

Lecture 1, Sep 2, 2025

Introduction to Robotic Vision

- Many vision problems are *ill-posed*, i.e. a solution may not exist, may not be unique, or may not be continuous with regard to change of the initial conditions
- Vision is an *inverse problem*
 - Much information is lost, so we need to appeal to probabilistic models
 - Most of our physical models are forward models, i.e. how should a particular object look under certain conditions?
 - The inverse is non-unique, e.g. a bigger object far away appears the same as a smaller, closer object
 - * This is known as *scale ambiguity*
 - By projecting 3D geometry to a 2D image we lose a lot of information – appearance only weakly depends on geometry

Definition

The *imaging function* (i.e. *perspective projection*):

$$I = \mathcal{P}(G, M, V, L, A, \epsilon)$$

where:

- G : Scene geometry (shape of the world)
- M : Materials
- V : Viewpoint (where the camera is located)
- L : Lighting (where light sources are)
- A : Atmospherics (how light interacts with the atmosphere)
- ϵ : Noise

We aim to find \mathcal{P}^{-1} to recover G, M, V from I .

- Parallel lines converge at “points at infinity” under perspective projection, and shapes are distorted
- Datasets typically contain synchronized, timestamped data from GPS/INS, LiDAR, stereo, etc. and accurate intrinsic/extrinsic calibration of sensors
 - Popular datasets include KITTI, Waymo Open (100x larger than KITTI), Nuscenes (object detection and tracking)

Lecture 2, Sep 5, 2025

Mathematical Fundamentals of Vision

- We need to reason over both the *geometric* (points, lines, shapes) and *photometric* (brightness, contrast, texture, shading) aspects of the scene, which are closely intertwined
- We will use \mathcal{F}_v to denote frame v (vetric notation) and column vectors like $\mathbf{x} = \begin{bmatrix} x \\ y \end{bmatrix}$

2D Transformations

- Points may also be represented in *homogeneous form*: $\tilde{\mathbf{x}} = (\tilde{x}, \tilde{y}, \tilde{w}) \in \mathbb{P}^2$
 - $\mathbb{P}^2 = \mathbb{R}^3 \setminus (0, 0, 0)$ is a *projective space*
 - This can be converted back to an inhomogeneous vector by dividing by \tilde{w} : $\tilde{\mathbf{x}} = \tilde{w}(x, y, 1) = \tilde{w}\bar{\mathbf{x}}$
 - $\bar{\mathbf{x}} = (x, y, 1)$ is the *augmented vector* (note the bar), with a canonical scale of 1
 - $\tilde{w} = 0$ represent *points at infinity* (aka *ideal points*); hence $(0, 0, 0)$ is undefined and excluded from \mathbb{P}^2
- Projective geometry allows us to represent and manipulate objects at infinity, which is necessary for cameras

- Since they are homogeneous (not affected by scalar multiplication), \mathbb{P}^2 is topologically equivalent to the unit sphere
- $\tilde{\mathbf{l}} = (a, b, c)$ represents the line $\tilde{\mathbf{x}} \cdot \tilde{\mathbf{l}} = ax + by + c = 0$ in 2D
 - This can be normalized to $\mathbf{l} = (\hat{n}_x, \hat{n}_y, d) = (\hat{\mathbf{n}}, d)$ where $\hat{\mathbf{n}}$ is the unit normal vector and d is the distance to origin
 - The intersection of two lines can be found by taking their cross product
- Define the *skew-symmetric form* $[\mathbf{u}]_{\times} = \begin{bmatrix} 0 & -u_3 & u_2 \\ u_3 & 0 & -u_1 \\ -u_2 & u_1 & 0 \end{bmatrix}$
 - This is skew-symmetric, i.e. $[\mathbf{u}]_{\times}^T = -[\mathbf{u}]_{\times}$
 - This allows us to write the cross product as $[\mathbf{u}]_{\times} \mathbf{v} = \begin{bmatrix} u_2v_3 - v_2u_3 \\ u_3v_1 - v_3u_1 \\ u_1v_2 - v_1u_2 \end{bmatrix}$
- A rigid transformation can be represented as $\mathbf{x}' = [\mathbf{C} \ \mathbf{t}] \tilde{\mathbf{x}}$ where \mathbf{C} is a rotation matrix, \mathbf{t} is a translation vector
 - Note $\det \mathbf{C} = 1$ and $\mathbf{C}\mathbf{C}^T = \mathbf{C}^T\mathbf{C} = \mathbf{I}$
 - We are rotating the vector while keeping the reference frame constant, instead of the other way around
- An *affine transformation* is $\mathbf{x}' = \mathbf{A}\tilde{\mathbf{x}}$ where $\mathbf{A} \in \mathbb{R}^{2 \times 3}$
 - Important to note parallel lines remain parallel after an affine transformation
 - This has 6 degrees of freedom
- A *projective transformation* or *homography* is $\tilde{\mathbf{x}}' = \tilde{\mathbf{H}}\tilde{\mathbf{x}}$
 - Straight lines remain straight, but parallel lines may not be parallel after the transformation
 - This has 8 degrees of freedom: $\tilde{\mathbf{H}} \in \mathbb{R}^{3 \times 3}$ (note below)
 - Note $\tilde{\mathbf{H}}$ is also *homogeneous*, i.e. defined up to scale only
 - * This is similar to homogeneous coordinates; if we multiply all 3 components by some scalar, we get the same point back just represented differently






Transformation	Matrix	# DoF	Preserves	Icon
translation	$\begin{bmatrix} \mathbf{I} & \mathbf{t} \end{bmatrix}_{2 \times 3}$	2	orientation	
rigid (Euclidean)	$\begin{bmatrix} \mathbf{R} & \mathbf{t} \end{bmatrix}_{2 \times 3}$	3	lengths	
similarity	$\begin{bmatrix} s\mathbf{R} & \mathbf{t} \end{bmatrix}_{2 \times 3}$	4	angles	
affine	$\begin{bmatrix} \mathbf{A} \end{bmatrix}_{2 \times 3}$	6	parallelism	
projective	$\begin{bmatrix} \tilde{\mathbf{H}} \end{bmatrix}_{3 \times 3}$	8	straight lines	

Figure 1: Hierarchy of 2D coordinate transformations.

3D Transformations

- Rotations preserve the length and orientating (handedness) of space
- Rotations have the following properties: let a, b, c be rotations, then:
 - Closure: $a \circ b$ is a rotation
 - Associativity: $(a \circ b) \circ c = a \circ (b \circ c)$
 - Invertibility: each rotation has a unique inverse rotation
 - Identity: the identity map is a rotation
- Therefore rotations form a *group* under composition; this is the *rotation group* or *special orthogonal group* on \mathbb{R}^3 , denoted $SO(3)$

- Rotations can be represented by matrices, Euler angles, axis/angle or quaternions
 - Euler angles decomposes rotations into the product of 3 elementary rotations about individual frame axes
 - * Due to different orders, there are 12 possible rotation sequences
 - * Suffers from gimbal lock
 - Axis-angle expresses rotations as angle θ around a unit vector $\hat{\mathbf{n}}$
 - * Note only the perpendicular component rotates
 - * *Rodriguez formula*: $\mathbf{C}(\hat{\mathbf{n}}, \theta) = \mathbf{I}_3 + \sin \theta [\hat{\mathbf{n}}]_{\times} + (1 - \cos \theta) [\hat{\mathbf{n}}]_{\times}^2$
 - This can be derived by decomposing the vector into parallel and perpendicular components and rotating the perpendicular component
 - Quaternions are hyper-complex numbers that take the form $\mathbf{q} = q_0 + q_1\hat{i} + q_2\hat{j} + q_3\hat{k}$
 - * $i^2 = j^2 = k^2 = ijk = -1$
 - * $ij = k, jk = i, ki = j, ji = -k, kj = -i, ik = -j$
 - Note i, j, k do not commute
 - * The set of quaternions is denoted \mathbb{H} and form a 4D non-commutative division algebra
 - * Unit quaternions satisfy $\|\mathbf{q}\|^2 = q_0^2 + q_1^2 + q_2^2 + q_3^2 = 1$ and can be mapped to rotations
 - * They have a direct relationship with the axis angle form: $\mathbf{q} = \begin{bmatrix} q_0 \\ \mathbf{q} \end{bmatrix} = \begin{bmatrix} \cos(\theta/2) \\ \hat{\mathbf{u}} \sin(\theta/2) \end{bmatrix}$
 - Using Rodriguez's formula, we have $\mathbf{C} = \mathbf{I}_3 + 2q_0 [\mathbf{q}]_{\times} + 2[\mathbf{q}]_{\times}^2$
 - Explicit form: $\mathbf{C}(\mathbf{q}) = \begin{bmatrix} 1 - 2q_2^2 - 2q_3^2 & 2q_1q_2 - 2q_0q_3 & 2q_0q_2 + 2q_1q_3 \\ 2q_0q_3 + 2q_1q_2 & 1 - 2q_1^2 - 2q_3^2 & 2q_2q_3 - 2q_0q_1 \\ 2q_1q_3 - 2q_0q_2 & 2q_0q_1 + 2q_2q_3 & 1 - 2q_1^2 - 2q_2^2 \end{bmatrix}$
 - * To compose two rotations represented as quaternions, we can multiply them, following the rules of quaternion multiplication (denoted \otimes)
 - $\mathbf{p} \otimes \mathbf{q} = (p_0 + \mathbf{p}) \otimes (q_0 + \mathbf{q})$

$$= p_0q_0 - \mathbf{p}^T \mathbf{q} + p_0\mathbf{q} + q_0\mathbf{p} + \mathbf{p} \times \mathbf{q}$$
 - Note order matters!
 - In 3D, rigid transformations take the same form of $\mathbf{x}' = \begin{bmatrix} \mathbf{C} & \mathbf{t} \end{bmatrix} \bar{\mathbf{x}}$
 - Affine transformations are analogous but $\mathbf{A} \in \mathbb{R}^{3 \times 4}$
 - Projective transformations now use $\tilde{\mathbf{H}} \in \mathbb{R}^{4 \times 4}$

Lecture 3, Sep 9, 2025

Probability Theory Review

- A random variable X is a variable whose value x may vary due to randomness
 - For *probability density function* (PDF) $p(x)$ we have $\int_x p(x) dx = 1$
 - $\int_a^b p(x) dx$ is the probability of $x \in [a, b]$
- For joint distributions $p(x, y)$ we can find the *marginal distribution* by integrating over one or more variables (marginalization): $p(x) = \int p(x, y) dy$
- A *conditional probability* is given by $p(x|y = y^*) = p(x|y) = \frac{p(x, y)}{p(y)} \iff p(x, y) = p(x|y)p(y)$
- This leads to *Bayes' rule*: $p(y|x) = \frac{p(x|y)p(y)}{p(x)}$
 - $p(y|x)$ is *posterior*
 - $p(y)$ is the *prior*
 - $p(x|y)$ is the *likelihood*
 - $p(x)$ is the *evidence*
 - e.g. if y is some robot state and x is a sensor measurement, we have that the probability of being in a state given that we have some measurement is equal to the probability of getting the

	Euclidean	similarity	affine	projective
Transformations				
rotation	X	X	X	X
translation	X	X	X	X
uniform scaling		X	X	X
nonuniform scaling			X	X
shear			X	X
perspective projection				X
composition of projections				X
Invariants				
length	X			
angle	X	X		
ratio of lengths	X	X		
parallelism	X	X	X	
incidence	X	X	X	X
cross ratio	X	X	X	X

Figure 2: Summary of different geometries, allowed transformations in each, and which quantities are invariant under the allowed transformations.

measurement if we are in that state, times the previous belief of our probability of being in that state, divided by the probability of getting the measurement in general

- The *expectation* of $f(x)$ is $\mathbf{E}[f(x)] = \sum_x f(x)p(x)$ (discrete) or $\int_x f(x)p(x) dx$ (continuous)
 - Note the expectation is a linear operator
- The *Gaussian/normal distribution*: $x \sim \mathcal{N}(\mu, \sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$ for mean μ , standard deviation σ
 - In higher dimensions: $\mathbf{x} \sim \mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma}) = 2\pi^{-\frac{n}{2}} \det \boldsymbol{\Sigma}^{-\frac{1}{2}} e^{-\frac{1}{2}(\mathbf{x}-\boldsymbol{\mu})^T \boldsymbol{\Sigma}^{-1}(\mathbf{x}-\boldsymbol{\mu})}$ for an n -dimensional Gaussian
- Often the PDF for a random variable is very complex, so to represent it in code we approximate it by one of many methods (e.g. single Gaussian for unimodal data, mixture of Gaussians for multimodal data, histogram, particle distribution for arbitrary distributions)

Estimation Techniques

- Consider the linear regression problem, where we are given a number of noisy points (x_i, y_i) , and we want to fit our model $y_i = f(x_i; \boldsymbol{\theta}) = mx_i + b$ where $\boldsymbol{\theta} = (m, b)$ is the vector of parameters we wish to determine
- We want to minimize $E_{LS} = \sum_i \|\tilde{y}_i - f(x_i; \boldsymbol{\theta})\|^2 = \sum_i \|\tilde{y}_i - (mx_i + b)\|^2$
 - \tilde{y}_i are the measurements and $r_i = \tilde{y}_i - (mx_i + b)$ are the residuals
- In matrix form, $y = \begin{bmatrix} x & 1 \end{bmatrix} \begin{bmatrix} m \\ b \end{bmatrix} = \mathbf{J}(x)\boldsymbol{\theta}$ where $\mathbf{J}(x) = \frac{\partial f(x; \boldsymbol{\theta})}{\partial \boldsymbol{\theta}}$ is the model Jacobian with respect to the parameters (this is possible since we have a linear model)
 - Substitute and expand: $E_{LS} = \boldsymbol{\theta}^T \left[\sum_i \mathbf{J}^T(x_i)\mathbf{J}(x_i) \right] \boldsymbol{\theta} - 2\boldsymbol{\theta}^T \left[\sum_i \mathbf{J}^T(x_i)\tilde{y}_i \right] + \sum_i \tilde{y}_i^2$
 - Since $\sum_i \tilde{y}_i^2$ is constant, we minimize the other part
 - By differentiation we get $\hat{\boldsymbol{\theta}} = \left[\sum_i \mathbf{J}^T(x_i)\mathbf{J}(x_i) \right]^{-1} \left[\sum_i \mathbf{J}^T(x_i)\tilde{y}_i \right]$ (*normal equation*)
- If data points are weighted differently, we change the loss to $E_{WLS} = \sum_i \sigma_i^{-2} \|r_i\|^2$
 - This is known as the *Mahalanobis distance*
- But notice due to the quadratic cost function, this method is sensitive to outliers; the model is distorted significantly to account for just a few outliers

RANSAC

- *RANSAC* or *Random Sample Consensus* is a method for outlier rejection:
 1. Determine the smallest number of data points required to fit the model
 - In the linear regression case, we have 2 points for a line
 2. Draw the smallest possible subset to fit the model
 - Usually this is drawn by uniform sampling, unless we have a prior
 3. Check the number of points in the whole dataset that are within some threshold of the model prediction – these are the inliers
 4. If we have enough points within the threshold, terminate
 - At this point we can re-fit the model to the inliers for better generalization/noise resistance
 5. Otherwise repeat from step 2 to generate a new hypothesis until we reached the max number of iterations
- RANSAC requires 2 parameters: the error tolerance for model compatibility, the max number of subsets to try, and the threshold for the number of inliers for a success
- Let w be the probability of drawing an inlier, then we can model the expected number of trials k needed to select n good data points

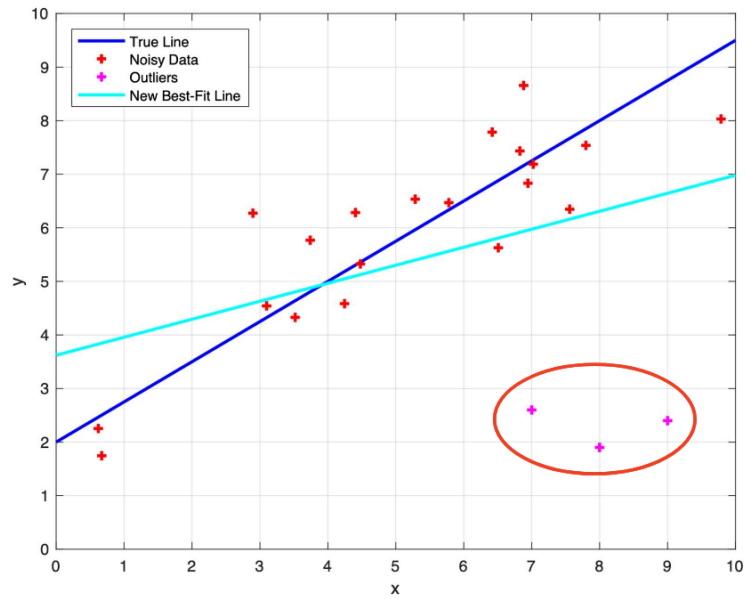


Figure 3: Example fit with outliers.

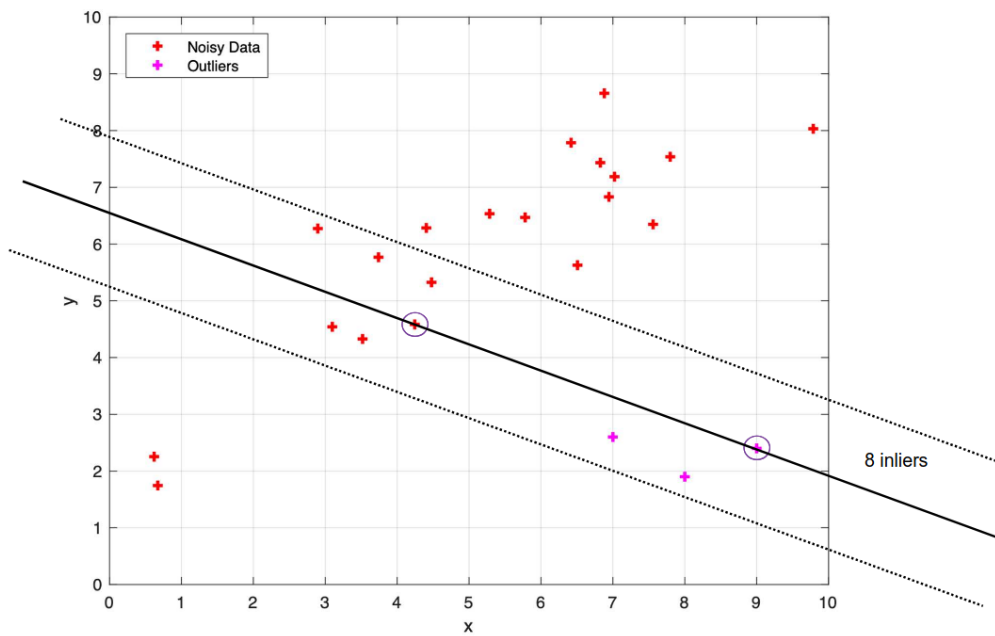


Figure 4: RANSAC on example data: incorrect hypothesis (containing an outlier) resulting in a model that captures very few data points.

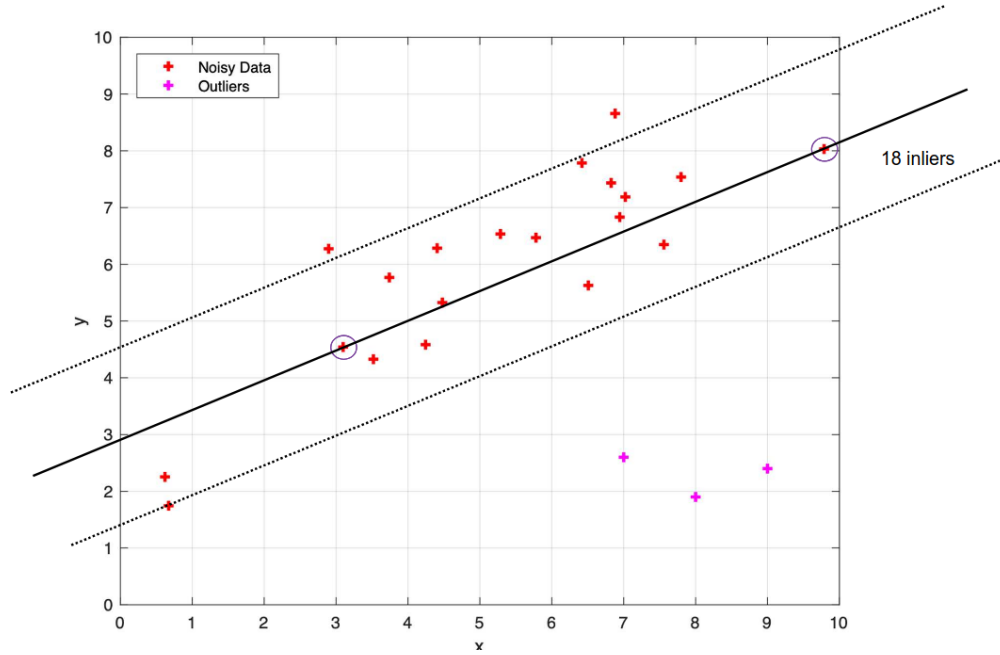


Figure 5: RANSAC on example data: correct hypothesis resulting in a model that captures most data points and rejects all outliers.

- Let $b = w^n$ be the probability of getting only inliers for our model, and $a = 1 - b$
- $E[k] = b + 2(1 - b)b + 3(1 - b)^2b + \dots + i(1 - b)^{i-1}b + \dots$

$$= b(1 + 2a + 3a^2 + \dots + ia^{i-1} + \dots)$$

$$= b \sum_{i=1}^{\infty} ia^{i-1}$$

$$= \frac{1}{b}$$

$$= w^{-n}$$
- How many iterations do we need to get at least one good sample with probability z ?
 - $z = 1 - (1 - w^n)^k = 1 - (1 - b)^k$
 - This gives $k = \frac{\log(1 - z)}{\log(1 - b)}$
 - Now we can substitute in the desired probability to get an outlier-free sample, z , and the probability of drawing a good sample $b = w^n$ to get the number of iterations that we should run RANSAC

Note

For RANSAC, we want the subset that we draw to fit the model to contain only inliers and generalize well enough to the rest of the dataset. In most cases this means drawing the minimum number of points to fit the model, since this makes it less likely that we will draw an outlier, decreasing the expected number of required iterations. However, drawing more points has the benefit of reducing noise and numerical sensitivity and risk of degeneracy. Especially in cases where the outlier ratio is known to be low, it may be beneficial to draw non minimal samples sometimes.

Lecture 4, Sep 12, 2025

Image Formation and Optics

- The simplest model is *orthographic projection*, $\mathbf{x} = [\mathbf{I}_2 \ 0] \mathbf{p}$, which simply discards the z component
 - This is only an approximation for very long focal lengths (telephoto lens)
 - The *scaled orthographic projection*, $\mathbf{x} = [s\mathbf{I}_2 \ 0] \mathbf{p}$ further scales by s to account for differences in coordinate systems
- The ideal perspective/pinhole model assumes a single point aperture, so all light rays intersect at the *optical center*, which is used as the origin of the camera frame
 - The optical axis pierces the image plane at the *principal point*, used as the origin of the image plane (note: not pixel) coordinate system
 - The camera frame is denoted \mathcal{F}_c
- To project $p = (x, y, z)$ in the camera frame onto the image frame, define the *projective map* $\bar{\mathbf{x}} = \mathcal{P}_z(\mathbf{p}) = \begin{bmatrix} x/z \\ y/z \\ 1 \end{bmatrix}$ which comes from similar triangles
 - This gives us the point's projection in the normalized image plane coordinate system (normalized by focal length)

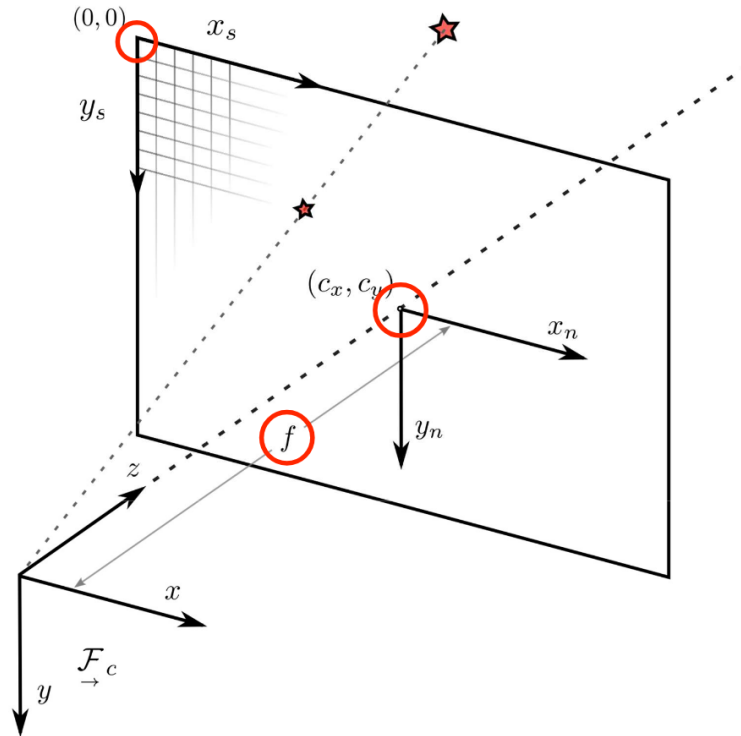


Figure 6: Diagram of the projection model.

- To relate actual coordinates to pixel coordinates, we need camera parameters:
 - (c_x, c_y) defines the principal point in pixel coordinates (note top left corner is the origin)
 - The focal lengths f_x, f_y , which may be different for the two axes
 - Sometimes, axes are not perfectly aligned, so the skew angle s models this
 - * This can often be ignored for modern cameras

- This is organized into the *intrinsic parameter matrix* \mathbf{K} :

$$\begin{bmatrix} x_s \\ y_s \\ 1 \end{bmatrix} = \begin{bmatrix} f_x & s & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x/z \\ y/z \\ 1 \end{bmatrix} = \mathbf{K} \frac{\mathbf{p}}{z}$$

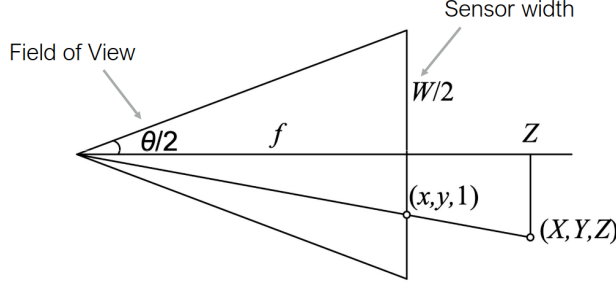


Figure 7: Relationship between field of view, imaging sensor width and focal length.

