix}$$
$$\frac{d\gamma}{d\theta}(\theta) = r\begin{bmatrix}-\sin(\theta)\\ \cos(\theta)\end{bmatrix}$$
$$T(\theta) = \begin{bmatrix}-\sin(\theta)\\ \cos(\theta)\end{bmatrix}$$
$$N(\theta) = \begin{bmatrix}-\cos(\theta)\\ -\sin(\theta)\end{bmatrix}$$

9 Local Differential Analysis of 2D Images for Edge Detection

9.1 Local 1st-Order Taylor Series Approximation of a 2D image

Locally, the intensity $f(x,\ y)$ near some point, let's say $(0,\ 0)$, can be expressed as a polynomial of $x,\ y$:

$$f(x,\ y) \approx f(0,0) + x\frac{\partial f}{\partial x}(0,\ 0) + y\frac{\partial f}{\partial y}(0,\ 0) + \text{higher order terms}$$

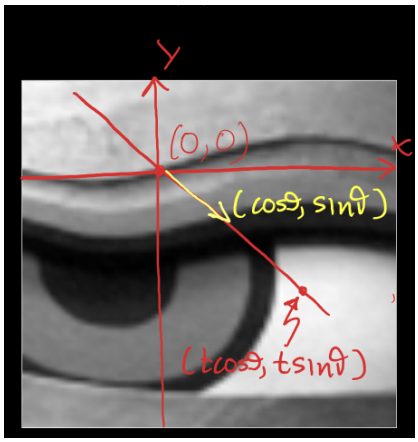
Now, let's consider what happens if we look at a specific direction away from that point, $(0,\ 0)$, and see how much the intensity varies on that direction.

Consider a 1D slice of the image through $(0,\ 0)$ in the direction of a unit vector $\vec{v} = \begin{bmatrix}\cos(\theta)\\ \sin(\theta)\end{bmatrix}$. As we move to different points, we can express it as a scaled version of that function: $t\begin{bmatrix}\cos(\theta)\\ \sin(\theta)\end{bmatrix}$.

This slice can be expressed as the 1D function:

$$g(t) = f\left(t\cos\theta,\ t\sin\theta\right)$$

So, we're just moving along that straight line and looking at the value of f, for every point on that straight line. And we can compute the derivative of g, which tells us how much the intensities change in that particular direction.



$$
\begin{aligned}
g(t) &= f\left(t\cos\theta,\ t\sin\theta\right) \\
&= f(0,0) + t\cos\theta\frac{\partial f}{\partial x}(0,\ 0) + t\sin\theta\frac{\partial f}{\partial y}(0,\ 0) \\
g'(t) &= \cos\theta\frac{\partial f}{\partial x}(0,\ 0) + \sin\theta\frac{\partial f}{\partial y}(0,0)
\end{aligned}
$$

We can write that concisely as the product of two vectors:

$$
g'(t) = \begin{bmatrix} \frac{\partial f}{\partial x}(0,\ 0) & \frac{\partial f}{\partial y}(0,0) \end{bmatrix} \begin{bmatrix} \cos(\theta) \\ \sin(\theta) \end{bmatrix}
$$

For a first-order derivative only, $g'(t)$ is constant. Remember that $g(t)$ is a first order approximation of the function.

So, if I'm considering how the image varies in different directions, I only need to know $\begin{bmatrix} \frac{\partial f}{\partial x}(0,\ 0) & \frac{\partial f}{\partial y}(0,0) \end{bmatrix}$.

Image derivatives along any direction v are given by $\begin{bmatrix} \frac{\partial f}{\partial x}(0,\ 0) & \frac{\partial f}{\partial y}(0,0) \end{bmatrix} \begin{bmatrix} \cos(\theta) \\ \sin(\theta) \end{bmatrix}$. At some point, the vectors are going to line up. When they are lined up, the dot product is maximized. When they are not, then the dot product is 0.

This vector $\begin{bmatrix} \frac{\partial f}{\partial x}(0,\ 0) & \frac{\partial f}{\partial y}(0,0) \end{bmatrix}$ points to where the intensity of the image changes the most and is perpendicular to where the image does not change at all.

