Good to know

1. Where and why ReLU outperforms Sigmoid in hidden layers

1. **Alleviation of the vanishing-gradient problem**
   - **Sigmoid**:

     $$\sigma(x) = \frac{1}{1 + e^{-x}}, \quad \sigma'(x) = \sigma(x)\,\big(1 - \sigma(x)\big) \leq 0.25.$$

     For large $|x|$, $\sigma'(x) \to 0$, so gradients vanish as they backpropagate through many layers.
   - **ReLU**:

     $$\mathrm{ReLU}(x) = \max(0,\, x), \quad \mathrm{ReLU}'(x) = \begin{cases} 1, & x > 0, \\ 0, & x \leq 0. \end{cases}$$

     When active ($x > 0$), the gradient is constant 1, so it preserves gradient magnitude better in deep nets.
2. **Sparse activations and computational efficiency**
   - ReLU sets all negative pre-activations to zero, producing sparsity in the network. Sparse activations reduce inter-neuron dependencies and can speed up both forward and backward passes.
   - The piecewise-linear form of ReLU is cheaper to compute than the exponential in $\sigma(x)$.
3. **Empirical performance in deep architectures**
   - Modern deep convolutional and feed-forward networks (e.g. AlexNet, ResNet) almost universally use ReLU (or its variants), because it leads to faster convergence and often better generalization.

---

2. When Logistic (Sigmoid) is preferable

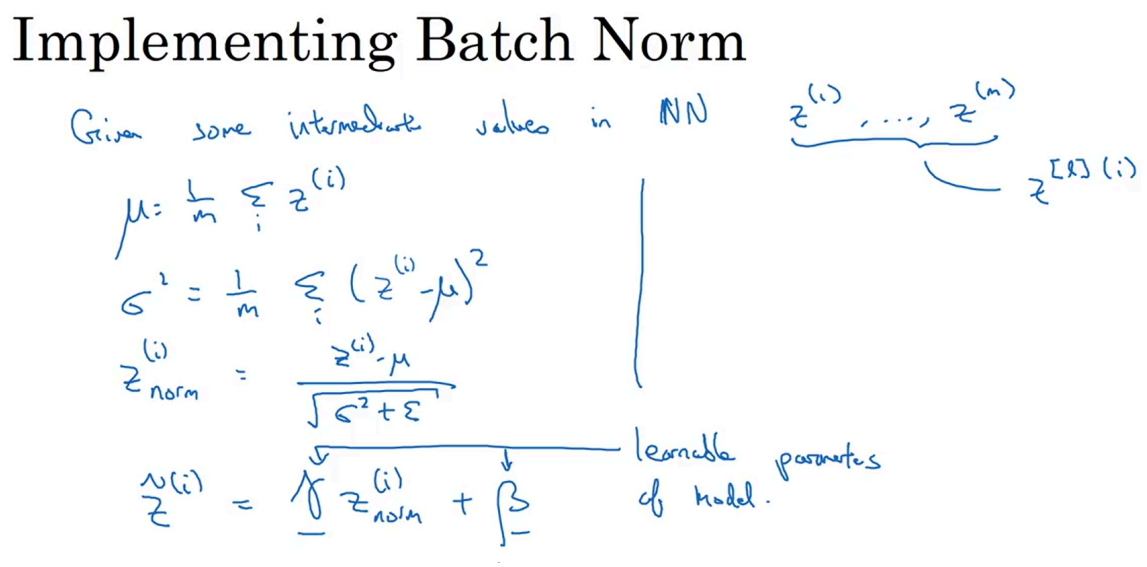
1. **Output layers for probabilistic interpretation**
   - For binary classification, the output neuron often needs to produce a probability in $[0, 1]$. The sigmoid's range matches this requirement directly, making it the natural choice for the final layer when used with cross-entropy loss.
2. **Gating mechanisms in recurrent networks**
   - In architectures like LSTMs or GRUs, gates (forget, input, output) must produce values in $[0, 1]$ to modulate information flow. Logistic activations perfectly serve this gating role, whereas ReLU's unbounded positive output and zero gradient for negatives would break the soft gating behavior.
3. **Shallow or specialized models**
   - In some shallow networks or autoencoders where depth is limited, vanishing gradients are less severe. Sigmoids can still work reasonably, especially when you need bounded, smooth outputs or want to exploit their probabilistic interpretation in energy-based models (e.g. Boltzmann machines).

Mathematical summary

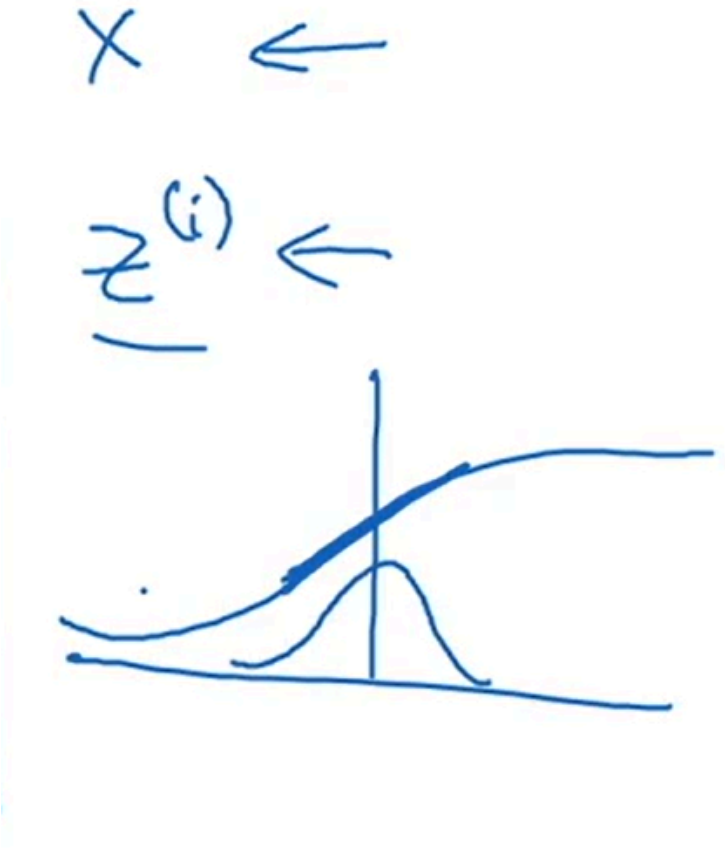
Property	Sigmoid $\sigma(x)$	ReLU $\max(0, x)$
Range	$(0, 1)$	$[0, \infty)$
Derivative	$\sigma(x)(1 - \sigma(x)) \leq 0.25$	1 for $x > 0$, else 0
Vanishing gradient	Severe for (	x
Sparsity	No	Yes (zeros for $x \leq 0$)
Computational cost	Higher (exponential)	Lower (simple comparison)

By choosing ReLU in deep hidden layers, you maintain stronger gradients, encourage sparsity, and speed up training. Conversely, Sigmoid remains indispensable where bounded outputs or soft gating are required.

Batch Norm (1)

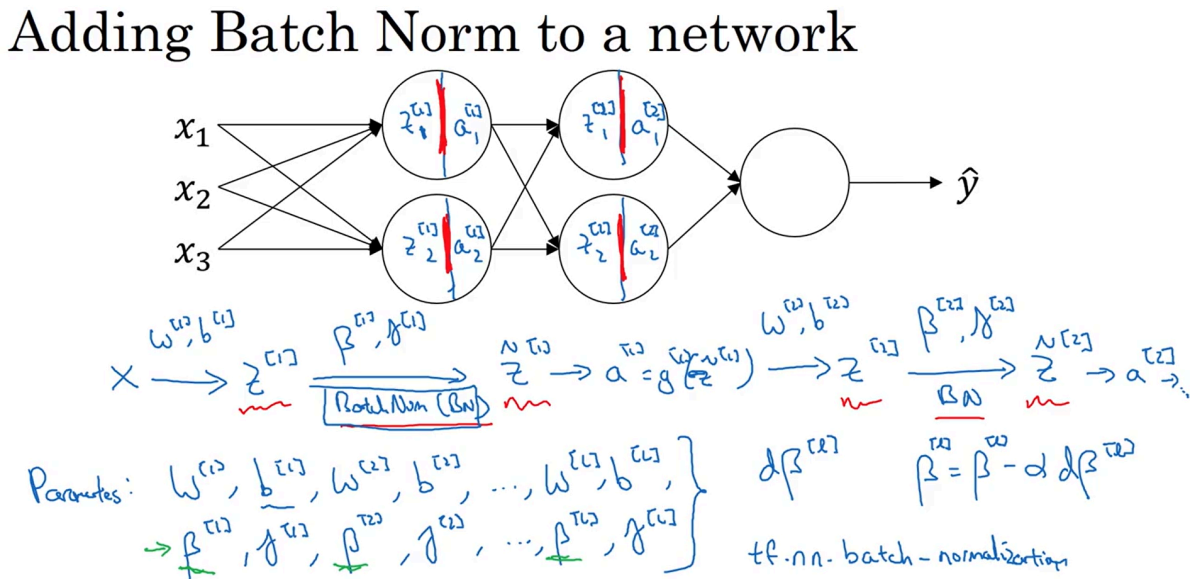


If gamma were sqrt(sigmasqrd + epsilon) and beta were mu, then no effect.

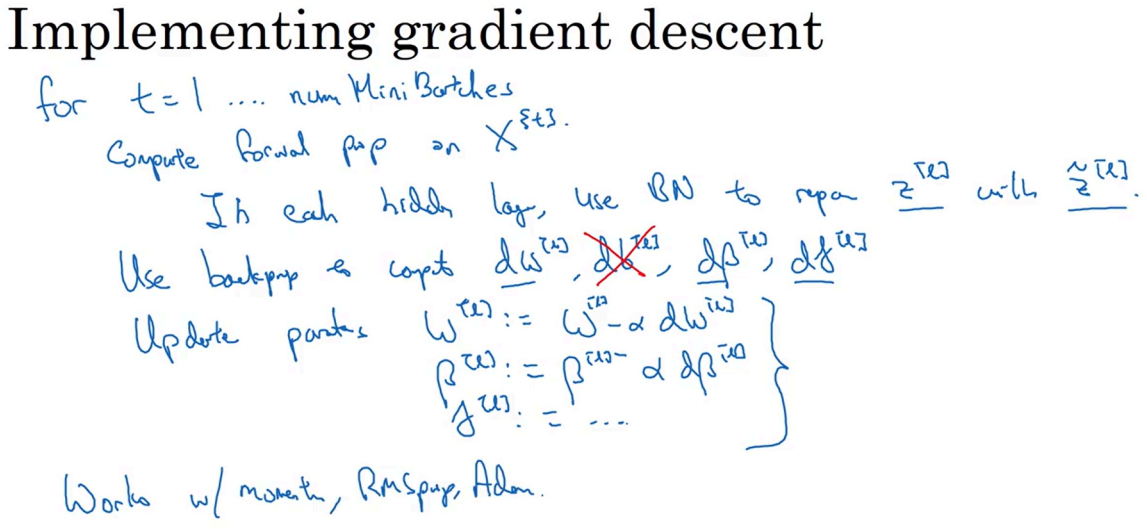


But we need to be careful to avoid the above situation (where the values are clustered around 0) and we are unable to take advantage of the non-linearity of sigmoid. That's where gamma and beta come into play

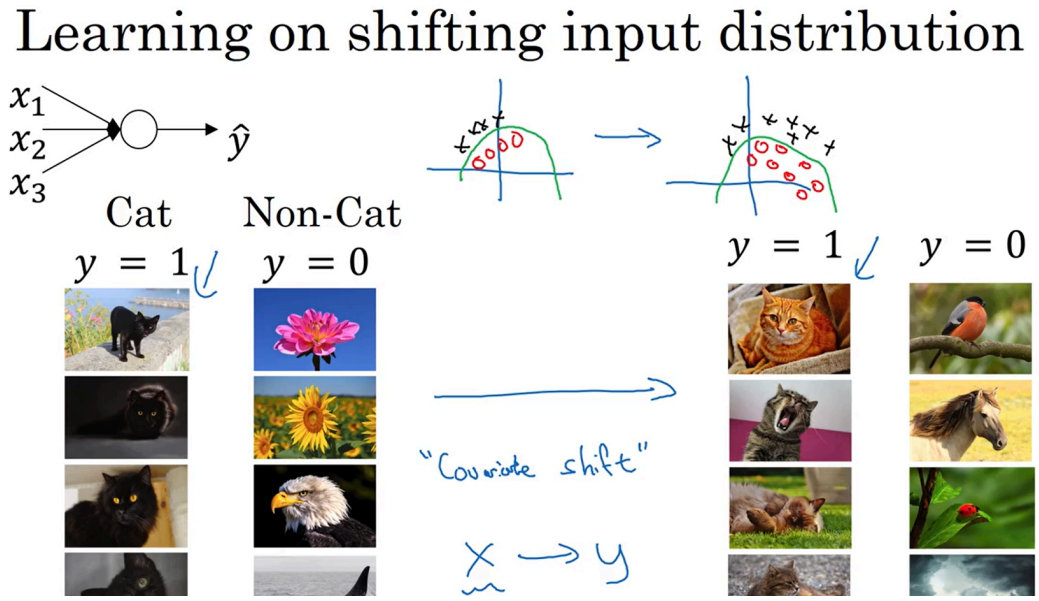
Adding BatchNorm to a network



If you use batch norm, just can ignore b

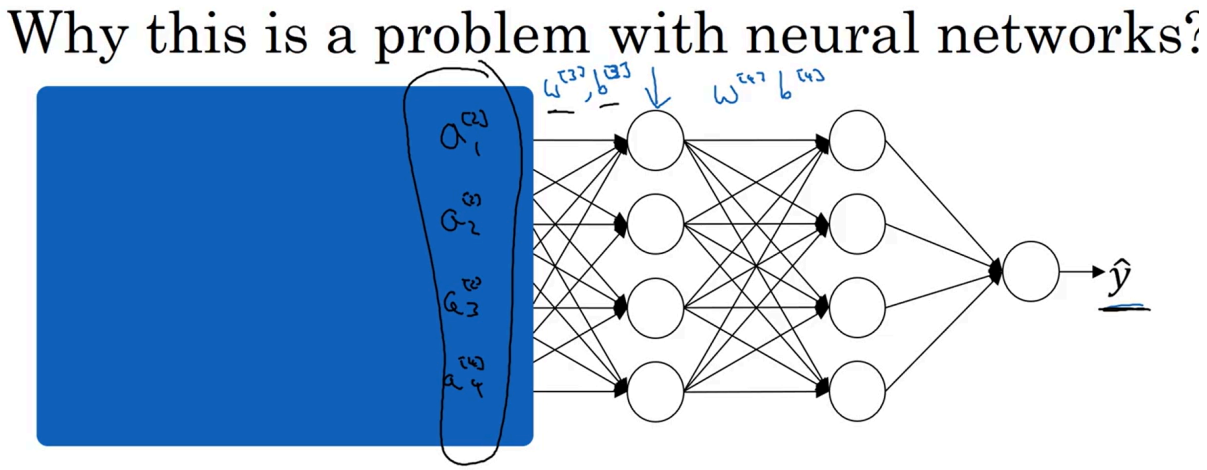


Why does BatchNorm work?



If you train to identify black cats and you test on coloured cats (:D), then your classifier may not work properly mah!

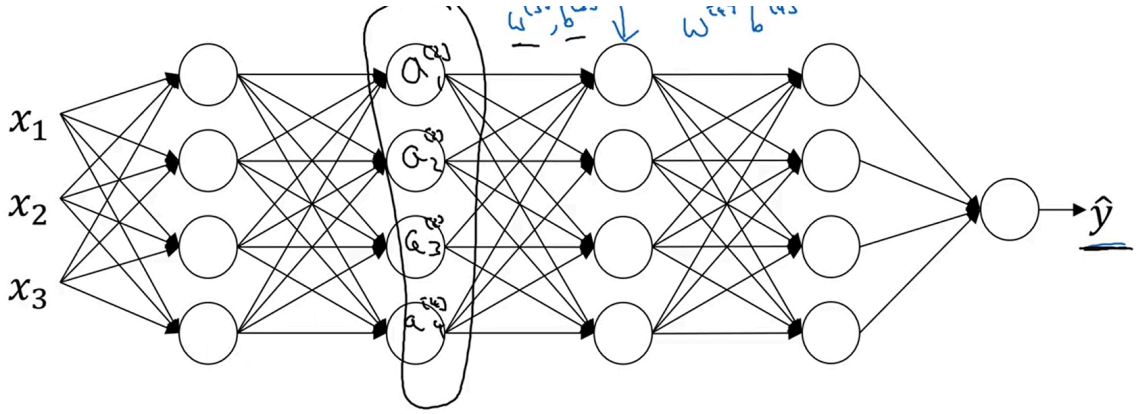
The change of the distribution from X (containing black cats) to X′ (containing coloured cats is known as covariate shift)



Ok bah?

The NN needs to try to work with the activations (can be interpreted as features). Job of the third layer needs to take these values and try to map to y

Let's uncover the blue mah?



From the view of the third layer, the inputs to it always keep changing! Come on lor! This is like that covariate shift bah!

So what batch norm does it that it reduces the amount that the distribution of hidden unit values shift around.

This is done by ensuring that the mean and variance doesn't change lor! Mean depends on gamma and beta for that BN layer

It limits the amount of how changing parameters in an earlier layer will affect the inputs to the current layer

i.e weakens coupling between weights parameters of learning almost making learning independent b/w layers.

Ok bah, got it? Sike! Here is another usage of BN! (You have more to think about lol)

Batch Norm as regularization

- Each mini-batch is scaled by the mean/variance computed on just that mini-batch.
- This adds some noise to the values $z^{[l]}$ within that minibatch. So similar to dropout, it adds some noise to each hidden layer's activations.
- This has a slight regularization effect.

Noise is added because the normalization is done w.r.t to the current minibatch and not w.r.t to the entire training set. (i.e estimate of the mean and st.dev is noisy lor!) This noise forces downstream hidden units not to be dependent on one hidden unit!

(So the larger mini-batch size you use, the less regularization you do! Ok mah? You seem confused lor!)