- We can relate the focal length, FoV and sensor width: $\tan \frac{\theta}{2} = \frac{W}{2f}$
- The camera also has *extrinsic parameters* \mathbf{E} that describe the relationship between the camera frame and another preferred frame (e.g. the base robot frame)
- Intrinsic and extrinsic parameters can be combined: $\mathbf{P} = \mathbf{K} [\mathbf{C} \quad \mathbf{t}]$
 - In matrix form: $\tilde{\mathbf{P}} = \begin{bmatrix} \mathbf{K} & \mathbf{0} \\ \mathbf{0}^T & 1 \end{bmatrix} \begin{bmatrix} \mathbf{C} & \mathbf{t} \\ \mathbf{0}^T & 1 \end{bmatrix} = \tilde{\mathbf{K}} \mathbf{E}$
 - Note we must be very careful that here \mathbf{E} is the world frame to camera frame transform (since we are discussing the forward model), not the camera pose in the world frame (which would be the camera-to-world transform)
- This allows us to map the point $\mathbf{p}_w = \begin{bmatrix} x_w \\ y_w \\ z_w \\ 1 \end{bmatrix}$ in world coordinates to pixel coordinates by $\tilde{\mathbf{x}}_s = \tilde{\mathbf{P}} \bar{\mathbf{p}}_w$
 - The resulting coordinates must be normalized by dividing by the third element!
 - Note $\mathbf{x}_s = \begin{bmatrix} x_s \\ y_s \\ 1 \\ d \end{bmatrix}$, where d is the inverse depth

Distortion

- Real lenses all have distortion, which displace points from their ideal perspective position (i.e. straight lines are no longer straight)
 - *Radial distortion*: points are displaced radially from the centre of distortion, due to the lens
 - * This is the most noticeable
 - *Tangential distortion*: pixels are shifted one direction, with further pixels shifted more, due to the lens and imaging plane not being perfectly parallel
- The most common distortion model is the *plumb bob model*, which uses a polynomial to model the distortion effects; this is a function that maps where the pixels should be to where they would appear due to distortion
 - $\begin{bmatrix} x_d \\ y_d \end{bmatrix} = \underbrace{(1 + \kappa_1 r^2 + \kappa_2 r^4 + \kappa_3 r^6)}_{\text{radial distortion}} \begin{bmatrix} x_n \\ y_n \end{bmatrix} + \underbrace{\begin{bmatrix} 2\tau_1 x_n y_n + \tau_2 (r^2 + 2x_n^2) \\ 2\tau_2 x_n y_n + \tau_1 (r^2 + 2y_n^2) \end{bmatrix}}_{\text{tangential distortion}}$
 - * $r = \sqrt{x_n^2 + y_n^2}$ is the distance from the image center; note all coordinates are in the normalized image plane coordinate system
 - Often κ_1 and κ_2 are enough to model the radial distortion, while tangential distortion might be negligible
- For radial distortion, we can have a “barrel” distortion (corner points pushed in) or “pincushion” distortion (corner points pushed out), or “mustache” distortion (straight line curves like a mustache)
 - Note for the “barrel” distortion, all pixels are being pushed inward, but the ones on the sides of the barrel are pushed in less; conversely for pincushion all pixels are being pulled out, but the

ones on the corners are pulled out more

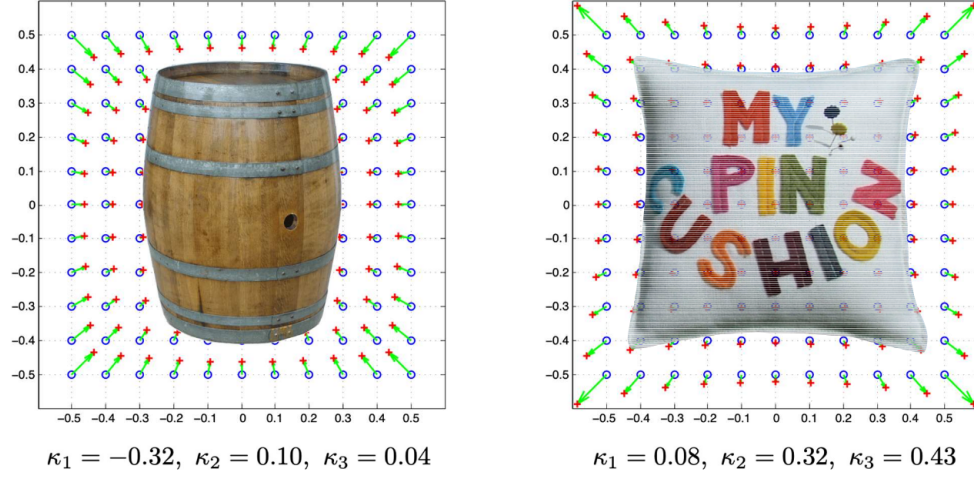


Figure 8: Visualization of two types of radial distortion.

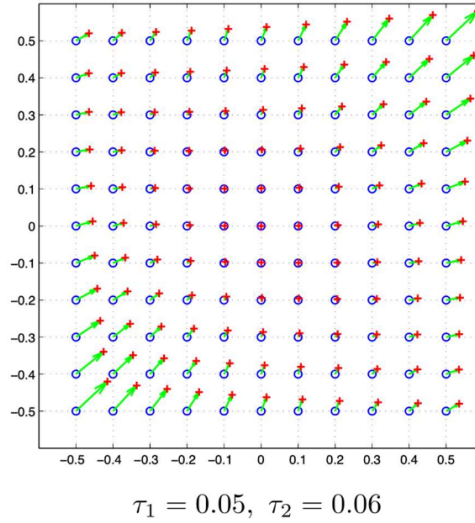


Figure 9: Visualization of tangential distortion.

- *Unwarping* is the process of removing distortion effects from the image, effectively reversing the distortion model
 - No analytical solution to the model exists in general, but we can precompute a nonlinear transform in a lookup table to approximate undistortion, since it only depends on image plane coordinates and not the image itself
 - This involves computing the distorted coordinates from normalized coordinates, and interpolating the pixel values
- Plumb bob is one of many models and is best suited for normal cameras; other types of cameras, e.g. fisheye (very large FOV) or cameras with different imaging geometry, e.g. catadioptric and omnidirectional cameras, require completely different camera models

Summary

To transform a world point $\bar{\mathbf{p}}_w$ to image pixel coordinates:

1. Transform into camera frame: $\bar{\mathbf{p}}_c = \begin{bmatrix} \mathbf{C} & \mathbf{t} \\ \mathbf{0}^T & 1 \end{bmatrix} \bar{\mathbf{p}}_w = \mathbf{E} \bar{\mathbf{p}}_w$
2. Project into image plane and normalize: $\begin{bmatrix} x_n \\ y_n \\ 1 \end{bmatrix} = \begin{bmatrix} x/z \\ y/z \\ 1 \end{bmatrix} = \frac{\mathbf{p}_c}{z}$
3. Apply distortion: $\begin{bmatrix} x_d \\ y_d \\ 1 \end{bmatrix} = \mathcal{D} \left(\boldsymbol{\kappa}, \boldsymbol{\tau}, \begin{bmatrix} x_n \\ y_n \\ 1 \end{bmatrix} \right)$

$$= (1 + \kappa_1 r^2 + \kappa_2 r^4 + \kappa_3 r^6) \begin{bmatrix} x_n \\ y_n \\ 0 \end{bmatrix} + \begin{bmatrix} 2\tau_1 x_n y_n + \tau_2 (r^2 + 2x_n^2) \\ 2\tau_2 x_n y_n + \tau_1 (r^2 + 2y_n^2) \\ 1 \end{bmatrix}$$
4. Transform to pixel coordinates: $\begin{bmatrix} x_s \\ y_s \\ 1 \end{bmatrix} = \begin{bmatrix} f_x & s & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_d \\ y_d \\ 1 \end{bmatrix} = \mathbf{K} \begin{bmatrix} x_d \\ y_d \\ 1 \end{bmatrix}$

Other Camera Effects

- *Chromatic aberration* is an effect caused by the difference in refraction due to wavelength (like in a prism), so different colours focus at slightly different distances and with slightly different magnifications
 - This is less of an issue with modern hardware
- *Vignetting* is the tendency for brightness to decrease towards the edge of the image
 - This is caused by the foreshortening of the lens, i.e. for rays coming in from higher angles, the size of the lens is effectively smaller, so less light can come in
 - This can be modelled (e.g. \cos^4) to remove the effect
 - More noticeable in cheaper systems and can still be an issue today



Figure 10: An example of a stitched image affected by vignetting.

- Two main sensor types exist: CCD and CMOS
 - CCD uses the entire sensor surface for imaging and reads everything through the same circuitry, so it's very sensitive and uniform, but is very expensive (used in e.g. telescopes)
 - CMOS is much more common but only uses 1/3 of the surface for imaging, while the rest is used to build a circuit for each pixel
- Due to spatial sampling of the light, *aliasing* is another effect that is unavoidable
 - This can cause Moire patterns and other effects

- Imaging sensors have twice as many green pixel sensors since humans naturally have more receptors for green light, so this looks more natural to us

Lecture 5, Sep 16, 2025

Image Operations

Point Operations

- Images can be thought of as functions, with the input being the pixel location and the output being the pixel intensity; we can apply operations to these functions
- The simplest operations are point operations, which operate on individual pixels – a pixel depends only on itself (or maybe global image characteristics)
 - e.g. thresholding (to get a binary image), brightness and contrast adjustments, histogram equalization, gamma correction
 - This can be a *monadic* (i.e. single image) or *dyadic* (two image) operation (e.g. HDR), etc
- An *operator* is a function that takes one or more input images and produces an output image: $g(\mathbf{x}) = h(f_0(\mathbf{x}), \dots, f_n(\mathbf{x}))$
 - $\mathbf{x} = (i, j)$ is the pixel location
 - Example:
 - * Gain and bias (brightness/contrast) adjustments: $g(\mathbf{x}) = af(\mathbf{x}) + b$
 - * Gamma correction (removes nonlinear radiance map): $g(\mathbf{x}) = (f(\mathbf{x}))^{\frac{1}{\gamma}}$
- *Histogram equalization* is a method to “spread out” the range of pixel intensities in an image, so pixels intensities are more evenly distributed over the dynamic range
 - A *histogram* is a function $h(r_k) = n_k$ is a discrete lookup function that maps an intensity $r_k \in [0, L - 1]$ to the number of pixels n_k with that intensity level
 - We often normalize this so $p(r_k) = \frac{n_k}{n}$ (where n is the total number of pixels) is an estimate of the probability of grey level r_k
 - The image looks more high-contrast if the histogram is more distributed, so we use more of the available dynamic range
 - We want to find an intensity mapping $T(r_k)$ to transform each pixel so that the resulting histogram is approximately flat
 - We use the CDF: $s_k = T(r_k) = \sum_{j=0}^k p(r_j) = \sum_{j=0}^k \frac{n_j}{n}$
 - * Can think of this as finding the percentile of each pixel – the relative intensity of a pixel is approximately the proportion of pixels that it’s brighter than
 - * Note this CDF is scaled $[0, 1]$, so we multiply by $L - 1$ (usually 255) to convert back to integer intensities

Neighbourhood Operations

- *Neighbourhood operations* operate on a pixel given information about its neighbourhood (i.e. Surrounding pixels)
- A *linear filter* is defined as $g(i, j) = \sum_{k, l} f(i - k, j - l)h(k, l)$
 - This operation is known as a *convolution* $g = f * h$; here $h(k, l)$ is the *kernel* (impulse response)
- When we’re at the end of the image, the kernel goes over the edge of the image, so we need to make a choice of *padding*
 - Zero: all pixels past the edge are treated as zero
 - Wrap: pixels on the edge wrap around to the other side
 - Clamp: pixels past the edge take on the same value as the edge pixel
 - Mirror: the edge is “mirrored”, so pixels past the edge take the same values as the pixels before the edge, in reverse order

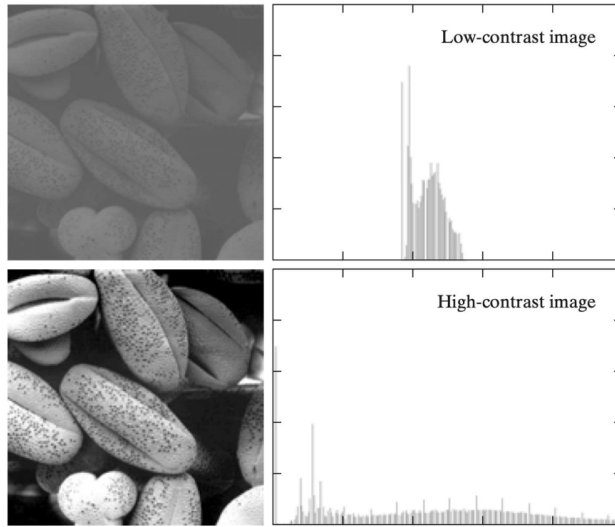


Figure 11: Illustration of histogram equalization.

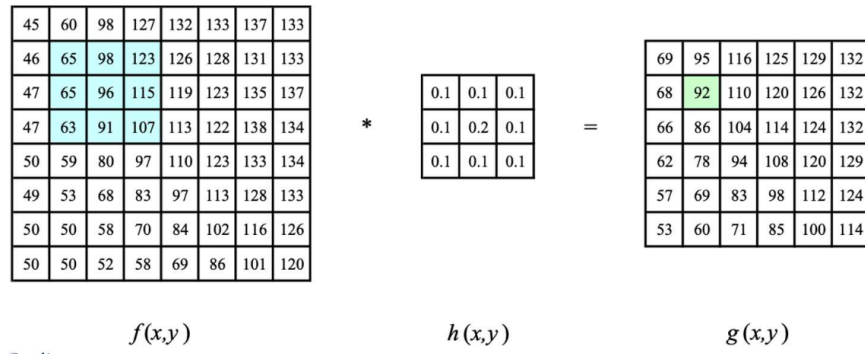


Figure 12: Example of a convolution operation.

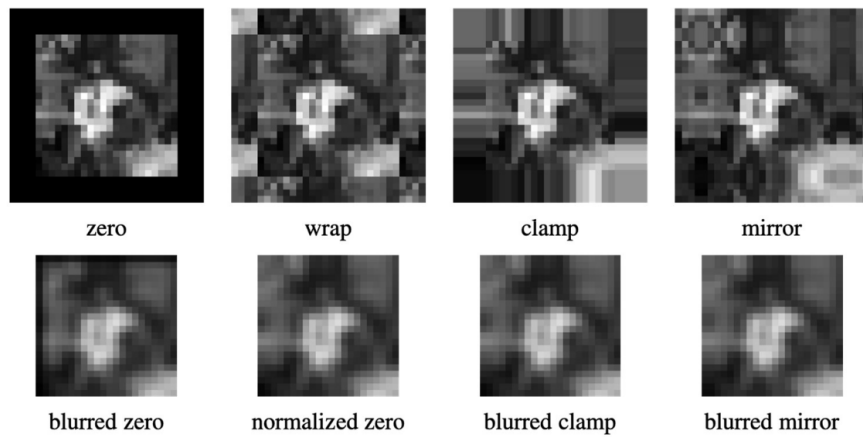


Figure 13: Effect of different padding choices.

- In the context of learning none of them seem to have a major impact, so zero padding is often used
- Generally convolutions require K^2 multiplication and additions per pixel where K is the kernel size
 - *Separable* linear filters can be decomposed into a 1D horizontal convolution followed by a 1D vertical convolution (only $2K$ operations)
 - A separable kernel can be decomposed like $\mathbf{K} = \mathbf{v}\mathbf{h}^T$
 - * We can tell whether a kernel is separable analytically using an SVD
- The *isotropic Gaussian kernel* (aka *Gaussian blurring*) is $G(x, y; \sigma) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}}$
 - This is a separable kernel, so we can compute it using fewer resources
 - Can be thought of a low-pass filter, since sharp details are smoothed out
- *Band-pass filters* removes low and high frequencies from an image
 - The *Laplacian of Gaussian* filter is $\nabla^2 G(x, y; \sigma) = \left(\frac{x^2 + y^2 + 2\sigma^2}{\sigma^4} \right) G(x, y; \sigma)$ where $G(x, y, \sigma)$ is the Gaussian filter
 - * This combines a Laplacian filter $\nabla^2 f = \frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2}$ which removes low frequencies with the Gaussian which removes high frequencies
 - This can be used for edge detection; noise in the signal is cancelled out by the symmetric filter, but zero crossings are amplified
 - Due to the shape, this is also known as the sombrero filter

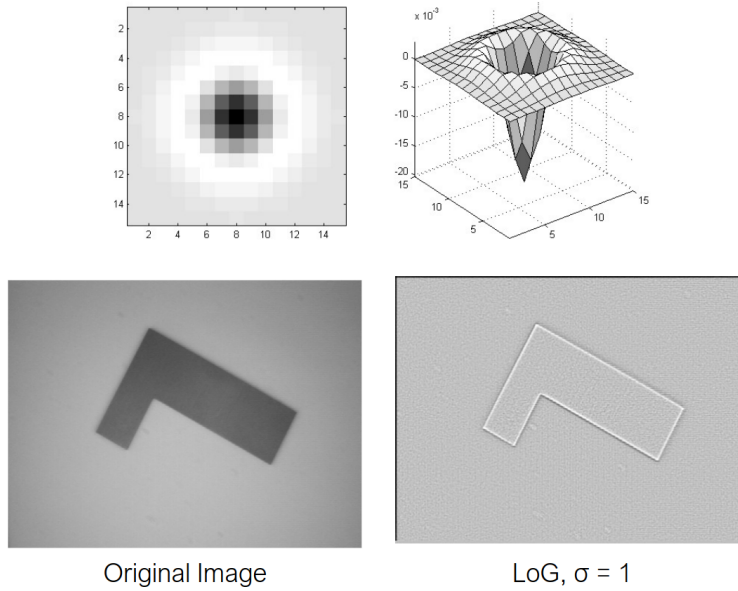


Figure 14: Shape of the Laplacian of Gaussian kernel (top) and effect of applying the filter (bottom). Notice the transformed image has a bright outline on the edge of the shape.

- A *summed area table* or *integral image* is an efficient technique for computing a sum of any rectangular area of an image
 - This can be efficiently computed recursively
 - Useful for certain applications like computing box filter convolutions
- *Median filters* are a type of nonlinear filter which selects the median pixel from each pixel's neighbourhood
 - This is good at removing *shot noise*
 - Min and max filters are other examples of common nonlinear filters
- The *bilateral filter* soft-rejects pixels whose value differs too much from the centre pixel
 - $g(i, j) = \frac{\sum_{k,l} f(k, l)w(i, j, k, l)}{\sum_{k,l} w(i, j, k, l)}$ where the weighing function is $w(i, j, k, l) = d(i, j, k, l)r(i, j, k, l)$

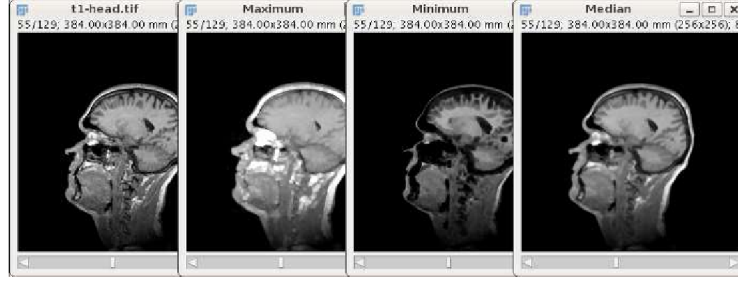


Figure 15: Example of the effects of max, min, and median filtering.

- $d(i, j, k, l) = e^{-\frac{(i-k)^2 + (j-l)^2}{2\sigma_d^2}}$ is the *domain kernel*
 - * Similar to a Gaussian centered around the pixel
- $r(i, j, k, l) = e^{-\frac{\|f(i, j) - f(k, l)\|^2}{2\sigma_r^2}}$ is the *data-dependent range kernel*
 - * This attenuates large differences in pixel intensities
- This filter removes noise while preserving edges and is often used to increase image resolution

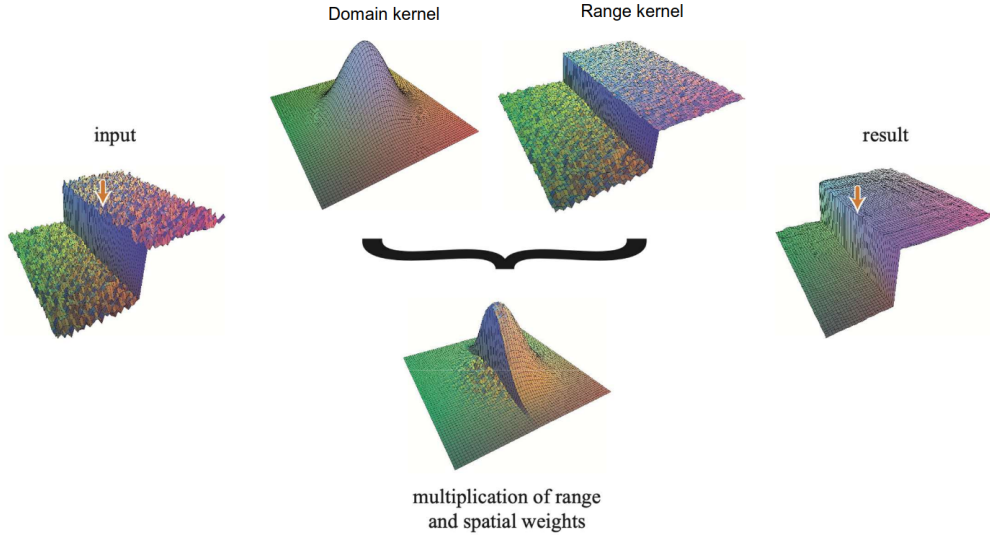


Figure 16: Illustration of bilateral filtering.

Geometric Transformations

- *Geometric transformations* apply to the domain of the image instead, i.e. $g(\mathbf{x}) = f(\mathbf{h}(\mathbf{x}))$, effectively moving pixels around
- A *forward transform* (or warp) maps each pixel location in the input to a new location, $\mathbf{x}' = \mathbf{h}(\mathbf{x})$
 - The problem is that starting with some integer pixel location in the input, we end up with a non-integer location in the output
 - Rounding to the nearest pixel results in aliasing and leads to holes and gaps
- The best way to map the pixel back is to perform an *inverse transform* on each pixel in the target image, which gives us a location in the original image; now we apply a bilinear interpolation to get the correct intensity for that pixel
 - This requires being able to invert the transformation, which is often the case (just need to invert a matrix for most cases)
 - This is used when removing radial distortion – the undistorted pixel location is mapped to the distorted location and interpolated to get its value

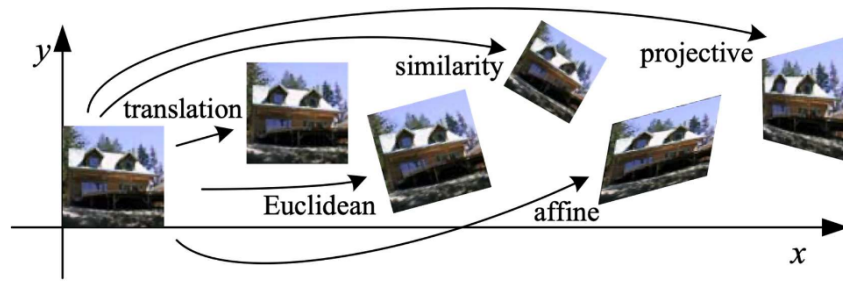


Figure 17: Illustration of different types of 2D parametric transformations.

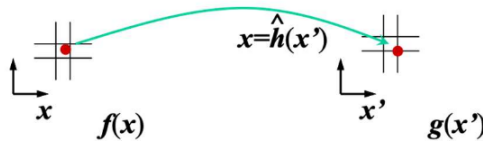


Figure 18: Performing an inverse mapping.

Regularization

- Many vision problems are ill-posed and ill-conditioned inverse problems, i.e. solutions are severely underconstrained; *regularization* is a technique to solve these problems
 - We also see this in training neural networks, e.g. weight decay
- We define a global *energy function* with some desired property (akin to applying a prior), and then finding a minimum energy solution
 - Often we know that the result we want is smooth, so we combine a smoothness penalty with a data penalty
 - $\varepsilon_1 = \iint f_x^2(x, y) + f_y^2(x, y) \, dx \, dy$
 - * f_x, f_y denotes gradients
 - * This forces the solution to have smooth gradients
 - $\varepsilon_d = \iint (f(x, y) - d(x, y))^2 \, dx \, dy$
 - * This ensures the solution respects the data points we have
 - We find the overall global minimum $\varepsilon = \varepsilon_d + \lambda \varepsilon_1$
 - Can think of this as “poking up a tent” – with only ε_1 we get a flat surface, and for each data point we add it “pokes up” the surface

Lecture 6, Sep 19, 2025

Image Features

- *Image features* (aka *interest points*) are regions in the image that are:
 - Salient: distinctive, identifiable, and different from its immediate neighbourhood
 - Local: occupies a small subset of the image pixels, so it has a precise location
 - Repeatable: can be found in other images, even if transformed a little
 - Compact: can be represented efficiently
 - Unique: should be different from the other features in the image, so matching can be done reliably
- Used for detection, matching, tracking
 - Identify a set of features in two different images, match them across the images (identify *correspondences*), and calculate motion/homography/etc
 - Can be used for e.g. making a panorama, tracking an object, SLAM
- Patches of low variation (e.g. sky) or repeating patterns (e.g. linear edges) are usually not good features,

while things like corners are much better

- A *stable corner* is a point in the image where we have a pronounced gradient in both x and y , i.e. a corner
- For an edge we only have a gradient in one direction, while for a textureless region we have low gradients in both directions
- Consider a simple way to match 2 patches across different images, with an offset u ; we compute a weight for each one of the surrounding pixels and sum them: $E_{WSSD}(u) = \sum_i w(x_i)(I_1(x_i + u) - I_0(x_i))^2$
 - The *autocorrelation* function $E_{AC}(\Delta u) = \sum_i w(x_i)(I_0(x_i + \Delta u) - I_0(x_i))^2$ is obtained by using I_0 for both images and small Δu
 - * Where we have a high autocorrelation, we have a distinctive region, since this means it stands out from surrounding locations
 - This can be approximated with a first-order Taylor series, using the gradients $\sum_i w(x_i)(\nabla I_0(x_i) \cdot \Delta u)^2 = \Delta u^T A \Delta u$
 - * This gradient can be computed using a convolution
- The autocorrelation matrix takes the form $A = w * \begin{bmatrix} I_x^2 & I_x I_y \\ I_x I_y & I_y^2 \end{bmatrix}$ where w is the weight kernel matrix
 - The inverse of this matrix is a lower bound on the uncertainty of the location of the feature
 - Therefore we can use eigenvalue analysis to determine the uncertainties – smaller eigenvalues corresponds to more uncertainty, so we want to choose locations so that the smallest eigenvalue is maximized
- This leads to the *Harris corner detector*, which gives each point a score: $\det A - \alpha \text{tr} A^2 = \lambda_0 \lambda_1 - \alpha(\lambda_0 + \lambda_1)^2$
 - When both eigenvalues are large, we have a high score, indicating a good feature
- Note the Harris detector only detects key points, but do not assign a descriptor to them, so it cannot be used for matching

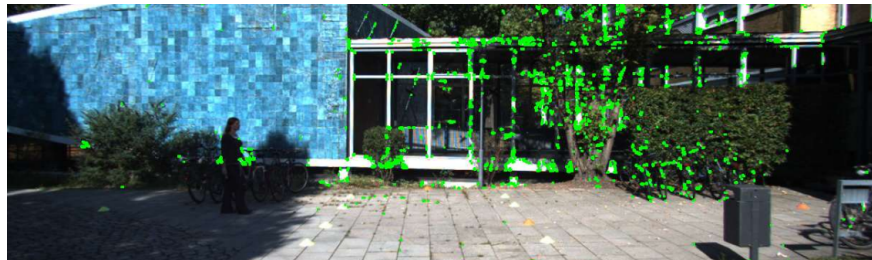


Figure 19: Harris corner detector on the KITTI dataset.

