CS 365 Computer Vision, Spring 2016

Dr. Bruce A. Maxwell
Department of Computer Science
Colby College

Course Description
Investigates designing computer programs that extract information from digital images. Major topics include image formation and acquisition, gray-scale and color image processing, image filters, feature detection, texture, object segmentation, classification, recognition, and motion estimation. Students are introduced to classic and contemporary vision techniques with examples for homework and programming assignments drawn from biological and medical imaging, robotics, augmented reality, and digital photography. Students will develop small and medium-scale vision systems to solve practical problems and possibly assist in active research projects at Colby.

Prerequisites: CS 251, linear algebra is recommended.

We live in a society exquisitely dependent on science and technology, in which hardly anyone knows anything about science and technology.

Carl Sagan

Desired Course Outcomes
A. Students understand the fundamentals of image formation and image acquisition.
B. Students understand and can implement image processing routines used in computer vision algorithms, such as filtering and morphological operations.
C. Students can discuss and implement algorithms for feature detection, segmentation, classification, and tracking.
D. Students work in a group to design and develop a medium-sized image analysis and computer vision application.
E. Students present algorithms and results in an organized and competent manner, both written and orally.

This material is copyrighted by the author. Individuals are free to use this material for their own educational, non-commercial purposes. Distribution or copying of this material for commercial or for-profit purposes without the prior written consent of the copyright owner is a violation of copyright and subject to fines and penalties.
1 Introduction to Computer Vision

Computer vision is the study of how to automatically extract knowledge from imagery. Many within computer vision would broadly define imagery as anything from 1-D to 4-D data captured by a sensor. Examples of data with different space and time dimensions include the following.

- 1-D: line scan camera
- 2-D: line scan camera over time
- 2-D: image
- 3-D: images over time (video)
- 3-D: image plus depth
- 4-D: video plus depth

While most computer vision is concerned with images created by sensors that capture information from the visible spectrum of electromagnetic radiation (380nm-780nm wavelength), the techniques and algorithms from computer vision also work on depth images, MRI images, infrared and ultraviolet images, X-rays, CT-scans, millimeter scans, and other 2D or 3D data such as might be captured by a capacitance sensor on a mobile phone. The key similarity of all of the potential sources of data is that they tend to exhibit coherence in space and time, and there are regularities of structure—due to regularities in the structure of the thing being imaged—that we can attempt to identify, recognize, and classify. In short, computer vision is the study of how to extract knowledge from data that has a coherent multi-dimensional structure (dimension ≥ 2).

Image data is generally organized by pixels. A pixel is a small spatial piece of an image. Pixels are usually organized in a regular rectangular grid, with the size of an image defined by the number of rows and the number of columns. The value of a pixel is one or more measurements that are captured at roughly the same time and the same spatial area.

The data at each pixel can be anything from one measurement (greyscale camera) to hundreds (hyperspectral camera). In general, each individual measurement at a pixel corresponds to a different range of frequencies. For example, in a standard color image each pixel contains three values, named R, G, B, that correspond roughly to measurements of the amount of low-frequency visible light (red), mid-frequency visible light (green), and high frequency visible light (blue).

Computer vision is a relatively new field that began in the early 60’s.

- 1960’s: first work with images, some scanned point by point by hand.
- 1970’s: filtering, segmentation, edge and line detection, first physics-based analysis (Horn)
- 1980’s: stereo, motion, more advanced physics-based vision
- 1990’s: cheap cameras, real time vision systems, appearance-based vision, structure from motion, stereo, content-based retrieval, object recognition
- 2000’s: object recognition, learning, SLAM, interactive vision systems, tracking, surveillance, image manipulation
- 2010’s: vehicle navigation and robotics, object recognition, biometrics, face recognition
1.1 Human Visual System

Humans have a very effective, working visual system. Most of what we do is unconscious processing, and it occurs in less than a few milliseconds. While scientists have been studying the human visual system for many years, we still know very little about the actual algorithms that take place in our brains.

Physical aspects

- Lens: gathers light and focuses part of the world into a sharp image.
- Iris: opens and closes to increase or reduce the amount of light entering the eye.
- Retina: sensor, has at least 4-5 different kinds of sensors on it.
- Color/Greyscale: each sensor has a different response to incoming energy, and the collection of responses generates our perception of color.
- Optical nerve: carries data from the eye to the brain.

While the physical aspects of the human visual system are fairly well known–because we can measure things like cell responses, and because most of it is visible and mechanical in nature–the information processing aspects are still relatively unknown.