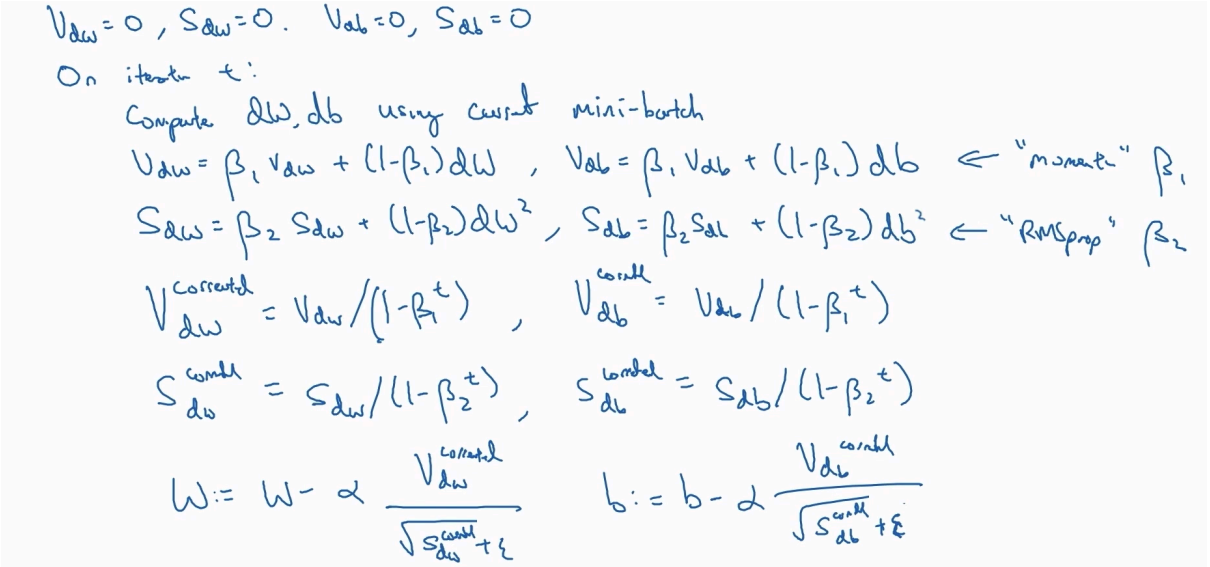
BN → Multiplicative noise + Additive noise

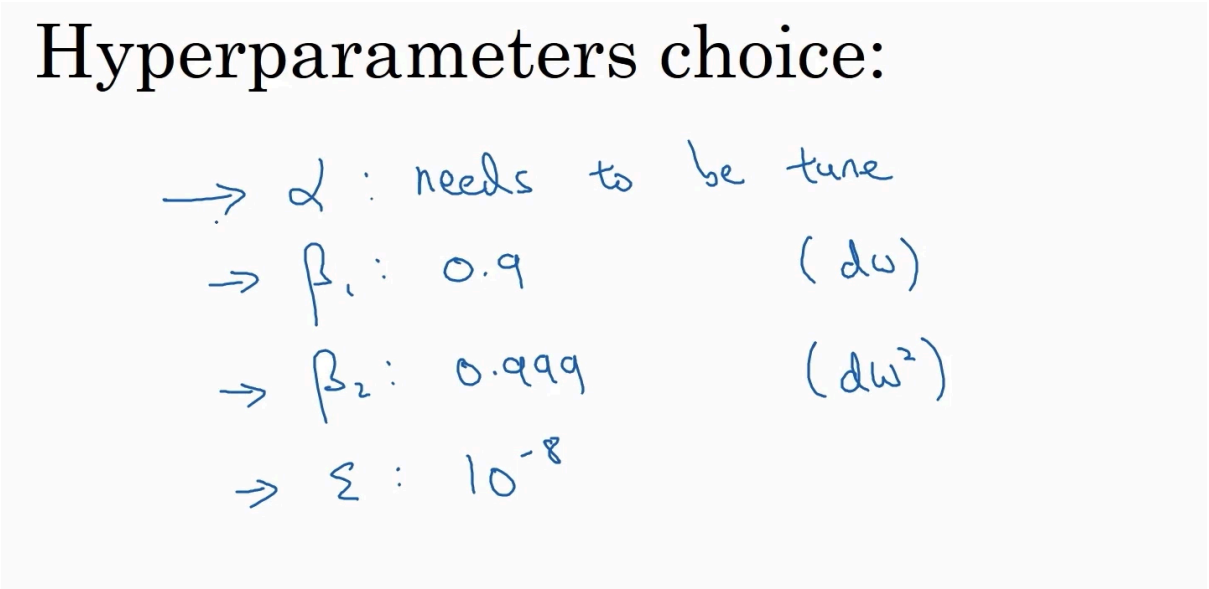
Dropout → Additive noise

Adam Optimizer

Adam = RMSProp + Momentum (Adaptive Moment Estimation)

- Start by initializing V_dW, S_dW and V_db, S_db to all zero
- Then do the Momentum calculations for V_dW, V_db
- Then do the RMS calculations for S_dW, S_db
- Then do bias corrections for all
- Then just combine Momentum update with RMS update 😀

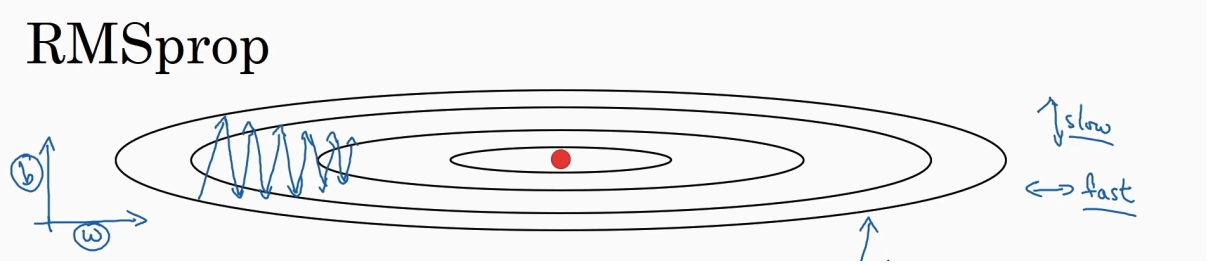




Only need to tune alpha usually. b1 and b2 were the default suggestions by the authors of the Adam optimizer paper

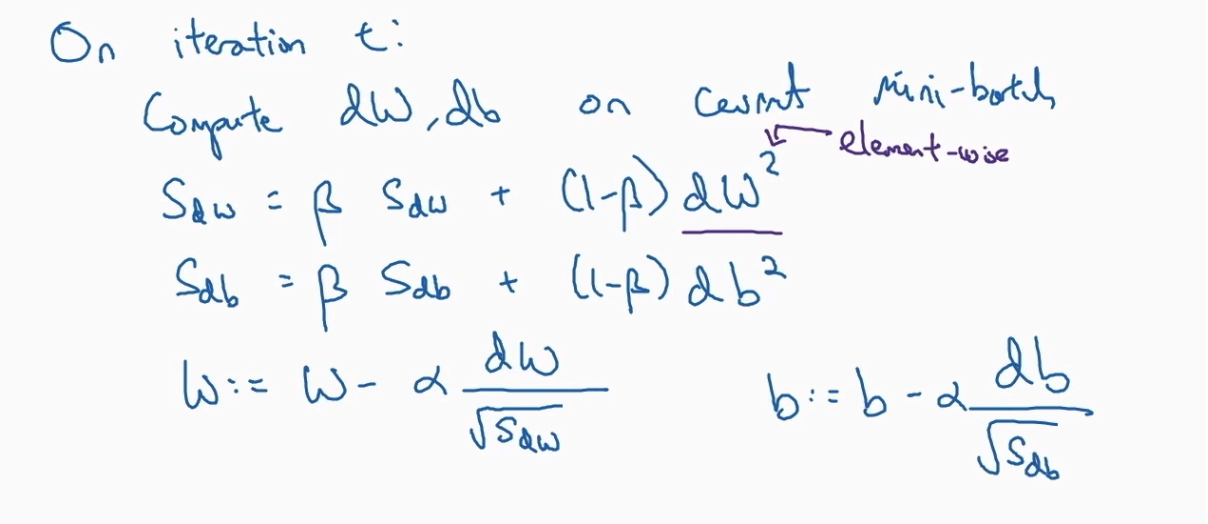
RMSProp

RMSPop → Root Mean Square Propagation



Want to:

Slow down learning in the b direction and increase learning in the W direction.



Do note that the squaring is done element wise!

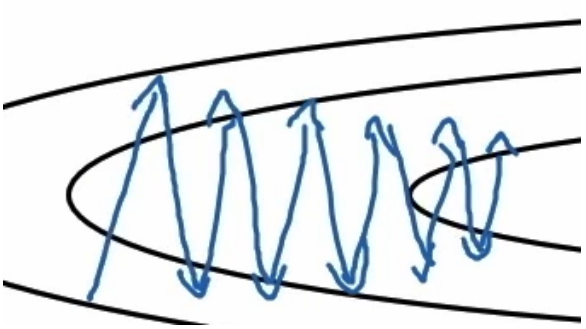
So what this is doing is keeping an EWA of the SQUARES of the derivatives.

Then look carefully at the update rule! We don't directly update by the EWA!

So, goal is to increase learning in the W direction and decrease/dampen learning in the b direction.

So what we hope is that S_dW is small so that when you divide by a smaller number, then the update to W is large. And we hope S_db is large so that when you divide by a larger number, then the update to b is small in order to slow down updates in the vertical direction.

And indeed, the dW's are much smaller than the db's!



Here, db is much larger compared to dW

Therefore oscillations in the vertical direction get damped out!

Think of this as adaptive learning rates!

Gradient Descent w/ Momentum

Almost always works faster than gradient descent.

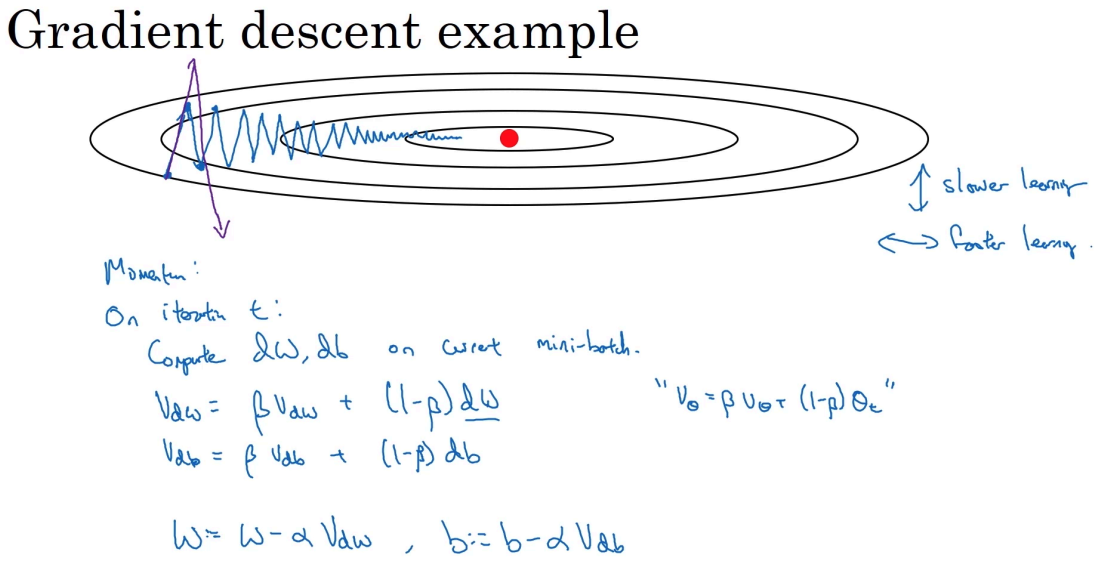
Basic idea: compute an EWA of gradients and use that gradient to update weights instead.'

I.e smooth out gradient descent

To prevent oscillations, we need the vertical learning to be slower and we need the horizontal learning to be faster.

To do this, for every iteration

- Just compute the gradients dW and db on current minibatch
- Simply get the EWA V_dW = beta V_dW + (1-beta) dW
- And same for the EWA V_db
- Then do the updates W = W - lr * V_dW and b = b - alpha * V_db



Eg: averaging over these gradients will cause vertical component to be almost zero and horizontal to be much more.

Now two hyperparameters: alpha (lr) and beta (most common in 0.9, average over last 10)

Implementation details

On iteration t:

Compute dW, db on the current mini-batch

$v_{dW} = \beta v_{dW} + (1 - \beta) dW$

$v_{db} = \beta v_{db} + (1 - \beta) db$

$W = W - \alpha v_{dW}, \quad b = b - \alpha v_{db}$

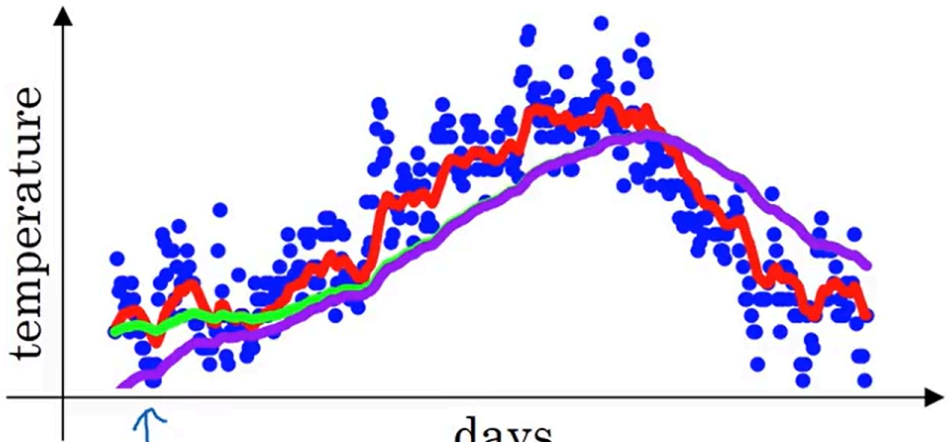
Hyperparameters: α, β $\qquad\qquad$ $\beta = 0.9$

Bias correction in exponentially weighted averages

beta = 0.9 → red line

beta = 0.98 → green line (actually you don't get the green line, you get the purple line)

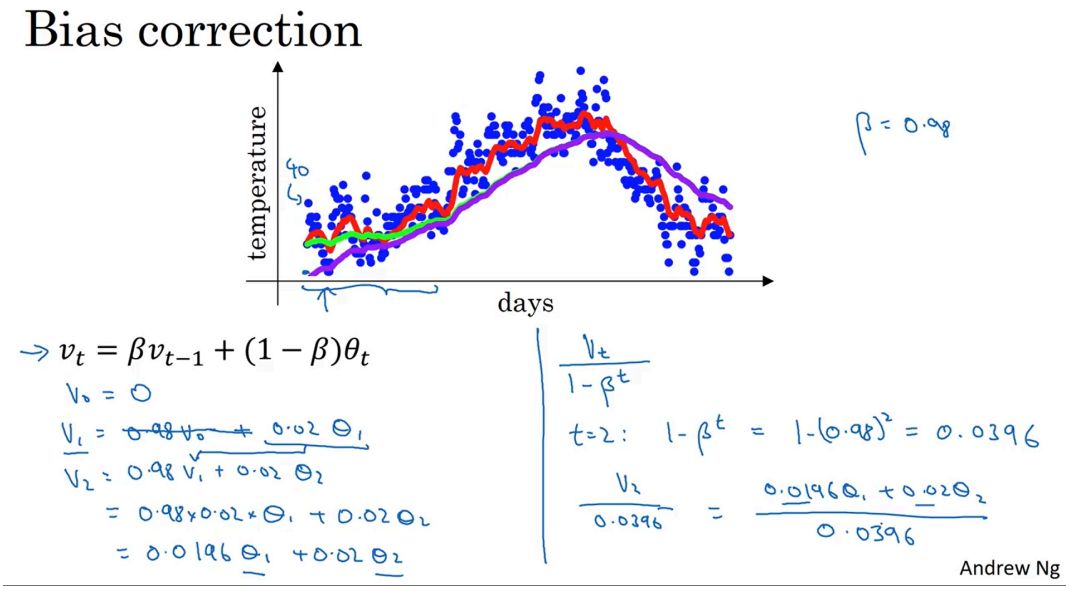
The purple line starts off low.



Initial EWA is bad, therefore use the following formula:

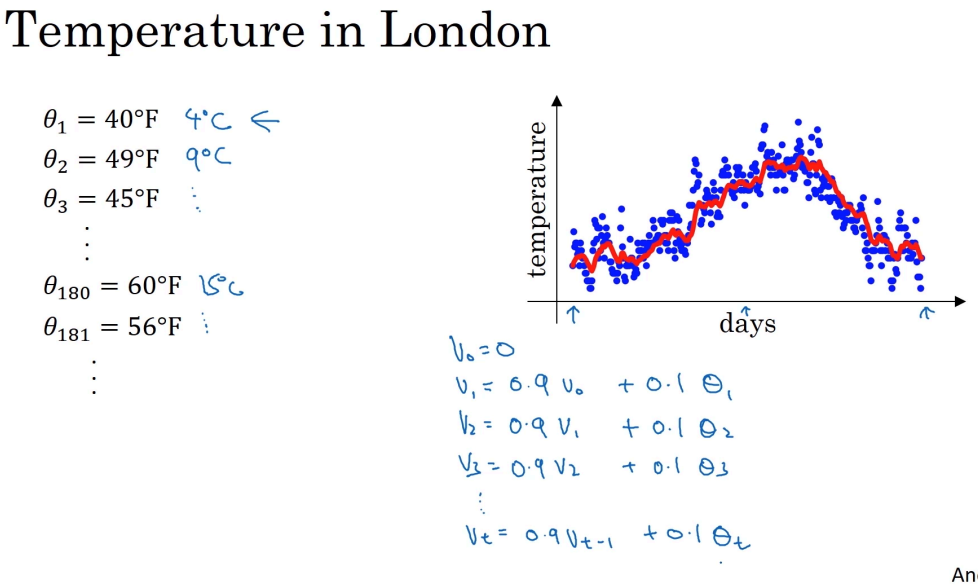
(essentially do the same EWA as before, but just divide each of those v_t's by (1-beta^t))

And also note that 1-beta^t is the sum of the coefficients of the summands, so it is a weighted average—it's at the scale of new temperatures.



Now, as t becomes large 1-beta^t will be 1 meaning bias correction is switched off.

Exponentially weighted averages

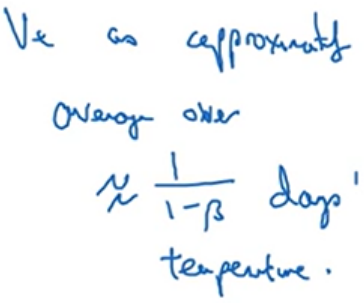


Andrew Ng

The red line is the exponentially weighted average

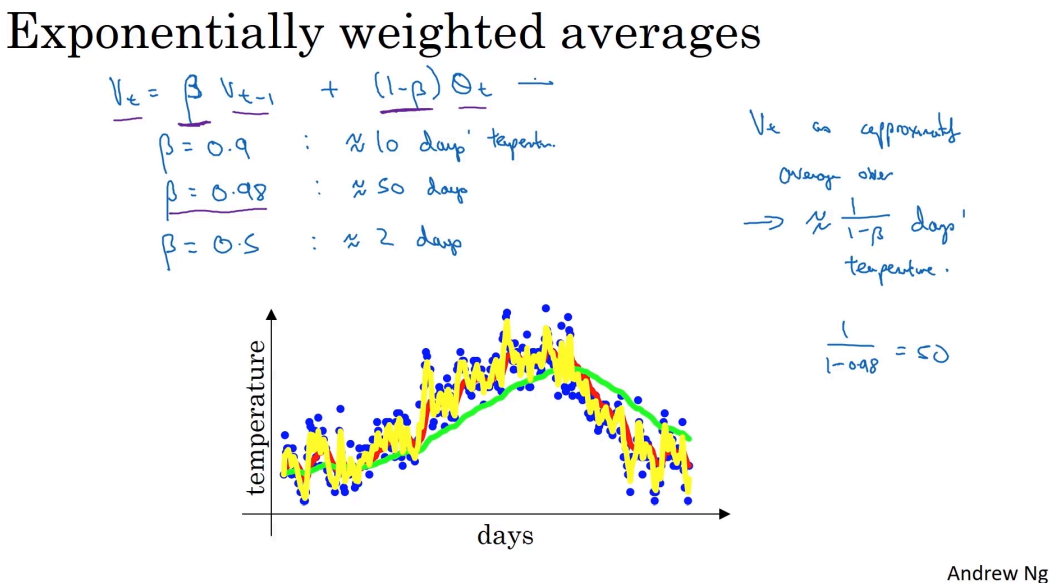
$$V_t = \beta V_{t-1} + (1-\beta)\theta_t$$

Here you can think of V_t as averaging over the past 1/(1-beta) days' temperature.

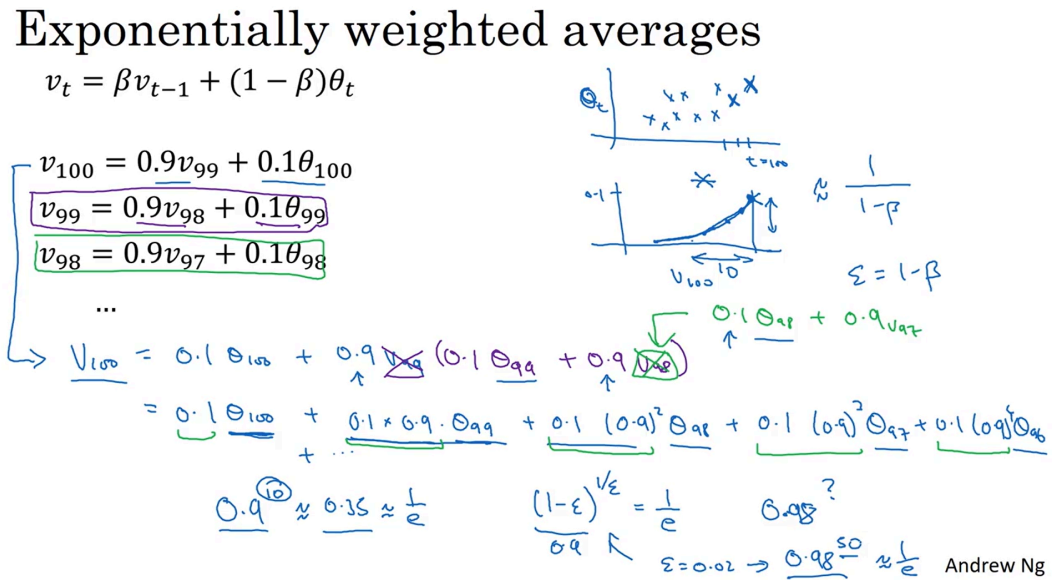


High beta → (eg: green line), the plot is smoother because it keeps track of a longer range of days' temperatures. Adapts slowly to newer temperatures

Very low beta → (eg: red line), very noisy yellow line since you are very susceptible to new temperatures.



Intuition for EWAs



Explanation of the graphs on the top right

Top graph: just the temperature measurements over time.