SIFT (Scale Invariant Feature Transform)

- The Harris corner detector depends highly on the patch size that we use to construct the autocorrelation function, therefore it is very sensitive to scale and also rotations (it would still detect them but might not match well)
 - A sharp corner under blur may look like it's rounded, or a rounded corner at a distance away looks sharp
 - We need a scale and orientation invariant feature detector/representation
- SIFT is a scale-invariant algorithm with the following steps:
 1. Construct a *scale space representation* of the input image
 - To find the scale space representation, we essentially apply Gaussian blurs of increasingly larger covariances, and then downsample (rescale) the image to be increasingly smaller
 2. Find keypoints with difference of Gaussian (DoG) operator
 - DoG is similar to LoG but cheaper to compute since we are already applying Gaussian blur

- * The extrema of the DoG is used as the keypoints
- * The idea is that sharp features like corners will be lost when applying Gaussians, so if we take the difference we can see where large changes occurred and it corresponds to features
- For each of the pixels at each scale, it is compared against immediate neighbours in both its scale and one scale up/down; if it is an extrema then we know that feature is the most prominent at that scale
- 3. Reject poor keypoints similar to the Harris approach
 - Compute the Hessian at each point $\mathbf{H} = \begin{bmatrix} D_{xx} & D_{xy} \\ D_{xy} & D_{yy} \end{bmatrix}$ and reject points for which $\frac{(\text{tr } \mathbf{H})^2}{\det \mathbf{H}} > 10$ (i.e. large difference between eigenvalues)
- 4. Assign an orientation to each keypoint
 - Identify the principal rotation direction of the feature
 - Construct a gradient magnitude histogram: sample points in a local region around the keypoint, and compute gradients and the direction of the gradient for each point; then create a 36-bin histogram (10 degrees per bin) from these orientations
 - The bin with the highest peak is used as the orientation; if there is a bin with a peak greater than 80% of the highest peak, it is also used (a new feature is created)
 - Note this is all done at the Gaussian scale that the keypoint was most prominent on
- 5. Assemble into a descriptor for the feature
 - Take the 16x16 patch around the keypoint and break it up into 16 subpatches of 4x4 pixels; for each subpatch, calculate the gradient direction on each pixel, and put into a histogram with 8 bins
 - The histograms of each subpatch is assembled together for the final feature vector (4x4 grid with 8-orientation histogram per grid gives 128 dimensions)
 - To achieve orientation invariance, subtract the feature's orientation (from the previous step) from each gradient direction before binning it

$$L = \sigma^2 (G_{xx}(x, y, \sigma) + G_{yy}(x, y, \sigma))$$

(Laplacian)

$$\text{DoG} = G(x, y, k\sigma) - G(x, y, \sigma)$$

(Difference of Gaussians)

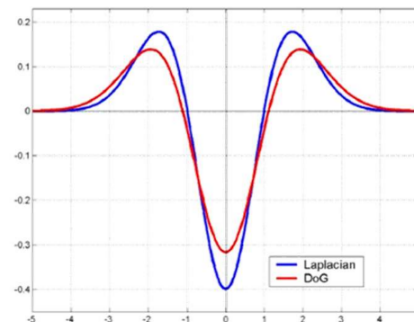


Figure 20: Comparison of LoG and DoG.

- Due to all the steps involved, SIFT is relatively expensive

SURF (Speeded-Up Robust Features)

- Another scale and rotation-variant detector designed to compete with SIFT
- Also uses Hessians and scale space, but to make it faster instead of using Gaussians it uses box filters, which can be computed very efficiently using integral images (see previous lecture)
- The descriptor size is only half as big as SIFT (64 elements)
- About 3x as fast as SIFT and often performs better in terms of accuracy

FAST (Features from Accelerated Segment Test)

- Simple algorithm:
 - For each possible pixel location, look at the 16 pixels in a circle (radius 3) around the pixel, and look at how many of them are brighter than the centre pixel by some threshold

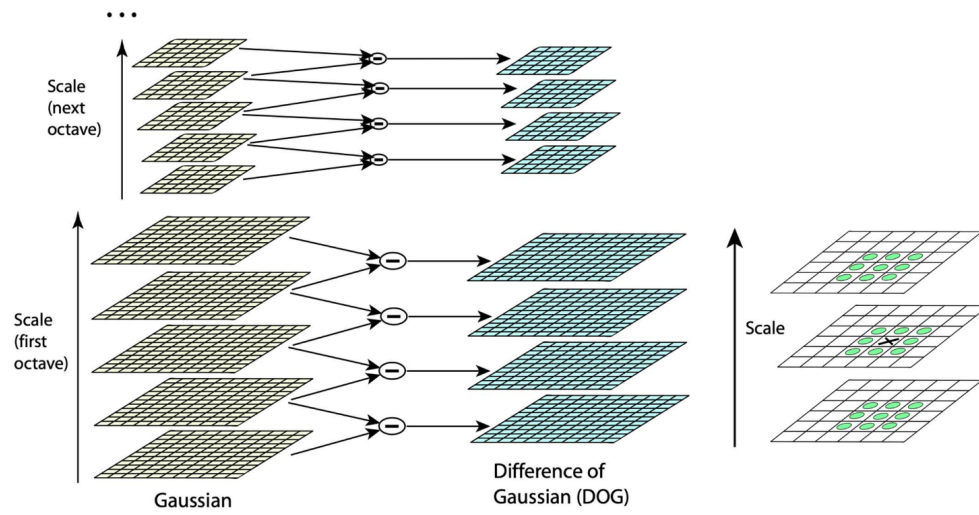


Figure 21: Keypoint identification in SIFT.

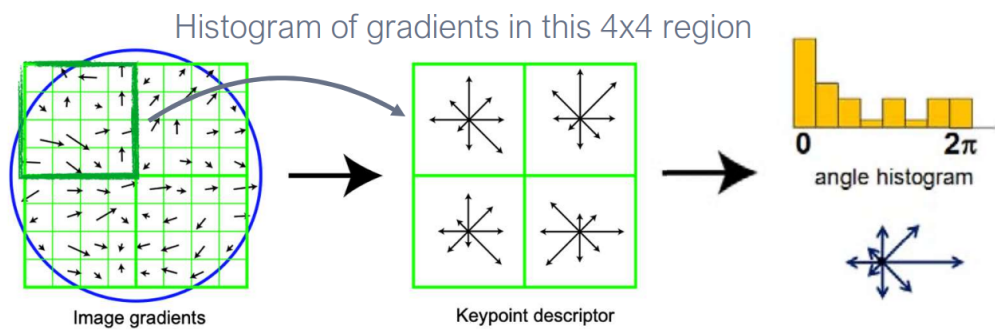


Figure 22: Process of calculating the orientation descriptor.

- p is a corner if there is a set of n contiguous pixels in the circle which are all brighter or all darker (in the original paper this was chosen as 12)
- We can make this even faster if we check only 4 points (1, 5, 9, 13), and if those do not pass the test then we don't need to test further
- The n parameter is very sensitive – any lower and there are much more features, higher and there are too few features
- Uses non-maximum suppression (i.e. for each maximum, look in an area around it and suppress all other maxima in that region)
- This method is very fast but can make matching hard

BRISK (Binary Robust Invariant Scalable Keypoints)

- BRISK is a much faster and much more compact feature detector
- It uses concentric circles around the keypoint, and samples pairs of points within the circles and compares their brightnesses, generating one bit for each comparison
 - Note we're not looking at individual pixels, but instead we apply a Gaussian to the surrounding region
 - The comparison of the points is done in a systematic pattern

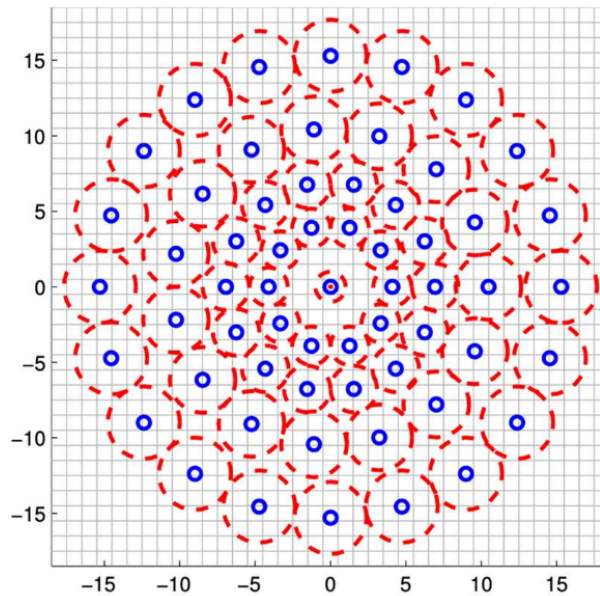


Figure 23: Sampling pattern for BRISK.

BRIEF (Binary Robust Independent Elementary Features)

- BRIEF draws pairs of points randomly around each pixel to be tested, according one of several distributions, and compares the pairs in a manner similar to BRISK and forms a descriptor
 - A number of different ways exist to sample the pixel pairs but they have similar performance
 - To draw the samples we can do it uniformly (GI), with a Gaussian around the center (GII), with a Gaussian around the location of the first pixel (GIII), draw both from a coarse polar grid (GIV), or use the center pixel and draw from the coarse grid for the other pixel (GV)
- This generates 512-bit vectors
- Note the locations of the pairs sampled are randomly chosen but is consistent across features/images, to facilitate matching
- It is highly efficient, but does not have scale or rotation invariance, and can be sensitive to noise

ORB (Oriented FAST and Rotated BRIEF)

- Combines FAST and BRIEF and adds orientation and scale invariance
- One of the most reliable free feature detectors, used by systems like ORB-SLAM
- ORB downsamples the image at multiple scales, and applies FAST across all layers to detect keypoints; for each feature the orientation is computed based on the intensity centroid of the patch, and uses the orientation to compute a rotated BRIEF descriptor across scales
 - For each patch, we find its centroid using the pixel intensity as weight, and the direction of the intensity centroid relative to the test pixel is the orientation
 - To compute the rotated BRIEF descriptors we rotate the sampling points by the orientation that we found in the previous step (equivalently, rotate the image)

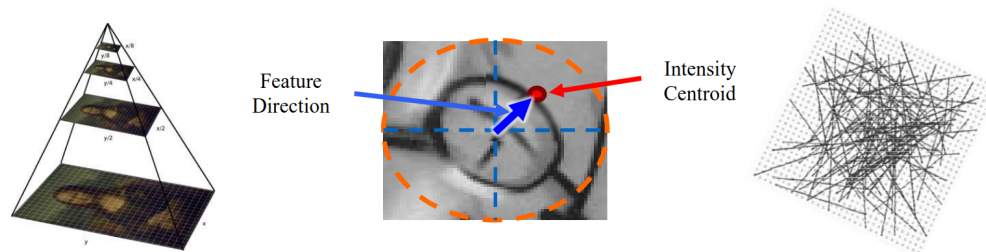


Figure 24: Illustration of the ORB feature detector procedure.

Deep Learning Based Detectors

- Learned feature representations have been becoming more popular
- Supervised methods can use hand-engineered features (e.g. SIFT or ORB) to match image patches to generate ground truth
- Self-supervised and unsupervised methods use sequences of images with pose information to learn how to match images
- Learned invariant feature transform (LIFT) is one of the first learned feature detectors, an end-to-end feature detector consisting of 3 CNNs for detection, orientation estimation, and generating a description
 - Trained with contrastive loss – maximize descriptor similarity of matching pairs, maximize disparity for different features

Lecture 7, Sep 23, 2025

Image Feature Matching and Tracking

- To use image features, we need to match them across images, i.e. identify correspondences
- We need a distance function that computes the similarity of two descriptors, so when matching we minimize this distance
 - For features without a descriptor, the simplest distance function is the *sum of squared differences* (SSD) between local patches around each feature
 - For vector-based descriptors like SIFT and SURF Euclidean distance is often used
 - Binary descriptors like SURF can use *Hamming distance* instead (number of bits that differ)
 - Also consider the ratio of the distance between the first best vs. second best match; if these are too close, the feature is likely ambiguous so we don't want to include it since it might lead to a false positive
- To quantify the performance of matching, we build a *confusion matrix* (aka *contingency table*) out of the true positives (TP , number of correct matches detected), false negatives (FN , number of matches missed), false positives (FP , number of incorrect matches), and true negatives (TN , number of incorrect matches correctly rejected)
 - Often false negatives are much less important since we often have enough true positives

- However false positives are very bad since they can significantly distort the resulting transformation
 - * RANSAC can be used to reject outliers (calculate homography with a subset of matches, and check if enough features match)

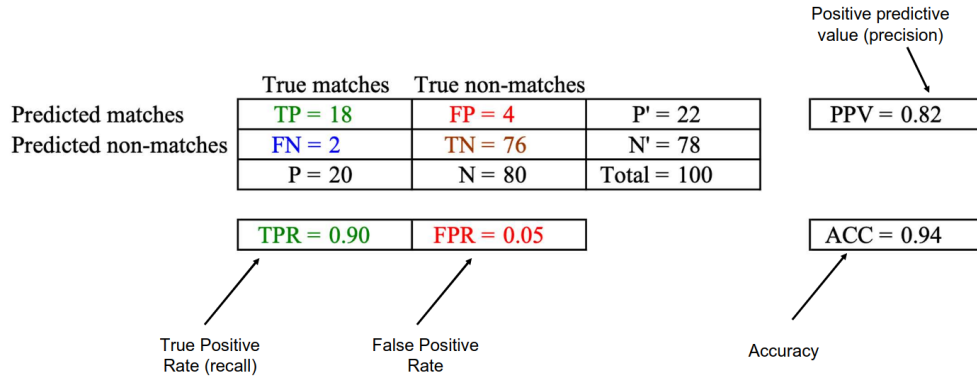


Figure 25: Confusion matrix and definition of precision, recall, and accuracy.

- We can compute a number of stats using the confusion matrix:
 - True positive rate: number of true positives out of all positives in the data $TPR = \frac{TP}{TP + FN}$ (aka recall)
 - False positive rate: number of false positives out of all negatives in the data $FPR = \frac{FP}{FP + TN}$
 - Positive predictive value: number of true positives out of all detected positives $PPV = \frac{TP}{TP + FP}$ (aka precision)
 - Accuracy: number of correct matches out of the total dataset size $ACC = \frac{TP + TN}{P + N}$
- We can evaluate the overall performance of a classifier using a *receiver operating characteristic* (ROC) curve, which compares the true positive and false positive rates for different threshold values
 - Often FP and FN rates go hand-in-hand as thresholds are changed, so the ROC curve allows us to see how they relate for a given classifier

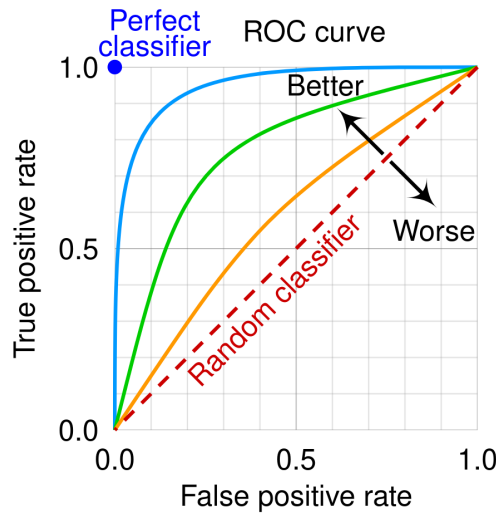


Figure 26: A Receiver Operating Characteristic (ROC) curve.

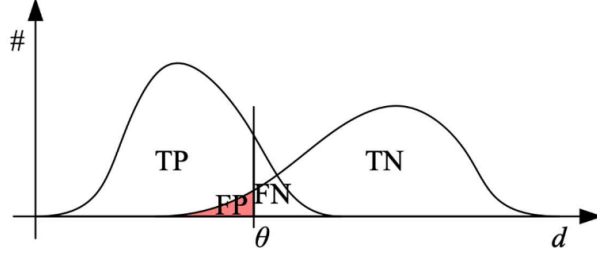


Figure 27: Illustration of the relationship between false positives and false negatives. Changing the threshold θ moves the cutoff along the axis, which changes the amount of each distribution included.

Efficient Matching

- If we brute force the feature matching, we need $O(n^2)$ complexity
 - This is often good enough if the feature detector is selective enough, e.g. SIFT, but descriptors like ORB or FAST generate more features and need more efficient matching
- If we have some idea of the relative transformation (i.e. assume we didn't move too much between images), we can use the spatial cue to match more efficiently
 - More often this is done in descriptor space, i.e. the k -D tree is built using the descriptor vector
 - However this does not make sense for binary descriptors
- k -D trees are a data structure that allows us to efficiently find the nearest neighbour, which is often the most likely feature to match
 - They are similar to a multi-dimensional binary tree, and allow us to do a repeated binary search to find the nearest neighbour
 - Construction can be done in $O(kn \log n)$ and takes $O(n)$ space, where k is the number of dimensions of the datapoints (considered constant) and n is the number of datapoints
 - Allows us to efficiently search for the nearest neighbour in $O(\log n)$ time
- To build a k -D tree, we split along the median of the next dimension at each level
 - Cycle through every dimension until there is at most one datapoint in each partition, i.e. we have a binary tree
 - Each node contains a key and value for the node, the dimension that the node splits on, and left and right subtree pointers

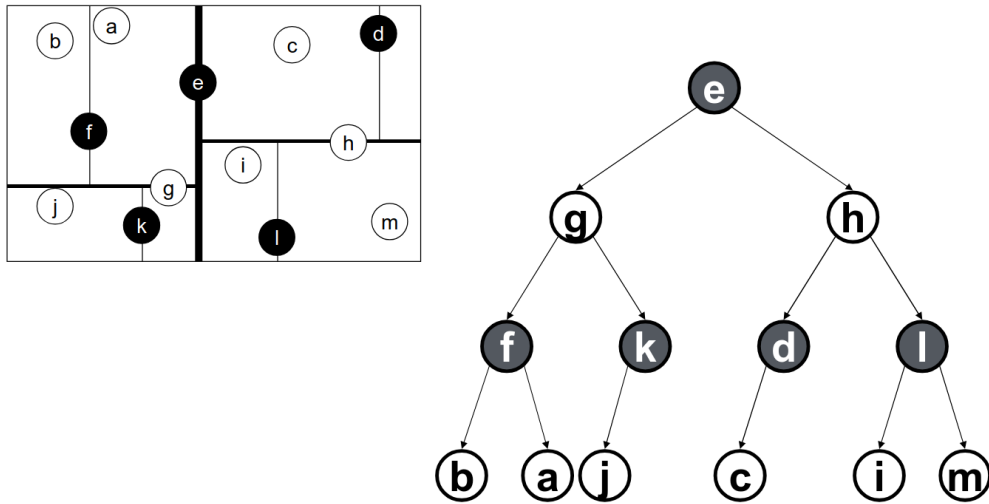


Figure 28: Example structure of a 2D tree. Black nodes split along the x dimension while white nodes split along the y dimension.

- To search for the existence of a specific value, we perform a search similar to a standard binary tree, except we cycle through dimensions as we move down the tree
- To find the nearest neighbour:
 1. Find the node in the same partition as the query point by searching down the tree, and set the current min distance to the distance between this node and the query point
 2. Go back up the tree, and at each parent node check the following:
 1. Check the node against the current best and update if necessary
 2. Check if the other branch (other than the one we came from) can possibly contain a point closer to the query point than the current best
 - In practice this is done by using the distance between the query point and the partitioning point, along the partitioning axis
 - If the current best distance is less than this, since all nodes in the subtree will necessarily have a distance longer than this, the entire subtree can be eliminated
 - Otherwise, run the entire search recursively down the subtree and update the current best in the process
 3. When the root node has been checked, the algorithm terminates
- In OpenCV, `cv2.BFMatcher()` does a brute-force search; `cv2.FlannBasedMatcher()` does a k -D tree based approximate nearest neighbour matcher
 - `FlannBasedMatcher()` can be configured to stop with an approximate result, but for feature matching we often want the exact best neighbour
- Recently learning-based matching approaches have become more popular and can significantly outperform some classical methods
 - Superglue is the first to use a GNN to demonstrate this
 - First use Superpoint (learned CNN feature detector) to detect features in two images, then construct a graph out of them and use GNN to find correspondences
 - GNNs do surprisingly well on repeating patterns; they can use features even though they look similar since they also use geometry information

Incremental Tracking

- In most applications, the camera only moves a small amount between frames, so we can search a smaller space for potential matches
 - We might have e.g. a Kalman filter giving us an idea of where we moved to
- The detect-then-track approach to matching searches a small region in subsequent images for a feature that appeared in the initial image, which works well if motion and scene deformation (e.g. viewpoint) is small
 - Good features for matching are also good for tracking
 - For larger changes we can use additional strategies using e.g. multi-resolution search (to eliminate scale differences) or using motion prediction models (for objects or egomotion)
- The Kanade-Lucas-Tomasi (KLT) tracker is a full tracking algorithm
 - Predictive motion models are used to refine the search space
 - Patches are matched between frames using gradient information
 - An additional feature selection algorithm similar to Harris is used
 - Keep track of the feature dissimilarity between the current frame and the first frame in the sequence, and discard the feature when this grows too large (RMS residual)
 - * i.e. in every frame, add new features, and discard features that have grown too dissimilar to when they were first added

Lecture 8, Sep 26, 2025

Camera Pose Estimation (Perspective-n-Point)

- The *pose estimation problem* is the estimation of the pose of an object (or camera) from 3D-2D correspondences

- Typically we know where the points lie on the object in 3D, and we also know where those points appear in our image
- This is known as the *perspective-n-point* or PnP problem
 - * For 3 points, this is known as P3P
 - * 3 points is the minimum number of points we need to get a solution, but often we want to use more points to remove ambiguity and reject noise
- This process can be used to estimate the camera pose for extrinsic calibration; intrinsic parameters can be estimated at the same time
 - * This is how camera calibration with a checkerboard works
- There are linear and nonlinear algorithms for this
 - The linear case locally linearizes the problem and may not be very accurate if the initial guess is bad
 - The nonlinear algorithm uses an initial guess and iterates to find the solution
 - We often start from the linear algorithm and then use nonlinear algorithm to refine
- Solving for $\mathbf{P} = \mathbf{K} \begin{bmatrix} \mathbf{C} & \mathbf{t} \end{bmatrix}$ requires at least 6 correspondences (since there are 6 degrees of freedom), but if we have intrinsics already we only need 3
- The linear algorithm for solving PnP is the *direct linear transform* (DLT), which stacks a system of equations
 - We have $\mathbf{P} \in \mathbb{R}^{3 \times 4}$ (the combination of the extrinsic and intrinsic matrices) with 12 unknowns
 - For each correspondence we can construct 2 equations from it; with 6 correspondences we can solve the whole system
 - * $x_i = \frac{p_{00}X_i + p_{01}Y_i + p_{02}Z_i + p_{03}}{p_{20}X_i + p_{21}Y_i + p_{22}Z_i + p_{23}}$
 - * $y_i = \frac{p_{10}X_i + p_{11}Y_i + p_{12}Z_i + p_{13}}{p_{20}X_i + p_{21}Y_i + p_{22}Z_i + p_{23}}$
 - * Note the division due to normalization
 - Because the intrinsics matrix \mathbf{K} is upper-triangular, we can do a QR factorization on \mathbf{P} to recover the separate intrinsic and extrinsic matrices
- However, DLT does not impose constraints on the structure of the resulting matrices, namely the structure of the rotation matrix, so after QR factorization we often end up with a result that does not fit into our camera models, forcing us to make an approximation; this is why DLT is not an exact solution

Nonlinear Least Squares

- Nonlinear least squares (Gauss-Newton optimization) is an optimization approach we can use to solve this system
 - For nonlinear least squares, we wish to optimize $E(\mathbf{x}) = \frac{1}{2} \mathbf{e}(\mathbf{x})^T \mathbf{e}(\mathbf{x})$ where $\mathbf{e}(\mathbf{x})$ is some nonlinear function
 - * Note $\mathbf{e}(\mathbf{x}) = \mathbf{f}(\mathbf{x}) - \mathbf{y}$, i.e. the prediction minus the observation; the order is important, otherwise we end up maximizing the error instead!
 - * In the linear case we can substitute $\mathbf{e}(\mathbf{x}) = \mathbf{A}\mathbf{x} - \mathbf{b}$ and expand the error function, then take a derivative to obtain the normal equation
 - For the nonlinear case we linearize around an initial guess (the operating point) \mathbf{x}_{op}
 - $\mathbf{e}(\mathbf{x}) \approx \mathbf{e}(\mathbf{x}_{op}) + \mathbf{J}_e \delta \mathbf{x}$ where $\mathbf{J}_e = \left. \frac{\partial \mathbf{e}}{\partial \mathbf{x}} \right|_{\mathbf{x}_{op}}$ is the Jacobian and $\delta \mathbf{x}$ is a small deviation
 - * Now substitute this back into the error function and notice we get an expression very similar to the linear form, so we can use the same techniques to solve this
 - We can solve the linearized system, and add the $\delta \mathbf{x}$ to our initial operating point \mathbf{x}_{op} to get the next linearization point
 - Do this until our Jacobian becomes sufficiently small which means we have converged
 - We are essentially approximating the cost function as a quadratic at each step and optimizing the quadratic for a local solution
- Important notes for nonlinear least squares:

- Choosing good initial guesses is important, otherwise the optimization process can get trapped in a local minimum
- The states we are solving for must exist in a vector space (i.e. we cannot apply constraints, since then the vector space is no longer closed)
- For multiple errors, we sum over all the errors, so in the normal equation we sum over $\mathbf{J}^T \mathbf{J}$ and $\mathbf{J}^T \mathbf{e}$ and multiply in the end
 - $E(\mathbf{x}) = \frac{1}{2} \sum_{i=1}^N \mathbf{e}_i(\mathbf{x})^T \mathbf{e}_i(\mathbf{x})$
 - $\delta \mathbf{x}^* = - \left(\sum_{i=1}^N \mathbf{J}_{e_i}^T \mathbf{J}_{e_i} \right)^{-1} \left(\sum_{i=1}^N \mathbf{J}_{e_i}^T \mathbf{e}_i(\mathbf{x}_{op}) \right)$
 - Note we need to reevaluate the Jacobian for each measurement i , since the linearization point is all different!
- Often we have associated uncertainties for each error, so we can do a weighted version of least squares, so $E(\mathbf{x}) = \frac{1}{2} \sum_{i=1}^N \mathbf{e}_i(\mathbf{x})^T \mathbf{W}_i \mathbf{e}_i(\mathbf{x})$
 - Now we have $\delta \mathbf{x}^* = - \left(\sum_{i=1}^N \mathbf{J}_{e_i}^T \mathbf{W}_i \mathbf{J}_{e_i} \right)^{-1} \left(\sum_{i=1}^N \mathbf{J}_{e_i}^T \mathbf{W}_i \mathbf{e}_i(\mathbf{x}_{op}) \right)$
 - \mathbf{W}_i are symmetric matrices, one describing the weight for each measurement
 - Often we want scalar weights so \mathbf{W}_i for each measurement i is just a multiple of the identity
 - For vector-valued observations, we can use the inverse of the measurement covariance matrix Σ , known as the *information matrix*
- Note nonlinear least squares is equivalent to a maximum likelihood estimate if we assume our data has additive noise drawn from IID multivariate zero-mean Gaussians; weights are the information matrices of each noise Gaussian in this case
- To use NLS for pose estimation, we optimize the reprojection error $\mathbf{e}_i = \mathbf{x}_i - f(\mathbf{p}_i; \mathbf{C}, \mathbf{t}, \mathbf{K})$
 - f is our projection function which takes \mathbf{p}_i , converts it to the camera frame using the extrinsic calibration, then to pixel space using the intrinsics and normalizes it
 - But, rotation matrices have constraints, so we can't use Gauss-Newton as-is; we need to either express \mathbf{C} in a way that ensures it is a rotation matrix, or modify our update so that \mathbf{C} remains orthogonal
- This leads to the *Wahba problem*, which involves identifying a rotation matrix \mathbf{C} between frames given corresponding unit vector measurements $\mathbf{u}_i, \mathbf{v}_i$ in two frames
 - The cost function is $E(\mathbf{C}) = \frac{1}{2} \sum_{i=1}^N (\mathbf{C} \mathbf{u}_i - \mathbf{v}_i)^T (\mathbf{C} \mathbf{u}_i - \mathbf{v}_i) = \frac{1}{2} \sum_{i=1}^N \mathbf{e}_i(\mathbf{C})^T \mathbf{e}_i(\mathbf{C})$
 - Approach 1: Euler angles
 - * Optimize over 3 Euler angles instead
 - * However, this runs the risk of Gimbal lock – if our solution ends up near points with Gimbal lock, we run into numerical sensitivity issues or our solution may collapse entirely
 - This can work if we have good initial guesses
 - * Let $\mathbf{C}(\boldsymbol{\theta}) = \mathbf{C}_3(\theta_3) \mathbf{C}_2(\theta_2) \mathbf{C}_1(\theta_1)$, then $\mathbf{e}_i(\boldsymbol{\theta}_{op} + \delta \boldsymbol{\theta}) = \mathbf{e}_i(\boldsymbol{\theta}_{op}) + \mathbf{J}_{e_i} \delta \boldsymbol{\theta}$
 - * The Jacobian $\mathbf{J}_{e_i} = \frac{\partial \mathbf{C} \mathbf{u}_i}{\partial \boldsymbol{\theta}}$ can be computed in each column as follows:
 - $\frac{\partial \mathbf{C} \mathbf{u}_i}{\partial \theta_3} = [\mathbf{1}_3]_{\times} \mathbf{C}_3(\theta_3) \mathbf{C}_2(\theta_2) \mathbf{C}_1(\theta_1) \mathbf{u}_i$
 - $\frac{\partial \mathbf{C} \mathbf{u}_i}{\partial \theta_2} = \mathbf{C}_3(\theta_3) [\mathbf{1}_2]_{\times} \mathbf{C}_2(\theta_2) \mathbf{C}_1(\theta_1) \mathbf{u}_i$
 - $\frac{\partial \mathbf{C} \mathbf{u}_i}{\partial \theta_1} = \mathbf{C}_3(\theta_3) \mathbf{C}_2(\theta_2) [\mathbf{1}_1]_{\times} \mathbf{C}_1(\theta_1) \mathbf{u}_i$
 - Note $\mathbf{1}_i$ is a zero vector with 1 in the i th spot
 - Approach 2: use axis-angle
 - * To avoid gimbal lock we keep the operating point \mathbf{C}_{op} in matrix form, and consider a