So:

- $\begin{bmatrix} \frac{\partial f}{\partial x}(0,\ 0) & \frac{\partial f}{\partial y}(0,0) \end{bmatrix} \begin{bmatrix} \cos(\theta) \\ \sin(\theta) \end{bmatrix}$ maxed out when $\begin{bmatrix} \frac{\partial f}{\partial x}(0,\ 0) & \frac{\partial f}{\partial y}(0,0) \end{bmatrix}$ is parallel to $\begin{bmatrix} \cos(\theta) \\ \sin(\theta) \end{bmatrix} = \vec{v}$
- Minimized when perpendicular

9.2 The Image Gradient and the Directional Derivative

$$\text{image gradient} = \nabla f(x,y) = \left[\frac{\partial f}{\partial x}(x,\ y) \quad \frac{\partial f}{\partial y}(x,\ y)\right]$$

- ∇f is the direction of the steepest intensity change
- ∇f is orthogonal to the direction where image intensity remains constant
- $\nabla f(x,\ y)$ is the normal of the isophte curve through $(x,\ y)$

From the image gradient, we can get the magnitude and the direction. $|\nabla f(x,\ y)|$ visualized looks like the DoG filter convolved with the image.

With that, we have a way to assess locally, the strength of an edge given on a point. Now, how do we actually detect them?

9.3 Edge Detection on Gradient Magnitude

"Binary out" $|\nabla f(x,\ y)|$": just threshold it for each pixel: $|\nabla I| > g_{\text{min}}$

What should we set g_{min}:

- Too low: thickness on the edges and dust. We do not want thickness on the edges
- Too high: not detectable

9.4 Hysteresis Thresholding Procedure

Choosing the right threshold is not easy. Well, we can use this insight saying, ideally, you can identify a bunch of pixels that can be classified as an edge, but there are other pixels nearby where the gradient is lower and belongs to the edge. So, we can define two thresholds:

$$g_{high}, g_{low}$$

- Strong gradients are more likely to be an edge
- Weaker gradients are more likely to be on an edge if they are near a strong gradient

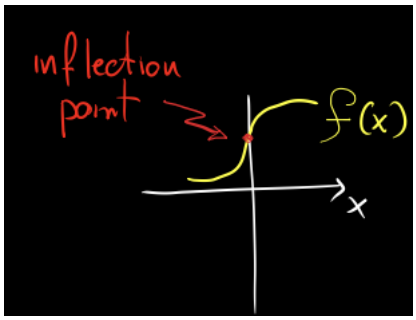
This is called the **hysteresis thresholding** procedure:

- Pass no. 1: mark as an edge pixel with $|\nabla I| > g_{high}$
- Pass no. 2: same with $|\nabla I| > g_{\text{low}}$ ONLY if adjacent to an image already marked as an edge previously

Can we do better than this?

9.5 Edge Detection By Locating Intensity Inflection Points

Give a very precise criterion to when a pixel lies on the edge. It won't be a threshold this time. We're going to treat the **inflection point** as the location of the edge.



The inflection point is a very specific location that corresponds to the extremum of the derivative. The second derivative, at 0, corresponds to an inflection point.

9.6 Laplacian

For 2D, we use the Laplacian:

$$\nabla^2 f(x, y) = \frac{\partial^2 f}{\partial x^2}(x, y) + \frac{\partial^2 f}{\partial y^2}(x, y)$$

That is the same as computing the second derivative in both directions.

9.7 Laplacian Zero-Crossings

A pixel (x, y) for which $f * \nabla^2 G_\sigma$ has a different sign from at least 1 3×3 neighbor.

We can eliminate zero-crossings for which the difference between positive and negative in a 3×3 neighborhood is below a threshold.

So, our operation here would be:

- Compute $f * \nabla^2 G_\sigma$
- Detect zero crossings of $f * \nabla^2 G_\sigma$
- Prune zero-crossings that are not significant

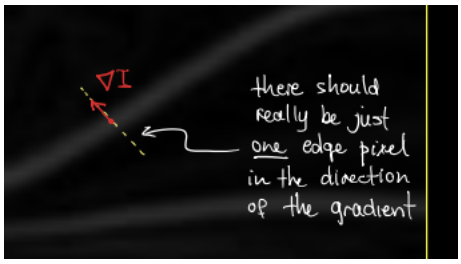
9.8 Canny Edge Detection Algorithm

Keep some of the benefits of the gradient magnitude $|\nabla f(x, y)|$, which gives information on edge normals, less sensitive to noise, but it needs thresholds.