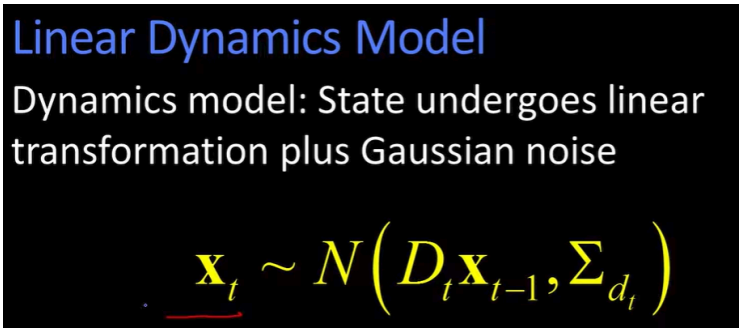
Bottom one: exponentially decaying function of weightage factor for each of the elements on top.

So v_100 = element wise product of those two graphs and sum them up

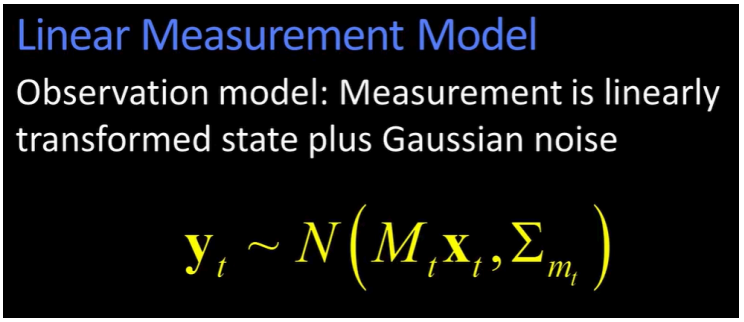
Kalman Filters

For the linear case, we have two models.

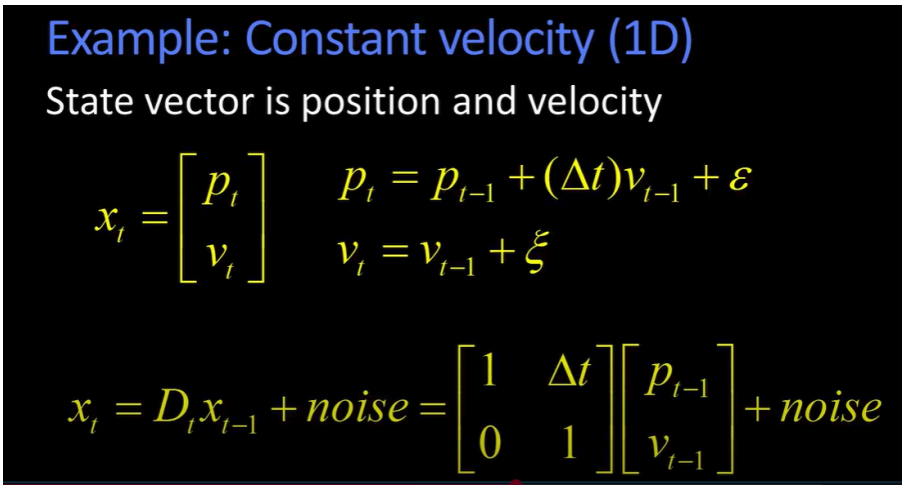
The linear dynamics model (how the state evolves over time)



The linear measurement mode



Constant Velocity (1D example)



Example: Constant velocity (1D)

Measurement is position only

$$y_t = Mx_t + noise = \begin{bmatrix} 1 & 0 \end{bmatrix} \begin{bmatrix} p_t \\ v_t \end{bmatrix} + noise$$

Constant Acceleration (1D example)

Example: Constant acceleration (1D)

State vector is position, velocity & acceleration

$$x_t = \begin{bmatrix} p_t \\ v_t \\ a_t \end{bmatrix} \qquad \begin{aligned} p_t &= p_{t-1} + (\Delta t) v_{t-1} + \varepsilon \\ v_t &= v_{t-1} + (\Delta t) a_{t-1} + \xi \\ a_t &= a_{t-1} + \zeta \end{aligned}$$

$$x_t = D_t x_{t-1} + noise = \begin{bmatrix} 1 & \Delta t & 0 \\ 0 & 1 & \Delta t \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} p_{t-1} \\ v_{t-1} \\ a_{t-1} \end{bmatrix} + noise$$

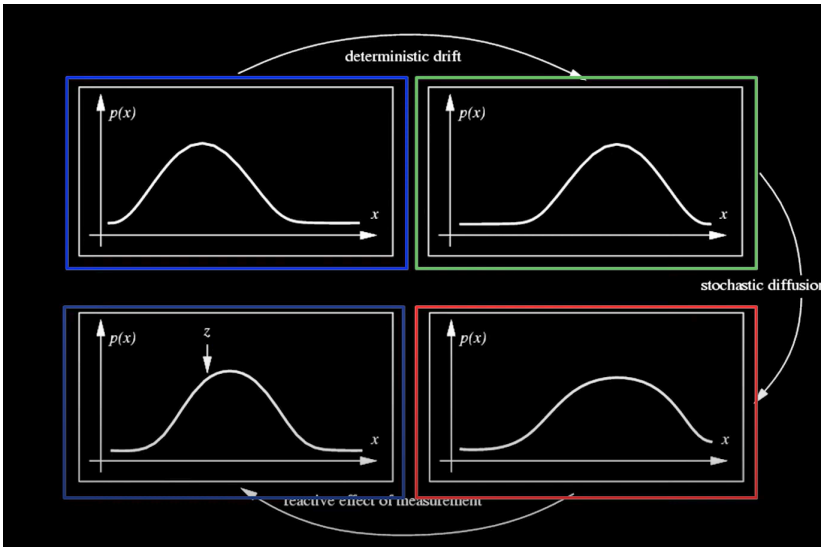
Example: Constant acceleration (1D)

Measurement is position only

$$y_t = Mx_t + noise = \begin{bmatrix} 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} p_t \\ v_t \\ a_t \end{bmatrix} + noise$$

Kalman Filter

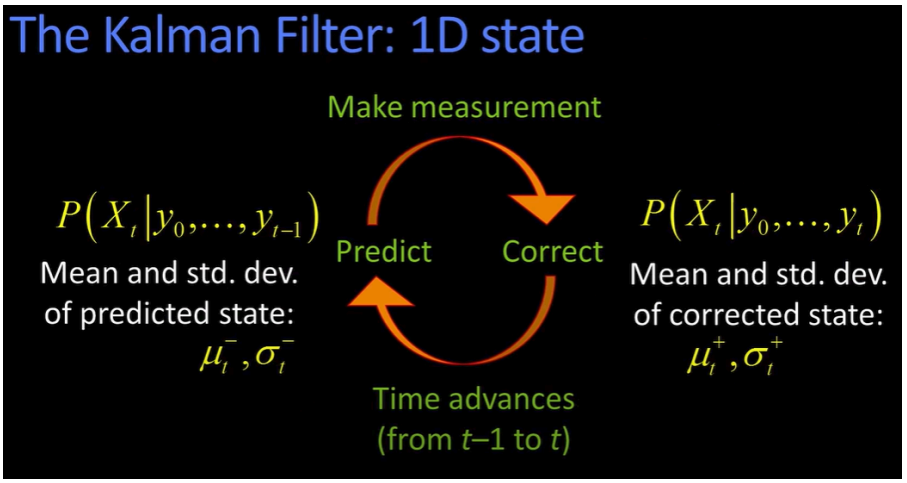
Method to track linear dynamical systems with GAUSSIAN NOISE. Both predicted and corrected states are both gaussian.



So you shift (scale mean) + expand (add noise) + measure + correct + shrink (because new information will only reduce uncertainty)

The first 3 images correspond to prediction.

Kalman Filter 1D State



Note that prediction uses mu minus and sigma minus (mean and covariance BEFORE I take the measurement, this is based on mu plus and sigma plus at t -1)

Then after you make measurement and correct mean to mu plus and sigma plus at t

1D Kalman Filter Prediction

In this linear dynamics model the next state is gotten by multiplying the previous state by a factor of d and adding some noise modeled by a gaussian.

1D Kalman Filter: Prediction

Linear dynamics model defines predicted state evolution, with noise $X_t \sim N\left(dx_{t-1}, \sigma_d^2\right)$

So, if that's the case, the distribution of the next predicted state is also gaussian:

The distribution for next predicted state is also a Gaussian

$$P\left(X_t \middle| y_0, \ldots, y_{t-1}\right) = N\left(\mu_t^-, (\sigma_t^-)^2\right)$$

Crucially note that they are mu and sigma minus because this prediction was made before measurement

And remember that we need to update mu and sigma

Update the mean: $\mu_t^- = d\mu_{t-1}^+$

Update the variance: $(\sigma_t^-)^2 = \sigma_d^2 + (d\sigma_{t-1}^+)^2$

1D Kalman Correction

In this linear dynamics model, the measurement is gotten by multiplying the previous state by a factor of m and adding some gaussian noise

1D Kalman Filter: Correction

Mapping of state to measurements: $Y_t \sim N\left(mx_t, \sigma_m^2\right)$

Now, given our predicted state (that we got above), we need to correct it using the measurement!

Predicted state: $P(X_t | y_0, \ldots, y_{t-1}) = N(\mu_t^-, (\sigma_t^-)^2)$

Want to estimate corrected distribution:

$$P(X_t | y_0, \ldots, y_t) = \frac{P(y_t | X_t) P(X_t | y_0, \ldots, y_{t-1})}{\int P(y_t | X_t) P(X_t | y_0, \ldots, y_{t-1}) dX_t}$$

Now, the corrected distribution is defined to be a new Gaussian with new mean mu plus and sigma plus.

Kalman: With linear, Gaussian dynamics and measurements, the corrected distribution to be:

$$P(X_t | y_0, \ldots, y_t) \equiv N(\mu_t^+, (\sigma_t^+)^2)$$

Formula for mu plus and sigma plus

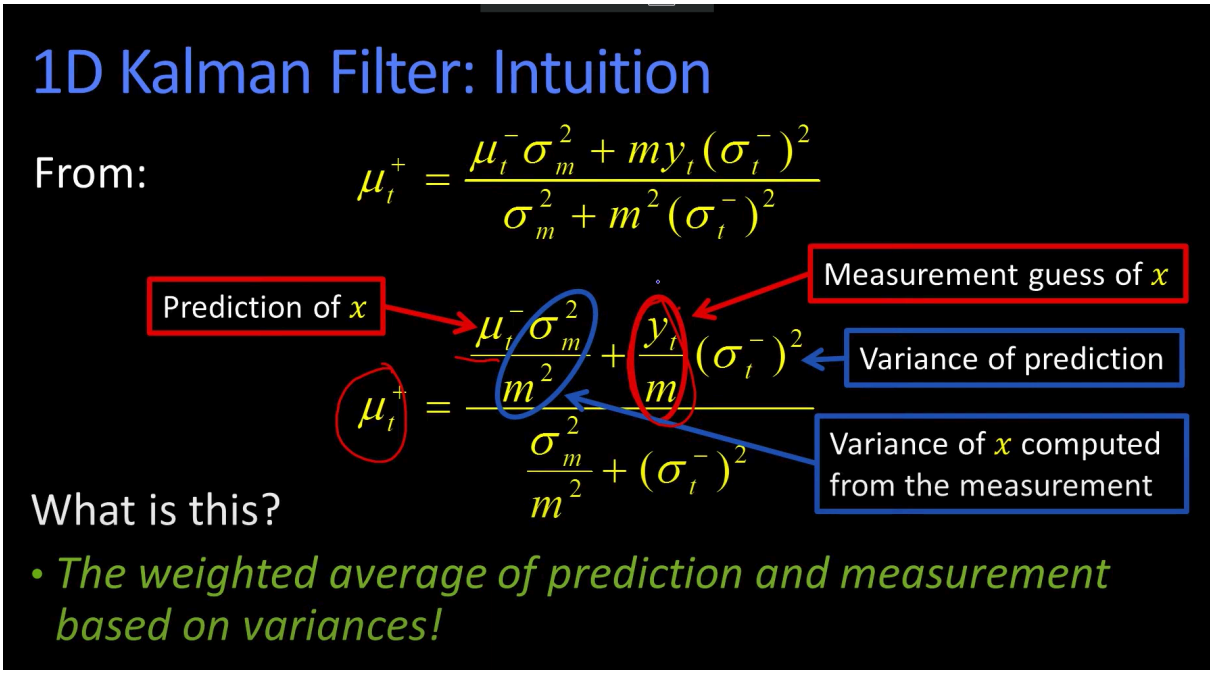
Update the mean:

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

Update the variance:

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

1D Kalman Intuition



Weights Based on Variances: The crucial insight is how the weights are determined. They are inversely proportional to the uncertainty (variance) of each estimate:

- The weight for the prediction (μt−) is proportional to the variance of the measurement (σm2).
- The weight for the measurement-derived estimate (yt/m) is proportional to the variance of the prediction ((σt−)2).

This means:

- If the measurement is very certain (small σm2), the weight for the prediction is smaller, and the weight for the measurement is larger. The filter trusts the measurement more.
- If the prediction is very certain (small (σt−)2), the weight for the measurement is smaller, and the weight for the prediction is larger. The filter trusts the prediction more.

Special Cases

Prediction vs. correction

$$\mu_t^+ = \frac{\mu_t^- \sigma_m^2 + m y_t (\sigma_t^-)^2}{\sigma_m^2 + m^2 (\sigma_t^-)^2} \qquad (\sigma_t^+)^2 = \frac{\sigma_m^2 (\sigma_t^-)^2}{\sigma_m^2 + m^2 (\sigma_t^-)^2}$$

What if there is no prediction uncertainty? $(\sigma_t^- = 0)$

$$\mu_t^+ = \mu_t^- \qquad (\sigma_t^+)^2 = 0$$

The measurement is ignored!

What if there is no measurement uncertainty? $(\sigma_m = 0)$

$$\mu_t^+ = \frac{y_t}{m} \qquad (\sigma_t^+)^2 = 0$$

Simplification of Mu+ and Sigma+

1D Kalman Filter: Intuition

Also:
$$\mu_t^+ = \frac{\frac{\mu_t^- \sigma_m^2}{m^2} + \frac{y_t}{m}(\sigma_t^-)^2}{\frac{\sigma_m^2}{m^2} + (\sigma_t^-)^2}$$

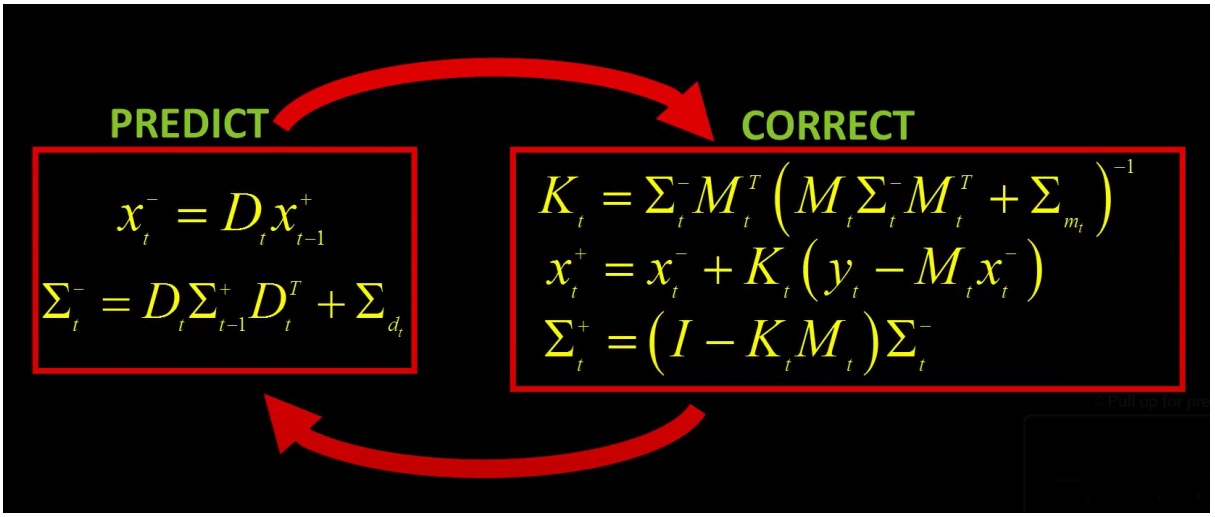
$$\mu_t^+ = \frac{a\mu_t^- + b\frac{y_t}{m}}{a+b} = \frac{(a+b)\mu_t^- + b(\frac{y_t}{m} - \mu_t^-)}{a+b}$$

$$\mu_t^+ = \frac{a\,\mu_t^- + b\frac{y_t}{m}}{a+b} = \frac{(a+b)\,\mu_t^- + b(\frac{y_t}{m} - \mu_t^-)}{a+b}$$

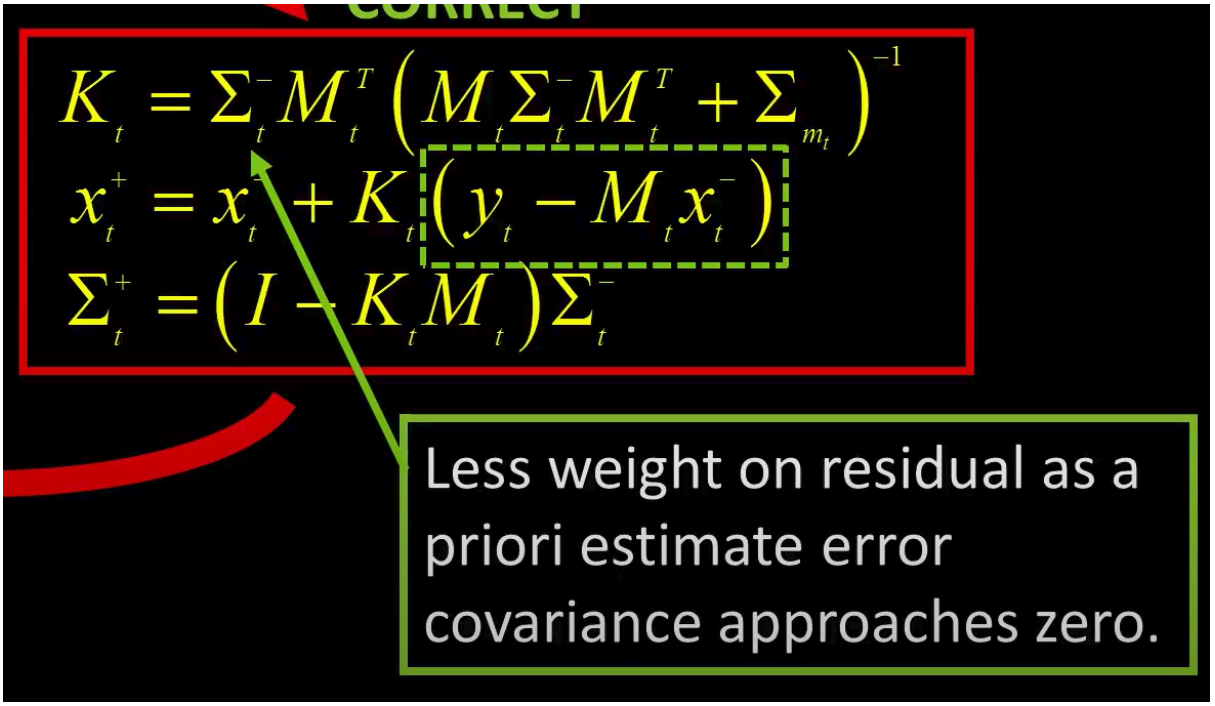
$$\mu_t^+ = \mu_t^- + \frac{b(\frac{y_t}{m} - \mu_t^-)}{a+b} = \mu_t^- + k(\frac{y_t}{m} - \mu_t^-)$$

Mu t - is the prediction. The difference (red) ika the residual. The residual is the difference between yt/m (the measurements guess of x) and mu t -. Something like TD updates.