- perturbation $\mathbf{C}(\delta\phi)$, so the update becomes $\mathbf{C}_{op} \leftarrow \mathbf{C}(\delta\phi)\mathbf{C}_{op}$
- * Recall the Rodrigues formula: $\mathbf{C}(\phi) = 1 + \sin \phi [\hat{\mathbf{n}}]_{\times} + (1 - \cos \phi) [\hat{\mathbf{n}}]_{\times}^2$
 - For small angles ϕ we approximate $\sin \phi = 0, \cos \phi = 1$
 - $\mathbf{C}(\delta\phi) \approx 1 + \delta\phi [\hat{\mathbf{n}}]_{\times} = 1 + [\delta\phi]_{\times}$
 - * Substitute the approximation for $\mathbf{C}(\delta\phi)$:
 - $\mathbf{e}_i(\delta\phi) = \mathbf{C}\mathbf{u}_i - \mathbf{v}_i$

$$= \mathbf{C}(\delta\phi)\mathbf{C}_{op}\mathbf{u}_i - \mathbf{v}_i$$

$$= (1 + [\delta\phi]_{\times})\mathbf{C}_{op}\mathbf{u}_i - \mathbf{v}_i$$

$$= \mathbf{C}_{op}\mathbf{u}_i - \mathbf{v}_i + [\delta\phi]_{\times}\mathbf{C}_{op}\mathbf{u}_i$$

$$= \mathbf{e}_i(\mathbf{C}_{op}) - [\mathbf{C}_{op}\mathbf{u}_i]_{\times}\delta\phi$$

$$= \mathbf{e}_i(\mathbf{C}_{op}) + \mathbf{J}_{e_i}\delta\phi$$
 - * Therefore the Jacobian is $\mathbf{J}_{e_i} = -[\mathbf{C}_{op}\mathbf{u}_i]_{\times}$, and with this we can use the normal equation to compute the update for $\delta\phi^*$ and update the operating point

Lecture 9, Sep 20, 2025

Stereo Vision

- *Stereo vision* is the technique of using 2 cameras with an offset (*baseline*) between them to infer depth, based on the differences in the appearances of objects between two images, similarly to how humans and animals perceive depth
 - Objects that are closer to the camera will appear to have moved more between the two images compared to objects further away; this effect is known as *parallax*
 - This process is known as *stereo triangulation*
 - Stereo can produce a depth value for almost every pixel in the image, as long as it can be matched across the images (which is often a difficult process)
- To calculate the depth of a pixel, we need to find and match its location between the two images, a process known as *stereo matching*
 - We can reduce the search space for matching using *epipolar constraints*, which arise from the geometry of the cameras
- Matching is made feasible due to *epipolar constraints* arising from the geometry of the problem
 - We can project the optical centre of one camera to the other camera to get the *epipole* or *epipolar point*
 - * Another way to think about this is where the line connecting the two camera centres intersect the imaging planes
 - Each point on one camera's image plane projects into an entire *epipolar line* on the other camera's image plane
 - * This applies for any orientation of the cameras
 - Therefore to do matching for a point p , we find its epipolar line in the other camera's image plane and search along this line
 - The epipole forms an *epipolar plane* along with the epipolar line of a pixel; projecting this plane onto the second camera gives the corresponding epipolar line in the second imaging plane, which is the line we will search on
- We can constrain the problem further and make the epipolar lines horizontal by applying *rectification*
 - Applying rectification and then searching along only a horizontal line makes our algorithms significantly faster and more accurate
 - This results in a *fronto-parallel* (aka *standard rectified*) plane geometry, where both image planes are parallel (looking forward) at the same depth and rotationally aligned
- The matching process for rectified images produces a *disparity* value $d = x_0 - x_1$, which is the difference between the x coordinates of the point between the two images
 - The disparity is inversely proportional to the depth

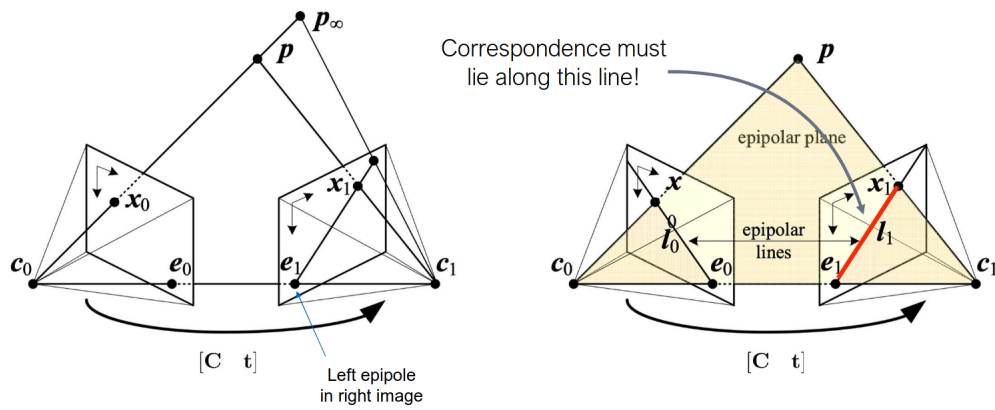


Figure 29: Epipolar constraints.

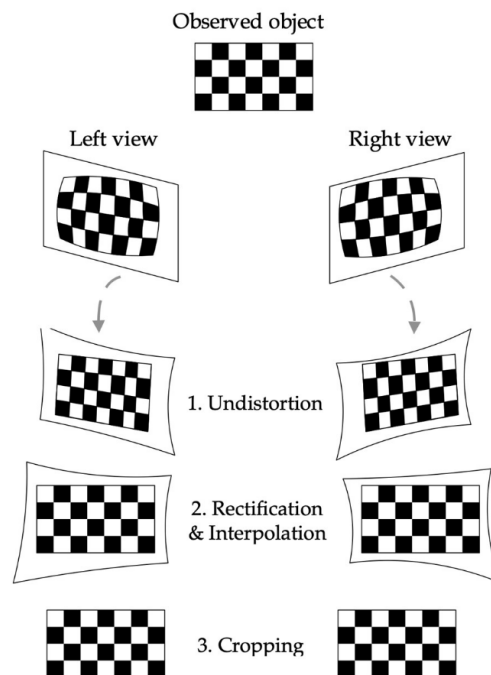


Figure 30: Typical stereo pre-processing pipeline, involving first undistortion, then rectification, and finally cropping to make both images rectangular.

- We can recover the depth using the disparity, baseline (distance between cameras), and focal distance using simple geometry
- By the geometry of the problem we will always get $x_0 > x_1$ since the second camera is on the right

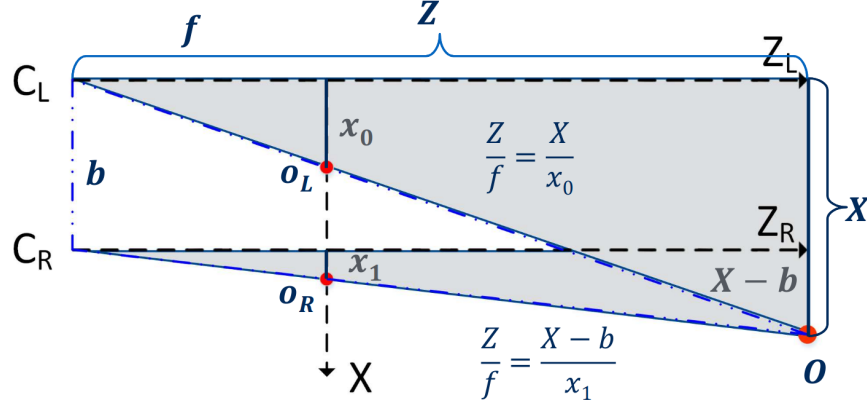


Figure 31: Geometry of the stereo camera model looking down, used to determine depth from disparity.

- We can derive the relation using similar triangles:
 - $\frac{Z}{f} = \frac{X}{x_0} \implies Zx_0 = fX$ for the triangle formed in the first image
 - $\frac{Z}{f} = \frac{X-b}{x_1} \implies Zx_1 = fX - fb$ for the triangle in the second image
 - $Zx_1 = Zx_0 - fb$
 - This gives rise to the final stereo equation $Z = \frac{fb}{d}$
 - We can also recover the X and Y location as $X = \frac{Zx_0}{f}, Y = \frac{Zy_0}{f}$
 - * This relation with the focal length means we can see depth in more detail using a long focal length camera (telephoto lens)
- This gives rise to the stereo camera model $\begin{bmatrix} x_0 \\ y_0 \\ d \end{bmatrix} = \mathbf{f}(\mathbf{p}_c) = \frac{f}{Z} \begin{bmatrix} X \\ Y \\ b \end{bmatrix} + \begin{bmatrix} c_x \\ c_y \\ 0 \end{bmatrix}$
 - Note the convention is to have the stereo camera frame be the left camera frame
- Observations about the properties of disparity:
 - The disparity is inversely proportional to depth
 - * The further a point, the smaller its disparity, and therefore the more noise in our measurement
 - * Points at infinity have no disparity
 - The disparity is proportional to the baseline
 - * Having a larger baseline improves measurement accuracy, but the overlapping FOV of the cameras decreases and might make matching difficult
- Typically disparity values are integers, but for longer range subpixel disparity can be critical
 - For probabilistic models, there is a disparity-dependent bias, i.e. there is a natural bias to estimate depth closer than it actually is
 - For narrow baseline values, the geometry means that we have a lot of depth noise; Gaussian approximations for the uncertainty are less accurate since we have a long “tail” (the uncertainty in front of the object is over-approximated, while the uncertainty behind is under-approximated)
 - * If we approximate as a Gaussian (e.g. for an EKF), we tend to think the object is actually closer
 - At wider baseline values the uncertainty is smaller and more Gaussian-like

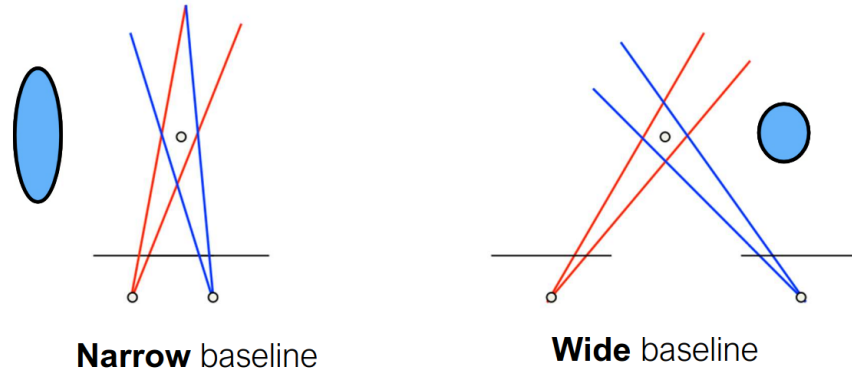


Figure 32: Comparison of the depth uncertainty for different baseline distances.

Stereo Matching Algorithms

- In robotics we often want dense depth, i.e. depth for every pixel, so we can't use only sparse feature matching
 - This can be fundamentally unreliable for textureless regions
- *Local* algorithms match only finite regions along the epipolar line while *global* algorithms aggregate information over the whole image
 - Due to speed constraints local methods are much more common
 - Recently (2020s) many learning-based matching algorithms have been proposed
- We need to define a similarity measure between pixels for matching
 - Sum-of-squared-differences and sum-of-absolute-differences are commonly used, basic (fast) similarity measures
 - Normalized cross-correlation and gradient-based techniques can be better for rejecting camera gain and bias differences
- Local methods slide a window over the epipolar line and compute the score at each point, which can be performed using a convolution $C(x, y, d) = w(x, y, d) * C_0(x, y, d)$
 - This can be accelerated using integral images for moving average box filters
 - Usually the pixel is just matched to the single pixel in the other image with the best similarity
- Larger windows result in smoother, less noisy images, and are less susceptible to *foreshortening*, but are less detailed, so the window size needs to be balanced
 - *Foreshortening* is the effect of oblique objects appearing as different sizes in the two images due to perspective change
 - This effect is more pronounced the larger the baseline is
 - Foreshortening means that a fixed window size is not always reliable, since it maps to different sized patches in real life for the two images
- Block matching and semi-global depth matching are some of the classic baseline methods, and are implemented in OpenCV
 - Often a good place to start!
- Modern state-of-the-art methods are learning-based, often using CNNs
 - This combines local matching with semi-global aggregation, e.g. GA-Net (2019)
 - Learning-based methods often produce depth maps that are smoother and with fewer holes, since the network learns to predict and fill in places where matches are bad
- Common standard datasets for stereo matching:
 - Tsukuba stereo dataset (1997): original dataset for stereo; contains ground truth from laser scanners
 - Middlebury stereo datasets (2001, 2014): contains sub-pixel accurate ground truth via structured light sensor
 - KITTI stereo (2012, 2015): outdoor dataset, with ground truth from interpolated LIDAR and



Figure 33: Comparison of the effect of window size. Middle image has a window size of 3, right image has a window size of 20.

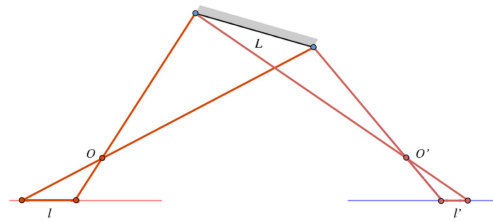


Figure 34: Illustration of foreshortening.

- CAD model fitting for segmented cars; only ~200 images for training
- Scene flow datasets (2016): synthetic datasets rendered in Blender, containing disparity and optical flow; a very large dataset designed for neural network training
 - * Beware of sim-to-real gap!

Lecture 10, Oct 3, 2025

Review Lecture

- Important content by lecture:
 - Lecture 2 (mathematical foundations)
 - * Different types of linear transformations and what properties they preserve, what situations they come up in
 - * Rotations in 3D (properties of $SO(3)$) and representations (matrix, Euler, axis-angle, quaternion)
 - Lecture 3 (probability and regression)
 - * Conditional probability (Bayes' rule)
 - * Linear regression, standard linear least squares formulation and solution
 - * RANSAC algorithm: working principles, expected/required number of trials to get an outlier-free sample with some probability
 - Lecture 4 (optics)
 - * Ideal pinhole camera model
 - * Definition of the camera reference frame, optical axis, image plane, principal point
 - * Projective map
 - * Camera matrices (intrinsics and extrinsics), projecting onto the image plane
 - * Lens distortion model
 - * Optical effects that degrade images (vignetting, other effects)
 - Lecture 5 (image operations)
 - * Point operations (thresholding, brightness, contrast, gamma adjustment, histogram equalization)

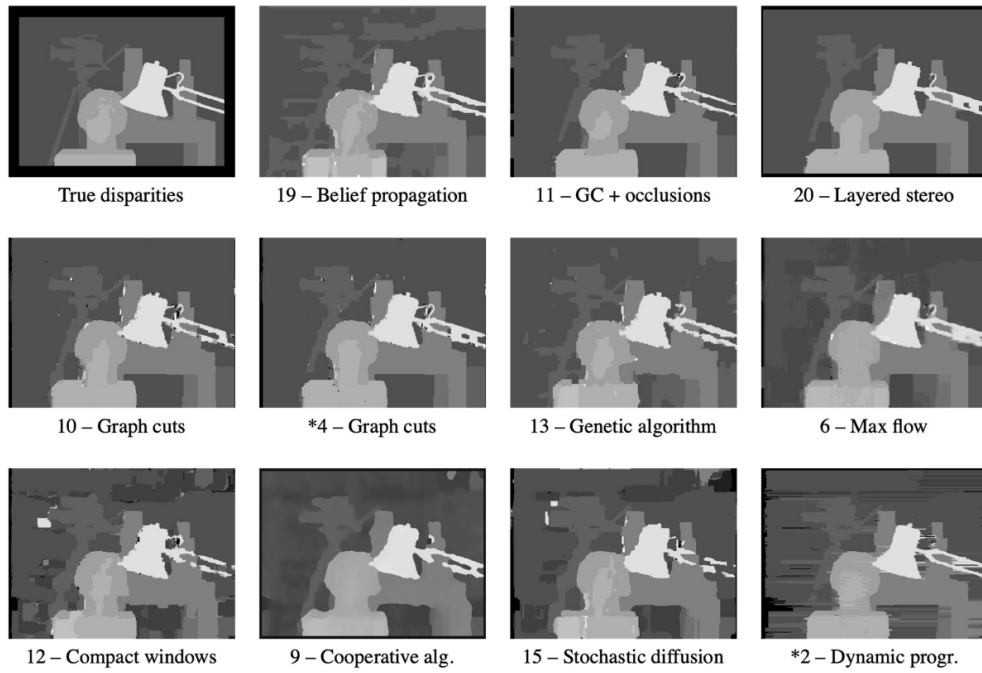


Figure 35: Comparison of some classic stereo matching algorithms (circa 2002), part 1.

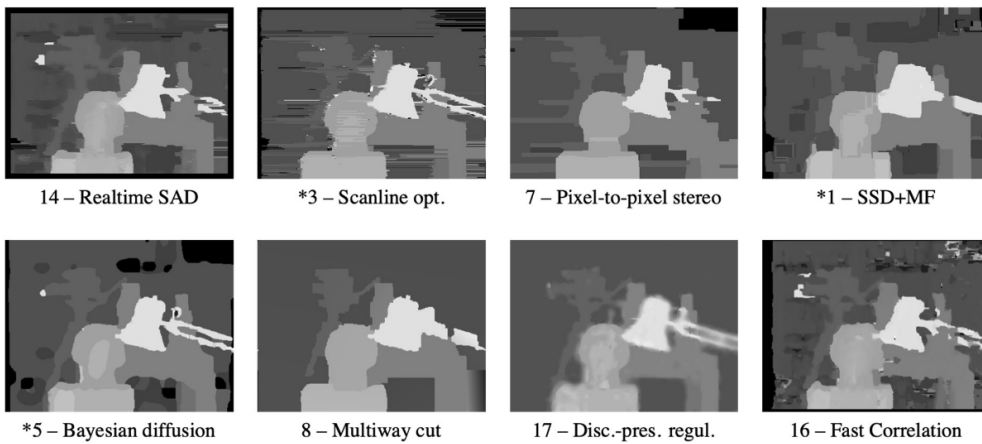


Figure 36: Comparison of some classic stereo matching algorithms (circa 2002), part 2.

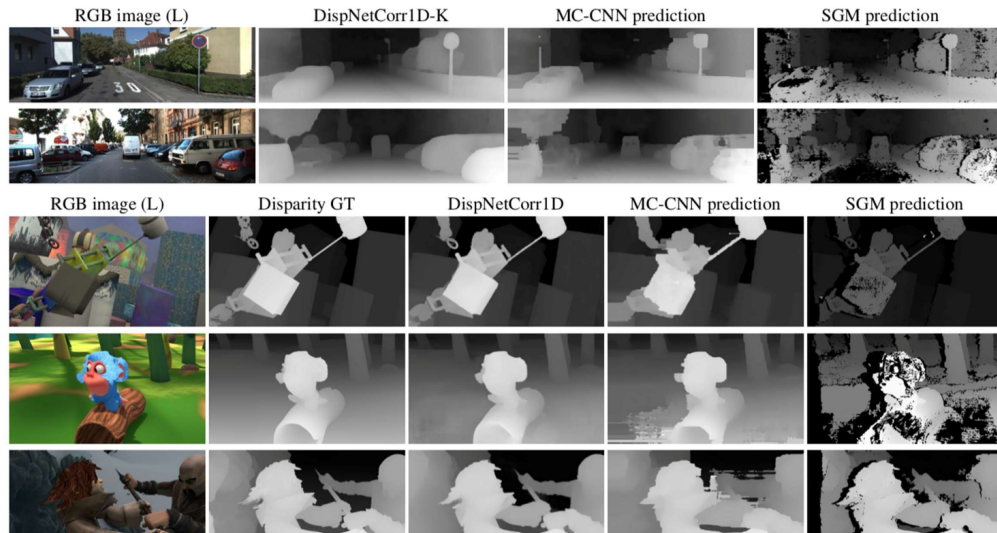


Figure 37: Comparison of some modern stereo matching algorithms (circa 2016).

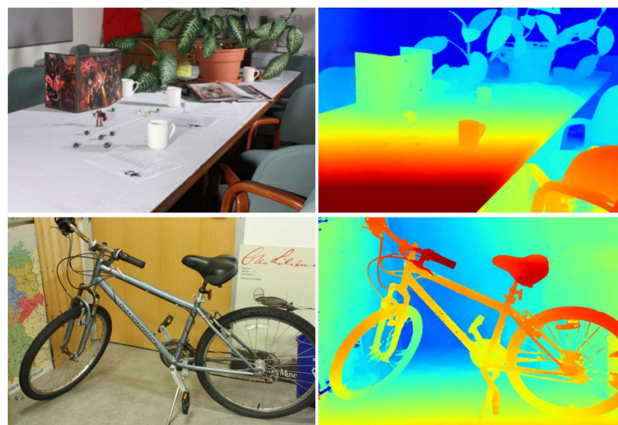


Figure 38: Results from GA-Net (2019) stereo matching.

- * Linear filtering (convolutions, separable filters)
- * Nonlinear filtering (band-pass, bilateral)
- * Geometric transformations (viewpoint transformation and bilinear interpolation)
- * Regularization?
- Lecture 6 (image features: detection and description)
 - * Important characteristics of features (saliency, locality, repeatability, compactness)
 - * Various feature detectors, classical (Harris, SIFT, SURF, FAST, BRISK, BRIEF, ORB) and learned (LIFT)
- Lecture 7 (image features: matching)
 - * Feature descriptor distance functions (SSD between patches, hamming distance, Euclidean distance)
 - * Rejecting outliers (RANSAC, ratio test)
 - * Binary classification evaluation (confusion matrix, ROC curve)
 - * Matching techniques (hashing, k -D trees)
 - * KLT tracker (local matching) as alternative to matching through the whole image
- Lecture 8 (camera pose estimation) (★)
 - * Perspective-n-point: problem definition
 - * Direct linear transform to solve the homography (as an approximation)
 - * Nonlinear least squares (iterative algorithm)
 - * Regressing rotations (Wahba problem, Euler angles and rotation matrix/axis-angle formulation)
- Lecture 9 (stereo)
 - * Epipolar geometry (epipolar planes, lines, epipolar point)
 - * Stereo preprocessing pipeline (stereo rectification)
 - * Stereo camera model (depth from disparity)
 - * Foreshortening problem
 - * Basic idea of stereo matching algorithms (local vs. global algorithms, window size)
 - * Learning based state-of-the-art methods

Lecture 11, Oct 10, 2025

Camera Calibration

- *Camera calibration* is the process of characterizing the geometric and photometric properties of the imaging system in use
 - Geometric detection, e.g. determining \mathbf{K} , is what we are primarily concerned with; photometric calibration (e.g. getting consistent brightness and colour values across pixels) is less important since we use techniques such as feature detection, which are invariant to photometric properties
 - This is done by using a calibration pattern with a known geometry, then solving for the camera parameters using the known quantities of the target
- The most common target for geometric calibration is a flat checkerboard pattern
 - Critical to keep the checkerboard flat and include a variety of views (multiple angles, distances, fully covering the image plane)
 - Other patterns exist, e.g. intersecting orthogonal planes, circle patterns, etc
 - Checkerboard is good because the cross junctions are not affected by foreshortening, and the simple square pattern is easy to detect, allowing for pixel or subpixel-level accuracy
 - For a circle, the effect of foreshortening distorts it into an ellipse, but critically the centre of this ellipse is not the same as the original centre of the circle
- Recall our forward imaging model: $\mathbf{x}_{ij} = f(\mathbf{p}_i; \mathbf{C}_j, \mathbf{t}_j, \mathbf{K}, \boldsymbol{\kappa}, \boldsymbol{\tau})$ for point i in camera j
 - For calibration, we want to recover all of these parameters; most often we're interested in $\mathbf{K}, \boldsymbol{\kappa}, \boldsymbol{\tau}$ but all parameters need to be estimated to solve the problem
 - The n -planes approach is to take n images of our known calibration target, so for each image we get a different set of extrinsic parameters
 - * This means we need at least a minimum number of points per image for this to work
- Targetless calibration and online calibration (as the robot is operating) is also possible, but much harder;

most often we want to solve the calibration problem given known positions of landmark points in some frame (usually the target's frame)

- DLT can be used, but it does not handle radial or tangential distortion, so it will be a very bad approximation
- DLT also doesn't work when all points are coplanar, which is often the case
- We can often get an initial guess and use NLS, using all feature points from all views in a big single optimization process
 - The structure of the Jacobian can be used for some speedups, but practically computers are already fast enough
 - For NLS, we optimize the sum of squared reprojection errors, i.e. predicted location minus observed location
 - $E_{NLS}(\delta\theta) = \sum_{ij} \left\| \frac{\partial f}{\partial \mathbf{C}_j} \delta \mathbf{C}_j + \frac{\partial f}{\partial \mathbf{t}_j} \delta \mathbf{t}_j + \frac{\partial f}{\partial \mathbf{K}} \delta \mathbf{K} + \frac{\partial f}{\partial \kappa} \delta \kappa + \frac{\partial f}{\partial \tau} \delta \tau - \mathbf{r}_{ij} \right\|^2$
 - * Note the partials should be taken with respect to each parameter individually (a bit of abuse of notation, especially for the matrix...)
- In the Jacobian, only the extrinsics for one measurement affects that measurement, but the intrinsics and distortion parameters affect all measurements, so we get a sparse structure like the following figure

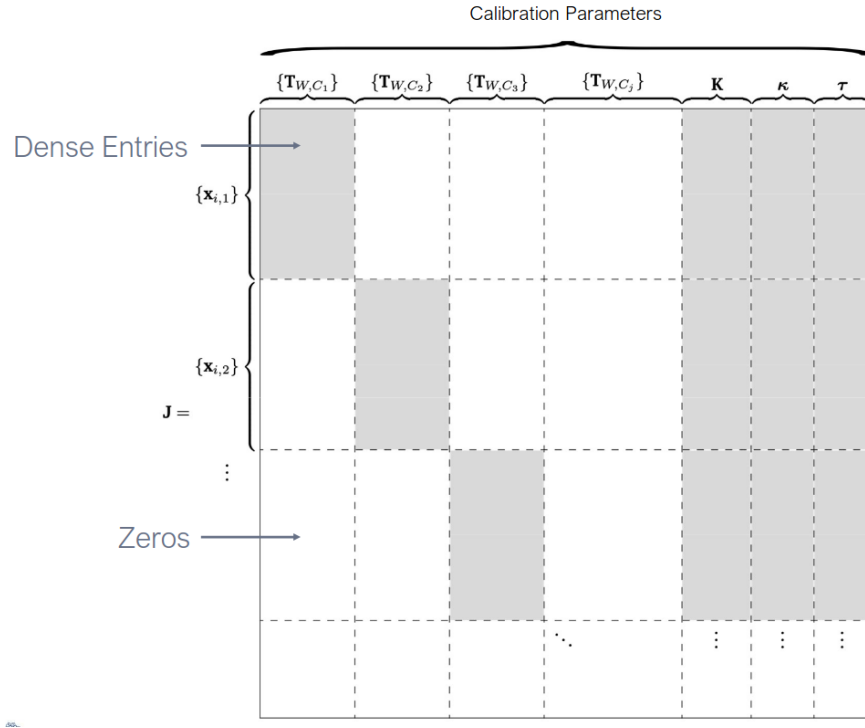


Figure 39: Structure of the Jacobian matrix.

