CMSC426 Notes

Jeremy Jubilee: Contact at jerrj33@gmail.com for revisions

Spring 2021

Contents

1 Math Primer

1.1 Linear Algebra: Common Terms and Definitions

Why is Linear Algebra Important to Computer Vision?

- We associate coordinates to points in an image (2D) or scene (3D)
- Use these coordinates to perform geometrical transformations, different coordinate systems, etc.
- We represent images as **matrices of numbers**

Matrices are used represent images and kernels. Many image processing operations involve matrix math.

- Consider a 2-D Matrix A with elements in A denoted as a_{ij}.

$$\underbrace{\begin{bmatrix} a_{11} & a_{21} & \cdots & a_{n1} \\ a_{12} & a_{22} & \cdots & a_{n2} \\ \vdots & \vdots & \ddots & \vdots \\ a_{1m} & a_{2m} & \cdots & a_{nm} \end{bmatrix}}_{A \text{ is an } n \times m \text{ matrix}}$$

- **Summation**: To have $A + B = C$ be a valid summation, they must have the **same dimensionality**.

$$\underbrace{\begin{bmatrix} a_{11}+b_{11} & a_{21}+b_{21} & \cdots & a_{n1}+b_{n1} \\ a_{12}+b_{12} & a_{22}+b_{22} & \cdots & a_{n2}+b_{n2} \\ \vdots & \vdots & \ddots & \vdots \\ a_{1m}+b_{1m} & a_{2m}+b_{2m} & \cdots & a_{nm}+b_{nm} \end{bmatrix}}_{A+B} = \underbrace{\begin{bmatrix} c_{11} & c_{21} & \cdots & c_{n1} \\ c_{12} & c_{22} & \cdots & c_{n2} \\ \vdots & \vdots & \ddots & \vdots \\ c_{1m} & c_{2m} & \cdots & c_{nm} \end{bmatrix}}_{C}$$

- For multiplication, we have $C_{n\times p} = A_{n\times m} + B_{m\times p}$. A must have same number of columns as B has rows. $c_{ij} = \sum_{k=1}^{m} a_{ik} b_{kj}$. In other words, in the corresponding row of A and column of B, get the sum of the products of each corresponding cell.
- Transposing is flipping a matrix. $C_{m\times n} = A^T_{n\times m}$, and $c_{ij} = a_{ji}$. $(A + B)^T = A^T + B^T$. $(AB)^T = (B^T A^T)$.

Vectors are often used to represent features of data or images. Consider two vectors of same dimensionality, v, w

- **Summation**: Good for normalizing vectors by mean centering around 0.

$$\begin{bmatrix} v_1 \\ \vdots \\ v_n \end{bmatrix} + \begin{bmatrix} w_1 \\ \vdots \\ w_n \end{bmatrix} = \begin{bmatrix} v_1 + w_1 \\ \vdots \\ v_n + w_n \end{bmatrix} = \begin{bmatrix} u_1 \\ \vdots \\ u_n \end{bmatrix}$$

- Can multiply every field in v with a scalar, a. Good for normalizing vectors by making variance 1.
- Magnitude of vector is just distance formula of all fields... get square of all fields, sum, then get root. Denoted as $||v||$
- Dot product is like multiplying matrices, imagine each "row" and "column" as vectors, then get the sum of the multiples of each one. If dot product is 0, vectors are perpendicular. Helps us find angle between vectors. $v \cdot w = ||v||||w||cos(\theta)$
- Cross product gives us another vector. The resultant vector is perpendicular to both v,w.

Norms: Different ways to calculate distances between vectors. Used for classification. Useful for measuring how accurate a prediction is by distance to the target.

- **L2 Norm:** $||x||_2 = \sqrt{\partial_{i=1}^{n} x_i^2}$. Most common, euclidean distance.
- **L1 Norm:** $||x||_1 = \sqrt{\partial_{i=1}^{n} |x_i|}$. Sum of absolute value of errors. "Manhattan distance".
- **Infinity Norm:** $||x||_{inf} = max(|x_i|)$. Get the worst error, and report that.
- **Generalized P-Norm** $||x||_p = (\partial_{i=1}^{n} x_i^p)^{1/p}$.

1.2 Least Squares Optimization

Imagine a 2-D scatter plot, where the x-axis is the "input" and y-axis is the "output". We can imagine this as how long someone spends out of the house with a certain amount of gasoline in their tank. Let's say we want to figure out how the amount of gas and the time spent are related.

Linear Regression is about getting a line of best fit for a set of data, in this case, somewhat of a formula to get the expected time based on gas. We can formalize mathematically.

- Let x be our input, and y be our output. We want to find the parameters, θ, of a formula across multiple (x, y) data points such that

$$y_i = \theta_0 + \theta_1 x_i$$

- We may not be able to get the parameters completely right. The difference between the expected $(\hat{y})$ and actual y value is known as error, ϵ. We can find the total error of our estimated parameters as the sum of these differences over our n data pairs,

$$\epsilon = \sum_{i=1}^{n} \epsilon_i = \sum_{i=1}^{n} (\hat{y_i} - y_i)^2$$

- We may not necessarily have just 2 inputs to our formula. We can generalize the formula to multiple variables:

$$\hat{y} = \theta_0 + \theta_1 x_1 + \cdots + \theta_m x_m$$

with m being the number variables. We create a "faux" x_0, where all the $x_0 = 1$, since it is a constant.

- We can rewrite this as a matrix!
  - We have n points of data to test, with m features for the input.
  - In a matrix X, we can list each row as a piece of data, with columns corresponding to a feature. X is an $n \times m$ matrix.
  - We can transform θ into a vector that when multiplied with X, matching with corresponding feature columns. θ is a $m \times 1$ vector.
  - We can transform $\hat{y}$ into a vector that is the product of X and θ that corresponds to each row in X accordingly. $\hat{y}$ is a $n \times 1$ vector.

$$\underbrace{\begin{bmatrix} x_{11} & x_{12} & \cdots & x_{1m} \\ x_{21} & x_{22} & \cdots & x_{2m} \\ \vdots & \vdots & \vdots & \vdots \\ x_{n1} & x_{n2} & \cdots & x_{nm} \end{bmatrix}}_{X} \cdot \underbrace{\begin{bmatrix} \theta_1 \\ \theta_2 \\ \vdots \\ \theta_m \end{bmatrix}}_{\theta} = \underbrace{\begin{bmatrix} \hat{y}_1 \\ \hat{y}_2 \\ \vdots \\ \hat{y}_n \end{bmatrix}^T}_{\hat{y}}$$

- We can get the total error by subtracting the actual y values from the generated expected values. We can replace $\hat{y}$ with our data matrix and our weights, with the formula for our error ends up being

$$\frac{1}{2}||X\theta - y||_2^2$$

To find the best θ, our goal will be to minimize this error. This is known as the **Least Squares Optimization**, since we want to minimize the squared error of our guesses by optimizing our parameters.

You may ask why we have a $\frac{1}{2}$ in our error formula. Since we want to minimize this, multiplying it by any positive constant wouldn't affect the results. However, it does make it easier to find the lowest error with calculus!

We want to find the point where the error is minimized, which is equivalent to when the derivative (or gradient) of the error in regards to θ is equal to 0. We know this is the minimum because the error is a **convex** problem, meaning there is a global minimum.

- We want to find the minimum of the error, otherwise known as the loss function:

$$\frac{1}{2}||X\theta - y||_2^2$$

- Since we can only change our parameters, θ, we have to find the point when the derivative (otherwise known as gradient) of the loss function *in regards to* θ is equal to zero. This makes our **objective**:

$$\nabla_\theta \frac{1}{2}||X\theta - y||_2^2 = 0$$

- Since θ is convoluted with X in the equation prior to the error being squared, we cannot directly derive it. Thus, we must use the chain rule:

$$\nabla_\theta f(X\theta) = X^T \nabla_{(X\theta)} f(X\theta)$$

  Letting us form the equation:

$$\nabla_\theta \frac{1}{2}||X\theta - y||_2^2 \rightarrow \nabla_{(X\theta)} X^T \frac{1}{2}||X\theta - y||_2^2$$

- Now that we have that out of the way, we have to get the gradient in regards to the L2 norm. We can do this with the following equation:

$$\nabla_\theta ||\theta - z||_2 = 2(\theta - z)$$

  This is where the $\frac{1}{2}$ comes in handy, letting us derive

$$\nabla_{(X\theta)} X^T \frac{1}{2}||X\theta - y||_2^2 \rightarrow X^T(X\theta - y)$$

- Finally, recall that we want to find θ to set this equation equal to 0.

$$X^T(X\theta - y) = 0$$

  First, we use the distributive property of matrix multiplication.

$$X^T X\theta - X^T y = 0$$

  Then, move the term without θ to the right hand side.

$$X^T X\theta = X^T y$$

  Finally, we left-multiply with an inverse to isolate θ, giving us our result.

$$\theta = (X^T X)^{-1} X^T y$$

This is known as the **Closed Form Equation** for Least Squares Optimization. Next, we want to try and expand this to fit with data that might not necessarily fit alongisde a straight line.

1.3 Linear Regression and Ridge Regression

What if our data doesn't fit a line? We may have some output points that scale quadratic or even higher based on some input variable. How can we use the Least Squares optimization then? The answer is creating mock variables.

- If the terms are of a higher order polynomial, we can simply create mock features that are equivalent to some polynomial of the input.

  $$\hat{y} = \theta_0 + \theta_1 x \longrightarrow \hat{y} = \theta_0 + \theta_1 x + \theta_1 x^2 + \cdots$$

  We call this **Linear Regression**, and using it to more accurately fit data.

- We call the number of polynomials **degrees**. Expanding an input n times means we are doing a linear regression of degree n. More degrees is more accuracy and less error, since we can fit to the points with more nuance.

- So why not do a really high degree? **Overfitting.** We would fit too much to the noise in the data rather than the general trends themselves.

- High degree linear regressions tend to have incredibly high θ values, as they are "bending" the line in strange ways to fit to all the data points.

This is where **Regularization** or Ridge Regression comes in. We want to make sure that the accuracy we gain by increasing the degree isn't offset by the crazy θ parameters we get as a result.