N-Dimensional Kalman Filters (the real deal)

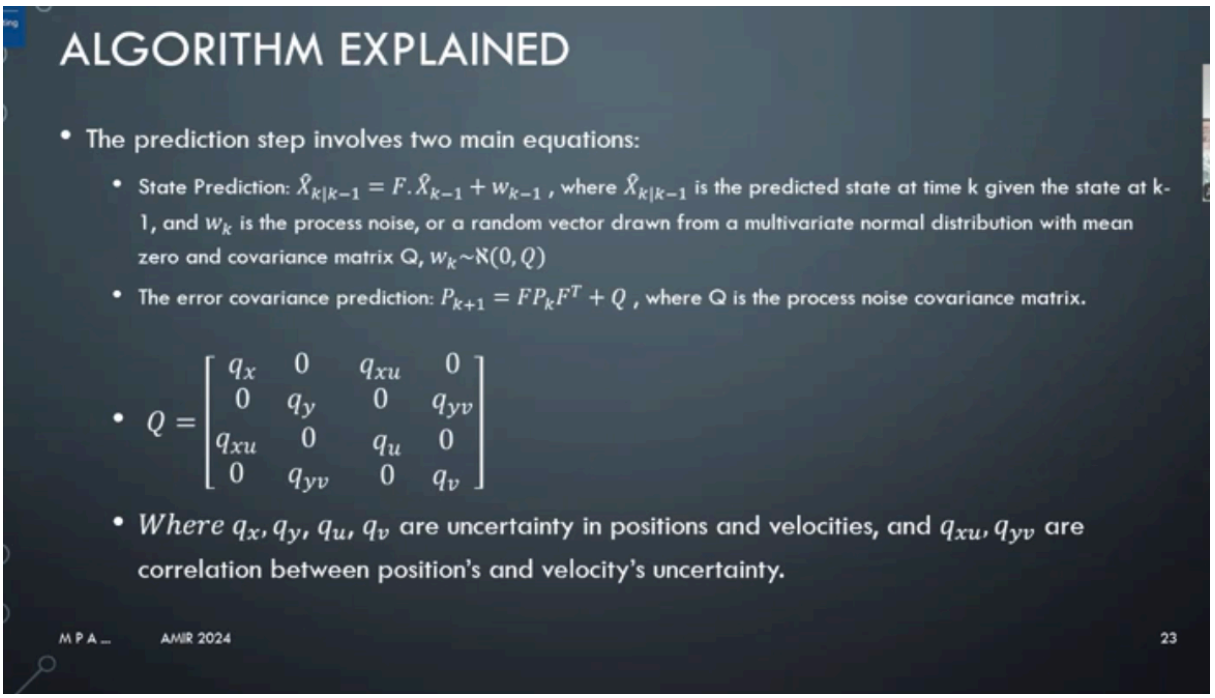


Σt−: This is the **predicted state covariance matrix** at time t. It represents the uncertainty in the predicted state xt−. The covariance matrix describes the variances of each state variable (diagonal elements) and the covariances between them (off-diagonal elements).

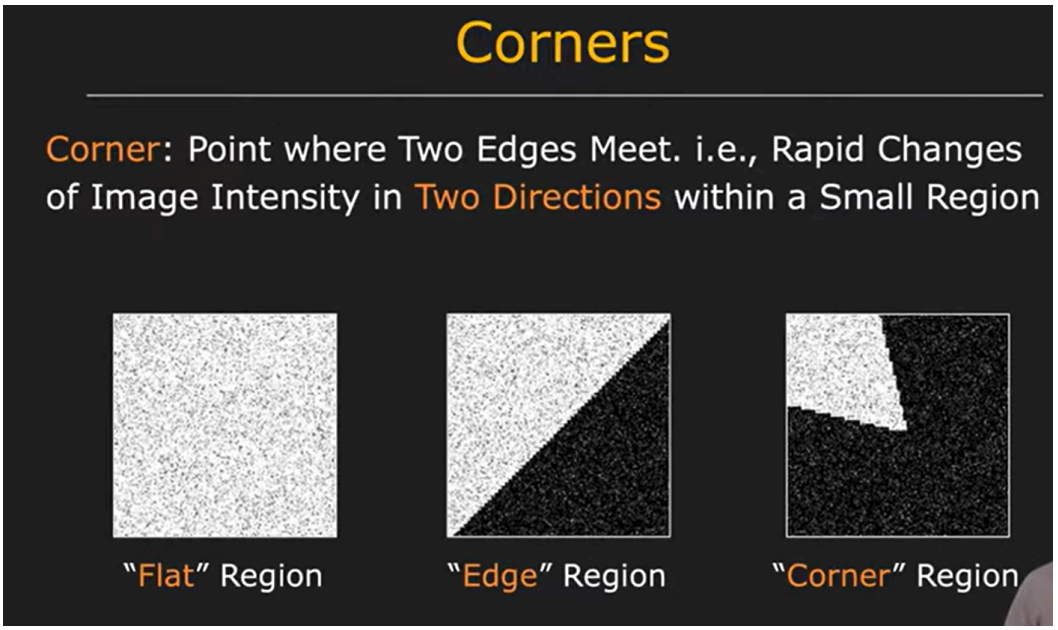


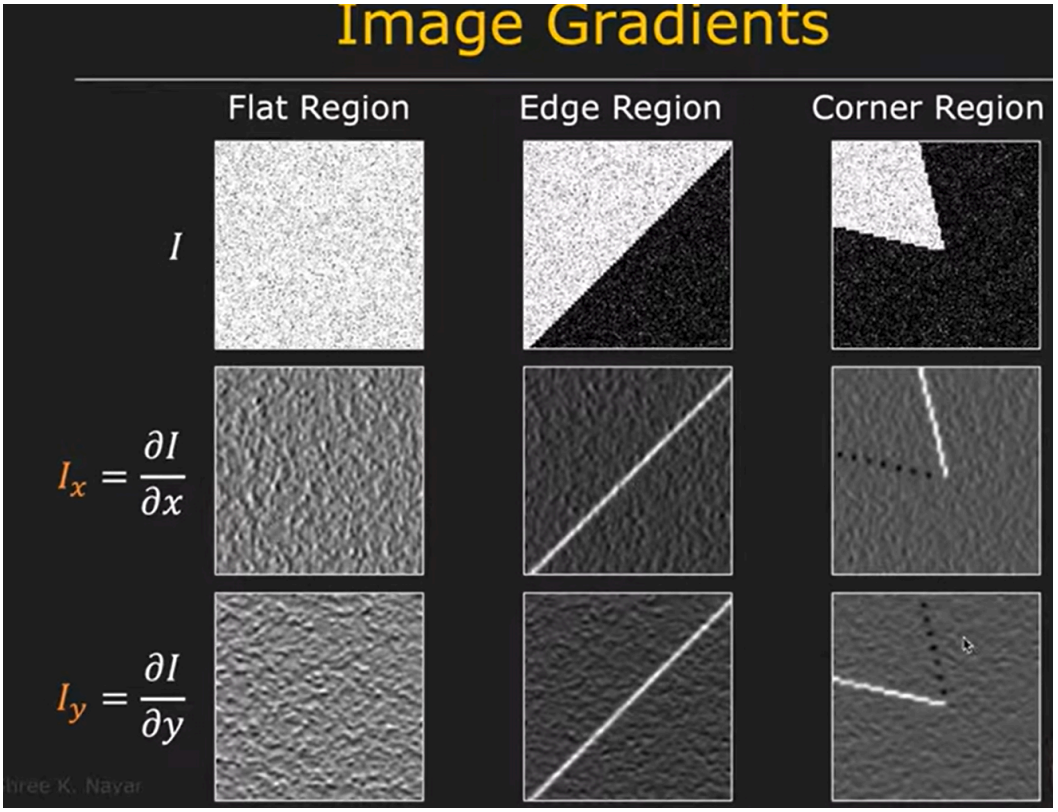
K_t is Kalman Gain The green box is the residual

Prof Amir's slide



Harris Corner Detector



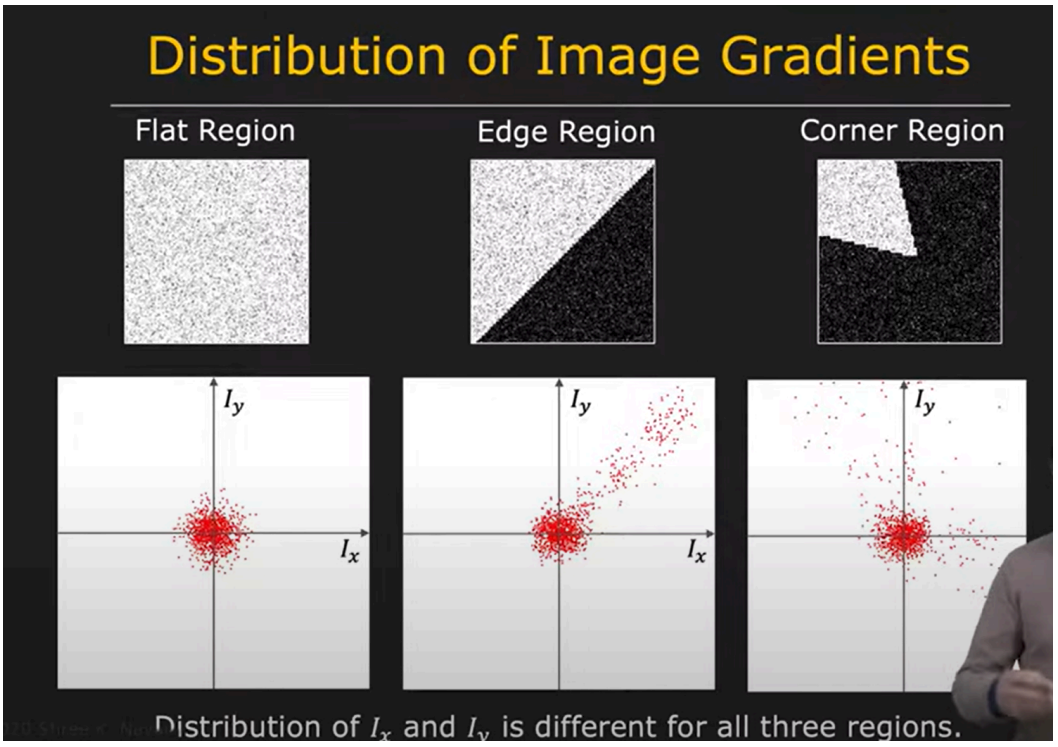


How come edge map can contain negative values?

If gradient of edge matches the gradient of the filter, then you will get some form of resonance and get a large positive number.

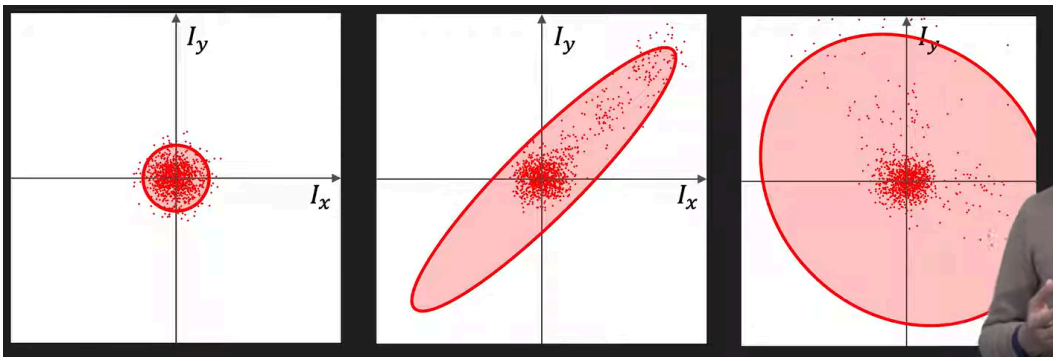
If the gradient of the edge is opposite to the gradient of the filter (for eg: the gradient of the edge is bright to dark and your filter is dark to bright), you will see on the edge map (in that direction, G_x, G_y or whatever), you will see large negative number at the position of that edge.

Images are normalized such that white means strong positive value, black means strong negative value and gray means near 0.

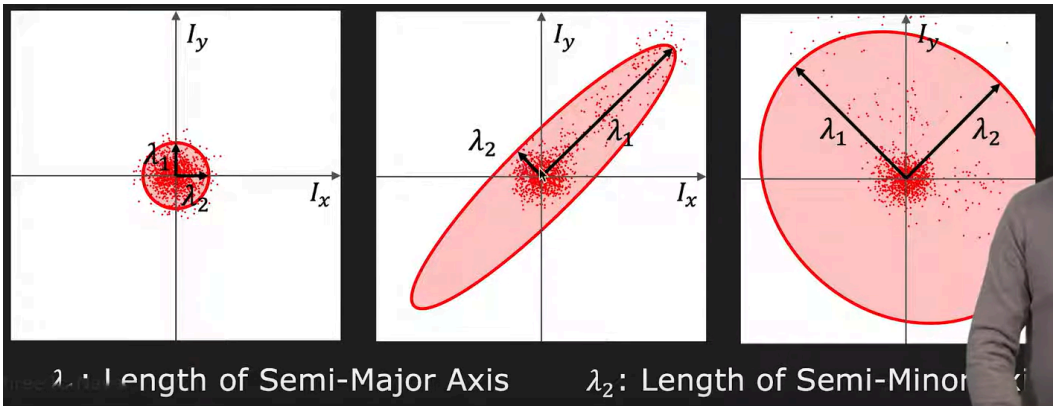


So the goal is to somehow quantify this structure and describe the structure of the distribution with a small number of parameters to classify the region.

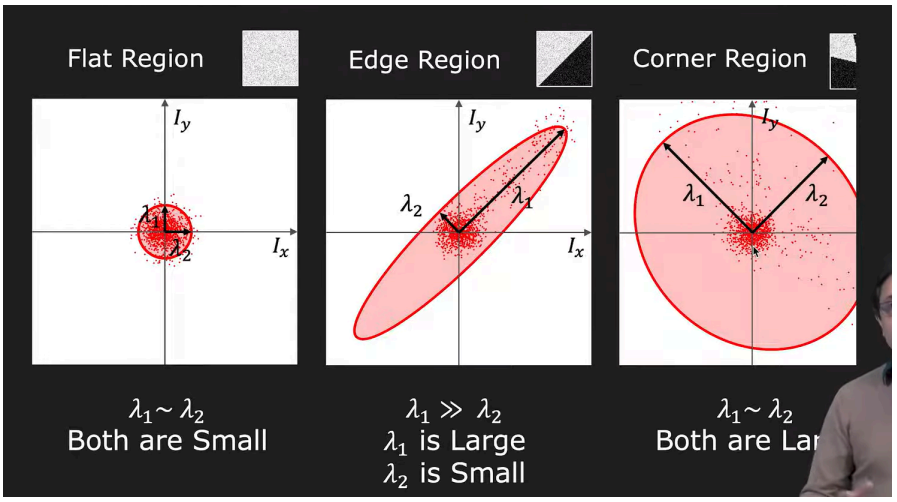
Do this by taking the distribution and fitting an ellipse around it.



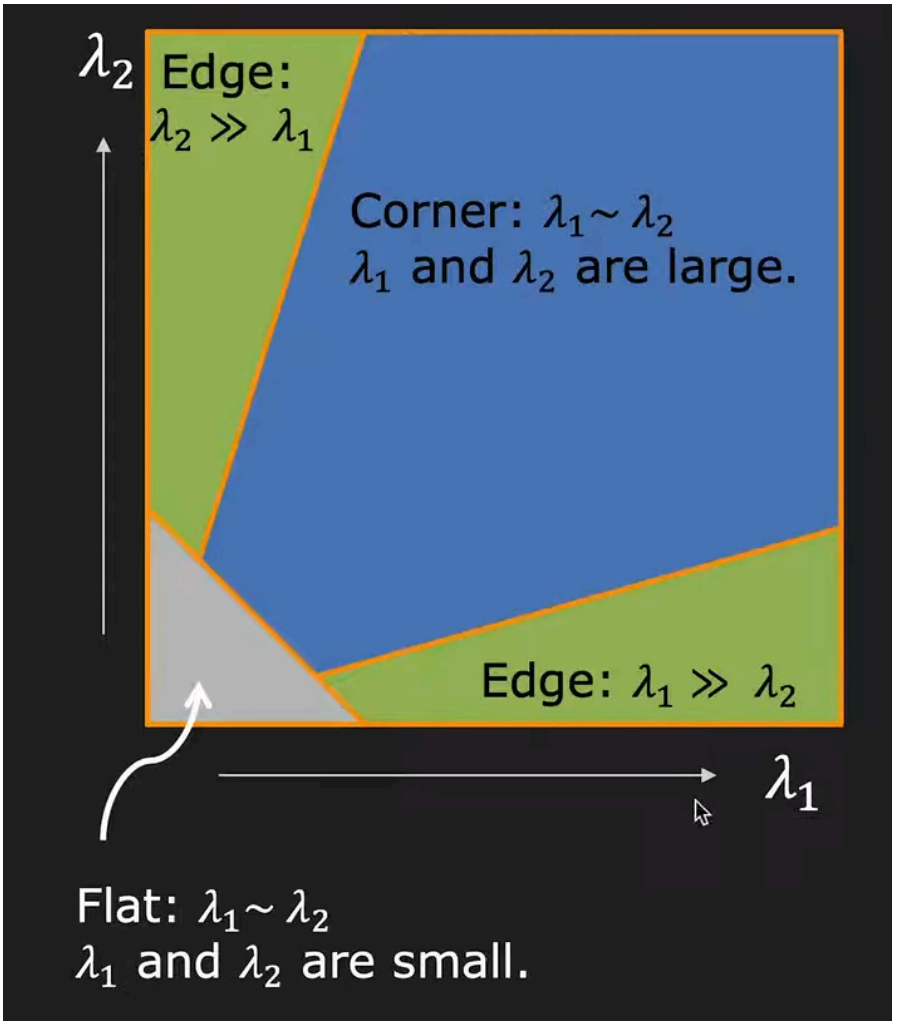
In doing so, you get a semi major (the longer) and a semi minor (the shorter) axis for the ellipse.



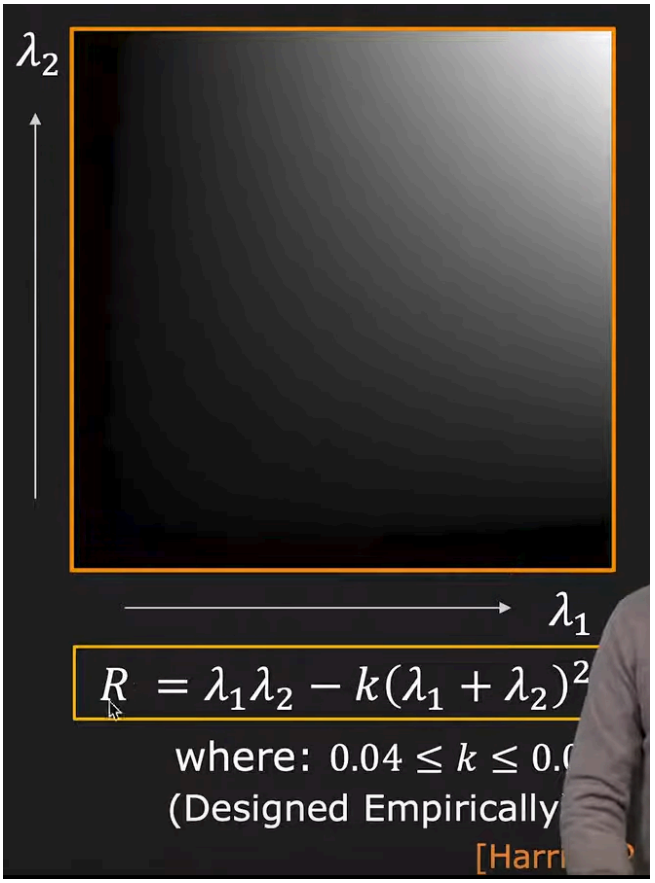
lambda1 is the length of the semi-major axis and lambda2 is the length of the semi-minor axis.