- There are many existing toolboxes for calibration: MATLAB, OpenCV, ROS1/ROS2
- For stereo calibration, we need to add a rigid body transformation between the cameras to our optimization parameters
 - We can either do intrinsics for the 2 cameras separately, or include both intrinsics along with the transformation between cameras in a big optimization
 - For stereo the projection matrices have the following form:

$$\begin{aligned}
 * \tilde{\mathbf{P}}_L &= \begin{bmatrix} \mathbf{K}_L^L & 0 \\ \mathbf{0}^T & 1 \end{bmatrix} \begin{bmatrix} \mathbf{C}_{L,W}^L & \mathbf{t}_{L,W}^L \\ \mathbf{0}^T & 1 \end{bmatrix} = \tilde{\mathbf{K}} \mathbf{T}_{L,W} \\
 * \tilde{\mathbf{P}}_R &= \begin{bmatrix} \mathbf{K}_R^R & 0 \\ \mathbf{0}^T & 1 \end{bmatrix} \begin{bmatrix} \mathbf{C}_{R,L}^R & \mathbf{t}_{R,L}^R \\ \mathbf{0}^T & 1 \end{bmatrix} \begin{bmatrix} \mathbf{C}_{L,W}^L & \mathbf{t}_{L,W}^L \\ \mathbf{0}^T & 1 \end{bmatrix} = \tilde{\mathbf{K}} \mathbf{T}_{R,L} \mathbf{T}_{L,W}
 \end{aligned}$$

- * Multiply a world point $\hat{\mathbf{p}}_i$ by these matrices to get the corresponding pixel locations in the left or right camera

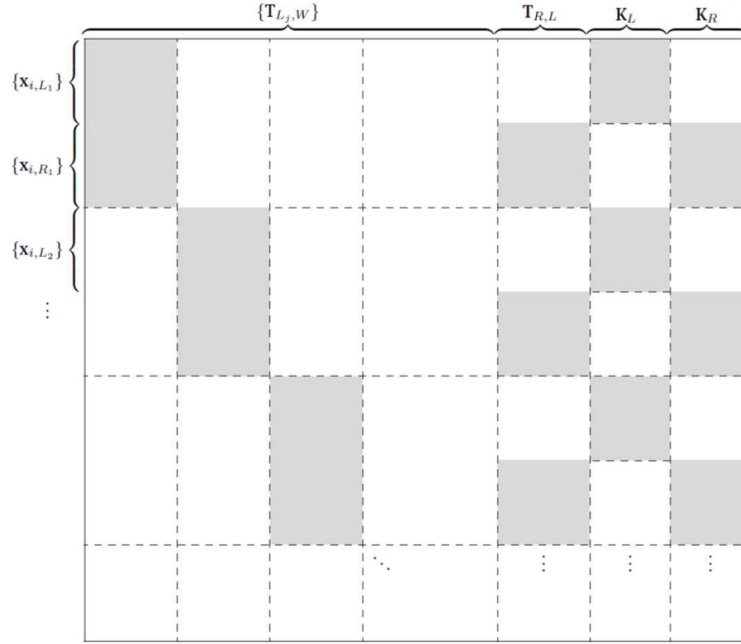


Figure 40: Structure of the Jacobian matrix for stereo calibration (note distortion parameters are left out).

Lecture 12, Oct 14, 2025

Structure From Motion and Bundle Adjustment

- *Structure from Motion* (SfM) is a problem of determining the 3D positions of landmark points (structure) and camera poses (motion) from a set of sparse image feature correspondences
 - The process of the optimization depends on whether we know the camera poses
 - If camera positions are unknown and no other data is available, everything can only be known up to scale
 - With known camera poses or other measurements, we can recover the scale
 - The process usually involves:
 1. Take lots of cameras of the object
 2. Identify features and match across images
 3. Apply joint optimization to find camera poses and 3D positions of feature points (and possibly other camera parameters)
- *Triangulation* is the process of finding the 3D position of a landmark point from multiple images, with known camera poses
 - Given 2D correspondences $\{\mathbf{x}_j\}$ and cameras $\{\mathbf{P}_j = \mathbf{K}_j [\mathbf{C}_j \quad \mathbf{t}_j]\}$ with optical centres $\{\mathbf{c}_j\}$, the goal is to find the point \mathbf{p} closest to all of the rays from each camera
 - Each ray has direction $\mathbf{v}_j = \mathbf{C}_j^{-1} \mathbf{K}_j^{-1} \bar{\mathbf{x}}_j$ (need to be normalized to get $\hat{\mathbf{v}}_j$)
 - There will always be some error, so the rays will never perfectly intersect, so we need to find the point that is the closest to all the rays
 - For each ray we try to find the distance from camera centre, d_j , to minimize the error $\|\mathbf{c}_j + d_j \hat{\mathbf{v}}_j - \mathbf{p}\|^2$
 - * Solution is $d_j = \hat{\mathbf{v}}_j \cdot (\mathbf{p} - \mathbf{c}_j)$
 - * The closest point on the ray to \mathbf{p} will be $\mathbf{q}_j = \mathbf{c}_j + (\hat{\mathbf{v}}_j \hat{\mathbf{v}}_j^T)(\mathbf{p} - \mathbf{c}_j)$
 - Now we can use all the \mathbf{q}_j in a least squares optimization to find \mathbf{p} , optimizing for the squared error between the \mathbf{q}_j and \mathbf{p}

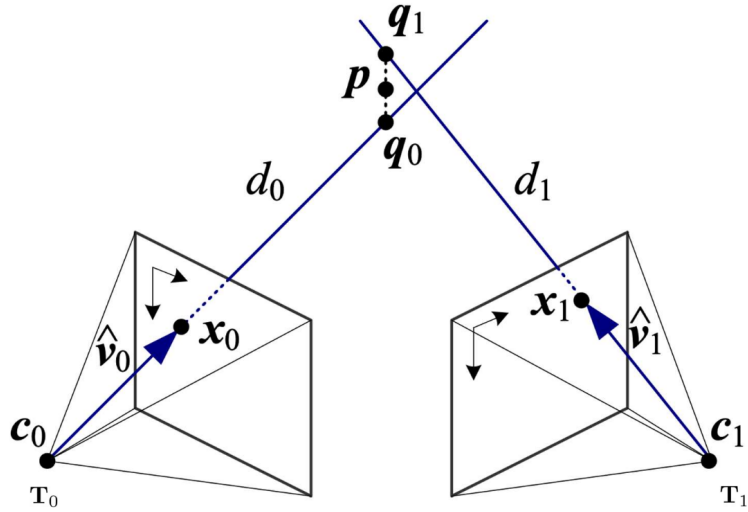


Figure 41: Structure of the triangulation problem.

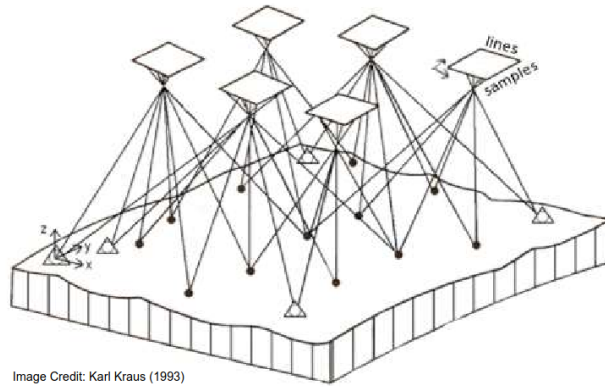


Image Credit: Karl Kraus (1993)

Figure 42: Visualization of the bundle adjustment problem. The top are the imaging planes for the cameras and the intersections are the camera centres; the bottom points are image features. The triangular markers are man-made landmarks that provide a failsafe correspondence, which can have more weight in optimization.

- *Bundle adjustment* (BA) is the problem of jointly optimizing the 3D structure and viewing parameters (camera poses and possibly calibration parameters)
 - The name comes from “adjusting” the “bundles” of rays coming from each camera
 - This is often a very large but sparse parameter optimization problem
 - We can solve this through nonlinear least squares
 - Modern techniques use *M-estimators* for outliers since they can significantly affect the result
- BA originated from *photogrammetry*, the science of measurement from photographs, often used in surveying
 - Early photogrammetry was analog, using devices such as the stereoplotter, a device that allows a human to look at 2 images in a stereo setup (anaglyph) and plot out a contour
- Since there are thousands of parameters involved, we need to take advantage of the structure of the optimization problem to make it tractable
 - Naive NLS results in cubic complexity
 - Most of the time we have sparse Jacobians and Hessians ($\mathbf{J}^T \mathbf{J}$), and we can exploit this by using techniques such as sparse Cholesky factorization
 - * We get a block between features when they can be seen together, and a block between a feature and a camera when the feature is visible from that camera
 - * In the Hessian, the location of features don’t affect each other directly (only indirectly through camera pose), so we have a somewhat diagonal structure
 - By ordering the parameters carefully we can make the matrix sparser (by grouping relevant parameters together to form a locally dense block), which improves performance
 - * Global parameters should be grouped (if optimized at all)

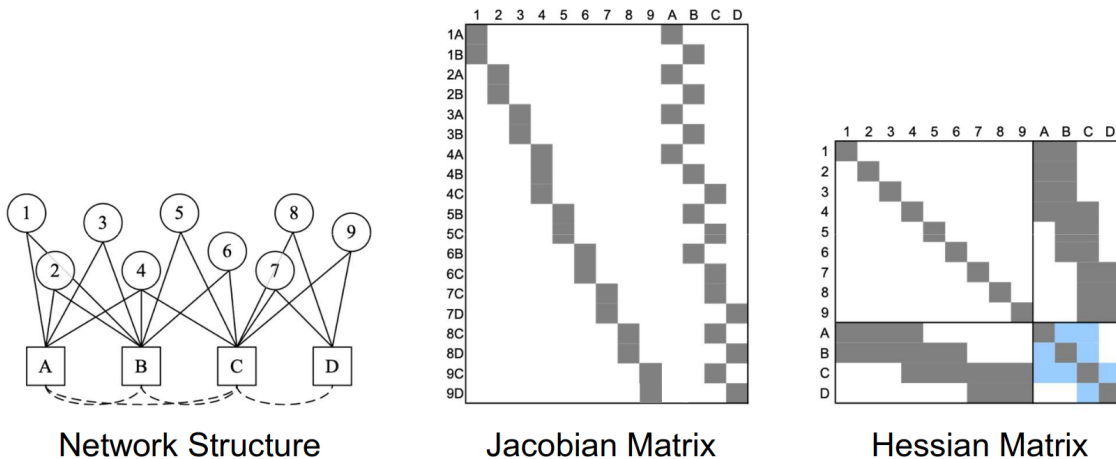


Figure 43: Structure of the Jacobian and Hessian for bundle adjustment. Dark squares indicate a dense block while white indicates zeros.

- If we add a new image/camera to an existing solution, we can use techniques such as the EKF to apply incremental updates
- For very large problems, there are techniques to reduce the computational complexity
 - Sliding window adjustment optimizes only the parameters within the window (e.g. temporal), instead of optimizing the global problem at every step
 - * Past measurements are marginalized away and the uncertainty incorporated in the environment representation
 - Keyframe-based algorithms only optimize a subset of relevant keyframes
 - * Keyframe determination can be based on spatial or temporal or feature-overlap thresholds
 - Both of these are often used in SLAM
- We can use different parametrizations for points and angles, which have a big impact on the numerical stability of computations
 - Ideally parameterizations should be locally linear for the chosen error model, so NLS has an easier

- time (i.e. we don't want dramatic changes in curvature)
- For points, we parameterize as $\tilde{\mathbf{x}} = (\tilde{x}, \tilde{y}, \tilde{z}, \tilde{w})$ to incorporate points at infinity
 - * Landmark points should be kept in homogeneous coordinates to prevent numerical instability, as we often get points near infinity in real life
- For rotations, the best choices are quaternions or rotation matrices (both subject to constraints)
 - * Quaternions in particular are numerically stable and compact
 - * Incremental updates can use any 3-parameter parametrization, e.g. Euler, Rodriguez (axis-angle), etc
- The optimization can be extremely sensitive to outliers, so we need robust error metrics and cost functions
 - RANSAC can be used to eliminate some outliers, but there might still be some remaining
 - * Sometimes the line between the two can be blurred, e.g. in the case of points that are close or in repeating patterns
 - * RANSAC requires knowing the outlier ratio and suitable thresholds, which makes it imperfect at filtering outliers
- We need to use *robust estimators*, where the cost of outliers is down-weighted, i.e. the estimator ignores data points if they appear to be an outlier
 - We need more than just a Gaussian to handle outliers, since Gaussians have short tails; outliers necessitate distribution with longer tails
 - When points get too far away, *robust cost functions* have a zero or small gradient so the outliers no longer affect the optimization
 - * They will look like the quadratic (L2) loss near the origin, but when further away the error flattens out to give less weight to outliers
 - * Note this means we need to make assumptions about convergence – in this case if we start too far away, we may end up in the flat region of the gradient and be unable to converge

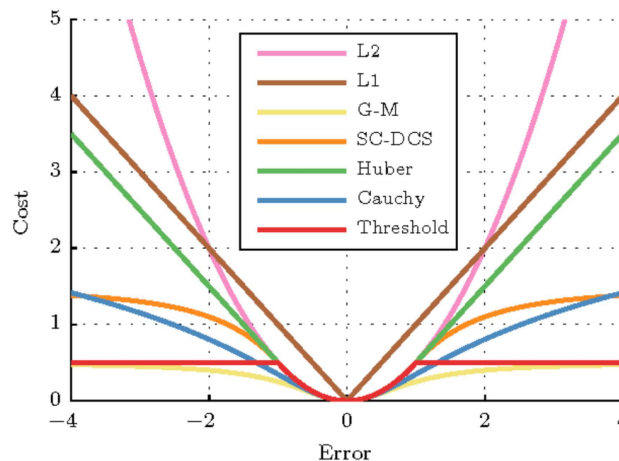


Figure 44: Comparison of different cost functions.

Lecture 13, Oct 17, 2025

Incremental SfM and Visual Odometry

- Suppose we're processing images as they arrive, can we build up the SfM problem incrementally instead of all at once as in bundle adjustment?
- This problem is known as *incremental SfM*, or *visual odometry* (VO) and *visual SLAM*
 - Visual odometry only computes the positions, while in visual SLAM we also build up a scene representation and potentially incorporates loop closures
 - Visual odometry methods compute the *egomotion* between frames, i.e. the motion of the cam-

- era/robot platform relative to the world
- We will discuss stereo visual odometry, since we need depth information when doing frame-to-frame motion estimation
- Visual odometry can be used in environments where normal wheel odometry is infeasible/inaccurate, e.g. high-slip environments like sand, aerial vehicles, or where we require higher short-term accuracy than wheel odometry can offer
- VO algorithms can be grouped into 4 categories, depending on whether they use monocular or stereo images, and whether they use image features or directly use pixel intensities
 - Feature-based VO algorithms run feature extraction and matching, while intensity based methods directly try to match pixel intensities from one image to the next
 - Importantly, stereo measurements have an invertible observation model, so we can take points in one frame and use the inverse model to calculate where we should expect it in the next frame
- Define the combined stereo camera model (note the origin is defined as the middle of the cameras, so the cameras centres are at $x = \pm b/2$)

$$- \text{ Assuming same intrinsics for both cameras, } \mathbf{M} = \begin{bmatrix} f_u & 0 & c_u & f_u \frac{b}{2} \\ 0 & f_v & c_v & 0 \\ f_u & 0 & c_u & -f_u \frac{b}{2} \\ 0 & f_v & c_v & 0 \end{bmatrix}, \mathbf{p} = \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

- The forward model is $[u_l \ v_l \ u_r \ v_r]^T = \mathbf{f}(\mathbf{p}) = \mathbf{M} \frac{1}{z} \mathbf{p}$ where (u_l, v_l) and (u_r, v_r) are the rectified pixel coordinates

$$* \text{ Jacobian given by } \frac{\partial \mathbf{f}}{\partial \mathbf{p}} = \mathbf{M} \frac{1}{p_3} \begin{bmatrix} 1 & 0 & -p_1/p_3 & 0 \\ 0 & 1 & -p_2/p_3 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & -p_4/p_3 & 1 \end{bmatrix}$$

$$- \text{ Inverse model: } \mathbf{p} = \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \mathbf{g}(y) = \frac{b}{u_l - u_r} \begin{bmatrix} \frac{1}{2}(u_l + u_r) - c_u \\ \frac{f_u}{f_v} \left(\frac{1}{2}(v_l + v_r) - c_v \right) \\ f_u \end{bmatrix}$$

- * This is only possible to do for a stereo camera since we have depth information

$$* \text{ Jacobian: } \frac{\partial \mathbf{g}}{\partial \mathbf{y}} = \frac{b}{(u_l - u_r)^2} \begin{bmatrix} -u_r + c_u & 0 & u_l - c_u & 0 \\ -\frac{f_u}{f_v} \left(\frac{1}{2}(v_l + v_r) - c_v \right) & \frac{f_u}{2f_v} & \frac{f_u}{f_v} \left(\frac{1}{2}(v_l + v_r) - c_v \right) & 0 \\ \frac{f_u}{2f_v} & -f_u & 0 & f_u \end{bmatrix}$$

- The VO problem involves finding the relative transformation between frames \mathbf{T}_{ba} , given an image captured in frame a and an image in frame b
- In a feature-based stereo VO pipeline, we run the standard stereo matching, then feature detection and matching of features between frames; then we can do outlier rejection (very important) and then solve for the pose transformation, potentially incorporating other sensors into the mix
- To compute the pose transformation between frames, we can try to align the point clouds we get from the two frames (since the stereo model is invertible)
 - Assuming we know correspondences perfectly, there is a closed-form solution
 - Let $\mathbf{p}_a^j, \mathbf{p}_b^j$ be the points in frames \mathcal{F}_a and \mathcal{F}_b respectively; we wish to minimize a squared cost function with respect to the rotation \mathbf{C}_{ba} and translation \mathbf{r}_a
 - $E(\mathbf{C}_{ba}, \mathbf{r}_a) = \frac{1}{2} \sum_{j=1}^J w_j \left(\mathbf{p}_b^j - \mathbf{C}_{ba}(\mathbf{p}_a^j - \mathbf{r}_a) \right)^T \left(\mathbf{p}_b^j - \mathbf{C}_{ba}(\mathbf{p}_a^j - \mathbf{r}_a) \right)$
 - Each data point can have a weight, usually based on our confidence of that feature; this can be a scalar or matrix weight
- Scalar weighted point cloud alignment has an exact solution without iteration:

Devon Island 2008

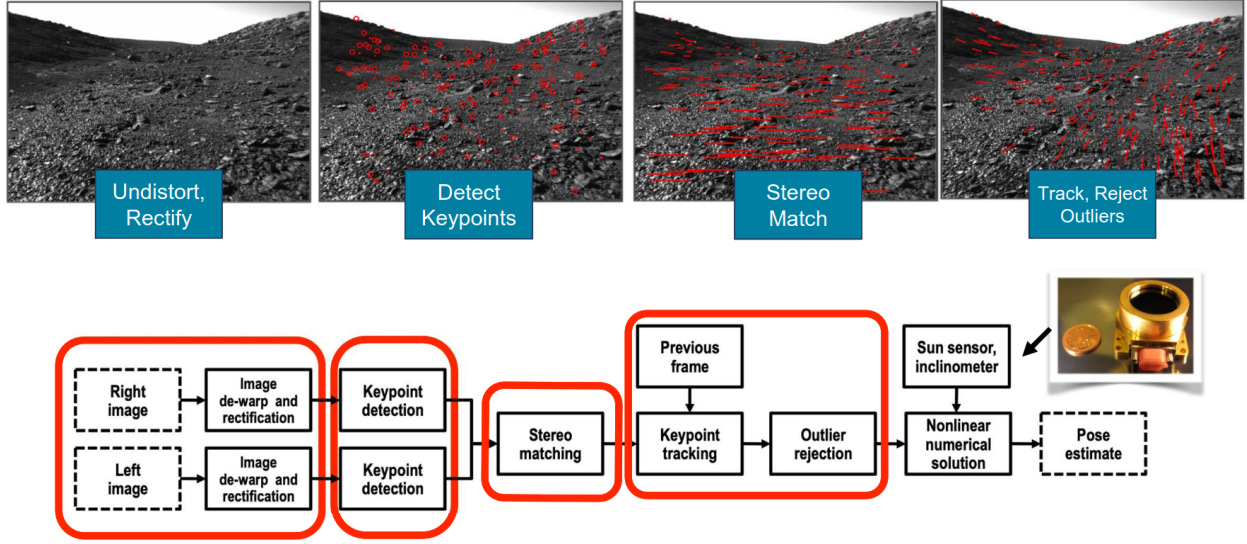


Figure 45: Feature-based stereo VO pipeline.

1. Compute centroids $\mathbf{p}_a = \frac{\sum_{j=1}^J w^j \mathbf{p}_a^j}{\sum_{j=1}^J w^j}$, $\mathbf{p}_b = \frac{\sum_{j=1}^J w^j \mathbf{p}_b^j}{\sum_{j=1}^J w^j}$
 - The idea is to align the two point cloud centroids, so we only have an alignment (rotation) to solve for
2. Compute the outer product matrix $\mathbf{W}_{ba} = \frac{1}{\sum_{j=1}^J w^j} \sum_{j=1}^J w^j (\mathbf{p}_b^j - \mathbf{p}_b)(\mathbf{p}_a^j - \mathbf{p}_a)^T$
3. Compute the SVD of the outer product matrix, $\mathbf{V} \mathbf{S} \mathbf{U}^T = \mathbf{W}_{ba}$
4. Compute the rotation using the SVD, and then use the inverse of the rotation to compute the correct translation vector
 - $\mathbf{C}_{ba} = \mathbf{V} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & \det(\mathbf{U}) \det(\mathbf{V}) \end{bmatrix} \mathbf{U}^T$
 - $\mathbf{r}_a = -\mathbf{C}_{ba}^T \mathbf{p}_b + \mathbf{p}_a$
 - Notice the translation is just subtracting the centroids (in the correct frame)
- Bonus: this can be used to align point clouds without registration by iterating, which is the ICP algorithm
- Due to the stereo geometry, we typically have much higher depth uncertainty than in x and y , and further points have more uncertainty; using simple scalar weights does not capture this complexity, so it's often inaccurate
 - Consider a Gaussian noise model (ignoring for now the long tail we discussed before)
 - $\mathbf{y} = \mathbf{f}(\mathbf{p}) + \mathbf{n}$, $\mathbf{n} \sim \mathcal{N}(\mathbf{0}, \mathbf{R})$ where $\mathbf{R} \in \mathbb{R}^{4 \times 4}$ is the pixel measurement noise covariance (in both cameras)
 - Linearize the inverse model: $\hat{\mathbf{p}} = \mathbf{g}(\mathbf{y}) = \mathbf{g}(\mathbf{f}(\mathbf{p}) + \mathbf{n}) \approx \mathbf{p} + \frac{\partial \mathbf{g}}{\partial \mathbf{y}} \mathbf{n}$
 - Therefore $\hat{\mathbf{p}} \sim \mathcal{N}(\mathbf{p}, \mathbf{G} \mathbf{R} \mathbf{G}^T)$ where $\mathbf{G} = \frac{\partial \mathbf{g}}{\partial \mathbf{y}} \Big|_{\mathbf{y}}$
- Now we can use the landmark covariances in the optimization as matrix weights
 - $E(\mathbf{C}_{ba}, \mathbf{r}_a) = \frac{1}{2} \sum_{j=1}^J \left(\mathbf{p}_b^j - \mathbf{C}_{ba}(\mathbf{p}_a^j - \mathbf{r}_a) \right)^T \mathbf{\Sigma}^j \left(\mathbf{p}_b^j - \mathbf{C}_{ba}(\mathbf{p}_a^j - \mathbf{r}_a) \right)$

- $\Sigma^j = \left(G_b^j R_b^j G_b^{jT} + C_{ba} G_a^j R_a^j G_a^{jT} C_{ba}^T \right)^{-1}$ where $G_b^j = \frac{\partial g}{\partial \mathbf{y}} \Big|_{\mathbf{f}(\mathbf{p}_b^j)}$, $G_a^j = \frac{\partial g}{\partial \mathbf{y}} \Big|_{\mathbf{f}(\mathbf{p}_a^j)}$ and R_b^j, R_a^j are the pixel measurement covariances in the two cameras for each point
 - * Note now we have covariances resulting from measurement errors in both frames, so we need to transform the uncertainty in frame a to frame b using C_{ba}
- With matrix weights, this no longer has a closed-form solution, so we use Gauss-Newton to iterate as usual
- For outlier rejection, we can first do RANSAC with scalar-weighted point cloud alignment to find the inlier set, then use matrix-weighted alignment to calculate the precise transformation
 - If we have other sensors, we can add another term to the cost function for point cloud alignment to incorporate it

Lecture 14, Oct 21, 2025

Case Study: Computer Vision on Mars

- The Descent Image Motion Estimation System (MER-DIMES) was a system used for descent of Mars Exploration Rovers to estimate horizontal velocity in order to determine when to fire rockets
 - Radar provided distance to ground (assuming a flat landing surface), monocular camera used to determine lateral velocity
 - Computation was very constrained (20MHz, no FPU, 128MB RAM)
 - Estimates horizontal motion over 3 frames
 - Uses a single feature (but looks for 2), fusing with IMU and altimeter data
 - Process:
 - * Image downsampling
 - * Masking shadow of parachute using estimation of sun location (this would give us an incorrect zero velocity)
 - * Determine image overlap region (based on altitude and velocity estimate) to define the search region
 - * Harris corners over a coarse grid, thresholding to 2 best features
 - * Correction for CCD effects (e.g. vignetting), patch rectification
 - * Feature matching using over 2-image pyramid for scale invariance

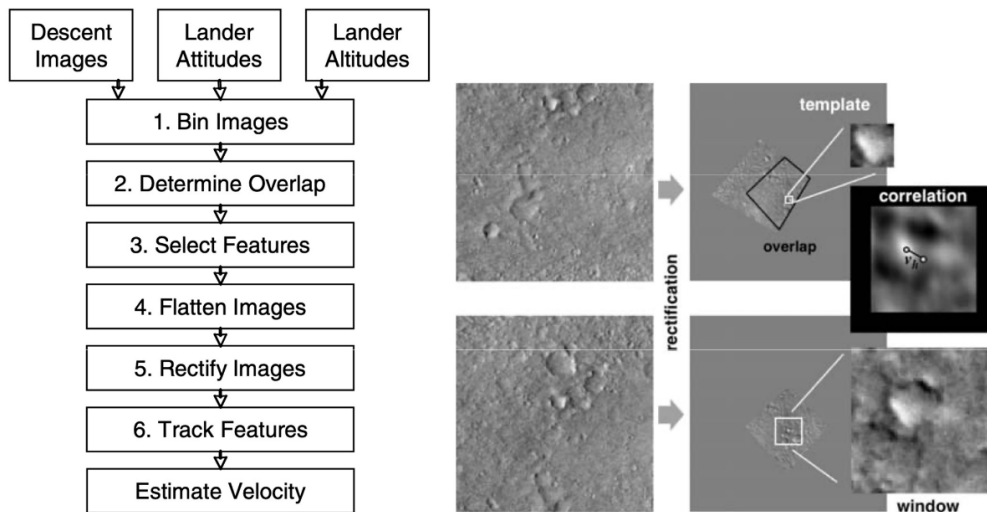


Figure 46: Illustration of the MER-DIMES pipeline.