- When we overfit, we notice that the weights become massive, to account for some of the noise that may be present in the data. So, we take the L2 Norm of the resulting θ parameters themselves.

- For this, we add a tuning parameter, λ, a value between 0 and 1. This essentially is the magnitude of consideration we give to the aforementioned coefficients (our bias). This gives us a regularization parameter:

  $$\lambda \Sigma_{j=1}^{m} \theta_j^2$$

- With a larger λ, we want to make sure the weights are not overfitting our data. High bias, low variance. With a small λ, we have lower bias, but higher variance, as we are allowing it to "fit" the data better.

- With this regularization parameter, we have our final loss function:

  $$L(\theta) = \frac{1}{2n}\left[\underbrace{\Sigma_{i=1}^{n}(\hat{y}_i - y_i)^2}_{\textbf{Least Squares}} + \overbrace{\lambda \Sigma_{j=1}^{m} \theta_j^2}^{\textbf{Regularizer}}\right]$$

Again, like earlier, we want to minimize this loss function, so again, we take the gradient. Let's take a step by step look at how to obtain the normal equation of **Closed Form Ridge Regression**.

- Consider the loss function

$$L(\theta) = \frac{1}{2n}\left[\Sigma_{i=1}^{n}(\hat{y}_i - y_i)^2 + \lambda\Sigma_{j=1}^{m}\theta_j^2\right]$$

- Keep in mind that the summation of errors would simply be the magnitude of the data vectors. This changes the loss function to:

$$\frac{1}{2n}\left[||X\theta - y||_2^2 + \lambda||\theta||_2^2\right]$$

  Since the magnitude can be calculated as the product of the transposed vector and itself, we get:

$$\frac{1}{2n}\left[\frac{1}{2n}[(X\theta - y)^T(X\theta - y) + \lambda\theta^T\theta\right]$$

- Using matrix math and transposing, we multiply this out to get:

$$\frac{1}{2n}\left[y^Ty + \theta^TX^TX\theta - 2(X\theta)^Ty + \lambda\theta^T\theta\right]$$

- Now, we want to take the gradient of this in regards to θ and set it to 0. Since the first term has no θ, we can eliminate it.

$$\frac{1}{2n}\left[\frac{\partial}{\partial\theta}\theta^TX^TX\theta - \frac{\partial}{\partial\theta}2(X\theta)^Ty + \frac{\partial}{\partial\theta}\lambda\theta^T\theta\right]$$

- We take the partial by treating the dot product of θ and itself as θ^2, letting us eliminate the first constant to get the gradient, which we want to set to 0.

$$X^TX\theta - X^Ty + \lambda\theta = 0$$

- Like in the closed form of least squares, simply solve for θ by first moving all terms without it to the right hand side.

$$X^TX\theta + \lambda\theta = X^Ty$$

  Then, use the distributive property to take θ out of the terms on the left hand side.

$$(X^TX + \lambda)\theta = X^Ty$$

  Finally, apply the left inverse of the term to both sides to get the final form of the equation.

$$\theta = (X^TX + \lambda)^{-1}X^Ty$$

As an aside, we can test for overfitting by separating our data into test and training data. **K-Fold Cross Validation** is one method of doing this.

1. **Separate our data into K equal parts**.
2. **Designate one part as test data, and the rest as training data**.
3. **Train our model using training data**.
4. **Test model accuracy using test data**.
5. **Repeat steps with other parts.**
6. **Report averaged accuracy**.

1.4 Singular Value Decomposition

Singular Value Decomposition is a useful way to represent matrices, since every matrix can be represented as an SVD. Before we go into it, we need to know some more linear algebra stuff.

- **Vector Spaces** are a generalization of Cartesian Plane. We denote vector spaces as $\mathbf{R}^d$, where elements are

  $$[x_1, x_2, \cdots, x_d]$$

  A set of all $n \times m$ matrices can be denoted as vector space $\mathbf{R}^{n \times m}$.

- **Linear Independence** is when a set of vectors in the same vector space cannot be represented as a linear combination of the other vectors.

- **Matrix Rank** is the number of linearly independent column or rows in matrix. For transformation matrices, rank tells us dimensions of output. Full rank matrices are $n \times n$ matrices with row-rank = col-rank = n.

- **Orthonormal** vectors are both

  1. **Orthogonal**: Perpendicular/Independent.
  2. **Normalized**: All of unit (1) length.

- Consider vector space $\mathbf{R}^d$. All vectors can be represented as a linear combination of **orthonormal basis vectors**, e_i, where

  $$e_1 = [1, 0, \cdots, 0], e_2 = [0, 1, \cdots, 0], \cdots, e_d = [0, 0, \cdots, 1]$$

  - **Span** of a set of vectors U is the set of all vectors that can be represented as a linear combination of vectors in U

    $$\text{For all } v \in Span(U) \rightarrow v = \sum_{i=1}^{u} x_i u_i$$

- **Linear Transformations** are functions $f : U \rightarrow V$ such that

  $$\text{For all } u_1, u_2 \in U \rightarrow f(\alpha u_1 + \beta u_2) = f(\alpha u_1) + f(\beta u_2)$$

  We can represent these as matrix multiplications!

- **Eigenvectors** x of matrix A is are non-zero vectors such that when we apply A to it, it does not change its direction, but rather scales it by a value λ, its corresponding **eigenvalue**.

$$Ax = \lambda x$$

  These can **ONLY** be found for square matrices, and not every square matrix has them.

  - IF they do however, the have n of them ($n \times n$ matrix).
  - Scaling an eigenvector does not affect its eigenvalue. Therefore, we normalize them to length 1.
  - Eigenvectors of the same matrix are perpendicular/orthogonal, meaning we can express data using eigenvectors as basis vectors of the data.
  - The *trace* of a matrix is the sum of its eigenvalues.
  - The *determinant* of a matrix is the product of its eigenvalues.
  - The *rank* of a matrix is the number of non-zero eigenvalues.

- **Spectral Theorem:** Consider *symmetric* $n \times n$ matrix A.
  - *Symmetric* implies that $A = A^T$.

  Then, there exist real numbers (eigenvalues) $\lambda_1, \cdots, \lambda_n$ and corresponding, orthogonal, non-zero real vectors (eigenvectors) $\phi_1, \cdots, \phi_n$ such that

$$A\phi_i = \lambda_i \phi_i \rightarrow A\phi - \lambda I \phi = 0 \rightarrow (A - \lambda I)\phi = 0$$

This information allows us to go into singular value decomposition.

- Every matrix A can be "decomposed" into $A = U\Sigma V^T$
  - Columns of U are orthonormal eigenvectors of $A \cdot A^T$
  - Columns of V are orthonormal eigenvectors of $A^T \cdot A$
  - Σ is a diagonal matrix containing the **square roots** of the eigenvalues from U or V in descending order (*singular values*).

$$\underbrace{\begin{bmatrix} a_{11} & \cdots & a_{n1} \\ \vdots & \ddots & \vdots \\ a_{1m} & \cdots & a_{nm} \end{bmatrix}}_{A=n\times m} = \underbrace{\begin{bmatrix} u_{11} & \cdots & u_{n1} \\ \vdots & \ddots & \vdots \\ u_{1n} & \cdots & u_{nn} \end{bmatrix}}_{U=n\times n} \underbrace{\begin{bmatrix} \lambda_1 & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & ? \end{bmatrix}}_{\Sigma=n\times m} \underbrace{\begin{bmatrix} v_{11} & \cdots & v_{m1} \\ \vdots & \ddots & \vdots \\ v_{1m} & \cdots & v_{mm} \end{bmatrix}^T}_{V=m\times m}$$

- Since AA^T is always a symmetric matrix, we can apply the spectral theorem to get this by hand! Then we can easily get the values.

Why is this representation useful to us?

- Gives a uniform way to get the eigenpairs of a matrix with spectral theorem.
- The eigenvectors are a basis for the matrix, allowing us to rerepresent the matrix in terms of its eigenvectors
- Corresponding eigenvalues represent the "impact" each eigenvector has on the shape of the matrix
- This allows us to find the most important eigenvectors, and create an approximated version of the matrix using those as the basis vectors.
- We do this by taking the top vectors with the highest eigenvalues, and then project the original data onto just these eigenvectors.

This lets us clear out a lot of time and space, translating nicely into the next segment: PCA.

1.5 Principal Component Analysis

First, let's talk about some background statistical stuff between two distributions, x and y. Let's imagine these as points on a scatter plot.

- **Mean:** $\bar{x} = \frac{1}{n-1}\sum_{i=1}^{n} x_i$. Note that we divide by $n-1$ since we are assuming a sample, not the population.
- **Variance:** $\sigma^2 = \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \bar{x})^2 = \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \bar{x})(x_i - \bar{x})$
- **Covariance:** $Cov(X,Y) = \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})$. Measures the relationship between two variables (dimensions).
- **Covariance Matrices:** For an n-dimensional data set, we have an $n \times n$ covariance matrix.

What is **Principal Component Analysis (PCA)**? When we work with data that has higher dimensions, it may be difficult to perceive it in a Cartesian plane. Higher dimensional data also introduces much more computational complexity. More features will scale for a very long time with more data, and this can become untenable in the long run when we perform many operations on high dimensional data. Our data collection can become more accurate, with more and more features, but the runtime can suck. Any features that are less relevant still take the same amount of time as other features. Some features may scale directly or inversely with each other (think elevation temperature), rendering calculating both of them slightly obsolete.

This is where PCA comes in. **Principal Component Analysis** is *technique to compress data by reducing its dimensionality and is useful in classification.* We do this by finding the basis vectors of the data matrix, known as the *principal components*, to represent the data. Instead of looking along the individual features, we want to look at the data by its variance. We use PCA

as a technique to find the directions along which the data lines up the best. To do this, we look at the *covariance* matrix of our data, which tracks how the dimensions are correlated with each other. For a matrix M, the covariance matrix is $C = M^T M$. The directions of the variance here are the *basis vectors* of this matrix, otherwise known as the **eigenvectors**. This is where **Singular Value Decomposition** relates back, as the eigenvectors of C is equivalent to U in $M = U\Sigma V$ in the singular value decomposition. With this, we know the directions of the variance, and we can find the most significant directions by looking at Σ for the eigenvalues.