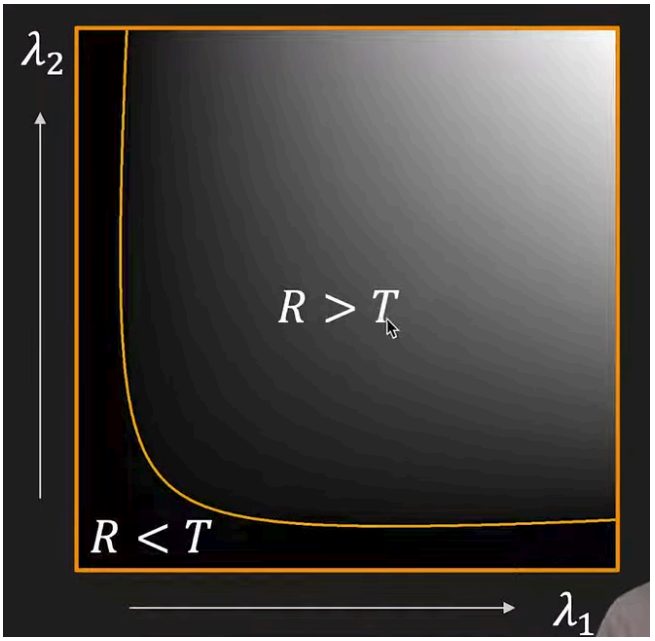
So in lambda1, lambda2 space, we classify accordingly:



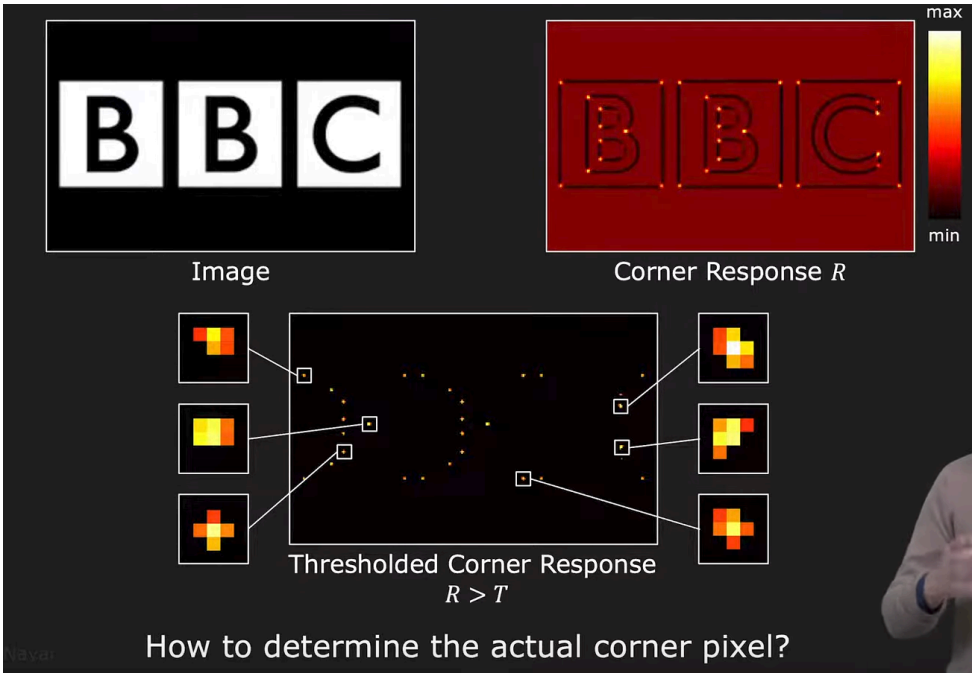
We want to do this simply, Harris came up with this expression where he maps the eigen values to a single real number. This is an empirically determined function ($0.04 \leq k \leq 0.06$ is also empirical). This is the Harris Corner response function



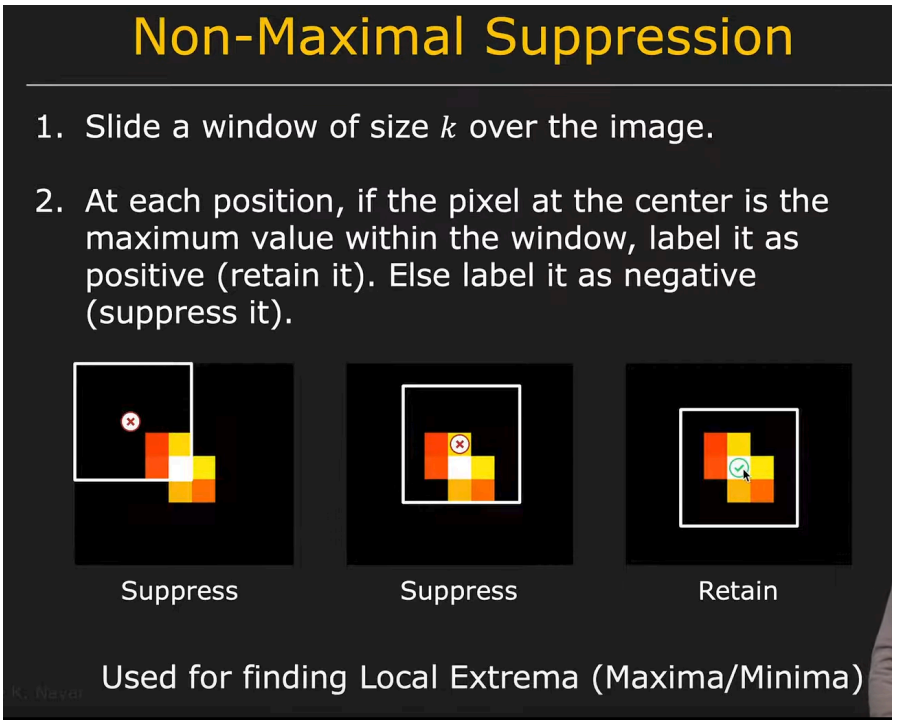
R is being plotted in this image where the brightness of the point is proportional to R. So if you threshold this space wrt R:

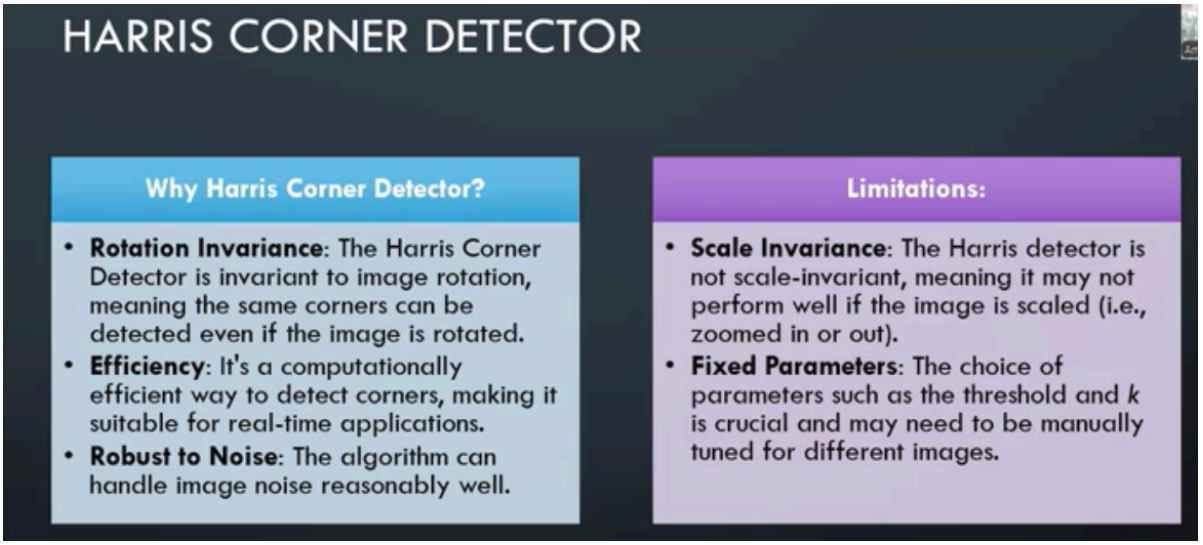


Eg:



The thing is, after you threshold, you get clusters of pixels near the corner signaled as the corner. But we want to single out the pixel that is the corner (this would be the peak), so we apply NMS





The biggest limitation is that it’s not scale invariant.

Specifically:

- **Changing Gradient Magnitudes:** Scaling an image affects the magnitude of the intensity gradients. If an image is scaled down, the pixel differences across a certain distance decrease, leading to smaller gradient values. Since the Harris detector's response is based on the squares and products of these gradients, the corner response will be lower for a scaled-down image, potentially causing genuine corners to fall below the detection threshold.
- **Fixed Window Size:** The local window used to compute the structure tensor has a fixed size in terms of pixels. When the image scale changes, this fixed window corresponds to a different physical area in the original scene. A corner feature that perfectly fits within the window at one scale might occupy a much larger or smaller area at a different scale. If the corner feature becomes larger than the window, only a part of it might be analyzed, potentially leading to it being misclassified as an edge or a flat region. If the feature becomes too small, its influence on the gradients within the window might be diminished.

Lucas Kanade Point Tracking

Gemini 2.5 Flash

Okay, building on your knowledge of the Lucas-Kanade optical flow algorithm, understanding Lucas-Kanade *point tracking* is straightforward.

Essentially, **Lucas-Kanade point tracking is an application of the Lucas-Kanade optical flow algorithm specifically tailored to find the motion of a *small number of predefined or detected feature points* from one image frame to the next.**

Here's how to think about it:

1. **Lucas-Kanade Optical Flow (General):** This algorithm estimates the 2D motion vector (optical flow, (u,v)) for pixels within a local window, based on the assumptions of brightness constancy and spatial coherence. It solves a system of linear equations (using least squares) derived from the optical flow constraint equation Ixu+Iyv+It=0 applied to all pixels in the window.
2. **Lucas-Kanade Point Tracking (Specific):**
    - **Goal:** Don't calculate the flow for *every* pixel or large regions. Instead, pick a few important points in the first frame (these are often "good features to track," like corners, detected using algorithms like Shi-Tomasi or Harris).
    - **Method:** For *each* point you want to track:
        - Define a small window (a patch of pixels) centered *at that specific point* in the first frame (It).
        - Apply the standard Lucas-Kanade optical flow calculation *just within that small window*, comparing the patch in frame It to the corresponding patch in the next frame (It+1).
        - The resulting optical flow vector (u,v) calculated for that specific window is the estimated movement of the feature point from frame t to frame t+1.
        - The new estimated position of the tracked point in frame t+1 is simply its old position plus the calculated flow vector: (xold+u,yold+v).
    - **Iterative Refinement:** The standard Lucas-Kanade flow assumes small motion. For point tracking, especially if the motion is larger, the algorithm is typically made iterative. It estimates the flow, warps the patch from the second image towards the first based on the estimate, recalculates the residual flow, and repeats until the estimate converges or a maximum number of iterations is reached. This helps handle larger displacements more accurately.

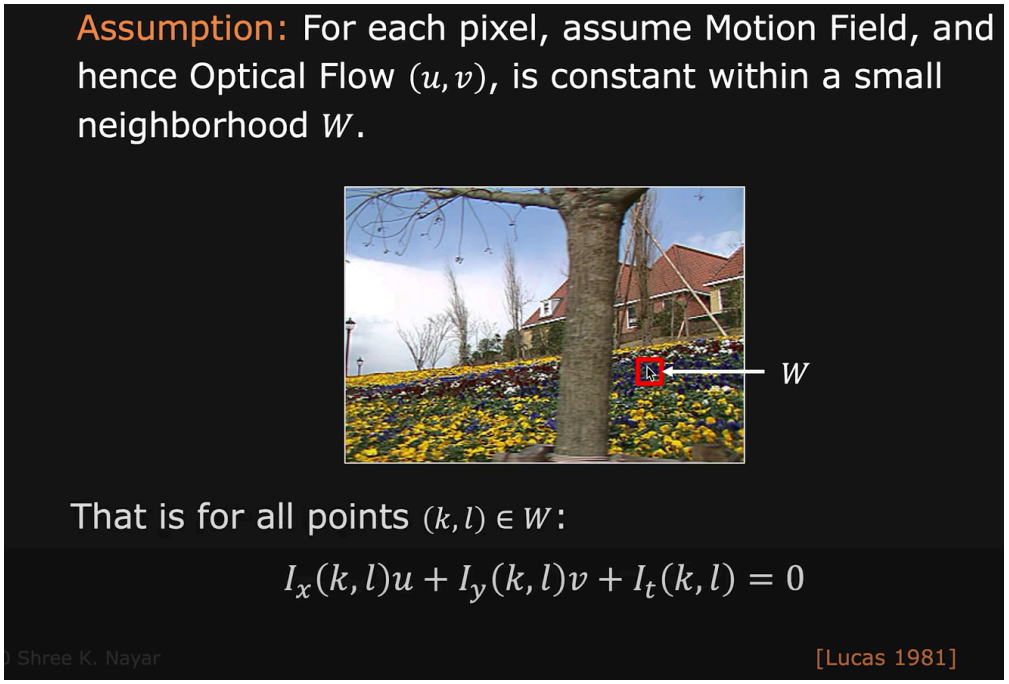
In simple terms:

If Lucas-Kanade Optical Flow is about calculating a *motion field* (a vector for many pixels or regions), Lucas-Kanade Point Tracking is about applying the *same local calculation* only *at the locations of specific points of interest* to follow their individual trajectories.

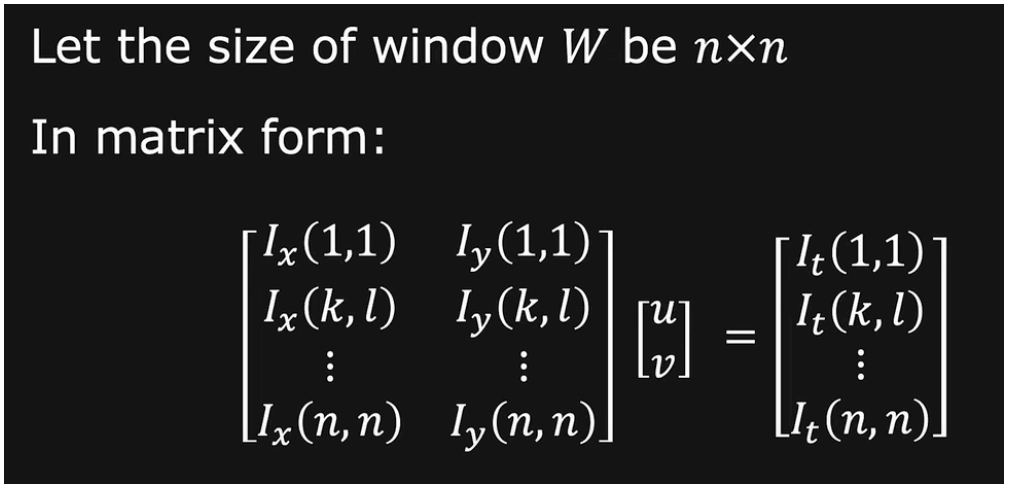
So, knowing the LK optical flow algorithm means you already understand the core computation behind LK point tracking. The "point tracking" aspect simply defines *where* in the image you apply that computation (only around the feature points) and what your goal is (following those specific points over time).

Lucas Kanade

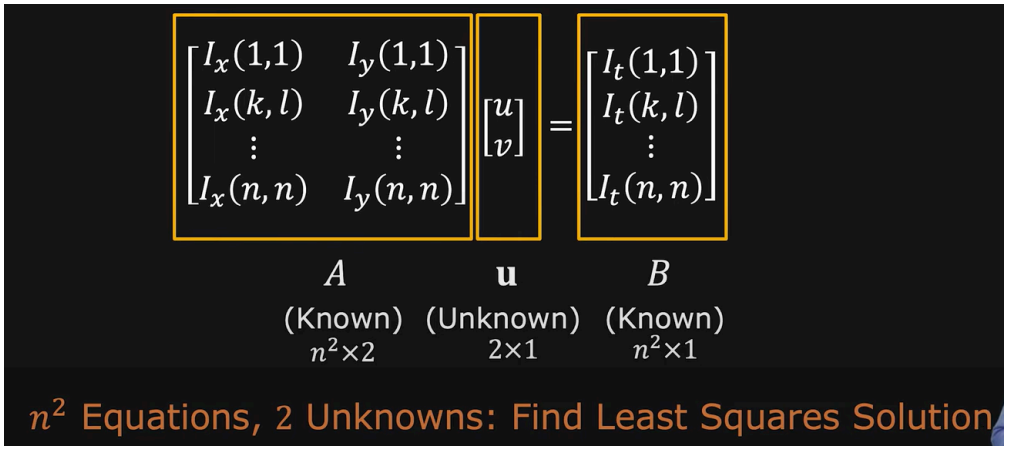
Adds additional constraint that for each pixel within a small local neighborhood W, assume that Motion Field and hence Optical Flow are constant

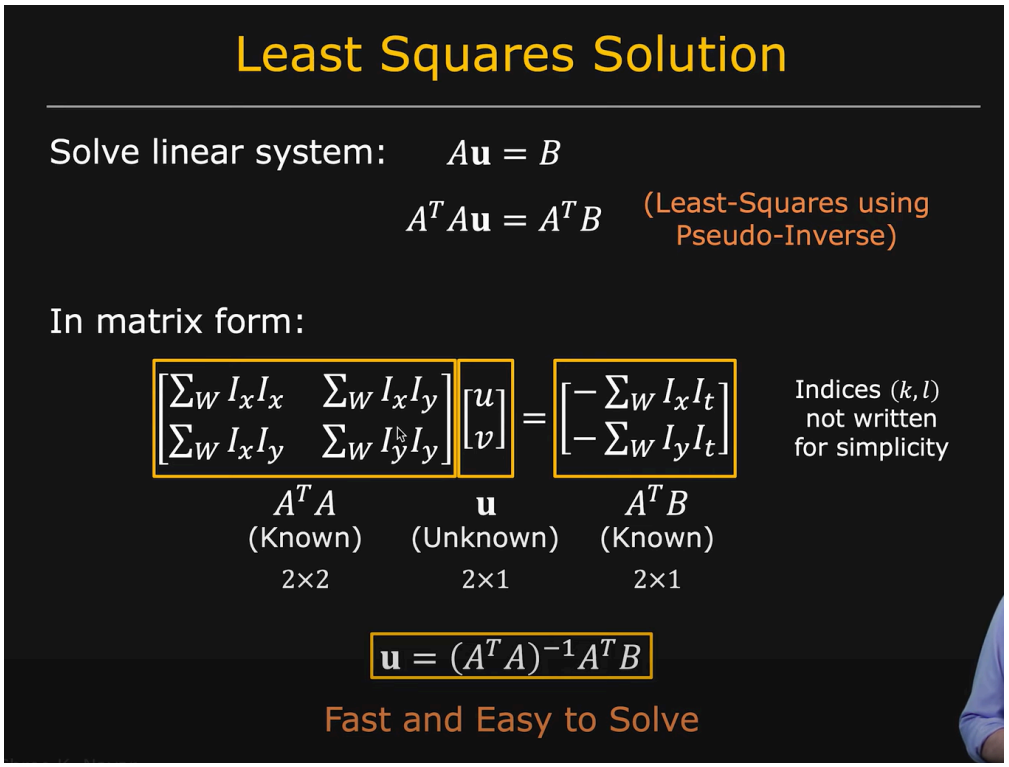


So, now you get a system of equations where the number of equations will be the size of the window.

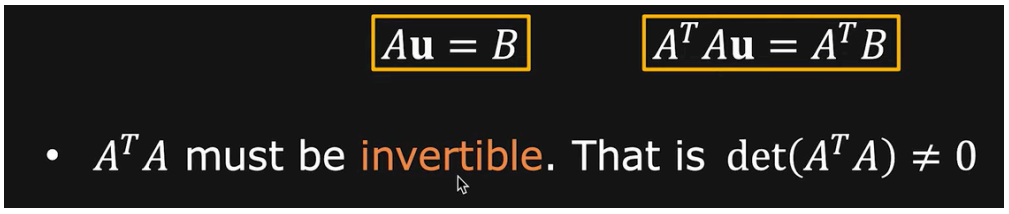


So, why does this work? Why would this matrix on the left be invertible? All of these equations tend not to be linearly dependent on one another. Because if you have a patch that has an interesting texture, then your Ix's and Iy's would most likely be different from pixel to pixel.





When would this not work again? What if AT @ A is not invertible?



More than wanting invertibility, we want A.T @ A to be “well-conditioned”

GPT: **Well-conditioned** means that the solution is *stable* with respect to noise or small changes in the input. A poorly conditioned (but still invertible) matrix can produce huge variations in the output for tiny variations in the input.

Also GPT:
Well-conditionedness is important because it ensures
stability in the solution. In the context of Lucas-Kanade, you are solving a system of linear equations derived from image gradients. If that system's coefficient matrix (often the structure tensor) is **ill-conditioned**, tiny amounts of noise or small changes in the gradients can lead to **large errors** in the computed optical flow. A well-conditioned matrix, on the other hand, keeps the solution more **robust** and **less sensitive** to inevitable noise or approximations in the image data.

- A^TA must be well-conditioned.

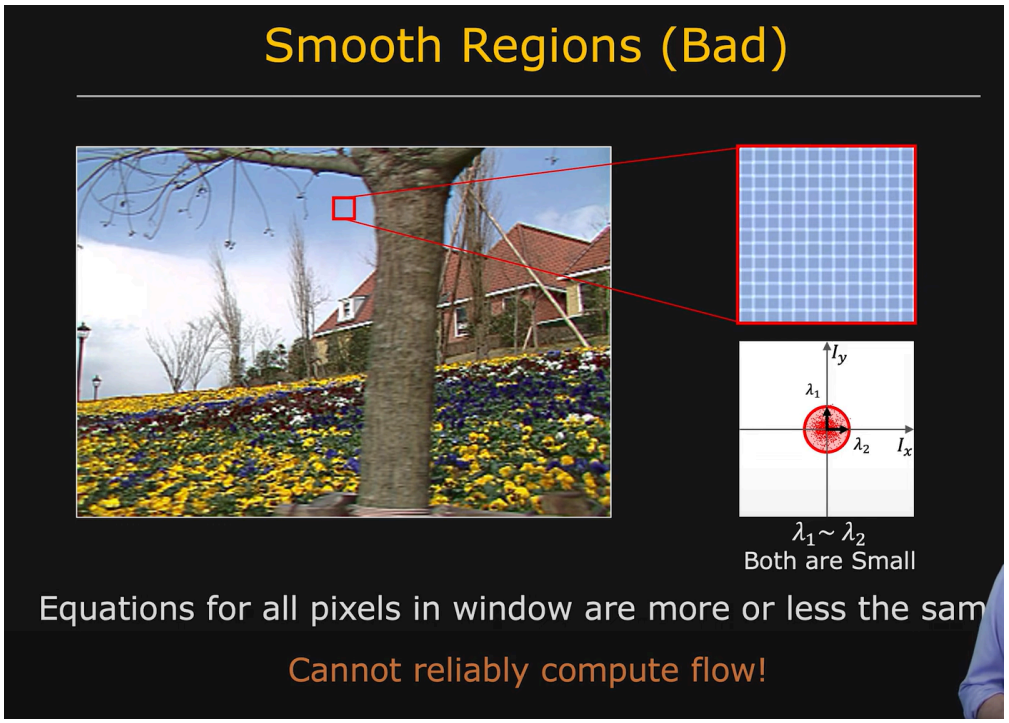
  If λ_1 and λ_2 are eigen values of A^TA, then

  $\lambda_1 > \epsilon$ and $\lambda_2 > \epsilon$

  $\lambda_1 \geq \lambda_2$ but not $\lambda_1 \ggg \lambda_2$

For A.T @ A to be well conditioned, the eigen values must be significant enough. But just that one eigen value cannot be SIGNIFICANTLY larger than the other one.