- Spirit and Opportunity rovers used 3 sets of stereo cameras:
 - Hazard cameras mounted under the solar panels on both sides, for close-range obstacle detection

- Navigation cameras on the main sensor mast, for visual odometry
- Panoramic cameras on the main mast, with different filters that can be applied, for long-range imaging and mineral classification
- Navcams and hazcams were 1024x1024, 12-bit color depth
- Wheel odometry was insufficient due to wheel slip and degradation of calibration errors; VO was used
- Traversability is estimated by fitting planes to small patches of stereo data, checking for protruding obstacles (by checking for residuals of plane fit), excessive tilt, then inflating obstacles on an occupancy grid and determine the path using trajectory rollout
- The Mars Science Laboratory ramped this up with many more cameras and a much larger vehicle
- Perseverance was able to estimate traversability and trajectories while driving, unlike previous rovers which needed to stop to take pictures; it also built maps
- The Ingenuity helicopter carried by Perseverance used visual odometry for navigation, using a single downward-facing camera
 - Used RANSAC for outlier rejection and to eliminate the shadow of the helicopter
 - At least 12 FAST features selected per frame for matching
 - Had to account for motion blur, lighting changes, and feature-poor terrain

Lecture 15, Oct 24, 2025

Optical Flow and Motion Tracking

- *Dense motion estimation* or *optical flow* is the process of estimating a motion vector (velocity) for every possible pixel in the image from a series of images
 - This can be applied in video compression, e.g. MPEG, H.263/4
 - Often regularization is needed to fill in missing pixels
- We need to define an error metric, similar to feature matching, e.g. SAD, SSD
 - Often subpixel accuracy is important (since we are defining very small vectors), so we use interpolation
 - Then use a search technique to identify motion in a pair of images
- Applications of optical flow includes motion estimation, moving object tracking, UAVs, optical mice, etc
- The most challenging version of the problem is to compute motion at each pixel independently over time to obtain an *optical flow field*
 - We rely on the *brightness constancy assumption*: over a time Δt , there is another pixel in the next image that has the same brightness, i.e. $I(x, y, t) = I(x + \Delta x, y + \Delta y, t + \Delta t)$
 - * $\Delta x, \Delta y$ is the optical flow vector that we are looking for; the smaller the Δt , the closer this assumption is to reality
 - * Using a Taylor expansion, $I(x + \Delta x, y + \Delta y, t + \Delta t) \approx I(x, y, t) + \frac{\partial I}{\partial x} \Delta x + \frac{\partial I}{\partial y} \Delta y + \frac{\partial I}{\partial t} \Delta t$
 - * Therefore $\frac{\partial I}{\partial x} \frac{\Delta x}{\Delta t} + \frac{\partial I}{\partial y} \frac{\Delta y}{\Delta t} + \frac{\partial I}{\partial t} \frac{\Delta t}{\Delta t} = 0$
 - The flow equation is $\frac{\partial I}{\partial x} v_x + \frac{\partial I}{\partial y} v_y + \frac{\partial I}{\partial t} = 0$ where v_x, v_y are the pixel velocities
 - In vector form, $I_x v_x + I_y v_y = \Delta I^T \mathbf{v} = -I_t$
 - * Physically we can interpret this as looking at the “flow of intensity” into and out of a pixel
- Due to the aperture problem, this is ill-posed, so we need some additional information in the neighbourhood to actually compute this
 - The *Lucas-Kanade method* assumes that a local patch has the same flow, so we compute the velocity for an entire patch
 - * This can be solved using least squares, by constructing a matrix equation for the entire patch, assuming the same flow

- $$* \begin{bmatrix} I_x(\mathbf{p}_1) & I_y(\mathbf{p}_1) \\ \vdots & \vdots \\ I_x(\mathbf{p}_N) & I_y(\mathbf{p}_N) \end{bmatrix} \begin{bmatrix} u \\ v \end{bmatrix} = - \begin{bmatrix} I_t(\mathbf{p}_1) \\ \vdots \\ I_t(\mathbf{p}_N) \end{bmatrix} \iff \mathbf{A}\mathbf{d} = \mathbf{b}$$
- $$* \begin{bmatrix} \sum I_x I_x & \sum I_x I_y \\ \sum I_x I_y & \sum I_y I_y \end{bmatrix} \begin{bmatrix} u \\ v \end{bmatrix} = - \begin{bmatrix} \sum I_x I_t \\ \sum I_y I_t \end{bmatrix} \iff (\mathbf{A}^T \mathbf{A})\mathbf{d} = \mathbf{A}^T \mathbf{b}$$
- The aperture problem can make motion seem as if it's in a different direction if we only look at a small part of the image (think the *barber pole illusion*)
 - This means it's often a good idea to incorporate some global information
 - The *Horn-Schunck method* imposes a smoothness constraint over the whole image
 - * Minimize $E = \iint [(I_x u + I_y v + I_t)^2 + \alpha^2 (\|\Delta u\|^2 + \|\Delta v\|^2)] dx dy$
 - $\mathbf{v} = \begin{bmatrix} u(x, y) \\ v(x, y) \end{bmatrix}$ is the flow vector, as a function of pixel location
 - α is a regularization constant and $\Delta u, \Delta v$ are the flow vector gradients
 - Practically we can take its derivative and convert to a summation, and solve the problem using NLS
 - This would be a very large problem so we'd need to exploit the problem structure as with bundle adjustment
 - * This imposes a smoothness constraint, which means adjacent pixels influence each other; large changes in the flow vector at adjacent points is penalized
 - * Can help fill in homogeneous (featureless) regions, at the cost of blurring some boundaries
 - Optical flow can give us information about the relative depth of objects, since objects further away move
 - This can be exploited for e.g. UAV navigation in urban canyons, where GPS is denied
 - Apparently bees use optical flows for navigation
 - A simple navigation rule, if we have 2 cameras pointed towards 2 walls on either side, is to simply steer so that the optical flows on the two sides are equal, since if we're closer to a wall it'll have higher optical flow

Lecture 16, Nov 4, 2025

Dense/Photometric Mapping

- In *dense mapping*, we try to match images between frames based on intensities only, as opposed to using features in sparse mapping
 - Instead of minimizing the geometric error of features, we minimize the photometric error of intensities between frames
 - Also known as *direct* methods, since we're directly matching pixel intensities
 - * Note pure "dense" methods recover depth for each pixel; most methods are somewhere in between
 - We are attempting to solve the camera motion estimation and pixel correspondence problems at once, so this is generally harder
 - * In feature-based matching we essentially decoupled the pose estimation and association problems into two distinct steps
 - Feature matching can be impacted significantly by blurring, noise and outliers, which is particularly relevant in the case of motion blur and camera defocusing
 - * By doing photometric mapping, we optimize over the entire image at once (global), so we can still recover matches even under significant blurring and noise
- In summary, dense vision:
 - Is robust to common failure modes, e.g. blur, self-similar textures, defocus
 - Uses almost all information in the image (as opposed to only pixels around features)
 - Can be used for scene understanding and path planning
 - Uses implicit data association by matching pixel intensities

- Very high dimensionality (10^5 to 10^6 pixels per image, making it very computationally expensive)
- Contains a lot more local minima (due to implicit data association), so initialization is important (i.e. works only for small camera movements)
- In a dense mapping pipeline, we use an estimate of the camera transformation between frames (e.g. from odometry) to predict the transformation/warping of the previous frame to the next frame, and then solve a nonlinear optimization problem for the pose
 - We use stereo as usual, since dense depth is needed to predict the warped image in general
- For a real-time scenario we can combine dense and sparse mapping by using sparse matching for pose estimates, and using dense mapping to reconstruct the map for planning purposes
- The dense map can be represented in different ways:
 - Surface-based maps: meshes, splines (fitting splines to points), and height/depth maps
 - Point-based maps: point clouds, surfels (like voxels but 2D), Gaussian splats
 - Volume-based maps: voxel grids (e.g. octrees for adaptive resolution), signed distance functions

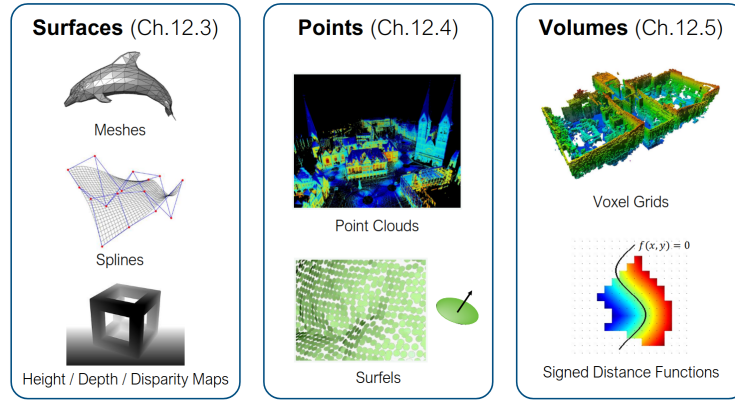


Figure 47: Common 3D map representations.

Dense/Photometric Mapping Pipeline

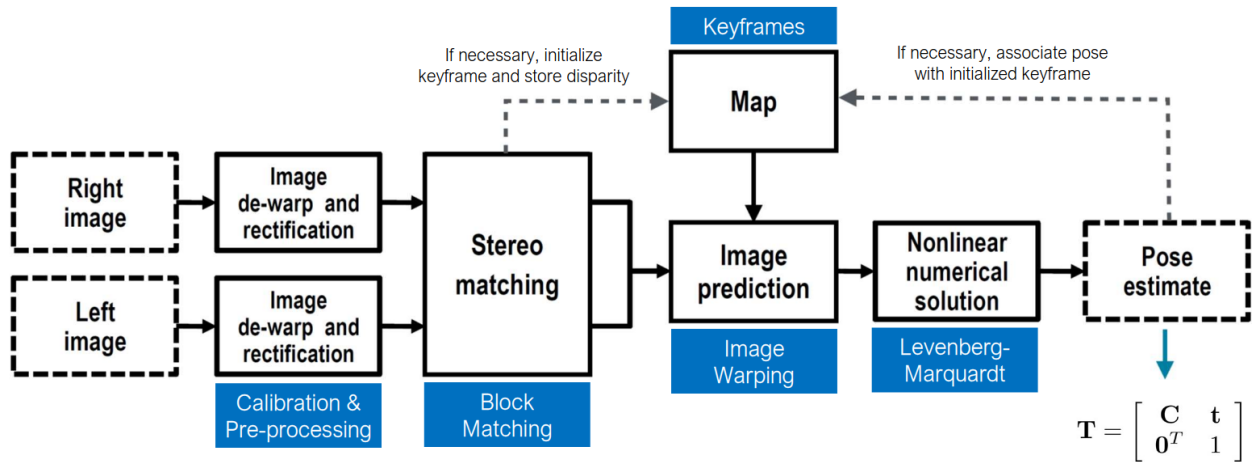


Figure 48: Example dense stereo mapping and localization pipeline.

- We use a modified stereo camera model, similar to the one presented in the VO lecture but this time using one set of pixel coordinates and the disparity

- Forward model: $\mathbf{y} = \begin{bmatrix} u_l \\ v_l \\ d \end{bmatrix} = \mathbf{f}(\mathbf{p}) = \mathbf{M} \frac{1}{p_3} \mathbf{p}$

* Modified camera matrix and homogeneous coordinates: $\mathbf{M} = \begin{bmatrix} f_u & 0 & c_u & 0 \\ 0 & f_v & c_v & 0 \\ 0 & 0 & 0 & f_u b \end{bmatrix}$, $\mathbf{p} = \begin{bmatrix} sx \\ sy \\ sz \\ s \end{bmatrix}$

$$* \frac{\partial \mathbf{f}}{\partial \mathbf{p}} = \frac{1}{p_3} \mathbf{M} \begin{bmatrix} 1 & 0 & -p_1/p_3 & 0 \\ 0 & 1 & -p_2/p_3 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & -p_4/p_3 & 1 \end{bmatrix}$$

– Inverse model: $\boldsymbol{\rho} = \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \mathbf{g}(\mathbf{y}) = \frac{b}{d} \begin{bmatrix} u_l - c_u \\ \frac{f_u}{f_v}(v_l - c_v) \\ f_u \end{bmatrix}$

$$* \frac{\partial \mathbf{g}}{\partial \mathbf{y}} = \frac{b}{d^2} \begin{bmatrix} b & 0 & c_u - u_l \\ 0 & \frac{f_u}{f_v}d & \frac{f_u}{f_v}(c_v - v_l) \\ 0 & 0 & -f_u \end{bmatrix}$$

- From the input, we select keyframes (based on e.g. spatial/temporal changes, information estimates, etc) and incorporate pointclouds from keyframes into the map
- Given a transformation between frames $\mathbf{T}_{k,k-1}$, we can map pixels in the previous frame (u_{k-1}, v_{k-1})

to the next frame (u_k, v_k) as $\begin{bmatrix} u_k \\ v_k \\ d_k \end{bmatrix} = \mathbf{f} \left(\mathbf{T}_{k,k-1} \mathbf{g} \left(\begin{bmatrix} u_{k-1} \\ v_{k-1} \\ d_{k-1} \end{bmatrix} \right) \right)$

- Use the inverse camera model to get 3D points from pixels, transform into the new frame, and project it through the forward camera model to get the expected pixel location and disparity
- This process will leave holes in the new image, so we need to interpolate
- For matching, again assume photometric consistency: $I_k(u_k, v_k) = I_{k-1}(u_{k-1}, v_{k-1}) + n$ where $n \sim \mathcal{N}(0, \sigma^2)$ is some noise
- Define the *photometric error* for each pixel, and sum over all pixels of interest to get the overall cost, which we can optimize to get the transformation
 - $e_i = I_{k-1}(\mathbf{u}_{k-1}^i) - I'_k(\mathbf{u}_k^i)$ where $I'_k(\mathbf{u})$ is an interpolated image indexed with continuous coordinates
 - $\mathbf{y}_k^i = \begin{bmatrix} u_k^i \\ v_k^i \\ d_k^i \end{bmatrix} = \begin{bmatrix} \mathbf{u}_k^i \\ d_k^i \end{bmatrix} = \mathbf{f}(\mathbf{T}_{k,k-1} \mathbf{g}(\mathbf{y}_{k-1}^i))$
 - Solve for $\hat{\mathbf{T}}_{k,k-1} = \underset{\mathbf{T}_{k,k-1}}{\operatorname{argmin}} \sum_{i=1}^N \left(\frac{1}{\sigma} e_i \right)^2$
- The disparity map of keyframes can be included in the optimization to generate better maps, combining the pixel intensity error, disparity difference in the active frame (actual vs. predicted disparity in next frame), and disparity difference from the original observation

$$- \mathbf{e}_i = \begin{bmatrix} I_{k-1}(\mathbf{u}_{k-1}^i) - I'_k(\mathbf{u}_k^i) \\ d_k^i - \bar{D}'_k(\mathbf{u}_k^i) \\ d_{k-1}^i - \bar{d}_{k-1}^i \end{bmatrix}$$

- * Note d_{k-1} is the disparity to be optimized (function of $\mathbf{T}_{k,k-1}$), \bar{d}_{k-1} is the actual observed disparity (from stereo matching) and \bar{D}'_k is the interpolated disparity in the current frame
- * First term is the usual photometric error term
- * Second term is the difference between the observed disparities in the next frame and the predicted disparities based on $\mathbf{T}_{k,k-1}$ and d_{k-1}
- * Third term is the difference between the optimized disparity map and original disparity observations, so the optimized map is not too different from the observed map
- We can use a matrix weight \mathbf{W}_i^{-1} , usually diagonal with variances, and optimize in both $\mathbf{T}_{k,k-1}$ and d_{k-1}

Tips and Tricks

- Due to the prevalence of local minima, we need a very good initial guess for the optimization
 - One method is to use an IMU/odometry
 - Another commonly used method is coarse-to-fine optimization, i.e. starting with a blurred and subsampled image, then repeatedly optimizing while increasing resolution

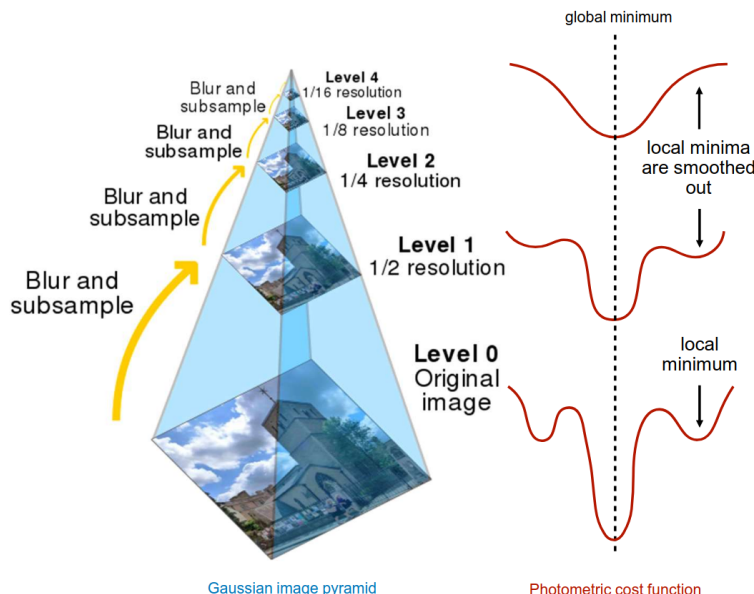


Figure 49: Coarse-to-fine optimization in dense mapping.

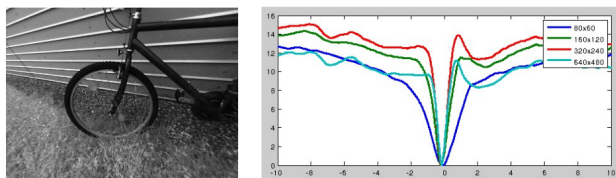


Figure 50: Comparison of the cost function shape under different degrees of blurring/downsampling for a real image.

- Instead of operating on all pixels, which is computationally intensive, we can choose to match over only informative pixels
 - Can be done based on gradient strength: $\|\nabla I(u, v)\| \geq \nabla_{min}$
 - We can also do this over a grid, and for each grid cell take the pixels with the best gradient
 - The result is retained pixels that are often concentrated along edges and corners
 - This trades off accuracy for computation time
- Using a GPU to parallelize the error computation can significantly speed up the matching
- Unlike features, direct photometric mapping is affected by effects such as gamma, exposure, vignetting, lens attenuation, etc
 - For even better accuracy, we can perform photometric calibration to build a photometric camera model incorporating all of these effects

Lecture 17, Nov 7, 2025

Image Segmentation

- *Gestalt theory* is the idea of “the whole is greater than the sum of its parts” – we perceive images as entire patterns, instead of individual components/pixels
 - Law of proximity: We naturally group things together based on proximity
 - Law of similarity: We group things based on similarity
 - Law of continuity: We group continuous objects together, and we understand that an object continues even if it’s partially occluded
- *Segmentation* is the task of finding pixels that “go together”, grouping them and potentially classifying them
 - Early techniques were *divisive* (breaking up an image/objects) or *agglomerative* (growing out from a point) and operate locally
 - More recent techniques are global and optimize across regions
 - Many types of segmentation:
 - * *Coarse segmentation*: bounding boxes for objects
 - * *Fine segmentation*: splines to describe boundaries between objects or pixel-wise labels/masks
 - * *Semantic segmentation*: segmenting based on what the object is, e.g. chair vs non-chair
 - * *Instance segmentation*: semantic segmentation, but being able to tell apart different instances of objects
 - * *Panoptic segmentation*: segmenting instances of countable objects (foreground “things”, e.g. people) and grouping together uncountable objects (background “stuff”, e.g. a road)

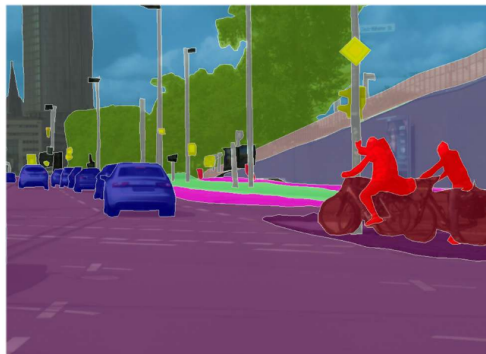


Figure 51: Example of semantic segmentation (but not instance-level).

Classical Segmentation Methods

- The simplest segmentation approach is to just to apply a threshold to the intensity levels (or to a specific channel in some colour space), then finding connected components to get the regions
- *Active contours* is an energy-minimization approach that fits a spline $\mathbf{f}(s) = (u(s), v(s))$ to minimize an energy function
 - Smoothness cost: $\mathcal{E}_{\text{int}} = \int \alpha(s) \|\mathbf{f}_s(s)\|^2 + \beta(s) \|\mathbf{f}_{ss}(s)\|^2 ds$
 - * Penalizes sharp changes/kinks
 - Image energy: $\mathcal{E}_{\text{image}} = w_{\text{line}} \mathcal{E}_{\text{line}} + w_{\text{edge}} \mathcal{E}_{\text{edge}} + w_{\text{term}} \mathcal{E}_{\text{term}}$
 - * The terms attract the spline to dark ridges, strong gradients (edges), and line terminations respectively
 - * In practice this is primarily the edge term, $\mathcal{E}_{\text{edge}} = \sum_i -\|\nabla I(\mathbf{f}(i))\|^2$, i.e. summing up the gradient magnitude at all the points on the spline
 - The *B-spline* is defined as $\mathbf{f}(s) = \sum_k B_k(s) \mathbf{x}_k$, where $B_k(s)$ are k basis functions and \mathbf{x}_k are

- control points (i.e. parameters)
 - * B-splines are generalizations of Bezier curves
- For evolving images/contours, we can model it with linear dynamics: $\mathbf{x}_t = \mathbf{A}\mathbf{x}_{t-1} + \mathbf{w}_t$ where \mathbf{x}_{t-1} are the contour control points from the previous iteration, \mathbf{A} is the transition matrix, and \mathbf{w}_t is some noise vector
 - This is known as *conditional density propagation* (aka *CONDENSATION*)
 - To implement this we can use a particle filter, where each particle is a contour, and we propagate the belief using \mathbf{A}
- *Split and merge* algorithms perform segmentation by a combination of recursive splitting of the image and merging together regions
 - The *watershed algorithm* starts from seed points, and performs watershed segmentation
 - * Interpret the grayscale image as a topographic image, i.e. darker regions are “valleys”
 - * The idea is to fill the local minima with “water”, i.e. propagate outward from the minima until we hit a high-intensity boundary
 - * This assumes that the boundaries are similar in intensity
 - * Locality constraints can be applied so that the regions to close off the regions

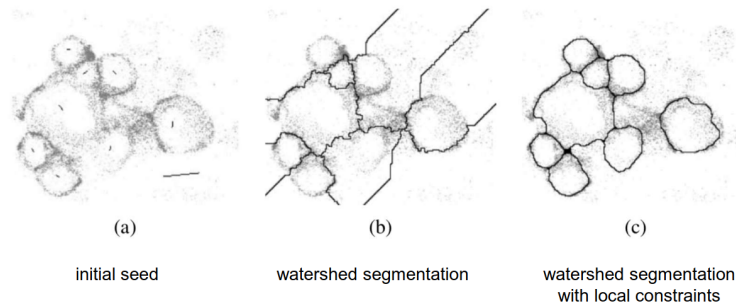


Figure 52: Example of the watershed algorithm.

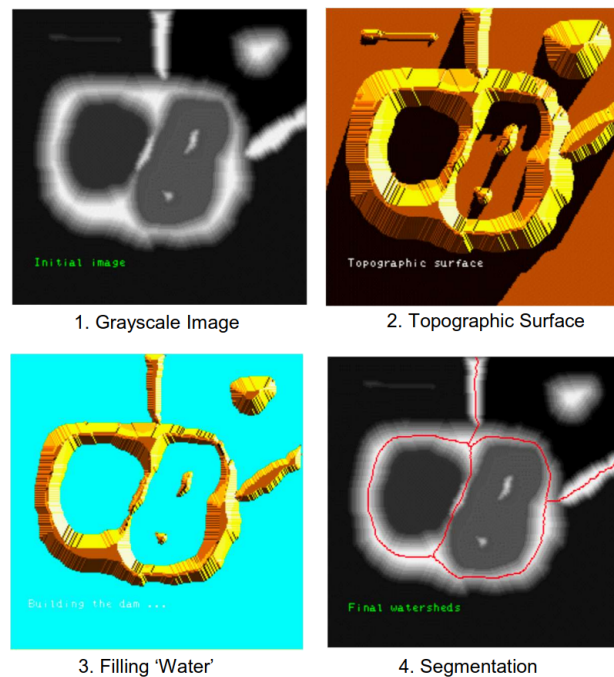


Figure 53: Watershed algorithm steps.