- There are at least two pathways for visual information. One pathway deals mostly with luminance information, while the other incorporates both luminance and chromaticity.
- The base area of the visual cortex (V1) tends to be viewed as a stack of layers that engage in massive parallel processing of the visual signals. By measuring the response of cells in the V1 layers, we know that individual cells are sensitive to specific features of an image, such as lines or dots with particular orientations or locations in the image.
- As information flows through the visual cortex, much of the spatial coherence is lost, and larger semantic objects develop.

The computer vision community used to (a long time ago) believe that algorithms needed to have a biological basis in order to be valid. But computers are not human brains, and the algorithms that are appropriate for computers to implement–and which can successfully accomplish many tasks in computer vision–are probably not the algorithms used by the human brain. In fact, there are some algorithms that would be difficult for the human brain to execute, but which are well-suited to computers.
2 Imaging

Hardware

The simplest camera is a pinhole camera.

- The entire world is in focus, just upside down and inverted.
- Conceptually, only one ray of light hits each sensing point.
- Not a lot of light gets in.

A typical camera has a lens, which gathers many rays of light emanating from an object and focuses them onto a point.

- Only parts of the world can be in focus, and the more light is gathered, the smaller the depth of field.
- The world is still upside down and inverted.
- Most lenses have an iris that can open or close to let in more or less light.
- As the iris gets smaller, the lens becomes more like a pinhole, and more of the world is in focus.

While the physical reality is that the image is upside down and backwards on the sensor, we can make the mathematical reality a bit easier by thinking about the geometry in terms of the image plane.

All cameras must have a sensor:

- Silver plates
- Film
- Photomultiplier tubes
- greyscale CCD
- single CCD color cameras (Bayer, Foveon, multi-spectral filters)
- multi-CCD color cameras
- CMOS sensors

2.1 CMOS v. CCD

In a CCD sensor, each row consists of a series of photon buckets (photon-sensitive silicon surrounded by a dielectric). Each row has a single A/D converter that can serially read out the sequence of buckets on that
row. The measurements on a row have high-uniformity, and the entire pixel area is dedicated to collecting light.

In a CMOS sensor, each pixel has its own A/D converter and gain circuitry. This reduces both the uniformity of the measurements and the area dedicated to light collection, but speeds up processing and reduces power consumption.

In practice, both sensor types are finding use in consumer and scientific applications, and the cost difference between them for any particular application with similar performance is minimal. CCDs have a slight edge in image quality, while CMOS sensors have a slight edge in speed and power consumption.

Sensing Color

Stuff is visible in the world because photons bounce off it and onto our sensors (eyes or cameras). Every photon has a characteristic frequency/wavelength that determines whether a sensor reacts when the photon hits it. To sense in color, our sensor has to have components that react differently to different portions of the EM spectrum. Combining readings from sensors with different sensitivity allows us to sense color. There are a number of different methods of achieving this.

1. Macro-filters: using a single broad spectrum sensor, put a series of filters in front of it and take multiple images of the scene.

2. Multiple sensors: split the incoming light into separate coherent beams and direct each beam through a different filter and then onto a broad spectrum sensor (e.g. 3-CCD cameras).

3. Micro-filters: Use a pattern of filters over individual sensor elements and interpolate the readings to get multiple sensor values at each location (e.g. Bayer pattern CCD).

4. Multi-layer CCDs: Divide each sensor element by depth and read out the energy collected at each level. Higher frequency/shorter wavelength photons will go farther into the sensor element (e.g. Foveon).

5. Prism: Using a slit lens, put the linear beam through a prism to spread out the signal by frequency onto a 2-D sensor. Each pixel element reads a different frequency range. Physically scanning the camera across the scene produces a 2D image.

The color that we sense from an object is the result of a long sequence of physical processes.

- A photon is produced by some physical event (e.g. a thermonuclear explosion in the sun)
- Scattering: the photon reacts with the atmosphere and may be scattered by collisions with matter, producing a different photon.
- Body reflection: the photon hits an object, penetrates the object and interacts with a pigment molecule, producing a different photon at a frequency that depends upon both the incoming photon and the pigment.
- Surface reflection: the photon may be reflected at the object’s surface because of the change in density.
- Transmission: the photon may pass through the surface, but its course is likely changed due to the density changes.
- Fluorescence: the photon hits an object, is absorbed by molecules in the object, and a photon is re-emitted at a different frequency.
2.2 Progressive scan v. Interlaced Scan

Many older cameras do not capture the entire frame at once. Instead, they first capture the even rows, then the odd rows. The capture process runs at twice the frame rate, which is the rate at which entire images are sent to the computer. In a progressive scan camera, the entire frame is captured at once in a single scan through the rows. Progressive scan cameras produce higher quality imagery, especially for moving objects, because there is no time aliasing as can exist between the fields of an interlaced scan image.