With these basis vectors, we can then re-project the data along the K most significant directions of variance, keeping a majority of the "information" in the data, while massively reducing the calculation time required to perform operations on the data. Keep in mind that these new principal components have no inherent meaning to them, unlike the features. We also have to decide how many of the basis vectors to keep. By summing the **eigenvalues** associated with the vectors, we can get the total variance of the dataset. We can then account for the variance each basis vector accounts for by taking its proportion in regards to the whole. So, we can keep taking the most significant basis vectors until we account for a certain percentage of the variance in the original data! Let's recap the steps of PCA and go through the math more rigorously.

1. Take our data matrix M and mean center it. This makes the math a lot easier to work with.
2. Calculate the covariance matrix of M, $C = M^T M$.
3. Get the eigenvalues and eigenvectors of C. Ensure that the eigenvectors are normalized to be of unit length.
4. Alternatively, use Singular Value Decomposition on M, Getting $M = U\Sigma V$, then use the columns of U as the eigenvectors, and the **SQUARED** values of Σ as the corresponding eigenvalues (since Σ is a diagonal matrix of *singular values*.)
5. Sum the eigenvalues, then take the top K eigenvectors and corresponding eigenvalues until they account for a target percentage of the sum of eigenvalues. This captures that percentage of the variance in the data. These eigenvectors are denoted as W
6. Project the data along these dominant principal components, with the end result being $T = WX^T$.

2 Introduction to Image Processing

2.1 Background Information

- Image can be thought of as a function $f : \mathbf{R}^2 \to \mathbf{R}$, where $f([x, y])$ returns the **intensity** of the image at the position (x, y)

  - We can realistically expect the image to be defined over a range, where $min(x, y) = (0, 0)$, $max(x, y)$ is the bounds of the rectangle, and $min/max(f([x, y]) = [0, 1]$, where the value from 0-1 is between a black-white spectrum, after being normalized.
  - A coloured image can be represented as 3 of these functions pasted together, corresponding to RGB values.

- Computer vision usually uses **digital (discrete)** images, where we *sample* the 2D space on a grid, then *quantize* each sample (rounding to nearest integer). With this, we can represent the image as a *matrix of integer values*.
  - Again, we simply represent an RGB image as 3 of these matrices. So in numpy, an $(x, y, 3)$ shape array
  - We can convert RGB to Grayscale, with the intensity being

$$I = 0.299 * R + 0.587 * G + 0.114 * B$$

- **Image Processing** operations define new images based on existing image. $g([x, y]) = t(f([x, y])$
- **Thresholding** is a simple image processing operation. If the value of a pixel is above T, set it to 1(255 for MAX whiteness), else 0.

2.2 Histograms and Images

- We can make **histograms** of the intensity values of images. By separating the range of intensities (typically 0-255) into bins, we can plot the distribution of intensities.
- For bins, can have equal length OR equal membership bins. For the former, allows us to see general "brightness" of image, but can dilute a lot of the information at denser bins. For the latter, equal membership can show us the general clumps of intensities, but can have trouble differentiating between dips and raises. Here are some formulae for bins and widths, where v are the intensity values, n is the number of values, b is the count of bins, and w is the width.
  - **Square:** $b = \sqrt{n}, w = \frac{max(v)-min(v)}{\sqrt{n}}$
  - **Sturges:**
  - **Rice:**
  - **Scott:**
  - **Freedman-Diaconis:**

- **Histogram Equalization** can be used as an image manipulation technique that balances the distribution of intensity values to be equivalent across the range of intensities. Here's how we do it:

  1. **Get intensity values** of all the pixels of the image. We assume these are *discrete* integers ranging from 0 to 255.
  2. **Create a histogram** of these values, ordered in increasing intensity.
  3. **Apply histogram equalization function**, $h(v)$ to the intensities of the bins. These will be the new bins values.
  4. **Replace pixels** of intensity v with intensity $h(v)$.

  The equalization function takes clumps of intensities and distributes them over the range of the function. Here it is:

  - v: Intensity value (of bin in histogram)
  - $cdf(v)$: **Cumulative Distribution Function**. Equivalent to the number of occurrences of intensity v or less.
  - cdf_{min}: Minimum value of $cdf(v)$, Equivalent to the number of occurrences of the *lowest intensity*.
  - H, W: Height and Width of the image. Equivalent to the total number of pixels in the image.
  - L: The range of intensities in the image. Equivalent to $v_{max} - v_{min}$.

$$h(v) = round\left(\frac{cdf(v) - cdf_{min}}{(W \times H) - cdf_{min}}\right) \times (L - 1)$$

- We can also compare two histograms to possibly find similarities between images. Here are a couple ways to compare $h(i), g(i)$, two histograms
  - **Sum of Squared Differences (SSD):** $||h - g|| = \sigma_{i=1}^{N}(h(i) - g(i))^2$
  - **Cosine Distance:** $cos(h, g) = \frac{h \cdot g}{||h|| \times ||g||}$
  - **Chi-square Distance:** $\chi^2(h(i), g(i)) = \frac{1}{2}\sum_{m=1}^{k}\frac{(h_i(m) - g_i(m))^2}{(h_i(m) + g_i(m))}$

2.3 Correlation and Convolution

- **Correlation** and **Convolution** are shift-invariant, linear operations on images.

- 1-D Example: Imagine a list of numbers. We can can do an averaging correlation by having the new value of any cell in the array equivalent to the average of itself, the prior number, and the upcoming number! We can use 0's or some other value to pad the edges. We can even go with larger "boxes" (len 5 vs 3)

- In 2 dimensions, we make a 3x3 "box" with different values. This is our filter (or kernel). The only difference between correlation and convolution is that **the filters are flipped horizontally and vertically**. For example:

  $$\text{Convolution} \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} \text{ is equivalent to correlation } \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

  which means that for symmetrical filters, they are the same

- Generally, **correlation** is used for template matching, and **convolution** is used for image processing operations (like smoothing)

- Here are some common filters with examples:
  - **Box/Mean Filter** is just a box where all the values are equivalent to 1 divided by the size of the box.
  - **Gaussian Kernel** takes a normal distribution of the points around it, weighing more towards points in the center.
  - **Median Filter** is like mean filter, but instead takes the median value of the box.

  $$\text{Mean} = \begin{bmatrix} \frac{1}{9} & \frac{1}{9} & \frac{1}{9} \\ \frac{1}{9} & \frac{1}{9} & \frac{1}{9} \\ \frac{1}{9} & \frac{1}{9} & \frac{1}{9} \end{bmatrix} \text{Gaussian} = \frac{1}{16} \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$$

- We can also have larger boxes (or kernels), which would increase the size of our mask. The increased mask means more neighbors contribute, leading to less noise variance, but we have a bigger spread, more blurring, and more compute cost.

- The Gaussian Filter is especially nice. It is an approximation of

  $$G_\sigma(x,y) = \frac{1}{2\pi\sigma^2} exp^{-\frac{x^2+y^2}{2\sigma^2}} = (\frac{1}{\sqrt{2\pi}\sigma} exp^{-\frac{x^2}{2\sigma^2}})(\frac{1}{\sqrt{2\pi}\sigma} exp^{-\frac{y^2}{2\sigma^2}})$$

  This is nice since it can be seperated into 2 1-dimensional filters rather than a 2-dimensional filter, meaning that we can apply them seperately at a cheaper cost combined rather than the whole filter together! We convolve each row with the 1D gaussian, then each column!

2.4 Fourier Domain

The way we have been percieving images from now is known as the **spatial domain**. The **Fourier (Frequency) Domain** takes the function and re-represents it as a set of sinusoidal waves of varying magnitudes. Basically, this is important for us because when we apply the filters we have been talking about, we have to do the individual calculations, which can be very computationally expensive. By transforming it, this allows us to perform multiplications directly, which save us a lot of time.

- Fourier transform provides an orthonormal basis for images with the following vectors:

$$\frac{1}{\sqrt{2\pi}}, \frac{\cos(kx)}{\sqrt{\pi}}, \frac{\sin(kx)}{\sqrt{\pi}}, \text{ for } k = 1, 2, \cdots$$

- **Fourier Transform** of a convolution of two functions is equivalent to the product of their respective transforms. The same is true for the inverse of the convolutions.

$$F[g * h] = F[g]F[h]$$

- Convolution in the spatial domain is equivalent to multiplication in the frequency domain! This is what makes Fourier so useful.

2.5 Summary

- **Digital Images** are represented as a grid of intensity values. *Colored* images are a combination of red, green, and blue intensity images.
- **Image Processing** operation are transformations on images to form different images, with an example being **thresholding**, setting intensities as black or white based on a target value.
- **Histograms** of intensity values are a good way to interpret images. They prove useful to categorize parts of an image.
- **Histogram Equalization** is a good way to transform images to make them clearer (*dark images brighter, vice versa*)
  1. Create a histogram of the intensities in the image.
  2. Calculate the equalization function for all the intensities represented in the histogram, $h(v)$
  3. Replace the intensities in the image with their equalized intensity.
- **Correlation, Convolution,** and **Cross-Correlation** are matrix operations that can be used for image processing.
  - These use various **filters / kernels** that are applied across the matrix.
  - Some 2-D filters, like the **Gaussian Kernel**, can be convolved as 2 separate 1-D filters, speeding up computation time.
- For filters that are more complex, the **Fourier Transform** converts images in such a way that convolution can be performed as multiplication, which can speed up convolution with complex filters.