- One category of segmentation methods attempts to cluster pixels on a feature level or directly over

some colour space, e.g. LUV

- We basically want to find blobs of similar pixels in some colour/feature space and cluster them together, where each cluster represents a region
- This can be done over colour, position, etc
- The *mean-shift algorithm* considers each pixel to be a sample from a PDF, with multiple means; *kernel density estimation* is used to estimate this PDF, and the modes of the PDF are used for the segments
 - The kernel density estimator is $f(\mathbf{x}) = \sum_i K(\mathbf{x} - \mathbf{x}_i) = \sum_i k\left(\frac{\|\mathbf{x} - \mathbf{x}_i\|^2}{h^2}\right)$
 - * \mathbf{x}_i are the samples and \mathbf{x} is the mean
 - * h is the *bandwidth* of the kernel and controls the spread of the distribution, i.e. how quickly the density varies
 - * We often use the Gaussian kernel $k_N(r) = e^{-\frac{1}{2}r}$
 - The mean-shift algorithm uses multiple restart gradient descent:
 1. For each mode \mathbf{y} , start with some initial guess \mathbf{y}_0
 2. Compute the next \mathbf{y} by adding the mean-shift, $\mathbf{y}_{k+1} = \mathbf{y}_k + \mathbf{m}(\mathbf{y}_k) = \frac{\sum_i \mathbf{x}_i G(\mathbf{y}_k - \mathbf{x}_i)}{\sum_i G(\mathbf{y}_k - \mathbf{x}_i)}$
 - * G is the derivative of the kernel function, so this is like computing an average gradient
 3. Repeat until convergence, $\|\mathbf{m}(\mathbf{y}_k)\| < \epsilon$
 - One simple approach is to initialize a mode at each input point, and iterate until all pixels have converged to a mode; then each distinct mode will be a segment, consisting of all the pixels that converged to it

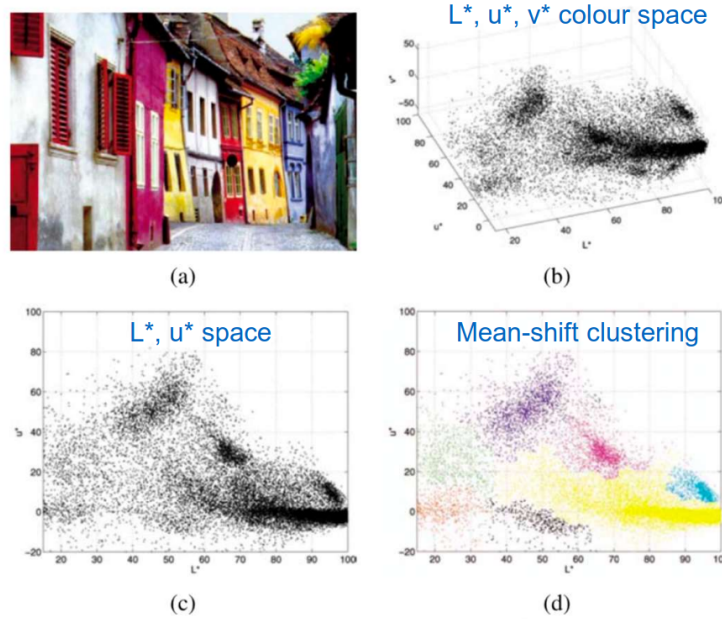


Figure 54: Example of mean-shift clustering in LUV colour space.

- The *graph cuts algorithm* constructs the image as a graph and attempts to cut the graph into regions
 - The idea is that pixels that should be grouped together should share affinity
 - The nodes of the graph are pixels
 - The edges are weighted using an affinity function based on salient properties, e.g. pixel distance, intensity difference, colour difference, texture metrics (e.g. by convolution)
 - We can make either minimum cuts (using the max-flow/min-cut algorithm), or normalized cuts (cost of the cut normalized by the size of segments)
 - * Minimum cuts can be solved efficiently but tends to penalize large segments, so it can over segment
 - * Normalized cuts are better, but is NP-hard in general, although we can use approximations

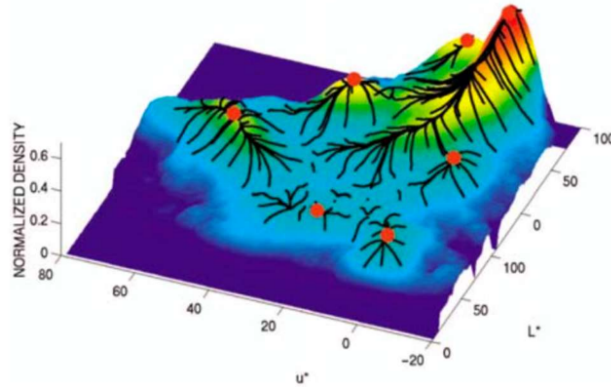


Figure 55: Example trajectory of points from the mean-shift algorithm. The black lines represent trajectories of each pixel point, which all converge to one of multiple final modes (red points).

- The segmentation problem can be formulated as a *Markov Random Field* (MRF), where we consider the segmentation solution as the “true” pixel states, and the image itself as the observed “evidence”
 - We want to minimize energy $E(x, y) = \sum_i \varphi(x_i, y_i) + \sum_{ij} \psi(x_i, x_j)$
 - Can be formulated as a min-cut problem for binary (foreground/background) labelling
- GrabCut is a classical segmentation method that uses a user-supplied bounding box, and an MRF model
 - A Gaussian mixture model (GMM) is fit to the foreground and background colour
 - The MRF energy is defined based on the GMM, and min-cut is applied to classify into foreground and background
 - Repeat until convergence, fitting a new GMM each time

Modern Approaches

- Modern approaches are mostly based on deep learning, focusing on pixel-level segmentation and classification
 - Obtaining ground truth data is difficult since images often have to be labelled manually per-pixel (nowadays we preprocess using existing segmentation networks)
 - SegNet is one of the first modern approaches for autonomous driving
- Many segmentation networks are based on a U-Net architecture, with down convolutions going to bottleneck layers, then up convolutions to recover the full resolution, and skip connections in between
- Datasets include KITTI (only 200 training/test), TUM SceneFlow (fully synthetic), and City Scapes (5000 real-life examples, instance-level segmentations across 50 cities for 30 classes)

Lecture 18, Nov 11, 2025

Visual Servoing

- A *servo mechanism* in general is a device that controls itself to drive to a desired position, by minimizing the error
- *Visual servoing* is the problem of controlling a robot (usually a manipulator) relative to a target of interest, by tracking visual features of the target
 - The camera can be mounted either on the robot (*eye-in-hand*) or mounted at a fixed external point
- Two different approaches:
 - *Position-based visual servoing* uses known calibration and target geometry to estimate target pose, and controls the robot in task space (e.g. $SE(3)$) to move to a desired pose

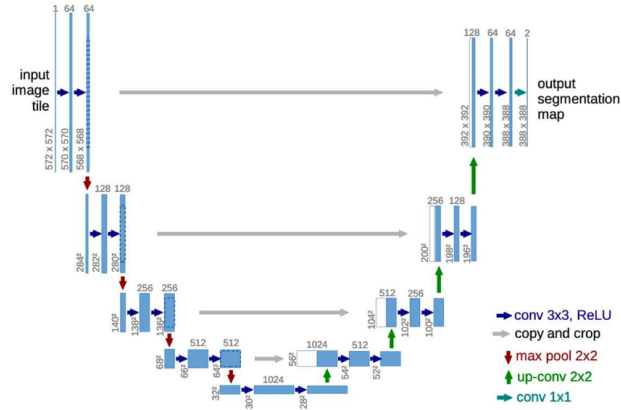


Figure 56: U-Net architecture used for medical image segmentation.

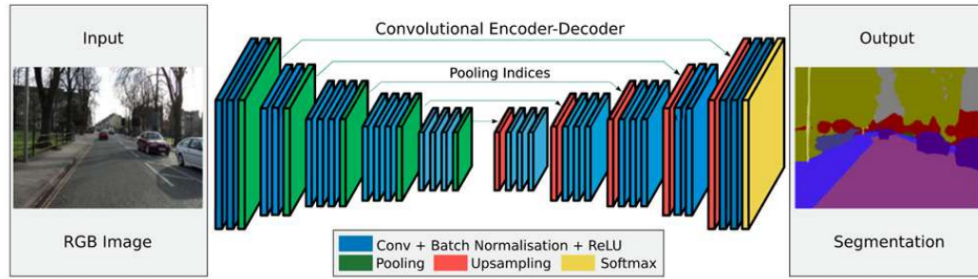


Figure 57: SegNet architecture.

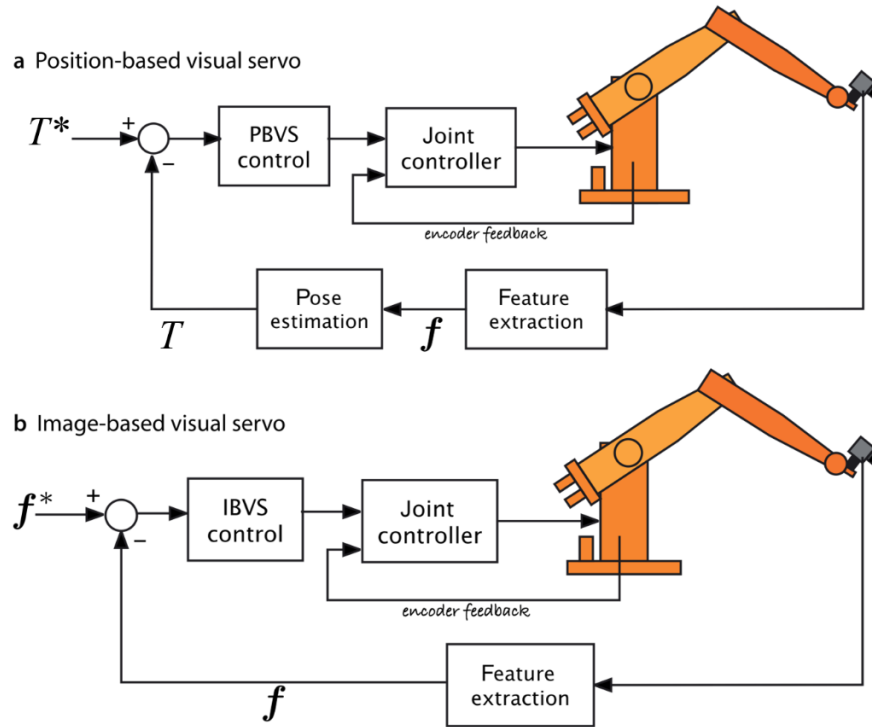


Figure 58: Position vs. Image-based visual servoing.

- *Image-based visual servoing* operates directly using feature correspondences on the image, and controls the robot in image space to align the image features to desired locations
- In visual servoing we want to make incremental steps; one problem with the eye-in-hand configuration is that if we move too much, we may end up in a position where the target is not visible

Position-Based Visual Servoing

- For PBVS, we need to know the 3D positions of the feature points on the target and the camera intrinsics
- We want to drive the end-effector pose to some \mathbf{T}_{TE}^* relative to the target
- Estimate the current pose of the target in the camera frame using camera pose estimation; then we minimize the error function $\mathbf{e}_i = \mathbf{x}_i - \mathbf{f}(\mathbf{p}_i; \mathbf{C}, \mathbf{t}, \mathbf{K})$
 - This is commonly done with fiducial markers like ArUco or AprilTag
- The target motion can be estimated using filtering approaches such as Kalman filtering or particle filtering, either in an inertial frame or a frame relative to the moving camera
- The controller can take one of two approaches:
 - *End-point open-loop*: Using inverse kinematics to move the arm directly to the desired location (often used for external camera)
 - *Eye-in-hand closed-loop*: Setting the end-effector velocity based on pose error
- PBVS requires that the depth/scale be known; this can be approximated, or we can lock in a scale on initialization
 - The depth can be estimated using the apparent size of the target, since we know the actual target size
- There is no direct constraint to keep the target within the camera FoV
 - Can be addressed with finite horizon planning and numerical optimization

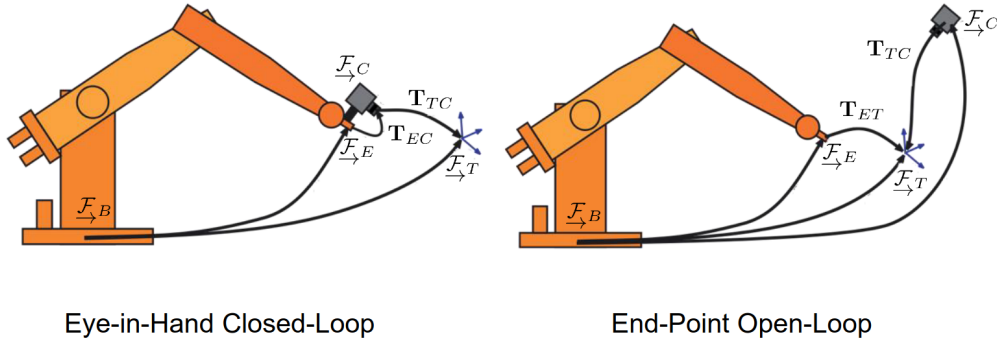


Figure 59: Two methods of controlling the end-effector pose.

Image-Based Visual Servoing

- In IBVS, the target pose is not explicitly computed; we instead directly track the image feature positions in image space
 - We assume that we have correspondences
 - Applies to the eye-in-hand case
- *Uncalibrated visual servoing* is when we don't have the intrinsic calibration or only know it approximately
 - This is still possible since we only care about the visual appearance, but we won't explore it here
- The control problem is formulated in the image space to align observed and desired image feature locations; usually a linear controller is used
- By differentiating the forward camera model, we can obtain the Jacobian:

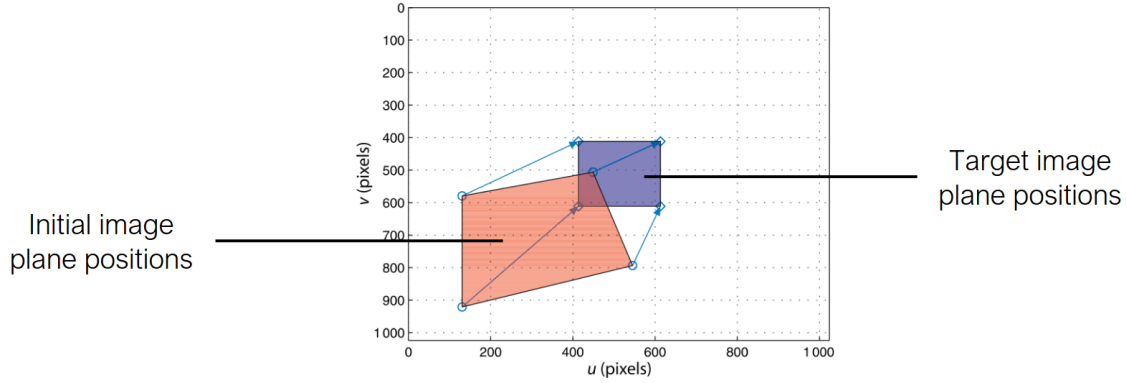


Figure 60: Illustration of IBVS.

$$- \dot{\mathbf{p}} = \begin{bmatrix} \dot{u} \\ \dot{v} \end{bmatrix} = \begin{bmatrix} -\frac{f}{Z} & 0 & \frac{u}{Z} & \frac{uv}{f} & -\frac{f^2 + u^2}{f} & v \\ 0 & -\frac{f}{Z} & \frac{v}{Z} & \frac{f^2 + v^2}{f} & -\frac{uv}{f} & -u \end{bmatrix} \begin{bmatrix} v_x \\ v_y \\ v_z \\ \omega_x \\ \omega_y \\ \omega_z \end{bmatrix} = \mathbf{J}_p \boldsymbol{\nu}$$

- This relates movements in Euclidean space to movements in image space
- Now we can stack the equations for at least 3 points and invert the stacked Jacobian (or use pseudoinverse) to solve for the desired camera motion
 - $$\begin{bmatrix} \dot{\mathbf{p}}_1 \\ \vdots \\ \dot{\mathbf{p}}_n \end{bmatrix} = \begin{bmatrix} \mathbf{J}_{p_1} \\ \vdots \\ \mathbf{J}_{p_n} \end{bmatrix} \boldsymbol{\nu}$$
 - Note if the points are collinear, the Jacobian ends up being singular
- Given the desired locations for the feature points, \mathbf{p}_i^d , we can now compute the camera velocity that we need to command:
 - $$\boldsymbol{\nu} = \lambda \begin{bmatrix} \mathbf{J}_{p_1} \\ \vdots \\ \mathbf{J}_{p_n} \end{bmatrix}^\dagger \begin{bmatrix} \mathbf{p}_1^d - \mathbf{p}_1 \\ \vdots \\ \mathbf{p}_n^d - \mathbf{p}_n \end{bmatrix}$$
 - This is a proportional controller, with λ as the gain
- Note calculating the Jacobian requires knowledge of the depth, which we can estimate; the algorithm is typically very tolerant of depth errors
 - We can also use stereo, which would also enable PBVS
- Like in eye-in-hand PBVS, this also has no explicit constraint to keep the target within FoV
- Since the camera motion is calculated implicitly from the desired movement of feature points, IBVS can fail to converge for some cases where there is ambiguity
 - e.g. if we rotate the camera by π for a rectangle, the desired trajectory of all image points will pass through the origin, so instead of rotating, the controller will actually move the camera further back; all feature points will end up converging at the origin as the camera moves further away, and the controller never converges

Lecture 19, Nov 19, 2025

Advances in Imaging: New Sensors and Systems

- How do we get depth information?
 - Stereo camera setup (e.g. Intel RealSense, which also uses lasers)
 - * The range is limited by the baseline – too close and we cannot find correspondences, too far

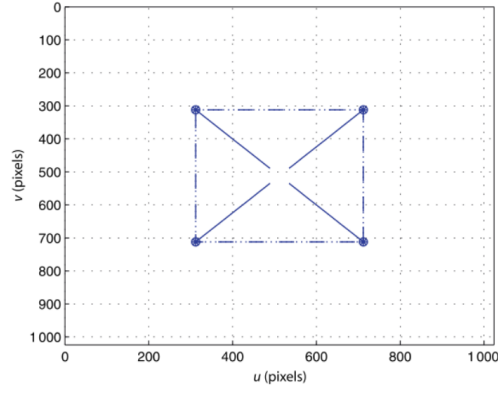


Figure 61: Example failure case of IBVS where the camera is rotated 180° from the desired pose.

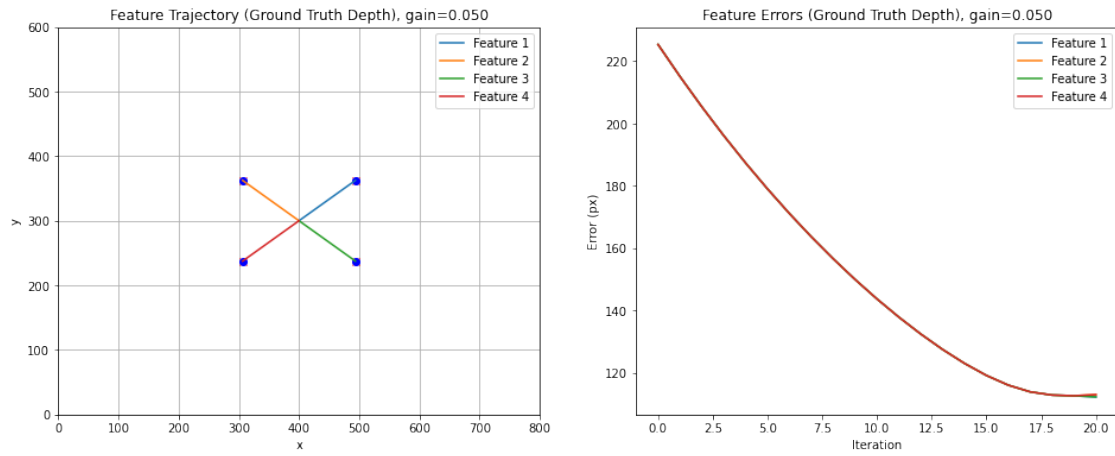


Figure 62: IBVS feature point trajectory when the camera is rotated 180° from the desired pose.

and the disparity will be too small, and a tiny disparity error will translate to a huge depth error

- Structured light (e.g. Kinect)
 - * A projector projects a pattern of points onto the scene, and a camera observes the pattern and uses the warping of the pattern to estimate depth
- Time-of-flight (ToF) sensors and LiDARs
- Comparing images taken with different focal lengths
- Neural network monocular depth estimation
- RGB-D cameras return colour values and a depth value for each pixel
 - Depth is typically returned as inverse depth or disparity map which is often better for numeric accuracy
- In addition to depth, normal cameras also suffer from other effects
 - Limited dynamic range, i.e. range of brightness intensities that it can measure in one image
 - * Typically measured as the ratio of the most intense measurable brightness divided by the least intense brightness in decibels
 - * Standard cameras usually have a dynamic range around 60 decibels
 - Motion blur from objects moving faster than the exposure time can handle
 - * Typically limited to 100 to 1000 fps
 - Inefficient bandwidth use – even when the scene is not changing we still need to use bandwidth

Event Cameras

- *Event cameras* instead measure light intensity variations in the scene
 - They have much higher dynamic range (140 dB), very low latency (1 MHz), low bandwidth use, and very low power consumption
- The output of an event camera is an *event*, a tuple containing the time, pixel location, and *polarity*, i.e. the sign of the intensity change, whether the pixel got brighter or darker
 - More precisely, if the log intensity of a pixel changes by more than some threshold C , we emit an event
 - When there is no change, the camera does not output anything
 - Each pixel is asynchronous, i.e. it can trigger independently of all others, and up to once per microsecond

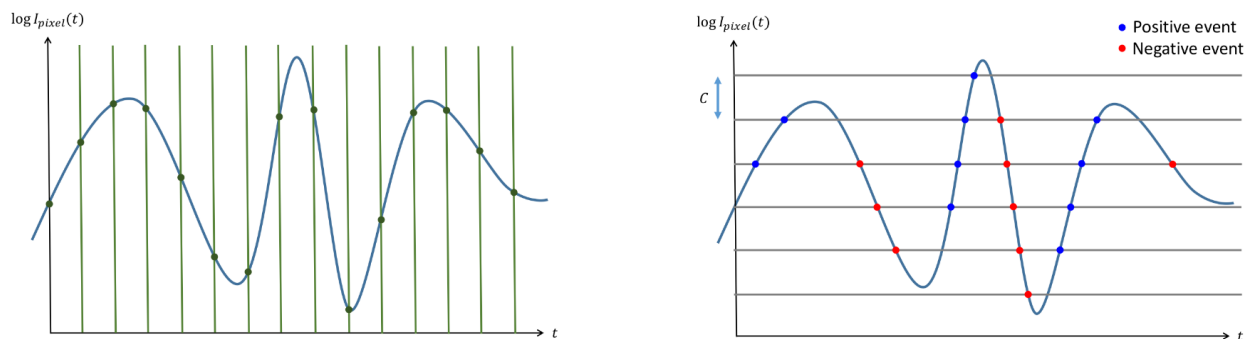
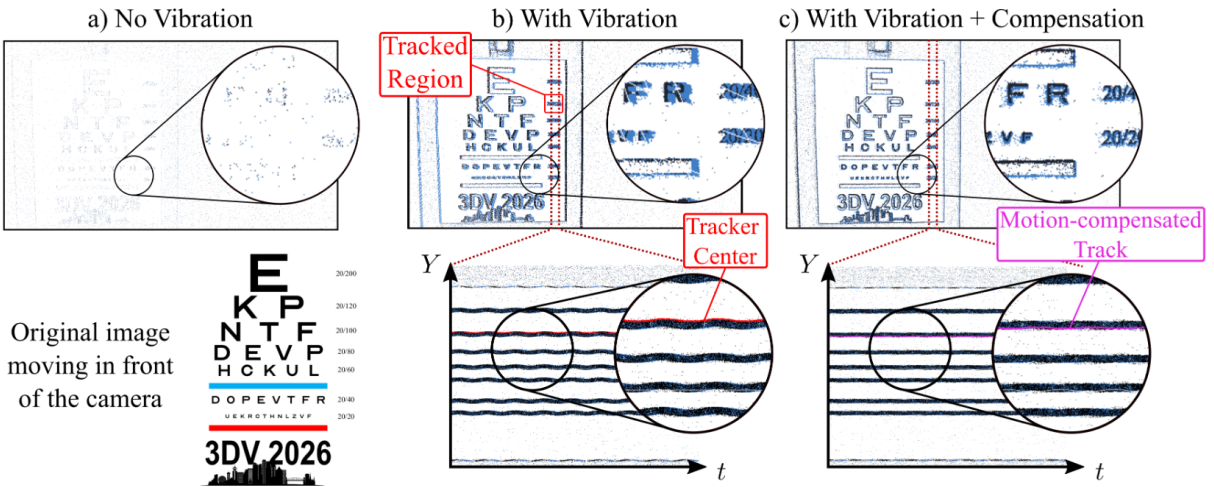


Figure 63: Sampling method of normal cameras vs. event cameras.

- The asynchronous pixels of the event camera is similar to how the human eye works; we don't have enough nerves to transmit signals from all the cone cells all the time, so all the cones are asynchronous like in an event camera
- The event camera uses a generative event model
 - When $L(x, y, t + \Delta t) - L(x, y, t) = \pm C$ where $L(x, y, t) = \log I(x, y, t)$, we get an event
- Consider a point moving with velocity (u, v) , then it moves from $(x, y) \rightarrow (x+u, y+v)$; we can once again make the brightness constancy assumption as we did for optical flow, $L(x, y, t) = L(x+u, y+v, t + \Delta t)$

- First-order Taylor approximation: $L(x + u, y + v, t + \Delta t) = L(x, y, t + \Delta t) + \frac{\partial L}{\partial x}u + \frac{\partial L}{\partial y}v$
- Taking the difference, $L(x, y, t) - L(x, y, t + \Delta t) = \frac{\partial L}{\partial x}u + \frac{\partial L}{\partial y}v$
- We can express the generative model as $-\frac{\partial L}{\partial x}u - \frac{\partial L}{\partial y}v = \pm C \implies -\Delta L \cdot \mathbf{u} = \pm C$
 - * Physically, we can interpret this as saying that we get the most events the movement of a point aligns with the gradient
 - * This means that events are triggered along the edges in the image, when the movement is perpendicular to the edge
- Since the event camera does not emit when there is no motion, to get a continuous stream we can explicitly induce vibrations using mechanical movement
 - This gives us continuous images where we can see the edges of objects
 - Since we know the vibration induced, we can explicitly compensate for it to get a stable image



V. Polizzi et al. "VibES: Induced Vibration for Persistent Event-based Sensing", 3DV 2026

Figure 64: Results from inducing vibrations in an event camera.

- The DAVIS sensor combines both events and frames, so the events give us information between frames
 - This information can be used to de-blur images (Qualcomm and Prophesee)
 - Motion blur is caused by the “summing of frames”, and event cameras give us the change between frames; this means that if we “integrate” the events, we can subtract it from the final image and remove the effect of summing the extra frames, removing blur

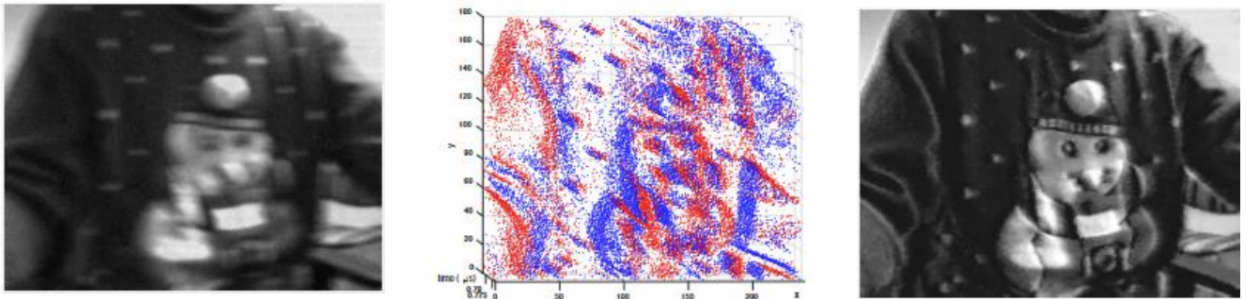


Figure 65: Image de-blurring using event cameras.