A line scan camera only has a single row of sensors. To capture 2D images, a line scan camera can either be swept across a scene, or the scene can be moved across the sensor. Often, line scan cameras are used for quality control monitoring on manufacturing lines for products such as paper.

2.3 Digital v. Analog world

All cameras are trying to capture images of a continuous world, at least until you reach the level of photons.

However, no sensor is perfectly continuous. Cameras have a spatial resolution, a time resolution, and a spectral resolution. When an object is too small, moving too fast, or its color spectrum is too complex, the sensor cannot accurately capture the scene.

Aliasing: When a system samples a frequency that is more than half the sampling frequency, then aliasing occurs. The result is that the sensor sees a ghost frequency that does not actually exist.

Aliasing is worse than not being able to see the signal at all; the sensor sees a signal that does not actually exist.

There are three types of aliasing that can occur:

- Spatial aliasing: the spatial frequency of the object, as projected onto the sensor, is higher than half the spatial frequency of the sensor elements. Example: a fine pinstripe suit will often flicker on TV because the pinstripes are at a higher frequency than the sensor can handle.

- Spectral aliasing: the spectral capability of a camera is determined by the sensitivity of the sensor to particular wavelengths and the number of different filters used. For some objects with complex spectra (e.g. spikes), two objects can appear the same color under one illumination, but a different color under other illumination conditions.

- Time aliasing: Cameras capture images at a regular frequency. When an object is moving fast enough, its motion in the image appears different than its actual motion. Example: a wagon wheel in a western will start out turning forwards, slow down, and start turning backwards while the wagon keeps on heading forwards.

2.4 Basic Imaging Geometry

The important parameters of imaging geometry are the focal length and the distance of the object from the camera.

Focal length: fixed physical constant for a particular lens configuration

- Longer focal lengths show you less of the world (zoom in / telephoto)
• Shorter focal lengths show you more of the world (zoom out / fish-eye)

Thin lens law:

\[ \frac{1}{Z_i} + \frac{1}{Z_o} = \frac{1}{f} \]  

(1)

\( Z_i \) is the distance between the lens and the sensor at which an object at distance \( Z_o \) is in focus give the focal length \( f \).

Distance from the camera: things get smaller as they get further away.

• Something twice as far away is half the size in the image.

• Something twice as far away and twice as big, looks exactly the same.

Perspective projection:

\[ x = -\frac{fX}{Z} \]  

(2)

The projected location of a point \( x \) is the original location of the point in camera coordinates \( X \), multiplied by the focal length \( f \) and divided by the distance from the lens \( Z \). As the focal length gets longer, things get larger in the image (zoom).

Perspective projection is how the 3D world is mapped onto the 2D world. Obviously, information is lost in the process, which is why depth illusions are possible. Perspective geometry is important in a number of applications.

• Camera calibration: calculating the intrinsic (focal length, optical center, pixel scaling) and extrinsic camera parameters (location and orientation).

• Geometric analysis: parallel lines are not preserved under perspective projection unless they are parallel to the image plane.

• Image stitching/mosaicing: perspective distortion must be adjusted to stitch together different images properly.

• Motion analysis: by tracking points in the world as the camera we can build geometric models of the world.
3 Color Spaces

Once we have an image in a computer, we have to represent the image in a way that enables us to process it. One issue is how to describe color.

RGB: the raw color space that comes out of the camera represents the three sensor values at each pixel. RGB is a linear color space, and a signal that is twice as strong will have twice the response from the sensors. Because RGB represents a frequency measurement, in many cases we can think of it as the multiplication of two signals: the illumination color and the body reflection color. The multiplicative model does not hold for surface reflection or fluorescence, but much of what we see is body reflection.

HSI: a different color space that is useful for picking colors and describing the attributes of color. HSI is a cylindrical color space with three axes: hue, saturation, and intensity. HSI is a nonlinear transformation of RGB.

- Hue is a circular color space that goes from blue to red and wraps. Spectral opposites are on opposite sides of the color wheel.
- Saturation is the distance out from the center of the color wheel. More saturated colors are on the outer edge of the wheel, and the central axis of the cylinder goes from black to white.
- Intensity is the distance along the central axis from black to white. It measures average energy output of the three color bands.

YIQ: a linear transformation of RGB that is used by the standard broadcast signal.