3 Edge Detection (Sobel, Canny)

3.1 Intro to Edges

In an image, an **edge** can be seen as a sudden shift in intensity values. While a gradual change can be attributed to just slight variations in lighting, a rapid one is likely a discontinuity in the image. Examples of these discontinuities can be

- Surface normal discontinuities
- Depth discontinuities:
- Surface *color* discontinuity
- Large illumination discontinuities

Imagine a 1-D image as a line. An edge would be a sharp variation. If we were to take the **derivative** of this, we would find that the places of maximum magnitude are the edges. In the **2nd derivative**, these edges would be 0. For a 2 dimensional image, we can denote this first derivative (gradient) as ∇f, getting the partial derivatives of the image in the x and y directions. We can get the direction of this gradient, θ, from these partial values. We can get the strength of this edge from the gradient's magnitude, $||\nabla f||$.

$$\nabla f = [\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}], \theta = \tan^{-1}(\frac{\partial f}{\partial y}/\frac{\partial f}{\partial x}), ||\nabla f|| = \sqrt{(\frac{\partial f}{\partial x})^2 + (\frac{\partial f}{\partial y})^2}$$

3.2 Working with Digital Images

How do we differentiate a digital image $f[x, y]$?

- Reconstruct continuous image, then take gradient, or...
- Take discrete derivative using finite difference:

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

- We can implement this discrete derivative as a *cross-corrrelation* using something known as a **Sobel Operator**. (*note that the* $\frac{1}{8}$ *is used to get the correct gradient value*)

$$s_x = \frac{1}{8}\begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}, s_y = \frac{1}{8}\begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$$

- With s_x, s_y, we have vertical and horizontal edge detectors, respectively.

What if the image is noisy? The small variations over a continuous function can make the derivative illegible. The answer to this is smoothing.

- Taking the image, f, We smooth with a Gaussian kernel, h. This smoothed image can be represented as $h * f$, where $*$ is the convolution.
- Next, we need the partial derivatives, but instead of taking the partials in regards to $h * f$, we use the *derivative theorem of convolution*,

$$\frac{\partial}{\partial x}(h * f) = (\frac{\partial}{\partial x}h) * f$$

  allowing us to save an operation.
- We can take the second partial derivative as well, since when it is 0, we reach points of extrema. This is known as the **Laplacian ∇ of Gaussian**.
- Note that as we increase the size of the smoothing, we remove more noise, but blur the edges, finding edges at different "scales".

3.3 Canny Edge Detection

What is our optimal edge detector? Assuming the image has some noise, and our edge detector must be composed of linear filtering, we want:

- **Good Detection:** Filter responds to edge, not noise
- **Good Localization:** Detected edge is near the true edge
- **Single Response:** one edge reported per edge.

To do this, we use a **Canny Edge Detection** algorithm, which has the following steps for an image, M:

1. **Smoothing:** Apply a gaussian kernel on the image via convolution. This reduces the impact of noise of the image.
2. **Find Derivatives (gradients):** We convolute the **sobel operators** to find the images of the partial derivatives:

$$F_x = s_x * M, F_y = s_y * M$$

3. **Find magnitudes and orientations of gradient:** We want to find how "strong" the edge is at a given point, alongside the direction it is facing.

$$G = \sqrt{(F_x^2 + F_y^2)}, \theta = \tan^- 1(\frac{F_y}{F_x})$$

4. **Non-Maximum Suppression:** If you recall, more blur = thicker edges reported. We want to trim these down. To do this, we look at the gradient intensities of the pixel (G) along the direction of the gradient (θ), and if it is the maximal value, then we keep it, else dump it.

- We can either look at *interpolated pixels* in the direction, which are sub-pixel values generated from the surrounding pixels, or we can use a more discrete method, which rounds θ and looks at the adjacent pixels in the horizontal ($\theta = 0°$), vertical ($\theta = 90°$), or diagonal ($\theta = 45°, 135°$ for NE/SW and NW/SE, respectively) directions.

5. **Hysteresis (Linking and Thresholding)** Our goal here is to remove edge pixels caused by noise, while linking together edge pixels to get continuous edges.

   - We want to define two thresholds, low and high, based on the maximum gradient intensity of G, say, 35 and 90 percent.
   - For edge candidates under the low threshold, they are too weak, likely being results of noise. Discard these. (*Thresholding*)
   - For candidates above the high threshold, they are significant enough to just outright keep. Keep these, marking them as "strong".
   - For those in the middle, mark them as weak. If any strong pixels are in the 8 pixels adjacent to it, keep them. Else, discard. This part is usually done as a mask. (*Linking*)

3.4 Summary

- **Edges** in an image are sudden shifts in intensity values.
- Edges can be found as **maxima** in the **first gradient** of the image and **zero-crossings** in the **second derivative.**
- We use **discrete partial derivatives** in the x and y directions to be able to operate on digital images.
- The **Sobel Operator** is cross-correlation kernel that can detect edges.
- The **Gaussian Kernel** can be used to smooth an image, reducing the impact noise on the gradients.
- **Canny Edge Detection** is an algorithm used to find edges in an image:

  1. **Smooth the image** by convolution with a *Gaussian Kernel*
  2. **Find Gradient Images** in the x and y directions by convolution with the *Sobel Operator*
  3. **Find Magnitudes and Orientations** of the gradient images at each pixel
  4. **Suppress Weak Edge Points** by looking at the strengths of gradients in the same direction, taking only the peak.
  5. **Hysteresis**, the process of *thresholding* away standalone weaker edges and *linking* weak edges with adjacent strong edges.

4 Keypoints and Features

We want to gleam useable information from images. Edges are some nice pieces of information that we can gain, and let us step our foot in the door in regards to identifying different things in images. However, edges often fail in many ways when images aren't taken with the exact same assumptions. This is where corners come in. Why find a corner? Corners have many advantages over edges:

- Corners are **invariant** to translation shifting, illuminations, and scaling. Edges can become too long or misinterpreted with these factors.
- A corner is an individual point, allowing us to identify these key points faster than edges, which may be regions.

Edge detectors usually fail to find corners, since they act as maxima and evade the detection. When a point is a corner:

- The gradients around the corner change in all directions. If just one direction, likely an edge.
- The change is significant. If the change is everywhere, but small, we're likely simply in a constant patch.

4.1 Harris Corner Detector

- Given a point in an image I, we want to find out if it is a corner. To do this, we look at a window around this image patch, calling it W.
- We want to shift this window around. This is because if the window is in a "flat" region, there will be no significant change in all directions, and for an edge, there will be no change along the direction of the edge. However, for a corner, there will be significant change in multiple directions! We denote this shifting as $\Delta x, \Delta y$.
- To detect the changes in the gradient, we use the **Sum of Squared Differences (SSD)**. We want to find that the gradient has significant changes in two directions, meaning the SSD is high. The equation for this is given as

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

- Using *Taylor Expansion*, we can approximate using the partial derivatves of the image, I_x, I_y, and find that:

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

$$I(x+\Delta x, y+\Delta y) = I(x,y) + \Delta x I_x(x,y) + \Delta y I_y(x,y)$$

Using this, we can generalize this back to the summation to find that SSD is simply

$$f(\Delta x, \Delta y) = \sum_{x,y \in W} [\Delta x I_x(x,y) + \Delta y I_y(x,y)]^2$$

Then, taking this equation, we can multiply it out to find that

$$f(\Delta x, \Delta y) = \sum_{x,y \in W} [\Delta x I_x + \Delta y I_y]^2 = \sum_{x,y \in W} [\Delta x^2 I_x^2 + \Delta y^2 I_y^2]$$

- The next step is representing this back in matrix form. Taking $\Delta x, \Delta y$ as scalars, and I_x, I_y as matrices, we get can convert the above equation into

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

  We call the middle part M, and if you notice, it is simply the **gradient covariance matrix**, indicating the covariance of the changes of the image in the x and y directions.

- When we look at M, we can find 2 *eigenvectors* to be the general directions of variance in the variance of the gradients (direction of edges) and the 2 *eigenvalues* to be the significance of these values. Due to this we can gleam significant information: If the two *eigenvalues* are vastly different, with one being much larger than the other, the point in question is likely an edge, since the change in the window is happening in one direction. If they are similar, but the values are small, the change is happening everywhere, but is insignificant, indicating a "flat" region. **When the two eigenvalues are similar and large, the point is likely a corner.**

- We can use some linear algebra to get hacky ways of getting the status of the eigenvalues.

$$det(M) = \lambda_1 \lambda 2, trace(M) = \lambda_1 + \lambda_2$$

  With the determinant of the matrix, we can see how significant the two eigenvalues are. With the trace, we can see how close they are to each other. Therefore, we create a score R from these two values, where $k = 0.040.06$ as an empirically determined constant.

$$R = det(m) - k(trace(M))^2$$

  When R is high, that means that the eigenvalues are both significant and similar to each other. This helps us in determining the likelihood of a point being a corner!

Let's sum all this math up into an algoritm known as the **Harris Corner Detector**.

1. **Prerequisites:** Convert the image to gray scale and apply a Gaussian blur to smooth out any noise.
2. **Get Partial Derivatives:** Apply the *sobel operator* to find the partial derivatives for the images.
3. **Construct the Gaussian Windows:** We do this by finding the second partial derivatives, then creating the windows by using a mask.
   - Compute I_x^2, I_y^2, I_xy
   - Convolve the images and find M for each pixel, $\sum_{x,y \in W} \begin{bmatrix} I_x^2 & I_xI_y \\ I_xI_y & I_y^2 \end{bmatrix}$
4. **Compute Detector Responses:** Recall the score function, $R = det(m) - k(trace(M))^2$. Compute this at each pixel.
5. **Find Maxima of Responses:** Get the local maxima of R in a neighborhood around it. Make sure that it is also above some minimal threshold. We can use an algorithm called **Adaptive Non-Maximal** suppresion for this to also make sure our corners are well distributed across the image.

4.2 Scale Invariant Feature Transform

The **SIFT** algorithm finds invariant local features in images using Laplacian of Gaussians for scale invariance. *Sharp local intensity changes* like this are good for identifying relative scales of regions. SIFT has 3 main sections when being used. Let's go through each of them!