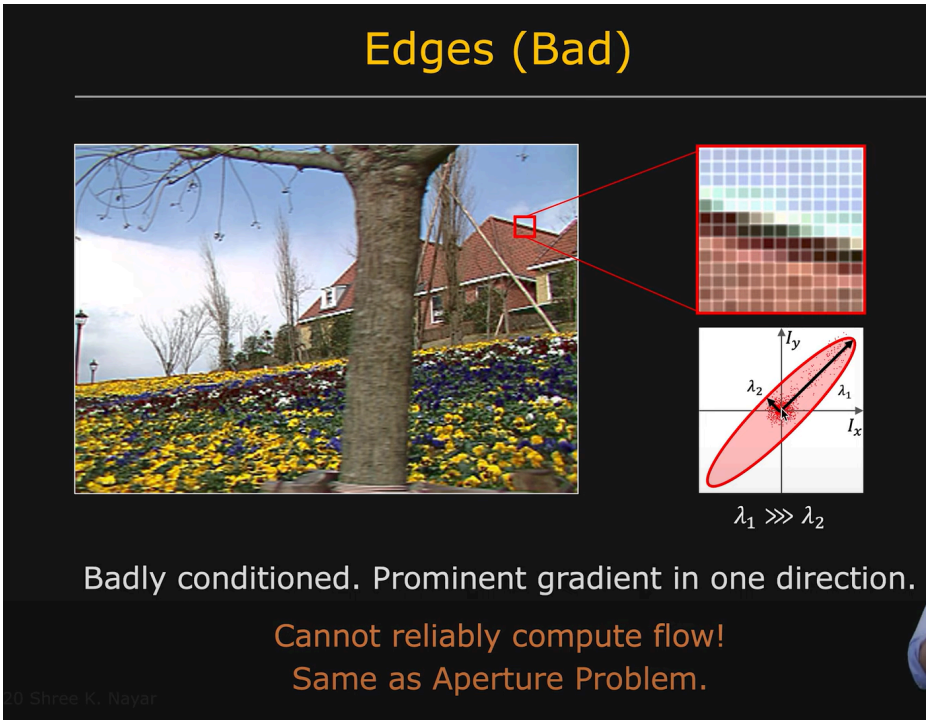
How non well-conditioned matrices manifest IRL



Note that A.T @ T is the covariance matrix and it's eigen vectors will show the direction where the data varies the most. If you have a smooth region then the Ix and Iy values are very small (these are the spatial gradients) then they will form a cloud round the origin. To find the eigen vectors of A.T @ T, fit an ellipse to these points. Then the semi major and major axis are the eigenvectors (of the covariance matrix A.T @ A). The covariance matrix also has very small eigen values.

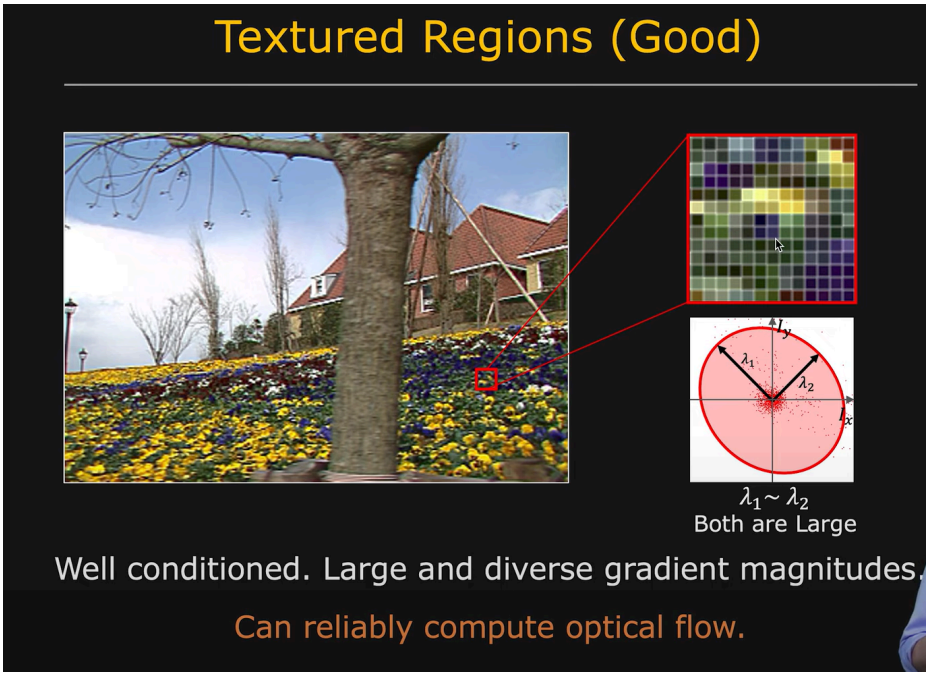
So optical flow cannot be computed reliably.

Another example of a bad region:



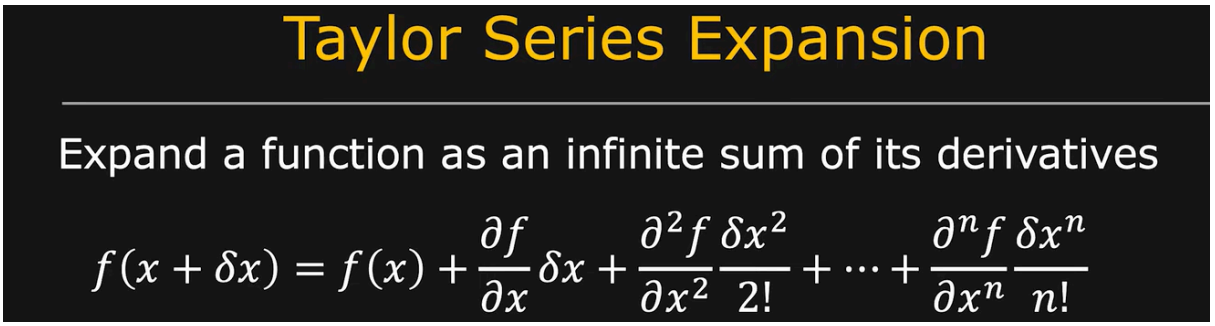
Edges are also bad because you have strong gradients (of pairs (Ix, Iy)) in one direction, so one of the eigen values is much larger than the other one.

How do well conditioned matrices manifest IRL?

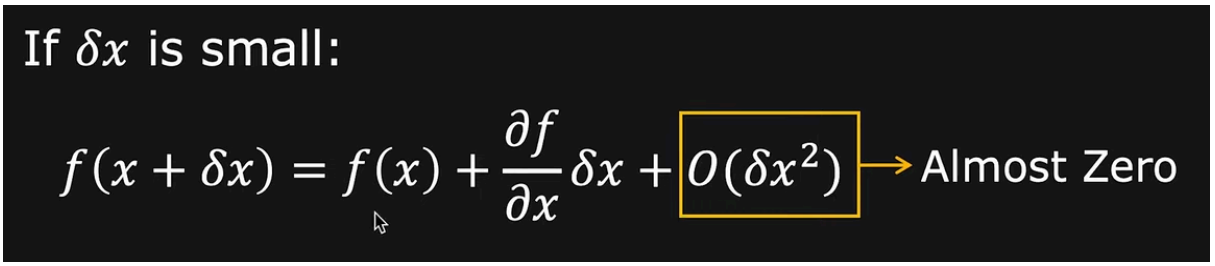


Good changes in brightness in both directions, so pairs of spatial gradient pairs (Ix, Iy) are well scattered. So that the eigen values are large but not large w.r.t each other. Therefore the matrix A.T @ T is well conditioned so that the optical flow computed is reliable.

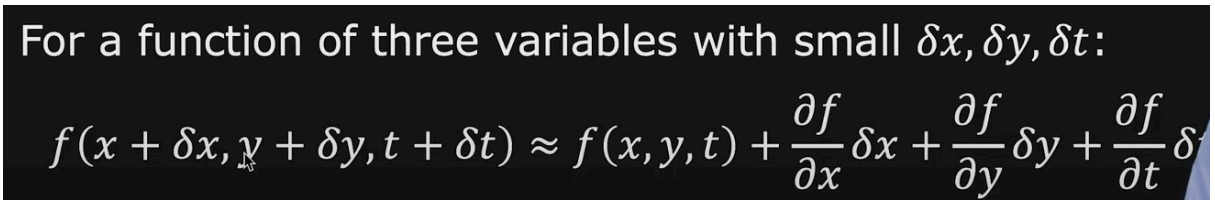
Math Primer



So, now you can see that if delta_x is really small, then the higher order terms are almost zero (i.e if dx is small to begin with, the delta_x^2 is even smaller)

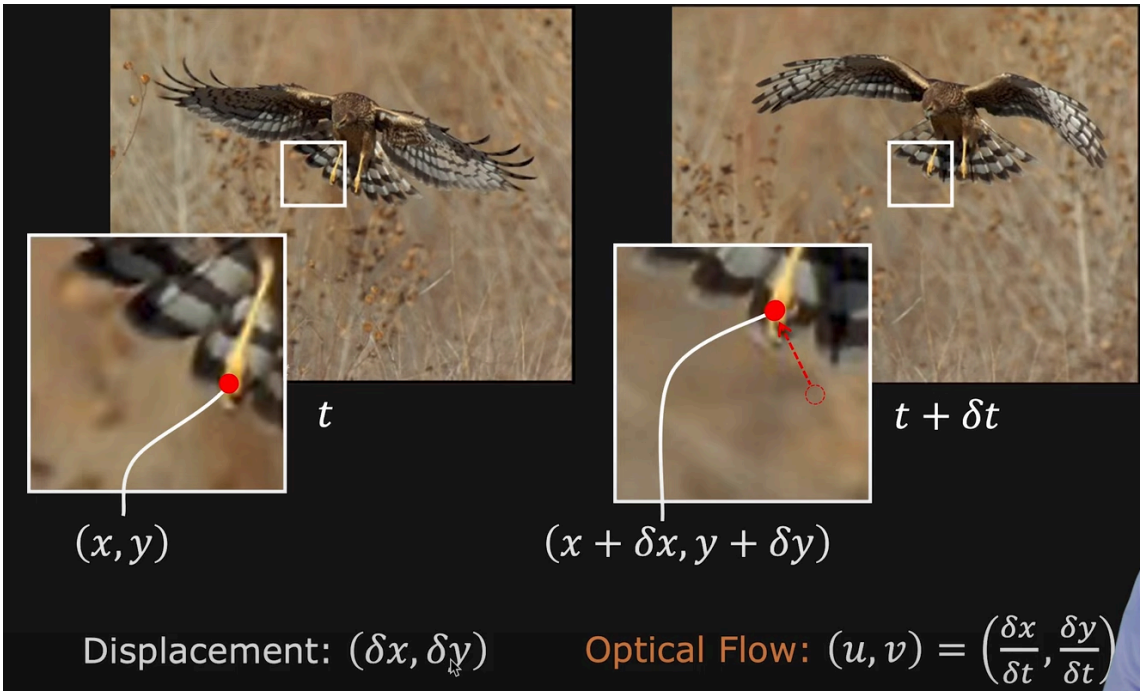


So you model f(x + delta_x) by f(x) + df/dx * delta_x. This is a linear approximation because the approximation is linear in dx. AKA First order Taylor approximation.



Optical Flow Constraint Equation

Consider a small window (that's the same in both images). Focus on a point in this window. Say we are focused on the foot. Then at time t + delta t, this point has moved to a new location which is x + dx, y + dy.



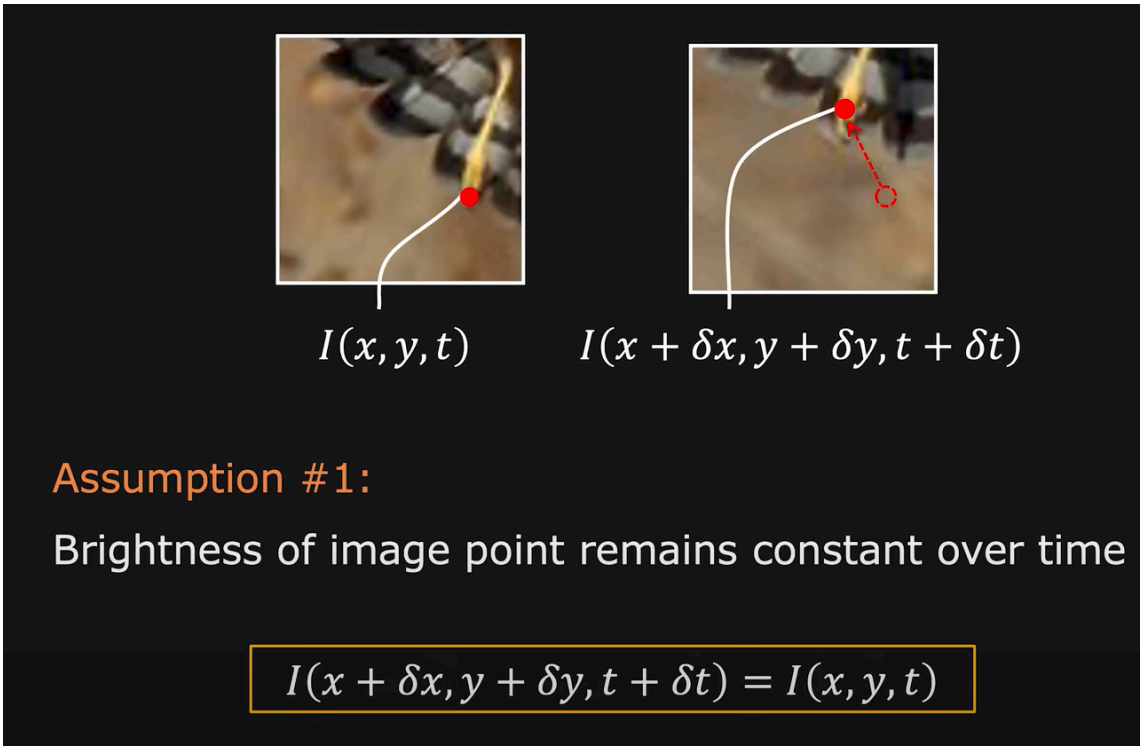
So, the displacement is (dx, dy)

So speed in x direction is dx/dt and speed in y direction is dy/dt. And this is the optical flow.

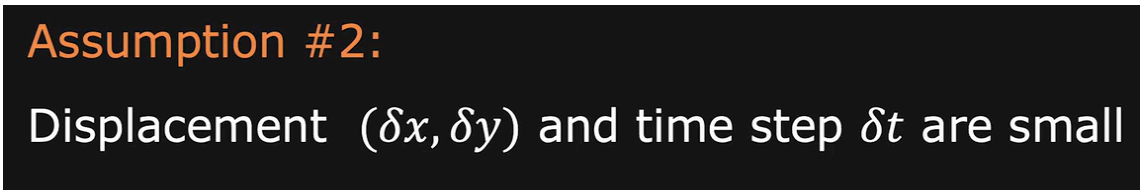
This is what we want to measure, (u, v)

Assumptions

Assumption #1: That brightness of image points remain the same, at least between consecutive images taken in quick succession. This is a reasonable assumption since dt is small.



Assumption #3: That the displacements of (dx, dy), and dt are small.



This has to be true for us to derive a constraint equation.

What this allows us to do is to come up with an approximation for I(x + dx, y + dy, t + dt). This approximation is based on the Taylor series expansion

Math Primer

$$I(x + \delta x, y + \delta y, t + \delta t) = I(x, y, t) + \frac{\partial I}{\partial x}\delta x + \frac{\partial I}{\partial y}\delta y + \frac{\partial I}{\partial t}\delta t$$

Alternatively,

$$I(x + \delta x, y + \delta y, t + \delta t) = I(x, y, t) + I_x \delta x + I_y \delta y + I_t \delta t$$

Results of assumptions

The two assumptions give rise to:

Optical Flow Constraint Equation

$$I(x+\delta x, y+\delta y, t+\delta t) = I(x,y,t) \quad \text{----------} \ (1)$$

$$I(x+\delta x, y+\delta y, t+\delta t) = I(x,y,t) + I_x\delta x + I_y\delta y + I_t\delta t \quad \text{----------} \ (2)$$

Then simply subtract:

Subtract (1) from (2): $I_x\delta x + I_y\delta y + I_t\delta t = 0$

Then divide by delta_t and take lim to 0

Divide by δt and take limit as $\delta t \to 0$: $I_x\frac{\partial x}{\partial t} + I_y\frac{\partial y}{\partial t} + I_t = 0$

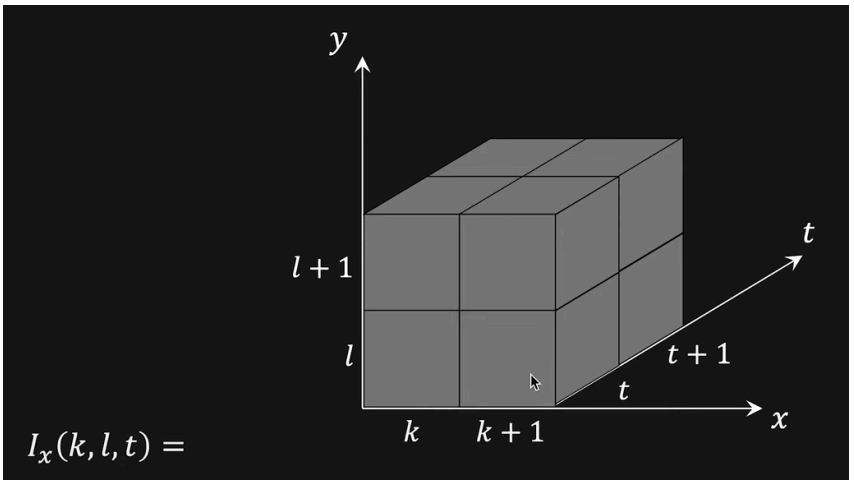
Then deltas turn into derivatives.

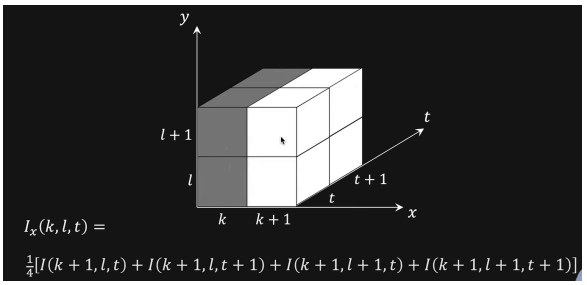
Then delta_x/delta_t becomes dx/dt = u, delta_y/delta_t becomes dy/dt.

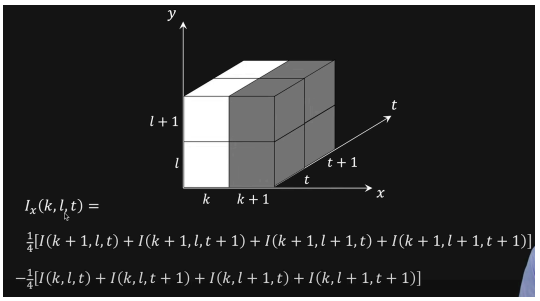
Constraint Equation: $\boxed{I_x u + I_y v + I_t = 0}$ (u,v): Optical Flow

Then! Given two equations taken in quick succession (at t and t + delta t), we can find these derivatives Ix, Iy and It. Done using finite differences using edge detection.