- Y is a weighted average of the RGB signals that mimics human perception (rods).
- I is approximately Red - Cyan.
- Q is approximately Magenta - Green.

\[
\begin{bmatrix}
Y \\
I \\
Q
\end{bmatrix} =
\begin{bmatrix}
0.30 & 0.59 & 0.11 \\
0.60 & -0.28 & -0.32 \\
0.21 & -0.52 & 0.31
\end{bmatrix}
\begin{bmatrix}
R \\
G \\
B
\end{bmatrix}
\]

YUV: a linear transformation of RGB used by some digital video products, including many analog capture cards. Note the conservation of the weights for each channel.

- Y is identical to the Y in YIQ.
- U is approximately Yellow - Blue.
- V is approximately Red - Cyan.

\[
\begin{bmatrix}
Y \\
U \\
V
\end{bmatrix} =
\begin{bmatrix}
0.30 & 0.59 & 0.11 \\
-0.30 & -0.59 & 0.38 \\
0.58 & -0.59 & -0.11
\end{bmatrix}
\begin{bmatrix}
R \\
G \\
B
\end{bmatrix}
\]

XYZ: CIE standard color space, used for comparing colors and for reporting experiments with color specifications. The definition of the XYZ color space is tied to human perception. Each channel is generally calculated as a linear transformation of RGB, but each channel carries color information, unlike YUV and YIQ. Conceptually, the XYZ color space represents the human response functions modified to avoid any negative components (the sensors in our eye actually have inhibitory properties in some frequencies).
In addition, there are two CIE color spaces that are intended to have perceptual meaning:

1. CIE-Lab: intended for simulating the perception of light reflecting off objects. The L channel is an intensity measure, while the ab channels are chromaticity. Small distances in Lab space are perceptually meaningful.

2. CIE-Luv: intended for simulating the perception of light generating by a device (e.g. computer screen). The L channel is intensity, and the uv channels are chromaticity. Like Lab, small distances in Luv space are perceptually meaningful.

Chromaticity

Chromaticity spaces attempt to represent a color independent of the intensity. Common chromaticity spaces include:

- rg: the R and G channels divided by the sum \((R + G + B)\).
- xy: the X and Y channels divided by the sum \((X + Y + Z)\)
- uv, ab: the uv and ab channels from the CIE-Lab and CIE-Luv spaces
- H: the hue measurement from the HSI color space (sometimes combined with saturation.
- UV: the UV channels of the YUV color space, since they are often provided directly by the video hardware.

Chromaticity spaces can remove much of the variation due to changing intensity levels, often caused by geometry or shadows.

- Highlights will generally cause a change chromaticity as surface reflection is the color of the light source.
- Shadows can cause slight changes in chromaticity when there are strong ambient illuminants.
- Interreflections–light bouncing off nearby colored surfaces–can cause significant changes in chromaticity.

4 Thresholding

It is often the case that we want to separate an image into foreground and background elements. In general, we can think of this as separating the image into two classes according to some criterion. The resulting image is a binary image, as each pixel contains one bit of information.

There are many ways of classifying pixels. One approach is to set thresholds on certain pixel characteristics and put a pixel in class A if it passes the thresholds and in class B if it does not.

- Single threshold: pixels that are brighter than a certain value are foreground, otherwise background.
- Multi-band thresholds: generate a threshold for each color channel, and foreground pixels must pass each threshold.
- Band-pass thresholds: have two thresholds per channel–hi and lo–and foreground pixels must be in-between.
• Histogram thresholds: specify which color values are in and which are out using a binary histogram.

• Functional thresholds: create thresholds that work on the result of functions applied to the pixel color values. For example, their ratio.

4.1 Picking Thresholds

There are a number of methods for picking thresholds. The best methods don’t involve manually selecting the thresholds.

• Manual selection: examine the values of the background and foreground pixels and select a threshold that provides a good solution. If you have a GUI interface, a slider lets a person be reasonably efficient at the process.
  – Generate a histogram of the pixels in the image and use it to select appropriate thresholds.
  – Easy thresholding situations will be bi-modal (have two peaks).
  – High gradient pixels (pixels where change is going on) will tend to be mixes of foreground and background values.

• Clustering: Let the computer automatically divide the pixels into two classes according to some criterion. The ISODATA algorithm is k-means clustering for $k = 2$.
  1. Initialize the cluster means to two initial locations (e.g. .25 and .75 of the range of the data)
  2. Classify all pixels as belonging to one of the two classes based on distance in the space.
  3. Recalculate a new mean for each class.
  4. Repeat from step 2 until the process converges.