1. **Detection:** Here, we want to detect the actual keypoints that are invariant to all of these affine changes.
   (a) Convolve the image with Gaussian Filters at $s + 3$ different **scales** (min 3). Assuming that the image is $I(x, y)$ and the Gaussian Blur is $G(x, y, \sigma)$ with the initial assumed blur of σ, we can find the Gaussian of an image and the scale of an image as:
   $$G(x, y, \sigma) = \frac{1}{2\pi\sigma^2} e^{-(x^2+y^2)/2\sigma^2} \; L(x, y, \sigma) = G(x, y, \sigma) * I(x, y)$$
   (b) Get the *Difference of Gaussians (DoG)* by subtracting the successive scaled blurred images. Denote this as $D(x, y, \sigma)$. k is the extra blur factor, determined as $k = 2^{1/s}$, meaning we want the final image scale to be twice as blurred as the initial image.
   $$D(x, y, \sigma) = L(x, y, k\sigma) - L(x, y, \sigma)$$
   (c) Find the local minima or maxima pixels in the DoG images by comparing them to 26 neighbors: The eight surrounding pixels in the same scale, then the 9 pixels centered around the pixel location in the neighboring scales. These points of interest become candidates for keypoints.

(d) Repeat the above steps with a copy of the image that has been scaled at a different size, usually cut by half. For example, if the image is originally size $x \times y$, we rescale the image to be size $\frac{x}{2} \times \frac{y}{2}$. We can do this to either the original image, or one of the blurred images in step (a). We call these different sized images **octaves**. The DoG images at these scales and octaves become what we call the DoG Pyramid.

(e) When we find these candidates at different octaves, the reported keypoint may be slightly off from the true keypoint, since the true keypoint may be in between pixels. To fix this, we want to **localize the keypoint** by fitting it to the data around it. Remember that in regards to derivatives, the location of a keypoint in an image is when the *first derivative is a high value* and when the *second derivative (laplacian) is* 0. We use Taylor expansion here.
We represent (x, y, σ) as a vector $\vec{x}$

$$\vec{x} = (x, y, \sigma)^T; D(\vec{x}) = D + \frac{\partial D^T}{\partial \vec{x}} + \frac{1}{2}\vec{x}^T \frac{\partial^2 D}{\partial \vec{x}^2}\vec{x}$$

Mathematically, we can represent these derivatives as vectors and matrices, where H is the hessian matrix for the second partial.

$$\frac{\partial D}{\partial \vec{x}} = \begin{bmatrix} \frac{\partial D}{\partial x} \\ \frac{\partial D}{\partial y} \\ \frac{\partial D}{\partial \sigma} \end{bmatrix}, H = \frac{\partial^2 D}{\partial \vec{x}^2} = \begin{bmatrix} D_{xx} & D_{xy} & D_{x\sigma} \\ D_{xy} & D_{yy} & D_{y\sigma} \\ D_{\sigma x} & D_{\sigma y} & D_{\sigma\sigma} \end{bmatrix}$$

i. The true location of the extrema based on the reported keypoint will get farther in the direction of the first derivative based on the size of the second derivative. We call this offset $\hat{x}$.

$$\hat{x} = -\frac{\partial^2 D}{\partial x^2}^{-1} \frac{\partial D}{\partial x}$$

ii. We want to eliminate points with low contrast, since they are likely just extrema in a flat location. We can compute this as $\hat{D(x)}$, and discard the points if they are below a certain contrast threshold, say $|D(\hat{x})| < 0.03$

$$D(\hat{x}) = D + \frac{1}{2}\frac{\partial D}{\partial x}^{-1} \hat{x}$$

iii. We also want to eliminate keypoints that are edges, since these are not invariant to scaling and transformation. Recall from the Harris Corner Detector that we can detect when a keypoint is an edge based on the variance in the x and y directions. We take the Hessian matrix and consider the top left corner with the partials in the x and y directions:

$$H = \begin{bmatrix} D_{xx} & D_{xy} \\ D_{xy} & D_{yy} \end{bmatrix}$$

Recall that this is like the covariance matrix, and that we can get the eigenvalues from trace and determinant. Since we already omitted keypoints with weak responses in the previous step, here we care about the ratio of eigenalues being too large. We call this ratio r.

$$Tr(H) = D_{xx} + D_{yy} = \lambda_1 + \lambda_2, Det(H) = D_{xx}D_{yy} - D_{xy}^2, r = \frac{\lambda_1}{\lambda_2}$$

We can get this ratio from an equation based on trace and determinant, and discard when the ratio is greater than some valeu (usually 10)

$$\frac{Tr(H)^2}{Det(H)} < \frac{(r+1)^2}{r}, r \approx \sqrt{\frac{Tr(H)^2}{Det(H)}}$$

(f) After localizing and filtering, scale the keypoints found at different octaves back to the size of the original image.

We can make the partials discrete and able to be operated on as a digital image with the following representations:

$$\frac{\partial D}{\partial \vec{x}} = \begin{bmatrix} \frac{\partial D}{\partial x} \\ \frac{\partial D}{\partial y} \\ \frac{\partial D}{\partial \sigma} \end{bmatrix} = \begin{bmatrix} \frac{D(x+1,y,\sigma) - D(x-1,y,\sigma)}{2} \\ \frac{D(x,y+1,\sigma) - D(x,y-1,\sigma)}{2} \\ \frac{D(x,y,\sigma+1) - D(x,y,\sigma-1)}{2} \end{bmatrix}$$

$$H = \frac{\partial^2 D}{\partial \vec{x}^2} = \begin{bmatrix} D_{xx} & D_{xy} & D_{x\sigma} \\ D_{xy} & D_{yy} & D_{y\sigma} \\ D_{\sigma x} & D_{\sigma y} & D_{\sigma\sigma} \end{bmatrix}$$

$$D_{xx} = D(x+1,y,\sigma) - 2D(x,y,\sigma) + D(x-1,y,\sigma)$$

$$D_{yy} = D(x,y+1,\sigma) - 2D(x,y,\sigma) + D(x,y-1,\sigma)$$

$$D_{\sigma\sigma} = D(x,y,\sigma+1) - 2D(x,y,\sigma) + D(x,y,\sigma-1)$$

$$D_{xy} = \frac{1}{4}[D(x+1,y+1,\sigma) - D(x-1,y+1,\sigma) - D(x+1,y-1,\sigma) - D(x-1,y-1,\sigma)]$$

$$D_{x\sigma} = \frac{1}{4}[D(x+1,y,\sigma+1) - D(x-1,y,\sigma+1) - D(x+1,y,\sigma-1) - D(x-1,y,\sigma-1)]$$

$$D_{y\sigma} = \frac{1}{4}[D(x,y+1,\sigma+1) - D(x,y-1,\sigma+1) - D(x,y+1,\sigma-1) - D(x,y-1,\sigma-1)]$$

2. **Description:** Here, we take the keypoints and assign an orientation of the keypoint, then construct a descriptor that is rotated based on the orientation. This image patch will be the feature vector we match between images. Recall that $L(x, y)$ is the Gaussian smoothed image. Remember that we can also calculate the magnitude of the gradient at a point along with the orientation of it with the following equations:

$$m(x,y) = \sqrt{(L(x+1,y) - L(x-1,y))^2 + (L(x,y+1) - L(x,y-1))^2}$$

$$\theta(x,y) = \tan^{-1}(\frac{L(x,y+1) - L(x,y-1)}{L(x+1,y) - L(x-1,y)}$$

   (a) **Find Orientation:** We take a patch around the keypoint who's size depends on the octave of the image. From this, we create a histogram with 36 bins of 10 degrees each, filling them with the magnitudes of the gradients of the pixels in the patch, rounded to the nearest direction. Then, we assign the keypoint with the orientation of the maximum of this histogram. Any other peaks in the histogram within 80% of the peak are also considered as seperate keypoints with the same location/scale, but different orientations.

   (b) **Create Descriptors:** We take a 16×16 patch around the keypoint. Then, we split it up into 16 4×4 subregions. In each of these subregions, we do the same thing as finding orientation and create a histogram of the gradients, but instead do only 8 bins. From these histograms in these subregions, we generate a feature vector. 16 regions with an 8 bin histogram form a 128 element feature vector.

3. **Matching:** This is the stage where we match keypoints between images to determine if the images are similar. Usually, we use some nearest neighbor stuff. We would find the nearest neighbor, then reject the keypoint if the distance to the nearest neighbor is within a threshold of some ratio to its distance to its next nearest neighbor (indicates ambiguity).

The main things to take away from **SIFT** is how it works, how each of the components of the stages work, and why they work.

4.3 Histogram of Gradients

The **Histogram of Gradients (HoG)** is a another technique to generate feature descriptors. This technique was designed and primarily used for pedestrian detection. Instead of focusing on keypoints, HoG focuses on regions of interest. We look at global features with a sliding window at multiple sized images. Here's a step-by-step look at the algorithm.

1. **Normalize and Gamma Color**: This is a preprocessing step, and although it has been found that this is not necessary for HoG, we should still think about it as a way to normalize all images prior to running any

feature detection algorithms on them. *Gamma Correction* is a way to evenly distribute the intensities along an image. We do the following:

$$I[x, y] = CI[x, y]^k$$

Where C is a constant and k is the normalization exponent. We set it to $\frac{1}{2}$ to make sure intense parts arent overly intense relative to the image. After this, we *normalize* the image to convert the numbers to floating point numbers between 0 and 1.

2. **Calculate Gradients**: Like many of the operations before, we calculate gradient at a point using the following formula:

$$\nabla f(x, y) = \begin{bmatrix} g_x \\ g_y \end{bmatrix} = \begin{bmatrix} \frac{\partial f}{\partial x} \\ \frac{\partial f}{\partial y} \end{bmatrix} = \lim_{d \to 0} \frac{f(x + d) - f(x - d)}{2d}$$

Like the Harris corner detector, we use partials, meaning we want to calculate the vertical and horizontal edges. We could use *sobel operators*, but instead use 1-D filter masks:

$$\begin{bmatrix} -1 & 0 & 1 \end{bmatrix}, \begin{bmatrix} -1 \\ 0 \\ 1 \end{bmatrix}$$

3. **Calculate Orientations and Magnitudes**: Like many of the operations before, we calculate the magnitude (g) and orientation (θ) of the gradients using the partial derivatives at a point.

$$g = \sqrt{g_x^2 + g_y^2}, \theta = \frac{180}{\pi}(\tan_2^{-1}\left(\frac{g_y}{g_x}\right) \text{mod } \pi)$$

Note that instead of letting the orientation be the full 360 degrees, the mod π limits this to angles between 0 and 180 degrees. This gives us *unsigned* gradients, which were tested to work better.