Computing Ix, Iy and It



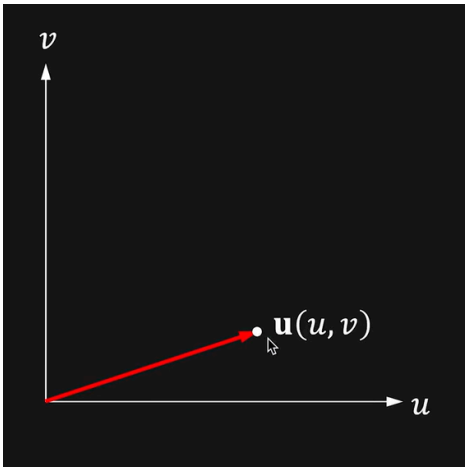




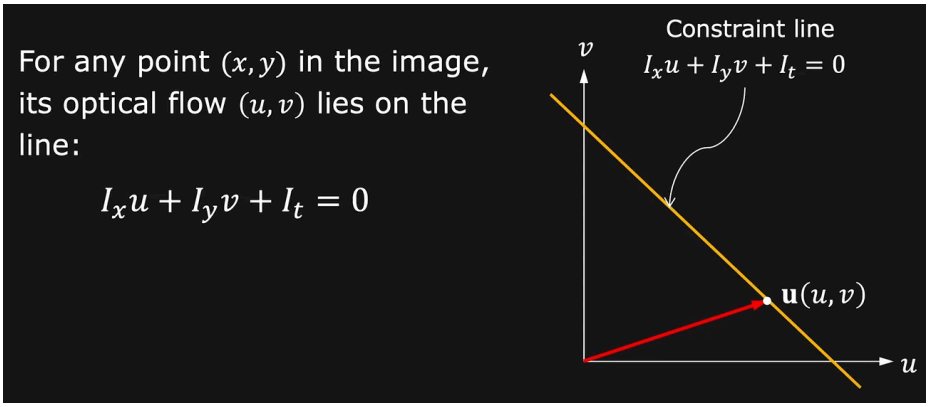
This will give you the derivative of the BRIGHTNESS in the x direction. Similarly you can do so for the y direction and t direction.

Geometrical Interpretation of Optical Flow (to show that u and v are really indeterminable, by showing under-constrainedness)

Let the true flow for a particular point in the image be (u, v)

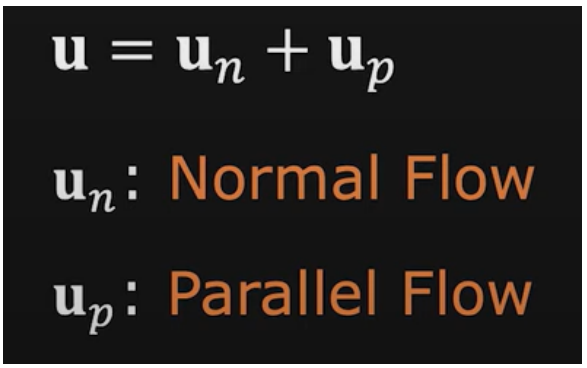


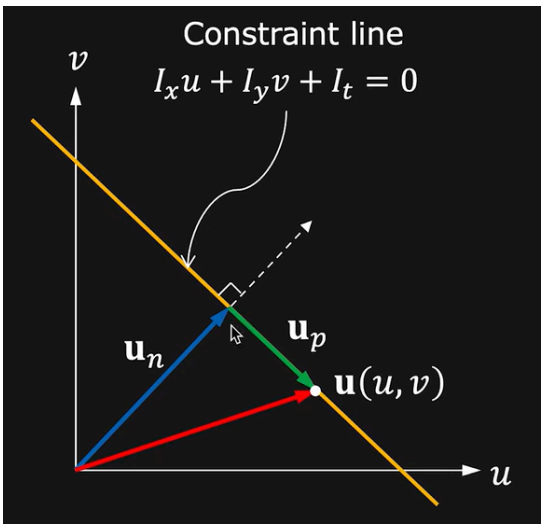
(Unfortunately) the only thing that we know about (u, v) is that it must lie on the constraint line.



We know that (u, v) lies on the line. But don't exactly know where. This is what makes optical flow an under-constrained problem.

First, note that the vector can be split into two components.





U_n is the normal flow and U_p is the parallel flow

U_n is obviously easy to calculate.

Direction of Normal Flow:

Unit vector perpendicular to the constraint line:

$$\hat{\mathbf{u}}_n = \frac{(I_x, I_y)}{\sqrt{I_x^2 + I_y^2}}$$

Magnitude of Normal Flow:

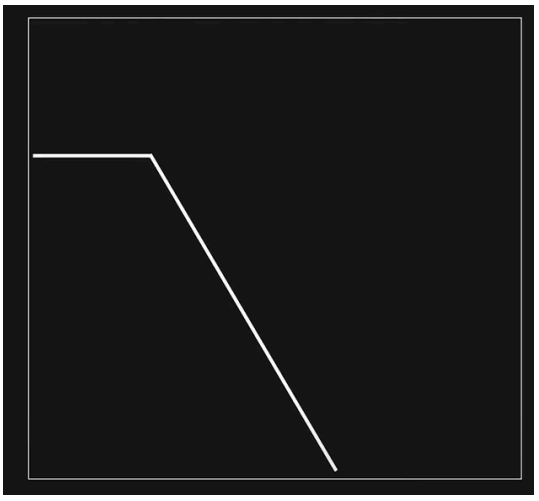
Distance of origin from the constraint line:

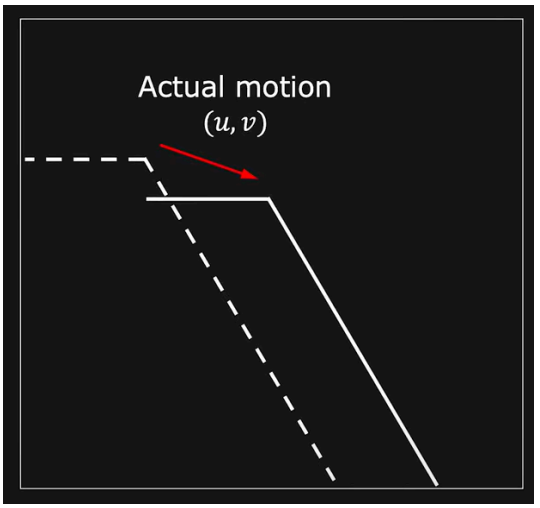
$$|\mathbf{u}_n| = \frac{|I_t|}{\sqrt{I_x^2 + I_y^2}}$$

$$\mathbf{u}_n = \frac{|I_t|}{(I_x^2 + I_y^2)}(I_x, I_y)$$

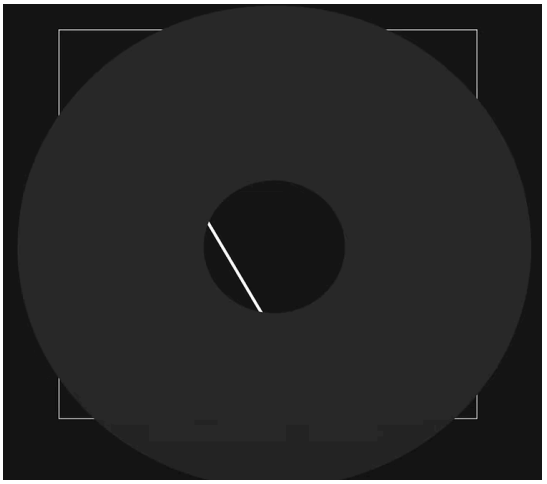
But we cannot compute U_p!

Example of how the problem of inability to calculate U_p manifest in a real scenario?

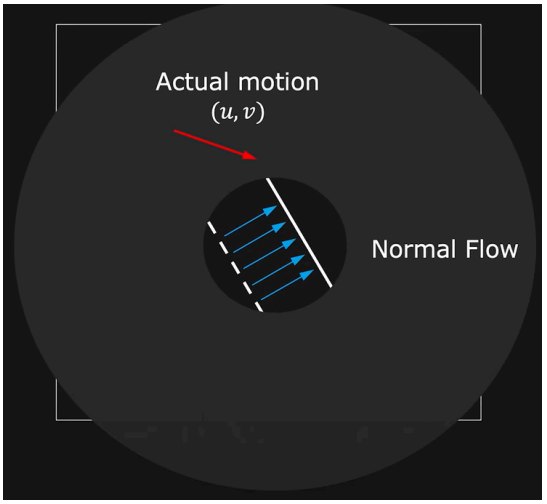




Our image is not one object, we have potentially different flows for every local region. So it necessitates to look at small local patches, called our aperture.



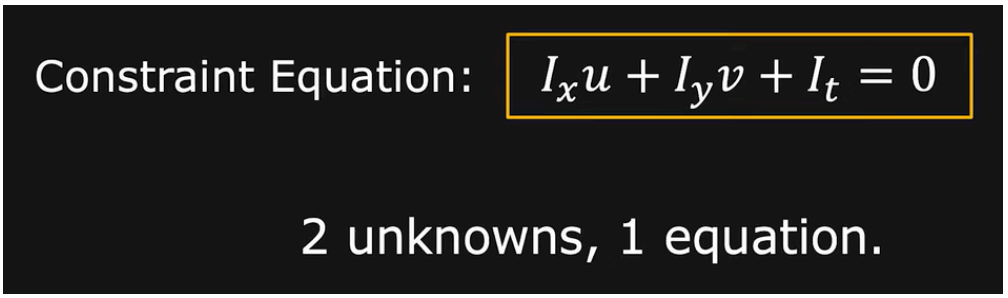
But if you look at the same motion through this aperture, you end up seeing ONLY the normal flow:



Unable to measure the parallel flow, and thus unable to measure the actual flow.

So locally, we are only able to determine the normal flow.

Optical Flow is under constrained!

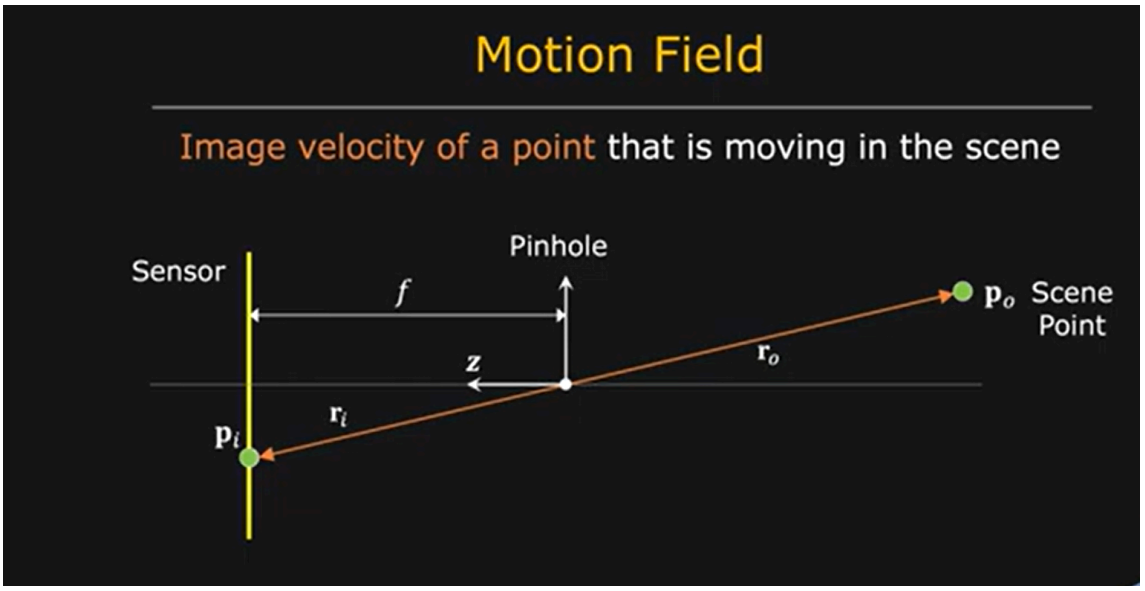


2 unknowns are u and v.

To solve this, we introduce additional constraints.

Motion Fields and Optical Flow

Motion Fields



p0 is a point in the 3d scene and it's location is given by the vector r0. p0 projects onto the point pi via perspective projection and it's location on the image plane is described by the vector ri.

If p0 has some velocity in the 3d world, say v0, then it's displacement is given by v0 * dt. So it's new location is r0 + delta r0.

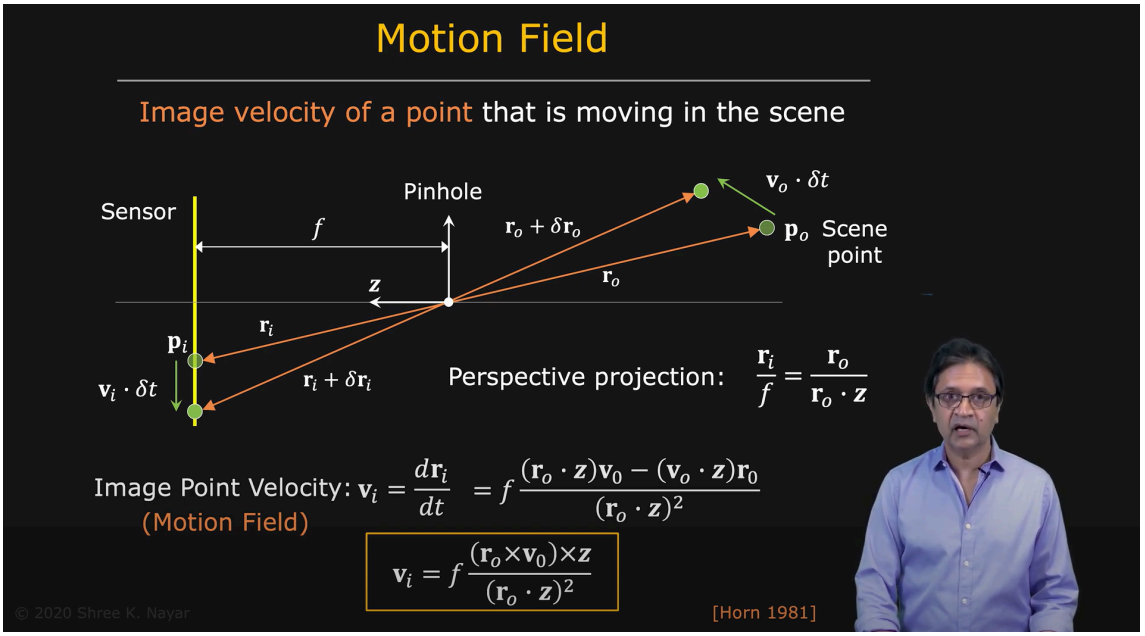
Also by definition, scene velocity v0 is the rate of change of r0, so

$$\text{Scene Point Velocity: } \mathbf{v}_o = \frac{d\mathbf{r}_o}{dt}$$

What we are interested in is the Image point velocity:

$$\text{Image Point Velocity: } \mathbf{v}_i = \frac{d\mathbf{r}_i}{dt}$$
$$\text{(Motion Field)}$$

We know v0 but we don't know vi. So try to relate these to each other using perspective projection. Then simply take derivatives (using quotient rule) and then use cross product and get the answer.



This vi is the motion field corresponding to the moving point in the 3D scene.

This is what we hope to measure, but there is no guarantee that we can measure this.

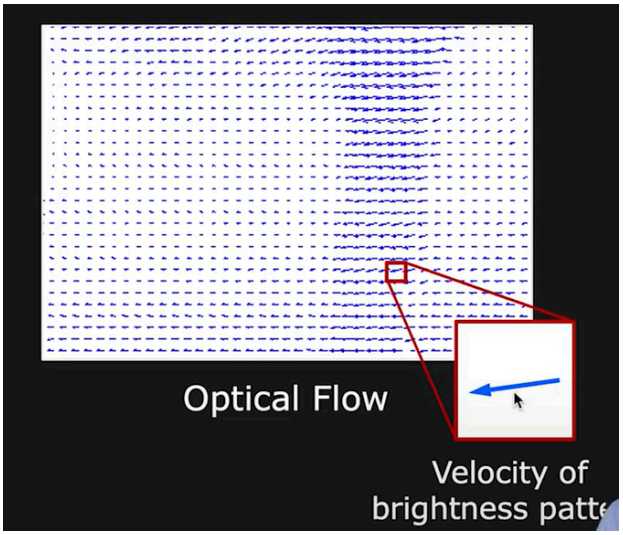
Optical Flow

Instead, what we hope to measure is the motion of brightness patterns in the image.

(Here, if you take each point in the first frame and try to figure out where the point landed in the second image, that would be the motion field)

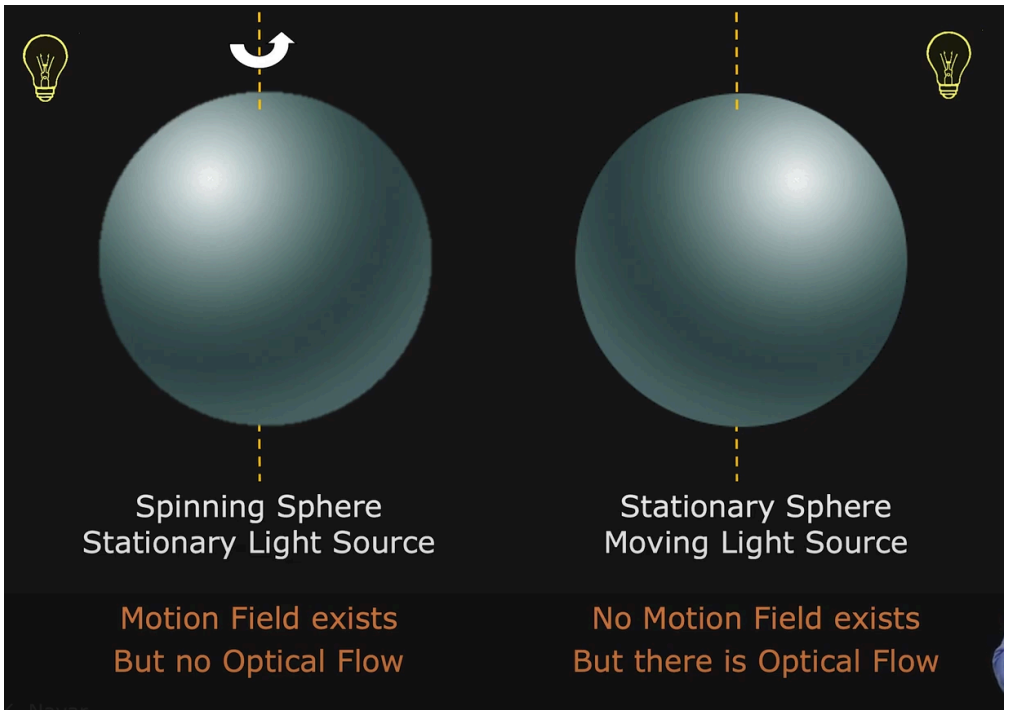
But what we can do is take a brightness pattern in one image and figure out where the brightness pattern is in the second image.

And if you have a successful algorithm that can do that, we can get a result that looks like this:

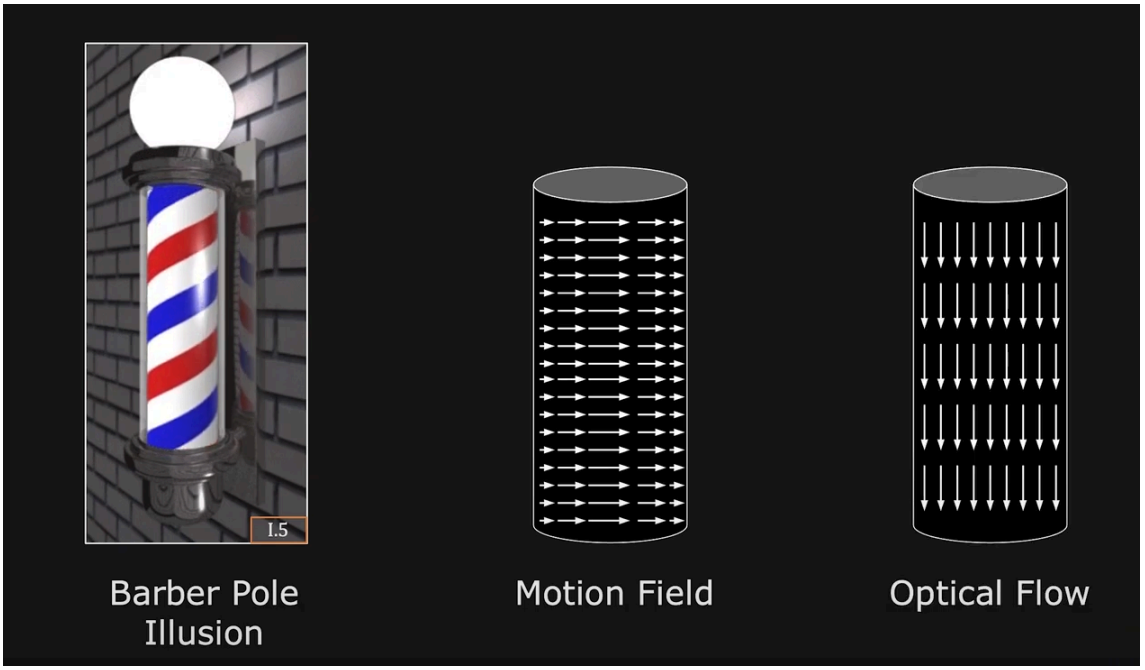


At each pixel, you have a vector that tells you what the optical flow at the point is. So, the motion of brightness patterns is the optical flow. And the optical flow vector's length tells you how fast its moving and direction tells you direction it's moving in the IMAGE PLANE.