And we have Laplacian zero-crossings, which has no normal information, more sensitive to noise, but at least the threshold is better for rejection of weak edges and has excellent localization.

We will look at the gradient magnitude, then we do a non-maxima suppression. We will look at a small area of the image. We won't be taking all the pixels with high gradients:

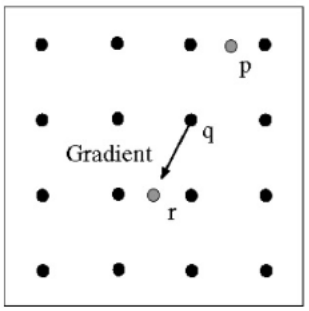
- We look at the pixels individually
- We draw a line in direction of ∇I
- In the direction, we only mark one pixel as the edge pixel



Actual procedure

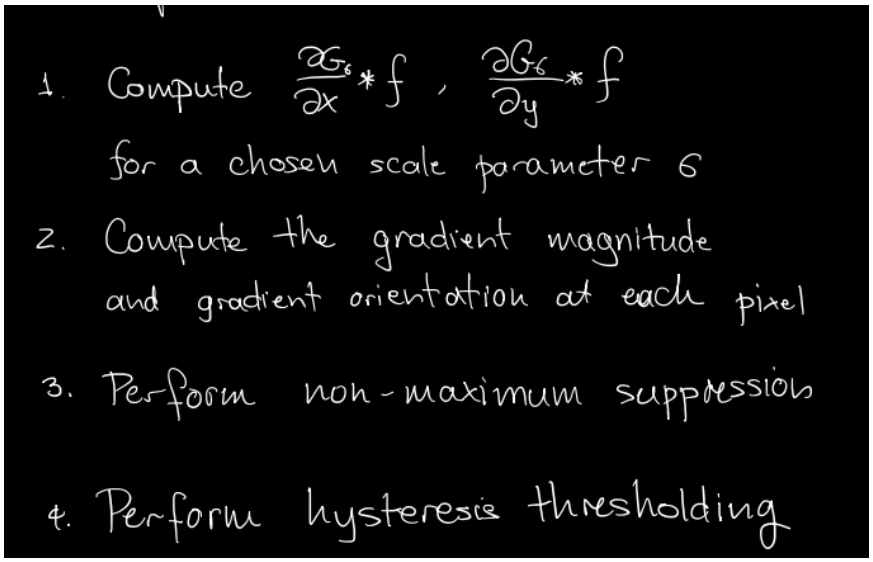
- Compute $\nabla I(q)$ at each pixel q. It's a general direction.
- Mark q as an edge if and only if $|\nabla I| < |\nabla I|(q)$ at the two neighboring pixels p, r in direction of $\nabla I(q)$

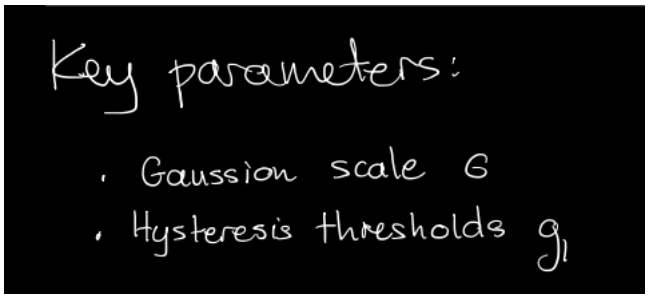
So, only two pixels in the neighborhood



We don't have these exact values for r, p, you got to interpolate them.

And with this, non-maxima suppression ensures that edges are thin. To get a slightly more complete set of edges:





9.9 Temp

$g - g'$ for aperture size $D - D'$. The larger the aperture, the more blurred the image is.

$$h(x,y) = \begin{cases} 1 & \text{if } \sqrt{x^2+y^2} \in \left[D, D'\right] \\ 0 & \text{otherwise} \end{cases}$$