4. **Create Patches and Blocks**: Remember that HoG is different in the sense that we do not find key points, rather we create features for an area.

   (a) **Create a patch of size** 64×128. Since HoG was made to be operated on human subjects, this patch size captures a standing pedestrian at the right scale. This will ultimately be the space that the feature vector reports on.

   (b) **Divide into Blocks of** 8×8**-sized cells**. These are the cells that will make up the sections of the histograms we make.

5. **Create the block-cells from the blocks**. We slide a 2×2 cell sized window over our entire patch. Each one of these windows will return a histogram of the magnitudes of gradient orientations.

6. **Quantize the gradient orientations of the block into a 9-binned histogram.** We choose 9 bins because they seem to work best.
    - *Quantizing* the magnitudes would mean that if the gradient was in an angle in the middle of two bin thresholds, we would split its magnitude proportionally into each of those bins based on distance.
    - We also *normalize* each block.
7. **Concatenate the histograms into a feature vector.** Recall that we have the 64×128-sized patch. Dividing this into 8×8 cells gives us a 8×16-cell-sized patch. Then, with a 2×2-cell window sliding over the patch, we report $15 * 7$ windows. Each of these windows has $2 * 2 = 4$ cells, and the histograms of each cell have 9 directions. This gives us a feature vector of size:
$$15 * 7 * 4 * 9 = 3780$$

The HoG is nice because it is robust to scaling, translation, and illumination. However, it not robust to orientation. It has coarse spatial sampling due to the large patches it takes. It has fine orientation sampling due to the many histograms. It also is invariant to illumination, due to the normalization of the patch as a whole. All of this makes it particularly well suited to detecting humans in images, specifically pedestrians. The way that HoG would form features over the whole image is by sliding the patch window over the image as a whole, meaning that instead of keypoints, we look at whole image patches.

4.4 Summary

- **Keypoints** of images are useful for identifying characteristics about them. We want them to be robust to possible changes in the image.
- **Invariance** is when a characteristic of an image would not significantly change with certain things.
- **Corners** are good candidate for keypoints for an image since they are invariant to translation, rotation, and illumination.
    - They can be defined as the points where two edges meet.
    - Edge detectors have trouble differentiating corners from flat patches.
    - We can look at the covariance of gradients to determine corner.
    - Two large eigenvalues of the covariance matrix likely indicates a corner.
- **Harris Corner Detector** is an algorithm to detect corners.
    1. **Preprocess with Grayscale and Normalization**
    2. **Generate Gradient Images**
    3. **Construct Gaussian Windows**

4. **Compute Detector Scores**
5. **Adaptive Non-Maximal Suppresion**

- **Scale Invariant Feature Transform (SIFT)** is another algorithm used to find features. SIFT features are robust to uniform scaling, orientation, and illumination.

  1. **Find Keypoints** using DoG pyramids. Localize them using Taylor expansions. Discard weak keypoints and edges.
  2. **Generate Descriptors** by finding the orientation of the keypoint. Generate a orientation histogram for the area around the keypoint, and rotate relative to the orientation of the keypoint.
  3. **Feature Match** using various algorithms.

- **Histogram of Gradients** is a method that looks at patches rather than keypoints. It is invariant to geometric and illumination transforms, with the exception of orientation. It is tailored towards pedestrian detection.

  1. **Preprocess** the image by gamma correcting, then normalizing it.
  2. **Calculate Gradient Images** using *Sobel Operators*.
  3. **Calculate Magnitudes and Orientations** of patches.
  4. **Take Patches and Blocks** from the image. Standard is patch size of 64×128, cells of 8×8, and blocks of 2×2 cells.
  5. **Generate Oriented Histograms** for each block.
  6. **Concatenate Histograms** into a feature vector. This ends up being $15 * 7 * 4 * 9 = 3780$ in the standard case.

5 Image Matching and Stitching

When we want to match images, we usually match certain **features** in the image. When we write detectors for these features, we want to make sure that they are robust to certain changes that can happen in images. These features should be

- Invariant to affine changes
- Invariant to translation
- Invariant to rotation
- Invariant to scale change

Corners help with translation and rotation, but corners are **NOT** invariant to scale change. When we zoom in or out of an image, the feature descriptors of corners may not be able to match each other. This is where the **SIFT** algorithm comes into play.

5.1 Cameras and Projective Geometry

When we have keypoints, something we can do other than matching is transforming. Say that we want to move these keypoints or rotate them or rescale them to stich images together, to normalize images, etc. How do we do this? **Geometric Transformations**. We do these to translate things from 3 Dimensions to 2 Dimensions as well!

- If we have a small hole with light coming through, we can essentially see the light coming through to see the 3 Dimensional world outside as a 2-dimensional inverted image on whatever the light lands on.
- Otherwise known as *Camera Obscura*, this essentially takes the **world coordinates**, projects them through the obscura point into **camera coordinates**, then the light ends up on the plane of the image as **image plane coordinates**.

$$\textbf{World} \begin{bmatrix} X_w \\ Y_w \\ Z_w \end{bmatrix} (R^3) \rightarrow \textbf{Camera} \begin{bmatrix} X_c \\ Y_c \\ Z_c \end{bmatrix} (R^3) \rightarrow \textbf{Image} \begin{bmatrix} x \\ y \end{bmatrix} (R^2)$$

- Think of x, y as how displaced from the focal point something is in the horizontal and vertical directions, and z as the depth of it. **The image plane is perpendicular to the z axis.** Objects that are "deeper" away are going to appear smaller in real life. So, when we get the camera coordinates from this focal point, we can approximate the x, y on the image plane as similar triangles based on a focal point f, giving us the following equations

$$\frac{x}{f} = \frac{X_c}{Z_c}, \frac{y}{f} = \frac{Y_c}{Z_c} \rightarrow x = f\frac{X_c}{Z_c}, y = f\frac{Y_c}{Z_c}$$

  This allows us to introduce homogeneous coordinates, representing them in matrix form for an image. $(x, y, z) \rightarrow (f\frac{x}{z}, f\frac{y}{z}, f)$. We ignore the third coordinate in the image plane since it is constant (The location of the image plane is the focal length).

- The homogeneous coordinates of (x, y) can be represented as a 3D point with a fictitious third coordinate, (x', y', z') We can recover the 2D point by getting $x = \frac{x'}{z'}, y = \frac{y'}{z'}$. This works for any non-zero z. We can translate this to

$$\begin{bmatrix} x' \\ y' \\ z \end{bmatrix} = \begin{bmatrix} f & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}$$

  This allows us to get homogeneous coordinates $\vec{x}$ from inhomogeneous coordinates $\tilde{X}$ with calibration matrix K.

$$\vec{x} = K\tilde{X} \rightarrow \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} f & 0 & 0 \\ 0 & f & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \end{bmatrix}$$

- Note that in this projection, a couple key things are preserved, but some are lost.
  - Points project to points
  - Lines project to lines
  - Planes project to either the whole image or half the image
  - Angles are **NOT** preserved
  - In degenerate cases,
    * Lines through focal point project to a point
    * Planes through focal point project to lines
    * Planes perpendicular to image plane project to a part of the image with horizon

Let's now see how a camera uses this math to actually its coordinates into a digital image.

- A couple formal definitions for the pinhole camera:
  - *Intrinsic Camera Calibration* means we know K.
  - *Principal Axis* is the line from the focal point of the camera perpendicular to the image plane.
  - *Normalized (camera) coordinate system:* The center of the camera is the origin, with the principal axis being the z axis
  - *Principal Point ($p = (p_x, p_y)$):* Where the principal axis intersects the image plane, the origin of the image plane system.
- We can apply the shift of coordinate to the principal point as another matrix, then add it to the calibration matrix K.

$$\begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & p_x \\ 0 & 1 & p_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} f & 0 & 0 \\ 0 & f & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} = \begin{bmatrix} f & 0 & p_x \\ 0 & f & p_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \end{bmatrix}$$

- Next, consider how this becomes pixels. We have to consider m_x, m_y as the pixel density per unit in the horizontal/vertical directions. Some points may not be an exact rectangle, so we have to account for the skew, s. Since in practice, pixel density $m \approx 1$ and skew $s \approx 0$, we get

$$\begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} m_x & s & 0 \\ 0 & m_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} f & 0 & p_x \\ 0 & f & p_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} = \begin{bmatrix} f & s & p_x \\ 0 & mf & p_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \end{bmatrix}$$

  Now, we have the intrinsic calibration matrix inherent to the camera itself.

Now, we must look at how the world translates into the camera. Since they are both in 3-Dimensional space with the same basis vectors, all that we have to do translate and rotate the world's coordinates. We do this with C_w as our translation vector and R as our rotation vector. We have the following:

$$\begin{bmatrix} X_c \\ Y_c \\ Z_c \\ 1 \end{bmatrix} = \begin{bmatrix} R & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} I_3 & -C_w \\ 0 & 1 \end{bmatrix} \begin{bmatrix} X_w \\ Y_w \\ Z_w \\ 1 \end{bmatrix} = \begin{bmatrix} R & -RC_w \\ 0 & 1 \end{bmatrix} \begin{bmatrix} X_w \\ Y_w \\ Z_w \\ 1 \end{bmatrix}$$

Where the translation is a vector $C_w = [C_x, C_y, C_z]^T$, R is a 3×3 rotational transformation matrix, and I_3 is teh identity matrix.

$$C_w = \begin{bmatrix} C_x \\ C_y \\ C_z \end{bmatrix}, R = \begin{bmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{bmatrix}$$

Now, let's take the steps overall mathematically. Given a 3D point X_w in the world coordinate system, to convert it to the image plane, we do the following:

1. Translate the world coordinates to the camera coordinates, given the translation vector $\tilde{C}$.

$$\tilde{X}_c = \tilde{X}_w - \tilde{C} = \begin{bmatrix} I_3 & -\tilde{C} \end{bmatrix} \tilde{X}_w$$

2. Rotate the world coordinate system to alight with the camera coordinates using the rotation matrix R.

$$\tilde{X}_c = R \begin{bmatrix} I_3 & -\tilde{C} \end{bmatrix} \tilde{X}_w$$

3. Apply the intrinsic camera matrix, K to project the camera coordinates to the image plane.

$$\tilde{x} = KR \begin{bmatrix} I_3 & -\tilde{C} \end{bmatrix} \tilde{X}_w$$

In short, we write this out as $\tilde{x} = P\tilde{X}_w$, where

$$P = KR \begin{bmatrix} I_3 & -\tilde{C} \end{bmatrix}$$

known as the camera matrix. K is the intrinsic qualities of the camera, and R, C are extrinsic to the world.