Ideally Optical Flow = Motion Field. Unfortunately, not most of the time.



Another case where motion field and optical flow exists, but not the same


I.5
Barber Pole Illusion
Motion Field
Optical Flow

Extra Hadamard Lecture

A = hah, a is the image, h is the sequency ordered hadamard matrix.

So if you just do H @ a , you've only done the first pass (transforms along the *horizontal* direction but leaves each column untouched) and haven't yet projected onto the vertical-basis patterns—hence it won't match the full 2D transform. After you have post-multiplied by h, this transforms both *rows* and *columns*, giving the true 2D Hadamard spectrum you see in your 4×4 coefficient matrix.

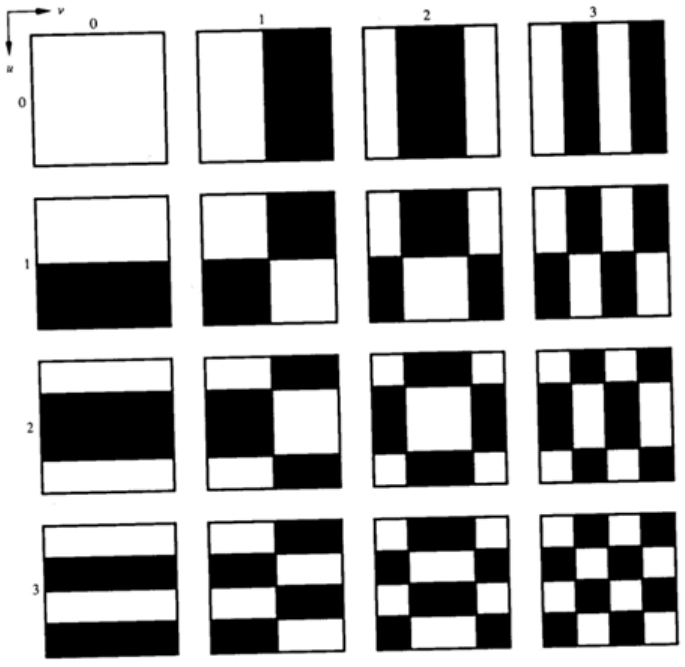
natural hadamard can do compression and coding

sequency ordered can do those plus filtering in hadamard doman

need bit reverse etc to convert natural hadamard to sequency ordered hadamard

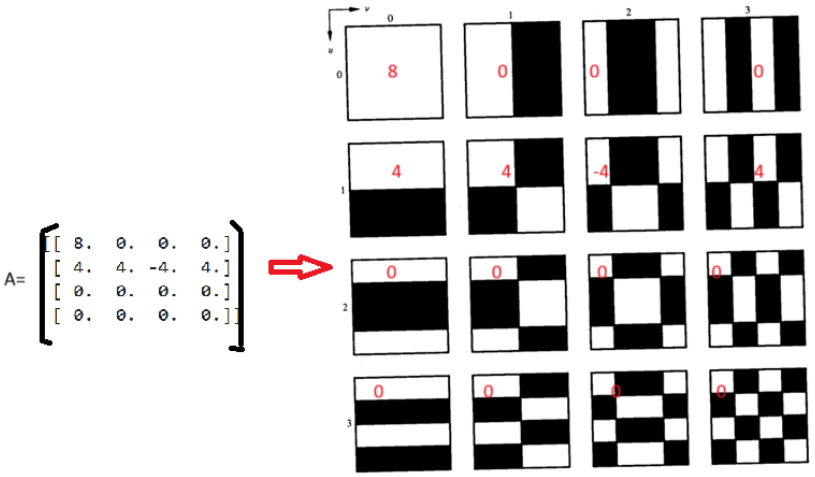
> Any 4×4 image can be represented as a weighted sum of the following 16 4×4 square waves in the SPATIAL DOMAIN. These are the Walsh basis patterns.

(white is 1, black is 0)



And what sequency hadamard transform actually does it calculating the magnitude of these weightages in order (low → high spatial frequency basis)

So A (4×4 matrix) = hah where a is the spatial image has these coefficients



Notice the u and v axis in the A matrix.

u represents the vertical frequency and v represents horizontal frequency.

Eg A[0,0] represents the wavelet with no horizontal and vertical frequency

A[0,1] represents the wavelet with low horizontal and no vertical frequency

A[0,2] represents the wavelet with higher horizontal and no vertical frequency

A[1,0] → no horizontal frequency but low vertical frequency

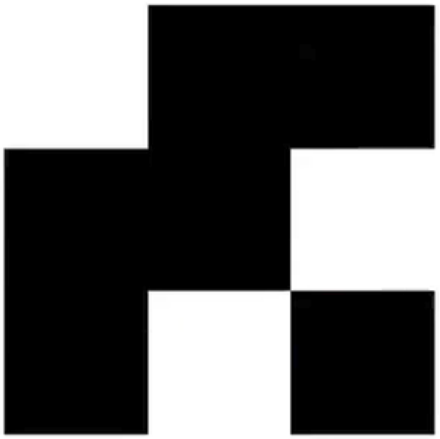
A[2,0] → no horizontal but higher vertical frequency

A[3,0] → no horizontal but more higher (lol) vertical frequency

Other A[i, j] has a combination of horizontal and vertical frequency

A[3,3] → highest horizontal and highest vertical frequency

Now, why is A[0][0] = sum of all the pixels? Look at hah and the sequency ordered hadamard matrix's first row and first column.



the (0,0) entry, A[0][0], is calculated using the first row of h on both sides. That is:

A[0][0] = (first row of h) · a · (first column of h)

So then since u and v represents frequency then you can filter out A in whatever way you like to preserve either high 2d frequencies by keeping coefficients of high frequency square waves intact or by removing them and preserving only the low frequency square waves by keeping their coefficients intact

How to convert Natural hadamard matrices to sequency ordered hadamard matrices?

(now the slide deck begins)

(by the way, take a look at the recursive formulation of the hadamard matrices)

Here, you only need to rearrange the rows, and the columns will be rearranged automatically.

1) Bit-by-bit GRAY 2 Bin algorithm

Let your Gray code be an nnn-bit string gn−1gn−2…g1g0g_{n-1}g_{n-2}\dots g_1g_0gn−1gn−2…g1g0. You build the binary bits bn−1…b0b_{n-1}\dots b_0bn−1…b0 as follows:

1. **MSB copies directly:**bn−1=gn−1.

   bn−1 = gn−1. b_{n-1} \;=\; g_{n-1}.

2. **Each lower bit is the XOR of the Gray bit with the previously computed binary bit:**bi=bi+1⊕gifor i=n−2,n−3,…,0.

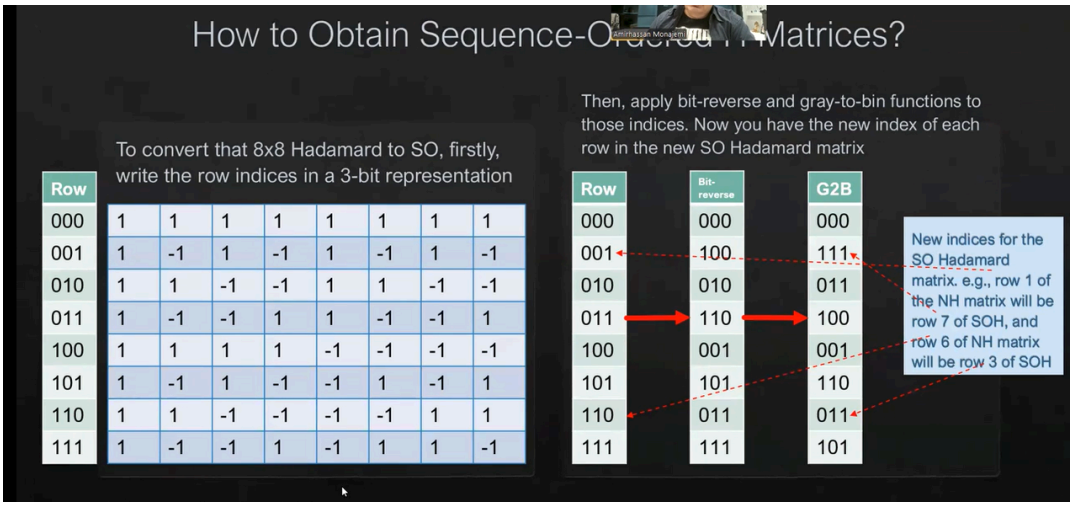
   bi = bi+1 ⊕ gifor i=n−2,n−3,…,0. b_i \;=\; b_{i+1}\;\oplus\;g_i
   \quad
   \text{for }i = n-2,n-3,\dots,0.

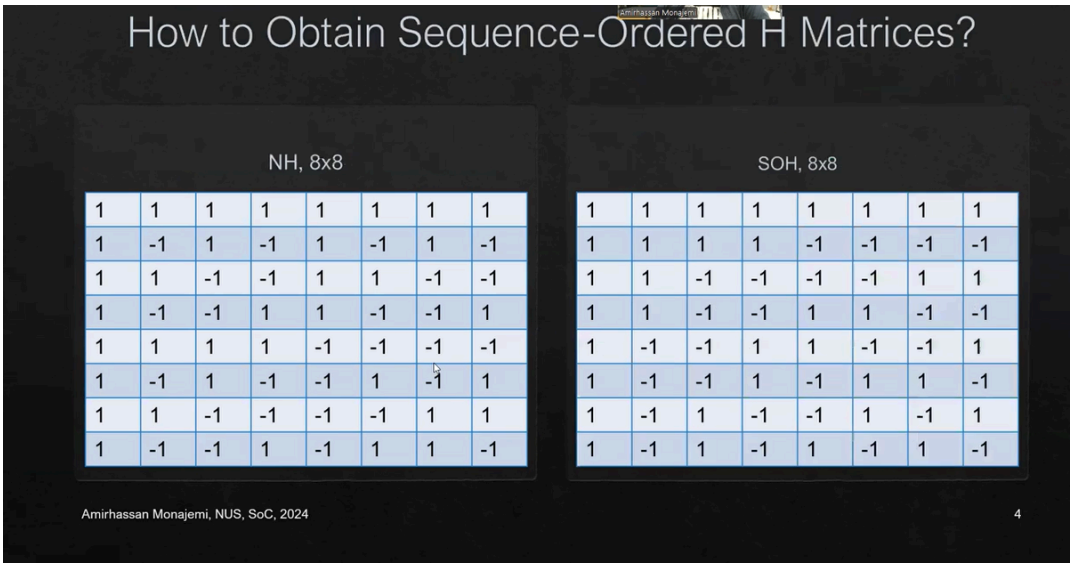
Example

Gray = `1101` (4-bit)

1. b3=g3=1b_3 = g_3 = 1b3=g3=1
2. b2=b3⊕g2=1⊕1=0b_2 = b_3\oplus g_2 = 1\oplus1 = 0b2=b3⊕g2=1⊕1=0
3. b1=b2⊕g1=0⊕0=0b_1 = b_2\oplus g_1 = 0\oplus0 = 0b1=b2⊕g1=0⊕0=0
4. b0=b1⊕g0=0⊕1=1b_0 = b_1\oplus g_0 = 0\oplus1 = 1b0=b1⊕g0=0⊕1=1

So `1101` (Gray) → `1001` (binary).



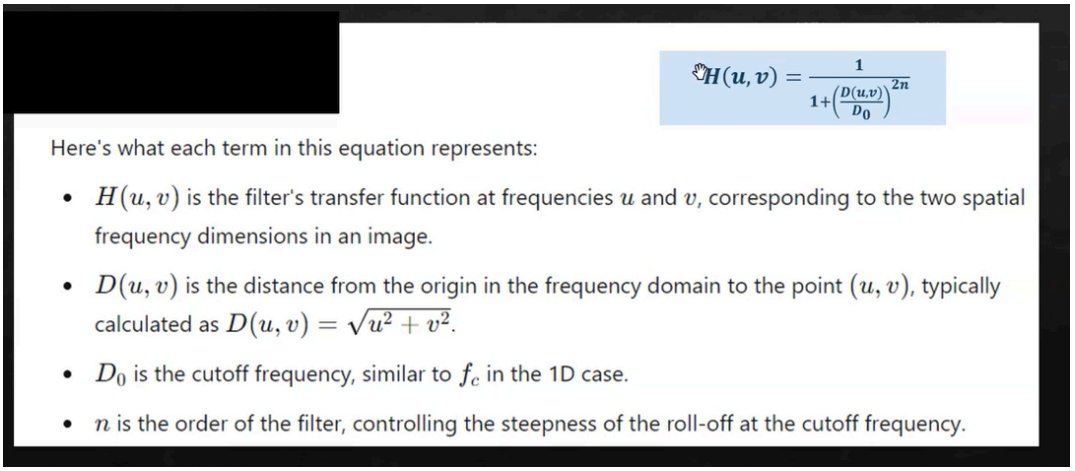


Butterworth low pass

For one 1:

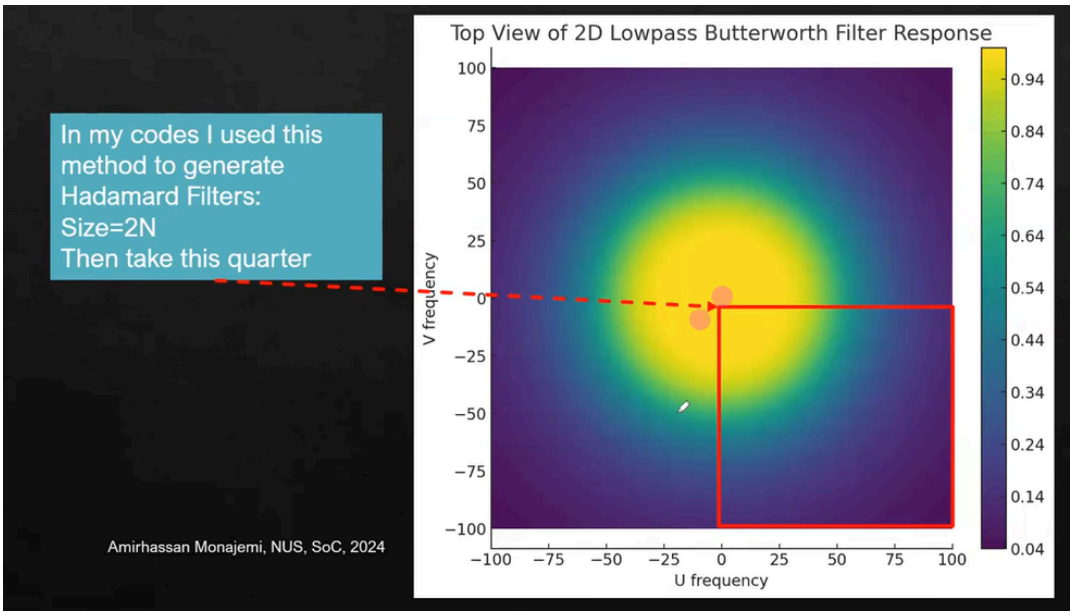
$$H(f) = \frac{1}{1+\left(\frac{f}{fc}\right)^{2n}}$$

This is a low pass filter



Look at how if D(u, v) < Do then high value.

But as soon as D(u, v) > Do, then quickly denominator explodes. Explosion is controlled by roll-off number n (order). Higher the order, higher the fall.



Butterworth highpass

But note that for ideal low pass butterworth, there is no gradual decay of magnitude, but a SHARP cutoff.

Why divide by 256^2?

For the *unnormalized* Hadamard matrix H of size N:

1. $HH = N\,I$.
2. Forward 2D transform:

$$A = H\,a\,H.$$

3. Inverse then is

$$a = \frac{1}{N^2}\,H\,A\,H,$$

since

$$H\,A\,H = H\,(H\,a\,H)\,H = (H\,H)\,a\,(H\,H) = N\,I\,a\,N\,I = N^2\,a.$$

So in code you divide by N^2. For a 256×256 image, $N = 256$, hence `…/256**2`.

Inverse via the matrix inverse

1. **General inverse formula**
   If your forward was

$$A = H\,a\,H$$

then the inverse is

$$a = H^{-1}\,A\,H^{-1}.$$

2. **Hadamard's special inverse**
   For the (unnormalized) Hadamard matrix $H \in \{+1,-1\}^{N\times N}$, one can show

$$H\,H = N\,I \quad\Longrightarrow\quad H^{-1} = \frac{1}{N}\,H.$$

   Moreover H is symmetric so $H^T = H$.
3. **Putting it together**

$$a = H^{-1}\,A\,H^{-1} = \left(\tfrac{1}{N}H\right)A\left(\tfrac{1}{N}H\right) = \frac{1}{N^2}\,H\,A\,H.$$

   Hence your code's division by `256**2` is exactly the two factors of $1/N$ coming from each inverse.

What is a normalized hadamard matrix

A **normalized** Hadamard matrix is just the usual $\{\pm 1\}$ Hadamard matrix H rescaled so that its rows (and columns) become orthonormal rather than orthogonal. Concretely, if H is an $N \times N$ Hadamard (so that $H\,H^T = N\,I$), then you define

$$H_n \;=\; \frac{1}{\sqrt{N}}\,H.$$

Key properties of H_n

- **Orthonormality:**

$$H_n\,H_n^T \;=\; \frac{1}{N}\,H\,H^T \;=\; I.$$

- **Symmetric and involutory:**

$$H_n^T = H_n, \quad H_n^{-1} = H_n.$$

Why normalize?

1. **Simplifies inversion.**
   With the un-normalized H you need

$$A = H\,a\,H \quad\Longrightarrow\quad a = \frac{1}{N^2}\,H\,A\,H.$$

   With H_n you get

$$A = H_n\,a\,H_n \quad\Longrightarrow\quad a = H_n\,A\,H_n$$

   with no extra $1/N^2$ factor.
2. **Unit-energy basis.**
   Each row of H_n has Euclidean norm 1, so your transform becomes an orthonormal change of basis—very handy for energy-preserving signal analysis.
3. **Interpretability.**
   The DC coefficient $A_{0,0}$ now gives the *average* pixel value (rather than the raw sum), because projecting onto an all-ones unit vector divides by $\sqrt{N}$.

What are possible applications of representing an image using the first few walsh basis images?

- **Image Compression**
  - **Low-complexity codecs**

    The Walsh–Hadamard transform (WHT) uses only additions/subtractions, so it's extremely fast in hardware. Truncating to the first KKK low-sequency coefficients yields a coarse but compact representation—ideal for very low-power or real-time systems (e.g. embedded vision).
  - **Progressive coding**

    Send only the DC term and a handful of first-sequency basis images to give a quick "preview" of the frame; refine gradually by adding higher-sequency terms as bandwidth allows.
- **Denoising and Smoothing**
  - Noise in images tends to manifest in high-sequency (rapid sign-changes) coefficients. By zeroing out all but the first few Walsh coefficients, you perform a form of low-pass filtering that suppresses speckle or salt-and-pepper noise while keeping the large-scale structure intact.
- **Watermarking & Steganography**
  - Embedding a watermark in mid- or low-sequency Walsh coefficients makes it robust to JPEG-style compression (which also retains low frequencies) but less perceptible to the human eye. You can add or modify a small offset in a few chosen basis coefficients to carry hidden data.
- **Feature Extraction for Recognition/Classification**

    - Projecting image patches onto the first few Walsh functions gives you a low-dimensional feature vector that captures bulk texture/brightness patterns. These “Walsh features” have been used successfully for face recognition, fingerprint matching, and texture classification because they’re rotation- and scale-invariant to a degree, and very fast to compute.
- **Template Matching & Fast Correlation**
    - Because convolution in the Walsh domain reduces to element-wise products, you can rapidly correlate a template against an image by transforming both to sequency order, multiplying only the first few coefficients (for a coarse match), and inversely transforming. This accelerates object detection or motion tracking in video.
- **Compressed Sensing & Sparse Sampling**
    - The Walsh basis is incoherent with many natural-image bases. If you know an image is sparse in the Walsh domain, you can randomly sample a small set of pixel measurements and reconstruct it by solving a sparse-recovery problem—keeping only the largest few Walsh coefficients.
- **Optical & Holographic Processing**
    - In optical computing, lenses can implement the WHT natively. Truncating to low-sequency patterns lets you perform rapid, analog smoothing, correlation, or pattern recognition directly in hardware without digital post-processing.
- **Progressive Rendering in Graphics**
    - Rather than rasterizing every pixel at full resolution, a renderer can project frames onto low-sequency Walsh patterns first to quickly display a blurred version, then refine it. This improves interactivity in remote or VR streaming scenarios.