- We can also use networks such as E2VID to reconstruct scenes from event cameras

Thermal Cameras

- Thermal cameras capture the infrared spectrum instead of visible light, which is emitted by warm objects
- This means we can see with no light (since objects emit their own light), and even see through smoke, fog, etc, and determine the temperature of objects
- Thermal cameras are not photometrically consistent over time
 - They can heat up themselves, which generates artifacts across time
 - The imaging sensors have non-uniform responses
 - There can be spatial artifacts
 - This breaks some systems since we made the brightness constancy assumption, which no longer holds
 - There are methods for photometric calibration to remove the artifacts and make the images more consistent
- The Luxonis OAK-T is an example combining thermal and normal imagery
- In low-light conditions, we can extract way more features from thermal imagery than visual imagery, so thermal cameras can be good for state estimation and localization

Lecture 20, Nov 18, 2025

Deep Learning for SfM and Localization

- *PoseNet* is a CNN for 6-DoF camera relocalization from a single image, i.e. identifying where you are in a scene
 - This solves the *kidnapped robot problem*, i.e. trying to relocalize a robot with no prior information about how it moved
 - Represents pose with position vector and orientation quaternion
 - Uses a geometric loss: $\mathcal{L}(I) = \|\hat{\mathbf{x}} - \mathbf{x}\| + \beta \left\| \hat{\mathbf{q}} - \frac{\mathbf{q}}{\|\mathbf{q}\|} \right\|$
 - * $\hat{\mathbf{x}}, \hat{\mathbf{q}}$ are the ground-truth position and orientation quaternion, and the non-hat variables are the network predictions
 - * β balances rotation and translation error, which needs to be tuned for each dataset
 - * Later work showed that β could be learned or selected based on pose uncertainty
 - Architecture based on GoogLeNet, containing 6 inception modules, with fully connected regression layers instead of classification
 - Can visualize the parts of the image that the network pays attention to with a saliency map, which shows that the network pays attention to distinctive feature points (similar to feature detectors) and large textureless regions, and ignore dynamic objects like pedestrians
 - Capable of relocalizing within 2 m and 3 degrees for very large scenes spanning 50,000 square metres
 - * An order of magnitude worse than classical methods, but faster and with less memory; instead of a large growing map, PoseNet compresses it all into a fixed-size network
 - We'd need to refine the pose with classical methods if we want to make it useable
 - * The challenge is that the network is directly regressing a 7D vector from individual images, so it cannot impose consistency between images in a pose sequence
 - * PoseNet is essentially just interpolating between poses in the training data, so it works more like image retrieval than actual camera pose estimation
- SfMLearner uses photometric loss for unsupervised learning, to predict depth of an image (monocular depth prediction) and relative pose difference between a pair of images (visual odometry)
 - During training, it attempts to recreate the next image in the sequence
 - We use the depth CNN to predict depth for the current image, and use the depth info along with predictions from the pose CNN to warp the current image to predict the next image, then minimize a photometric loss between the prediction and the true next image
 - It also learns an “explainability mask”, which identifies regions of the image that should not be

- used for VO (e.g. dynamic objects, occlusions, reflective surfaces, etc)
- * Pixels that are masked have their importance decreased or ignored when computing the photometric loss
 - * To prevent the mask from just being driven to zero everywhere, add a regularization term that encourages nonzero terms
 - * Use another regularization term that encourages smoothness (i.e. penalizes second order gradients) to get geometrically plausible masks
 - Uses a U-Net architecture similar to segmentation networks for depth
 - * Pose predictions and explainability mask are produced by parts of the same network; pose is taken with some fully connected layers at the bottleneck while explainability is taken at the later layers
 - Depth map can hallucinate objects, e.g. predicting car shapes where there are none due to the dataset typically having cars in a scene

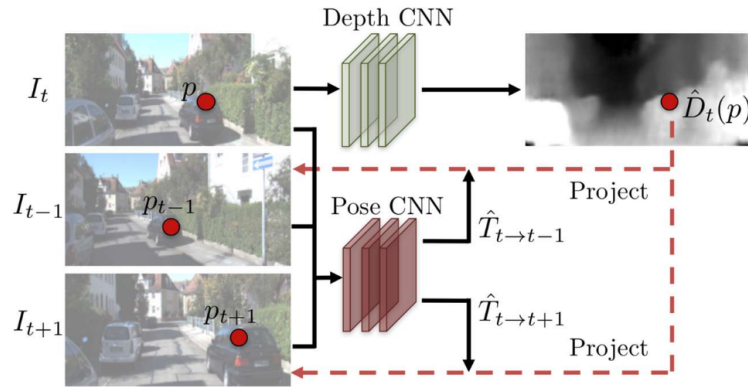


Figure 66: SfMLearner training process.



Figure 67: Examples of the explainability mask highlighting moving objects (top) and occlusions (bottom).

- Single-image depth prediction networks tend to rely on the vertical position of an object to estimate depth, since it learns to associate objects that are higher with being further away due to the structure of the dataset
 - They can generalize to unseen objects, but only if the objects have shadows under them
- Often best results can be achieved by fusing classical and learning based approaches, for which there are 3 general approaches:
 - Correction: Use classical techniques to estimate a solution, then use a learning based method to apply a correction
 - * e.g. correcting systematic bias in pose estimation resulting from parameter errors

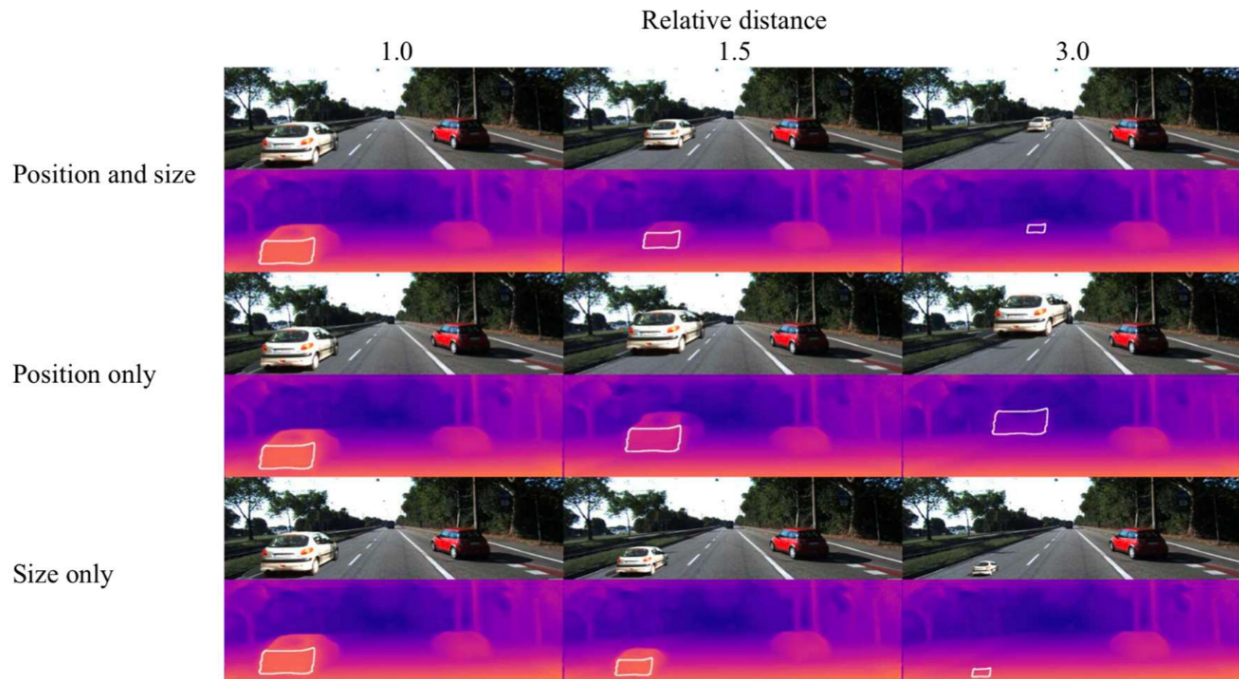


Figure 68: Single-image depth prediction reliance on vertical position.

- Augmentation: Using learning based methods to get additional information, then using a classical method to predict the solution with the extra information
 - * e.g. estimating sun direction to limit orientation error
- Initialization: Using learning methods to get an initial guess for the solution, then using classical methods to refine the solution accuracy
 - * e.g. using PoseNet for initializing, then normal feature matching to get an accurate solution
- *Sun-BCNN* is an example of the augmentation approach; the network learn to predict the sun direction from an image, which can be fed to correct orientation errors in visual odometry, so we don't need a dedicated sun sensor
 - Orientation errors cause VO error to grow super-linearly, compared to just linear growth if orientation is corrected
 - Uses cues such as shadows, lighting, reflections, etc
 - * Saliency maps shows that the network pays attention to the sky, well-lit regions, and shadows
 - Uses a Bayesian GoogLeNet, regressing a 3D vector and an uncertainty estimate, which can be used to fuse the sun orientation only when the network is certain
 - Used a cosine distance loss $\mathcal{L} = 1 - \hat{\mathbf{s}}_k \cdot \mathbf{s}_k$

Lecture 21, Nov 21, 2025

Deep Learning for Computer Vision Tasks

- A variety of tasks in computer vision:
 - Image classification: classify an object based on the dominant object inside it
 - Object localization: predict the image region that contains the dominant object
 - Object detection: localize and classify all objects in the image
 - Semantic segmentation: label each pixel in the image by the object class it belongs to
 - Instance segmentation: label each pixel of an image by the object class and instance it belongs to
 - Panoptic segmentation: combining semantic and instance segmentation to label both foreground object instances and background “stuff”

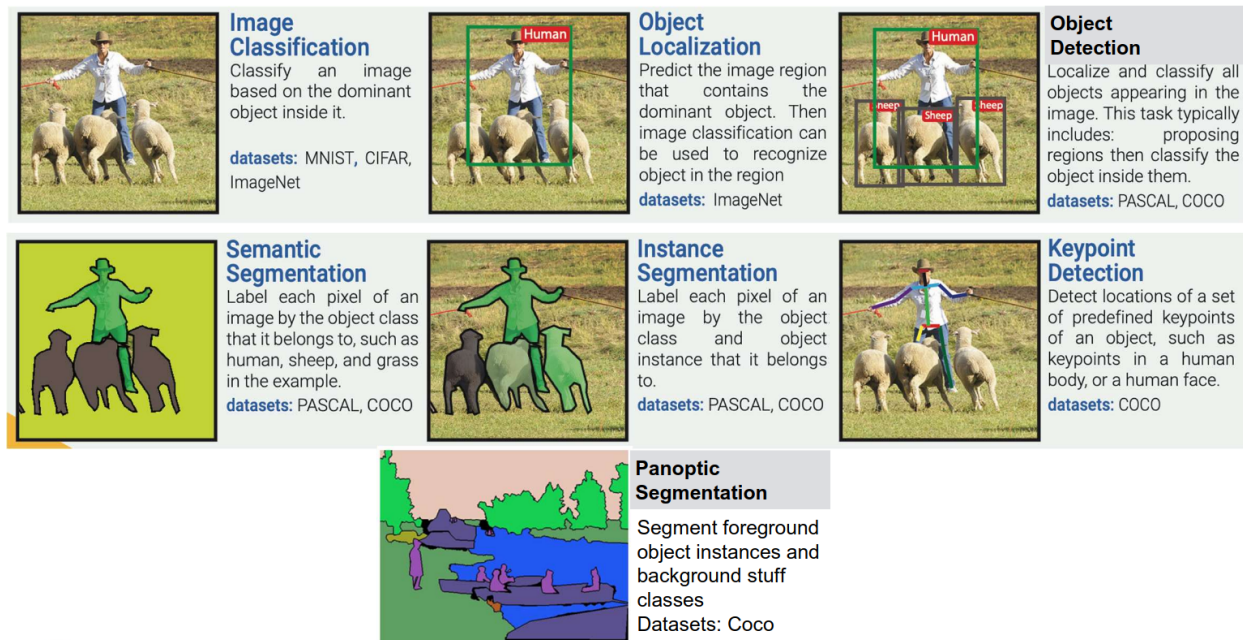


Figure 69: Summary of some of the task types in computer vision.

- Keypoint detection: detect locations of a set of predefined keypoints of an object, e.g. points on a human body or face
- COCO (Common Objects in Context) is a large-scale object detection, segmentation, and captioning dataset
- Many segmentation networks use the SegNet shape where we have downsampling and then upsampling convolutions, with a bottleneck layer in the middle
 - The simplest upsampling method is nearest neighbours, which replaces each cell with the value of its nearest neighbour, or “bed of nails”, where all cells without a value are replaced with zero
 - *Max-unpooling* is a technique where during the max pooling downsampling we remember which indices had the max value, then using a bed of nails unpooling with these indices
 - *Learnable upsampling* (aka upconvolution or transposed convolution) uses a kernel, which is multiplied by the input value and summed in the output, giving weighted copies of the filter
 - * The parameters of the kernel are learnable, hence the name

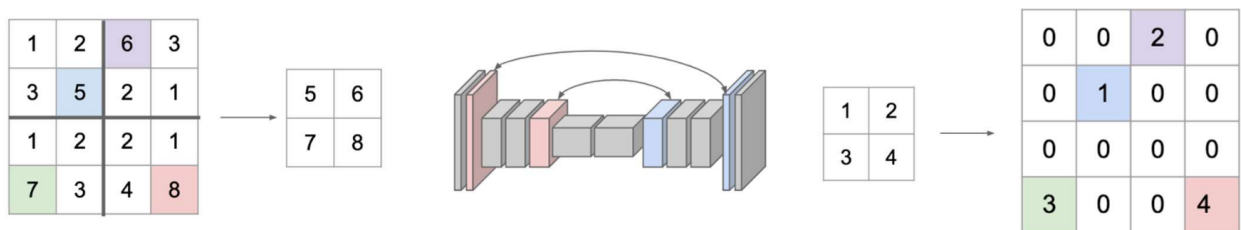


Figure 70: Illustration of max-unpooling.

Object Localization and Detection

- Image classification has long been a solved problem, but how do we move from classification to localization and recognition?
- Object localization outputs a *bounding box proposal*, which is a rectangular image region that potentially contains the object

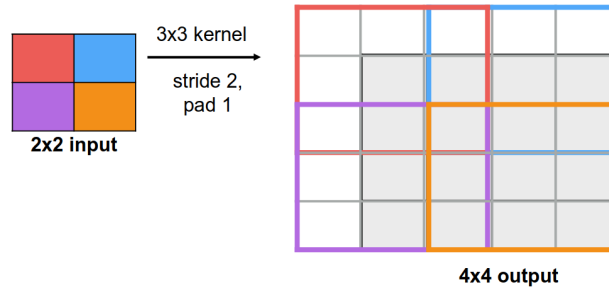


Figure 71: Illustration of learnable unsampling.

- Usually represented as a vector (x, y, w, h) along with a confidence score
- To measure accuracy, we can use the $L2$ distance of the vectors, or more commonly use the *Intersection over Union* (IoU)
- To remove duplicates, we can use non-maximum suppression, by thresholding by IoU and pick the highest confidence box
- To train an object localization network, we use a multitask loss consisting of the sum of the classification loss and bounding box regression loss
- For object detection, we need to detect all instances of objects, which we can do through region proposal methods
- The first approaches were the R-CNN family
 - In the original R-CNN, a proposal algorithm identifies potential regions of the image that can contain an object; we crop the image to each one of the regions, use a CNN and SVM to calculate the class scores and a correction to the box coordinates
 - * This is slow because we need to run it for each proposal region
 - Fast R-CNN learns a feature map for the entire image – we pass the whole image through a CNN to get a feature map, and do the same proposal + crop process on the feature map, so that we don't have to redo the CNN for each region
 - Faster R-CNN uses a neural network for the proposal as well – the feature map is passed to a region proposal network, which predicts a number of proposals, where we take the top predictions and run the same crop + detection process

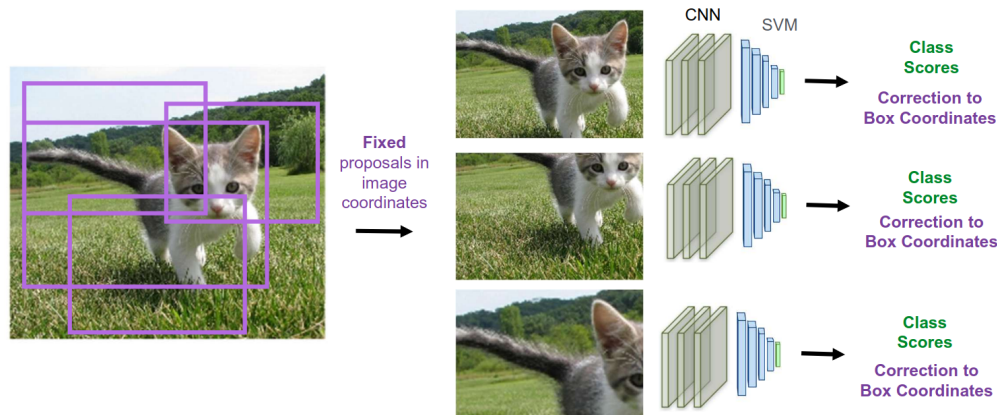


Figure 72: Structure of R-CNN.

- This gave rise to a standard architecture, where the image is passed through a feature extractor, then prior boxes are used to crop each region, which is passed through output layers, and finally non-maximum suppression for the output
 - Most common extractors were VGG, ResNet, and Inception layers back in the day; nowadays vision transformers are becoming more popular

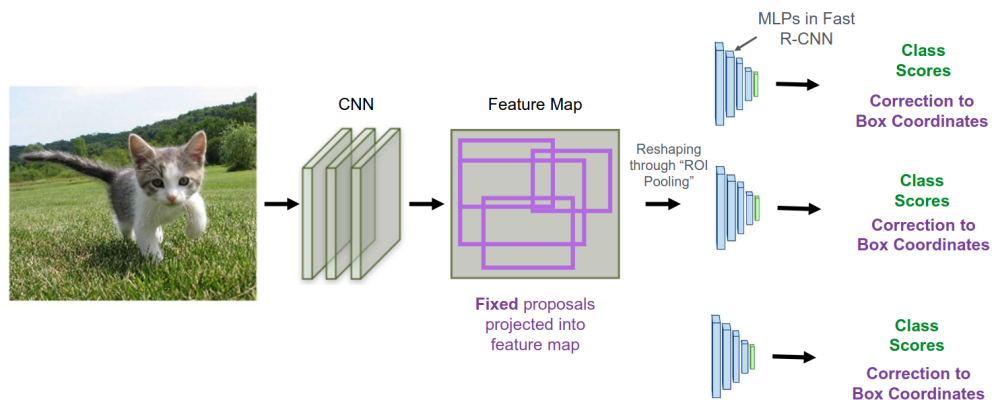


Figure 73: Structure of Fast R-CNN.

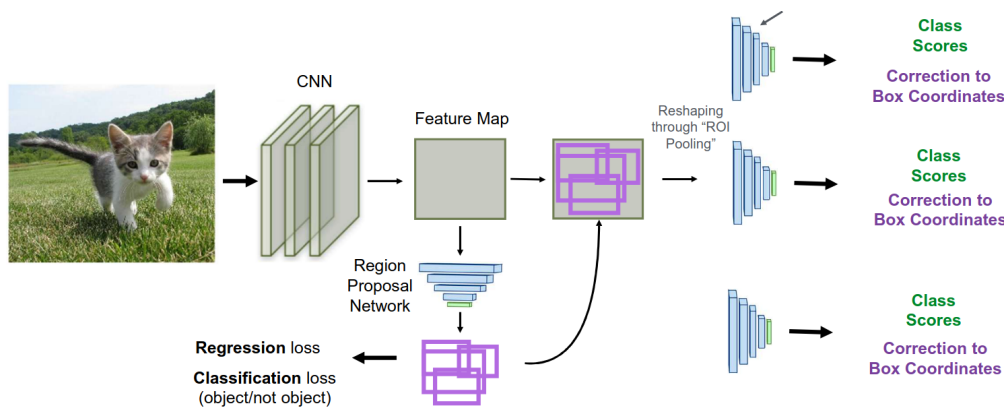


Figure 74: Structure of Faster R-CNN.

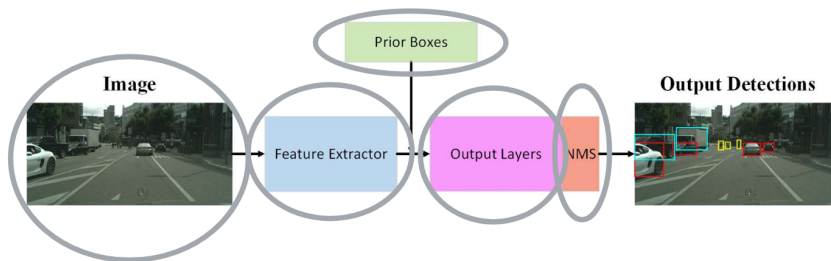


Figure 75: Standard architecture of region proposal networks.

- Output layers take a feature vector and outputs a class prediction and regresses a bounding box correction (i.e. a classification and a regression head)
- During training, we evaluate the network output on all region proposals (anchors), instead of just the best ones after NMS

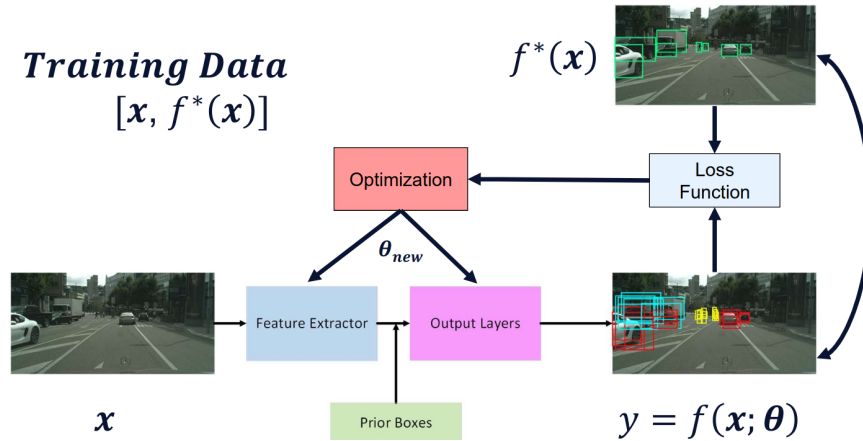


Figure 76: Training process for region proposal networks. Notice how nonmaximum suppression is not used during training.

- Since we evaluate the training on all region proposals, not just the best ones, we end up with a lot more negative examples than positive examples
 - If we have too many background proposals, it'll just learn to classify everything as negative
 - To address this, we choose the “hardest” ones, i.e. the ones where the network's output is right on the threshold for determining positive/negative
 - Usually we take a 3:1 ratio of negative to positive anchors
- To perform NMS, we sort the predictions by confidence score, then starting with the highest confidences, we include each one, then prune out all the remaining boxes that have IoU with the box we just included over a certain threshold
 - This doesn't handle overlaps very well, and introduces another threshold parameter we must tune
 - Make sure to take class labels into account to handle overlapping objects

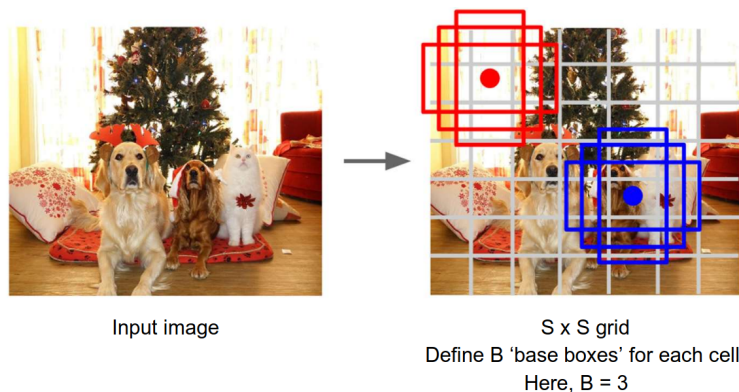


Figure 77: Architecture of YOLO.

- The other detection framework is YOLO (You Only Look Once), where detection is done without proposals
 - The original YOLO used a GoogLeNet structure to predict both bounding box coordinates and class probabilities directly from the whole image

- Since predictions are made in a single pass, single-shot approaches run much faster than region proposal-based approaches, and can run accurately in real time
- YOLO breaks the image into an $S \times S$ grid, where each grid cell predicts B possible bounding boxes and confidences, and C class probabilities (including the background class)
 - * This results in $S \times S \times (5B + C)$ total output values, since each bounding box + confidence consists of 5 values
 - * To get the final output, the bounding box confidence is multiplied by the class probability
 - * For each class the boxes with the highest combined confidence is taken, and NMS is applied to eliminate duplicates as usual
- In general, Faster R-CNN gives better accuracies but is much slower than YOLO

Lecture 22, Nov 25, 2025

3D Object Detection

- 3D object detection aims to identify the 3D position, orientation, and bounding box extent for every relevant object, using LIDAR and RGB measurements
- To quantify detection performance for all thresholds, the average precision (AP) metric can be used, which measures the area under the ROC curve
- The AVOD-FPN architecture uses proposal generation from an image feature map and a birds-eye-view LIDAR feature map, trained on the KITTI dataset
 - Proposals were anchored to the ground to significantly reduce the search area
 - Achieved 10fps at the top of the KITTI benchmark
- Training both the vision and LIDAR detectors were difficult; one method of fusing the visual and LIDAR information was to first use the 2D object detector to get masks, which are projected into the pointcloud and used to paint points with bounding box or class information
- Detection accuracy drops off sharply with distance in the LIDAR pointcloud as it becomes very sparse, so fusing 2D and 3D information can achieve better results
 - This can be done in different stages of the network, early stage (e.g. pointcloud painting), intermediate stage (e.g. fusion of feature maps) or late stage (e.g. running detections on both independently and fusing in the end)
 - Dense Voxel Fusion is an early fusion approach that uses visual detections to colour voxels in the scene to get denser data
- 3D object detection is also possible using only monocular input, where the network tries to estimate depth
 - One approach is to try to reconstruct the object using the depth prediction and comparing this with 3D models of objects constructed using LIDAR depth completion in the training data
 - Instead of generating a single depth, uncertainty can be incorporated by generating a depth distribution for each pixel (i.e. a histogram)
- Stereo matching can be enhanced with object detection, by performing instance segmentation in the two images and associating them, and performing stereo matching on the individual instances
 - Similarly this can be used to reconstruct objects and trained against completed 3D LIDAR data
 - Learned stereo depth maps tend to be smooth and have points halfway in between the background and foreground, since this results in smaller loss than having sharp separations that might be slightly wrong
- How can we get probabilistic uncertainty estimates (and not just confidences) from 3D object detection, which would allow us to fuse it properly with other sensors?
 - Instead of non-maximum suppression, we can use clustering and Bayesian fusion and treating each detection as a measurement
 - The network also regresses a variance
 - The idea is to use dropout to get different predictions, and using the difference between predictions as an estimate of uncertainty (or difficulty of estimating an object), and fusing these to get a distribution
 - Other approaches either directly generate uncertainty or output a set of samples directly

- *Domain adaptation* involves transferring knowledge from a labelled source dataset to an unlabelled target dataset, e.g. making a network trained mainly on clear weather handle adverse weather conditions
 - We can create pseudo labels generated using the teacher network
 - The student network is taught to ignore variations between the datasets
 - We can also use foundation models such as DINOv2