5.2 Homography

Homographies are line-preserving projections from a space to itself. In computer vision, we mostly care about 2-dimensional images.
A **2D Homography** is an *invertible mapping* $h : R^2 \to R^2$ such that any three co-linear points x_1, x_2, x_3 have $h(x_1), h(x_2), h(x_3)$ co-linear as well. We

use 3 dimensions since we are imagining different images as 2-d planes in the 3-d space.

$$\begin{bmatrix} x_2 \\ y_2 \\ z_2 \end{bmatrix} = H \begin{bmatrix} x_1 \\ y_1 \\ z_1 \end{bmatrix} = \begin{bmatrix} h_{00} & h_{01} & h_{02} \\ h_{10} & h_{11} & h_{12} \\ h_{20} & h_{21} & h_{22} \end{bmatrix} \begin{bmatrix} x_1 \\ y_1 \\ z_1 \end{bmatrix}$$

Why are homographies important?

- Homographies allow us to project an image onto another one, letting us stich images together
- We can unwarp (rectify) an image
- We can use homographies to match keypoints quicker (more on this in next section)

Solving for homographies

- Looking at the homography formula, we want to use it to project one image onto another based on a keypoint. Recall that we are imagining these images in multi-dimensional space, which is why we use 3 dimensions. However, remember that in these homogeneous coordinates, we consider $x = \frac{x'}{z'}, y = \frac{y'}{z'}$, and we usually just set $z = 1$ as a mock variable. This would make the homography matrix look like:

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

- When we multiply this out, we get the following equations:

$$x' = \frac{h_{00}x + h_{01}y + h_{02}}{h_{20}x + h_{21}y + h_{22}}, y' = \frac{h_{10}x + h_{11}y + h_{12}}{h_{20}x + h_{21}y + h_{22}}$$

  If we take each equation and multiply both sides by their denominators, we get:

$$x'(h_{20}x + h_{21}y + h_{22}) = h_{00}x + h_{01}y + h_{02}$$

$$y'(h_{20}x + h_{21}y + h_{22}) = h_{10}x + h_{11}y + h_{12}$$

  Then, we just set the right side to zero with subtraction:

$$x'(h_{20}x + h_{21}y + h_{22}) - h_{00}x + h_{01}y + h_{02} = 0$$

$$y'(h_{20}x + h_{21}y + h_{22}) - h_{10}x + h_{11}y + h_{12} = 0$$

  If we rearrange the terms in the equation such that the h_{nn} terms line up, we get:

$$xh_{00} + yh_{01} + 1h_{02} + 0h_{10} + 0h_{11}0h_{12} + -x'xh_{20} - x'yh_{21} - x'h_{22} = 0$$

$$0h_{00} + 0h_{01} + 0h_{02} + xh_{10} + yh_{11}1h_{12} + -y'xh_{20} - y'yh_{21} - y'h_{22} = 0$$

- We can re-represent this system of equations as matrix multiplication!

$$\begin{bmatrix} x & y & 1 & 0 & 0 & 0 & -x'x & -x'y & -x' \\ 0 & 0 & 0 & x & y & 1 & -y'x & -y'y & -y' \end{bmatrix} \begin{bmatrix} h_{00} \\ h_{01} \\ h_{02} \\ h_{10} \\ h_{11} \\ h_{12} \\ h_{20} \\ h_{21} \\ h_{22} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

- This makes it easy to expand to multiple keypoints we match. If we want to find a homography for n keypoints across two images, we create a $2n \times 9$ matrix A and a 0 vector of length $2n$, attempting to solve for h in the formula $Ah = 0$.

$$\begin{bmatrix} x_1 & y_1 & 1 & 0 & 0 & 0 & -x'_1x_1 & -x'_1y_1 & -x'_1 \\ 0 & 0 & 0 & x_1 & y_1 & 1 & -y'_1x_1 & -y'_1y_1 & -y'_1 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ x_n & y_n & 1 & 0 & 0 & 0 & -x'_nx_n & -x'_ny_n & -x'_n \\ 0 & 0 & 0 & x_n & y_n & 1 & -y'_nx_n & -y'_ny_n & -y'_n \end{bmatrix} \begin{bmatrix} h_{00} \\ h_{01} \\ h_{02} \\ h_{10} \\ h_{11} \\ h_{12} \\ h_{20} \\ h_{21} \\ h_{22} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \\ 0 \end{bmatrix}$$

- Solving for this is computationally complex. Firstly, we want to solve for the unit vector of h, since it is only defined up to scale. We can approximate h by minimizing the magnitude of the vector produced by Ah.

$$||Ah||^2 = (Ah)^T Ah = h^T A^T Ah$$

  With this information, we can approximate h as finding the **eigenvector of A^TA** with the smallest eigenvalue!

- We can create an image homography with 4 or more points (as long as no 3 are colinear)

- Since h can only be computed up to scale, we can impose a constraint for some $h_{nn} \in \vec{h}$ for computational speed. However, do not set $h_{22} = h_9 = 0$, as results for this close to 0 fails.

5.3 Random Sample Consensus

Motivation for RANSAC How do we match keypoints between images *robustly*? We do the following

- **Work with individual features:** We can iterate through each of the features and find their nearest neigbhbors, rejecting ambiguous matches with too many similar points.
  - Use 2-NN: get the **SSD** of closest and second closest neighbors, then accept if closest is significantly closer than 2nd closest.
- **Work with ALL the features:** After having some good feature matches, We can generate a *homographies* between the two images, rejecting ones that don't have many feature matches.

Generating homographies, as we saw earlier, can be incredibly expensive. Some key points may be false positives, and could be outliers that massively reduce the speed and quality of our answer. RANSAC is a quick way to fit data, and it generally works like this:

1. Given a dataset of size n, we want to generate a model M that fits it well.
2. Select d random data points from the dataset, and generate a model M' to fit the dataset.
3. Test the model M' on the points in the dataset, and if the error on a point is less than some threshold t, mark it as an **inlier**.
4. Keep the model M', noting the amount of inliers it has.
5. Repeat steps 2-4, replacing M' if we find a new model that has more inliers than it **or** has the same amount of inliers but less error.
6. Repeat step 5 until one of the following conditions have been met:
   (a) We iterate through the process k times
   (b) We find a model M' that has a significant amount of inliers, usually some ratio of n (80%)
7. Return M' as M, the best model.

This is a massive speed up over computing a model over all the data, especially as the size increases. We can translate this RANSAC algorithm to generating a homography for the images by doing the following: Given two images with feature pairs (p, p')

1. Select 4 feature pairs at random.
2. Compute the exact homography H for the 4 pairs. (maps $p \to p'$)
3. Compute inliers, where $||p' - Hp|| < t$, where t is the error threshold
4. Repeat the above steps (k times), keeping the ***Largest*** *set of inliers*. We are not keeping the homography, but rather the set of feature pairs.

5. Re-compute the least-squares homography H as an estimate using aforementioned largest set of inliers.

Let's do some quick math on the robustness of the algorithm. Say we have g as the ratio of good inlier feature pairs in our data. We want to find n good pairs (we *need* 4 for a homography). With this, we can compute the odds of us *not* finding a good set of inliers after k iterations as a simple exponential formula:

$$P(\text{good inlier set}) = (1 - g^n)^k$$

Basically this means the following for the effects of the parameters on the robustness of our data:

- **Number of necessary pairs(4):** In general RANSAC, when this is higher, it makes us less likely to get a good model over k iterations.
- **Percentage of Inliers:** This is the base of the exponential.
- **Number of Iterations:** This is a "good" exponential. The more we iterate, the more likely we will be to get a good set of inliers.

5.4 Summary

- **Projective Geometry** is a form of geometry that does not focus on the inherent distance between points, but rather, their colinearity.
- **Cameras** use projections to convert coordinates in the real world to camera coordinates, then to image planes.
- This works by translating and rotating the real world into the camera's coordinates, then transforming it with a homography to the image plane.
- **Homographies** are line-preserving projections from a space to itself. Answers to homographies can be roughly estimated with *linear least squares*.
- We can stitch images by warping matching keypoints onto each other with homographies.
- **Random Sample Consensus** is a way to speed up model creation and point validation. Here's how we do it to generate homographies from feature matches:
    1. Sample 4 random feature pairs.
    2. Compute the homography H
    3. Compute inliers
    4. Repeat, keeping the ***Largest*** *set of inliers.*
    5. Re-compute least-squares homography H using largest set of inliers.

6 Optical Flow

6.1 Definitions and Assumptions

Here, we move from images to videos. Since a video is a collection of adjacent images, we must be able to gleam information from it. We want to find objects in the video moving in the 3D plane. This is the idea of the **Motion Field:** a projection of 3D velocity vectors onto the 2D image plane. We can estimate this using something called the **Optical Flow** of an image, the observed 2D displacements in the intensity patterns of the image. First, let's list some assumptions:

- **Brightness consistency**: The intensities of an object in an image do not change between consecutive images.
- **Temporal regularity**: Time between frames is short enough for us to find motion using differentials.
- **Spatial consistency**: Neighboring pixels have similar motion.

We need these assumptions, as without them, here are some examples of what can go wrong:

- Imagine a smooth sphere rotating. The image projection itself would not change, but the motion field of the objects would not.
- Imagine the same sphere statically, but the illumination itself is moving around the object. The motion field would not change, but the optical flow field would due to the intensities changing.

With that out of the way, let's list the possibilities of when optical flow and motion fields can change:

- Camera is still, objects/scene is moving
- Camera is moving, objects/scene are still
- Both camera and scene are moving

This leads us to some Motion Analysis Problems:

- **Correspondence Problem:** Tracking elements (*objects*) across frames (*the video*).
- **Reconstruction Problem:** Given the corresponding elements and camera parameters, how can we reconstruct the 3D motion field? Think of this as gleaming information about moving objects from the optical flow, like when an object moves behind another in a video.
- **Segmentation Problem:** Finding the regions of the image plane that correspond to different moving objects.

Let's go into more depth how the motion field into optical flow. Objects in 3D moving around have velocity vectors. When moving this to a camera, the velocities are the *relative motion* between the camera and the world. This motion field is the projection of these 3D velocities onto the 2D image plane. The optical flow is the observed 2D displacements of intensity patterns in the image, meaning we approximate these vectors with these intensity displacements.
Let's talk about an issue known as the **Aperture Problem**. Imagine we are looking at a barber pole through a small hole. The line appears to go upwards, but it could also be going right. We cannot find where each point on the line has exactly moved from just the measurements from the whole. This can cause the percieved and actual motion to be inconsistent.

6.2 Brightness Constancy Equation Math

With the assumptions, we can form the brightness constancy equation, which can measure the changes in an image. Let us consider the image I and time t.

- Image intensity at a time t is

$$I(x, y, t)$$

- If the pixel were to move (displaced) over time, we would find the pixel at time $t + dt$ as:

$$I(x + dx, y + dy, t + dt)$$

.

- We use Taylor Series Expansion with partial derivatves to approximate this:

$$I(x + dx, y + dy, t + dt) \approx I(x, y, t) + \frac{\partial I}{\partial x}dx + \frac{\partial I}{\partial y}dy + \frac{\partial I}{\partial t}dt$$

- With the *Brightness Constancy* assumption, we assume

$$I(x, y, t) = I(x + dx, y + dy, t + dt)$$

- Therefore, we can cancel out the intensities on both sides to find that

$$\frac{\partial I}{\partial x}dx + \frac{\partial I}{\partial y}dy + \frac{\partial I}{\partial t}dt = 0$$

- Since we want to find how the image changes over time, we take the derivative with regards to time:

$$\frac{d}{dt}\left(\frac{\partial I}{\partial x}dx + \frac{\partial I}{\partial y}dy + \frac{\partial I}{\partial t}dt\right) = \frac{\partial I}{\partial x}\frac{dx}{dt} + \frac{\partial I}{\partial y}\frac{dy}{dt} + \frac{\partial I}{\partial t} = 0$$

- Let's think about what we can get. We can get the partial derivatives of the image in regards to x, y as the **Frame Spatial Gradient**, ∇I. We can also get the partial derivative of time, I_t. What's left are the partials of x, y in regards to time. This is what becomes our optical flow, (d). We can represent these as vectors and scalars:

$$\nabla I = \begin{bmatrix} \frac{\partial I}{\partial x} \\ \frac{\partial I}{\partial y} \end{bmatrix}, d = \begin{bmatrix} \frac{dx}{dt} \\ \frac{dy}{dt} \end{bmatrix}, I_t = \frac{\partial I}{\partial t}$$

- Let's convert the equation to solve for optical flow:

$$\frac{\partial I}{\partial x}\frac{dx}{dt} + \frac{\partial I}{\partial y}\frac{dy}{dt} + \frac{\partial I}{\partial t} \rightarrow I_x u + I_y v + I_t = 0$$

  where we note the partials of the image as I_x, I_y, I_t, and $u.v$ as the partials of x, y in regards to time. This essentially becomes a line equation where

$$\text{Given } u = \frac{dx}{dt}, v = \frac{dy}{dt} \rightarrow v = -\frac{I_x}{I_y}u - \frac{I_t}{I_y}$$

6.3 Lukas-Kanade flow

With the above formula, we can get the optical flow of a pixel. However, the hard part is solving this system of equations with only one input. We may not have a correct output.

- When we use the *Spatial consistency* assumption, we can use the neighbors of a pixel, since we assume the flow field is smooth locally.
- We use a 5×5 window around our target pixel, denoting the pixels in the window as p_i
- We can reshape the brightness constancy equation to work with these multiple pixels.

$$I_t(p_i) + \nabla I(p_i) \cdot d = 0 \rightarrow \nabla I(p_i) \cdot d = -I_t(p_i)$$

  This can also be re framed as a matrix equation, $Ad = -b$ where:

$$\begin{bmatrix} I_x(p_1) & I_y(p_1) \\ \vdots & \vdots \\ I_x(p_{25}) & I_y(p_{25}) \end{bmatrix} \begin{bmatrix} u \\ v \end{bmatrix} = \begin{bmatrix} I_t(p_1) \\ \vdots \\ I_t(p_{25}) \end{bmatrix}$$

- This means to solve this equation, we want to minimize $||Ad - b||^2$. We can do this by solving the least squares problem,

$$Ad = b \rightarrow (A^T A)d = A^T(b)$$

  With this, we can rewrite the equation as:

$$\begin{bmatrix} \Sigma I_x I_x & \Sigma I_x I_y \\ \Sigma I_x I_y & \Sigma I_y I_y \end{bmatrix} \begin{bmatrix} u \\ v \end{bmatrix} = \begin{bmatrix} \Sigma I_x I_t \\ \Sigma I_x I_t \end{bmatrix}$$

- Note that A^TA is like the covariance matrix used in Harris corner detection, but as a summation of the region of points.
- Looking at the Harris corner math again, $det(A^TA) = \Pi\lambda_i = 0$, one or more eigenvalues are 0.
  - If one eigenvalue is 0, then it is not a corner, but an edge.
  - If both are 0, then it is a homogenous patch.

  This means that we should be mainly tracking corner points between images.
- The last part of the Lukas-Kanade Algorithm is **iterative refinement** to remove outliers:
  1. Estimate velocity of pixel using Lucak-Kanade equation
  2. Warp $I(t-1)$ to $I(t)$ using the estimated flow field
  3. Repeat until convergence: until the change in displacement is small.
- For things with large areas of motion, we can reduce the resolution to find more prominent features. We can do **Coarse-to-Fine** optical flow estimation by running the L-K Algorithm on the most downscaled image, warping and upsampling, then repeating it for every resolution.

7 Classification

One of the things we can do in image processing is classifying images. Is something a human or not? Is something a bird? a plane? We call this Classification, and have many algorithms to help us do so.

7.1 Support Vector Machines

A **Support Vector Machine (SVM)** is a supervised learning model for binary classification.

- Imagine a 2-Dimensional grid with points scattered across it. Some of the points are red, and some are blue. The goal of a SVM is to draw a *hyperplane (decision boundary)* that separates the data such that all points above the plane are red, and all points below are blue.
- Most of the time, there can be many lines that we make that separate the data. How can we say one line is better than the other?
- The closer points get to the line, the worse the line is, since it is less likely to be generalized. We call the distance from the points to our classifying line as the **margin**. Maximizing the margin is the goal of the SVM.

- Some of the math: Given a point with feature vector $\vec{x}$, the vector perpendicular to our hyperplane, $\vec{w}$, and a bias for the hyperplane, b, we can find the project the point to boundary as

  $$\vec{w} \cdot \vec{x} + b$$

  With this, we can get the margin, d, as the distance from this projected point to the real point as the following:

  $$d = \frac{2}{||\vec{w}||^2}$$

  which is equivalent to minimizing a loss function $L(w)$:

  $$L(w) = \frac{||\vec{w}||^2}{2}$$

  As long as it is subject to correctly classifying the objects. This allows us to have a constrained optimization problem.

- If the problem is not linearly separable, we want to just apply some penalty based on how far off the point is from the correct direction of the line.

- If the boundary is not linear, we can transform the data into higher dimensional space like we did with the least squares regression.

.

8 Bag of Visual Words.

Some of the things we can do with images:

- Image Classification
- Image Tagging
- Object Detection
- Activity Recognition
- Image Parsing
- Image Description

Bag of Features/Visual Words is a pipeline to detect different types of objects across images.

- Imagine an image. There are important pieces in the image, such as the nose and eyes of person vs. the wheels and windows of a car.

- We call it a Bag of Words because we represent the document as an orderless frequency of words from a dictionary.

  - For sentences, we would match the frequencies of words.
  - We can tell what a sentence is about somewhat by looking at the word frequencies. Math sentences have lots of numbers, animal sentences have lots of animals.
  - We create term frequency vectors, then have a linear classifier based on the weight of the words in the dictionary.

- The Bag of Visual Words is motivated by this. Instead of being words, we use the feature vectors of images as our "visual words"

The basics of the BoVW algorithm are as follows:

1. **Extract Local Features**
2. **Learn the "Visual Vocabulary"**
3. **Quantize the local features to the vocabulary**
4. **Represent images as frequencies of the "visual words"**

Let's start with Extracting Local Features.

- These features can be any of the above stuff that we learned!
- We can sample patches across the image and report them (Histogram of Gradients)
- We can also look at patches around the keypoints (Corner detection)
- With these, we would send all of the features of all of the images into the next step, finding the visual vocabulary.

How do we learn the visual vocabulary?

- We do this by doing **Clustering**, an *unsupervised* learning algorithm.
- Clustering is essentially grouping data points based on similarity (usually euclidian distance).
- Imagine we have some features for fruits, then plot a bunch of fruits over the feature space. The same fruits are likely to be close to each other, defining our clusters.
- We do this with **K-Means Clustering**, which works as follows:
  1. Given the dataset, we want to form K clusters. Randomly generate K clusters over the feature space.
  2. Assign each point in the dataset to its nearest cluster.
  3. Remake the clusters by setting the new value to the average of the points determined to be part of the cluster.

4. Repeat until satisfied.

In practice, since this is not a convex optimization problem, we may not reach a viable solution. We usually intialize this randomly and run it several times, taking the best result.

- We graph the sum of squared errors over different K and use the elbow point as the optimal K.

Next, we quantize the local features to the cluster centroids. and represent them as histograms of the visual words.

- If a feature that belonged to a centroid occurred in an image, we would increment the count of that "visual word" in the histogram of the image.

Finally, we would train some SVM on the resulting histograms.