• Training: Give the computer examples of foreground and background pixels and let it discover the best thresholds.
  – Build histograms of the background or foreground and use histogram-based thresholding.
  – Search various thresholds to discover which values provide the best separation.
  – Linear search is useful: hold all thresholds but one constant while varying the one threshold to achieve the optimal separation; then rotate through the rest of the thresholds using the same process.
4.2 Reflection Models

Understanding the appearance of surfaces is an important part of making use of color to identify objects or targets and to properly understand a scene. Most light captured by a camera is the result of illumination falling on a surface, being modified by that surface, and then reflecting to the camera sensor. Therefore, the signal measured by the camera is a result of a physical process involving the properties of light and the properties of the surface.

Light consists of photons that have properties of frequency, wavelength, and polarity. The frequency and wavelength describe the energy of the light, and higher frequency (smaller wavelength) visible light is in the blue spectrum, middle frequency visible light is in the green part of the spectrum, and lower frequency visible light is in the red part of the spectrum. The polarity of a photon describes the orientation of how it moves through space. Most light is randomly polarized, but certain situations create light with a non-uniform distribution that can be used, for example, in polarized sunglasses to reduce highlights in a scene.

4.2.1 Dichromatic Reflection Model

Light interacts with materials in a number of different ways. One method of interaction is surface reflection. Surface reflection occurs any time there is a difference in density at a boundary, such as when light moves from air into an object. Surface reflection is a mirror-like reflection phenomenon that depends upon the local geometry of the surface, which includes micro-facets as might appear on rough surfaces.

Materials like metals, that are conductive, tend to exhibit primarily surface reflection. Any photon that breaks through the surface tends to quickly be absorbed by the material and get converted into energy (heat). Therefore, metals exhibit primarily a mirror-like reflection, though a rough metal surface can look more diffuse. Because of the directed nature of the reflection, the amount of surface reflection measured by a camera sensor will depend not only on the incoming energy and angle, but also the viewing angle.

There are many materials in our environment, however, that fall in the category of inhomogeneous dielectrics. They are inhomogeneous (not all the same) because they consist of pigment particles in a substrate. They are dielectrics because they are non-conductive. Inhomogeneous dielectrics include paint, plastic, wood, skin, most cloth, paper, and many other materials. In most cases, the substrate material is clear in order to allow the pigment particles to control of the perceived color of the surface. Consider, for example, the difficulty of mixing paints with a yellow substrate.

Because of the inhomogeneous nature of these materials, there are two different types of reflection that occur. First, there is surface reflection, as will occur at any surface boundary with differing densities. The surface reflection tends to generate reflected light that is the same color as the light source, since the substrate material tends to be clear and exhibits neutral interface reflection [NIR]. Any light that penetrates the surface, however, is then likely to hit a pigment particle. While some of that energy may be converted into heat, what is more likely to occur is that the pigment particle with re-emit a photon of a particular color in a random direction, creating the phenomenon of body reflection, or diffuse reflection. Some of those re-emitted photons will exit the surface and strike the camera sensor. Note that, since the re-emitted photon is sent out in a random direction, the intensity of the surface will depend largely on the amount of energy striking the surface, not on the viewing angle. This observation is the basis of the Lambertian reflection model, which says that the reflected light is proportional to the color and intensity of the light source, the color and intensity of the pigment particles, and the cosine of the angle between the surface normal and the light source direction.
The combination of the body reflection and surface reflection is called the dichromatic reflection model. It is dichromatic because, if the pigment particles are a different color than the illumination, then the surface reflection and body reflection will be different colors. The body reflection will be a combination of the light and material colors, while the surface reflection will match the light source in the case of an NIR material.

The body reflection, in isolation, depends on the color and intensity of the illumination and the color and intensity of the pigment particles. When there is no light, the body reflection goes to black. As the illumination increases, the reflected color moves along a vector away from the origin, creating what we can call the body reflection cylinder (no material color is perfectly uniform) in the RGB linear color space.

The surface reflection also forms a rough cylinder in RGB space, but it does not occur in isolation from the body reflection. Instead, we can visualize the surface reflection as starting at a point on the body cylinder (no surface reflection) and then heading towards the light source color in the RGB cube as the surface reflection increases.

An important result of these dichromatic reflection model observations is that all of the colors an object will appear are the result of the addition of two colors that can be represented as vectors in the RGB color cube. The weighted sum of two 3D vectors forms a planar parallelogram. Therefore, all of the reflected colors of a uniformly colored inhomogeneous will fall within the plane defined by the body reflection and surface reflection vectors. What’s more, surface reflection cannot occur in the bottom 50% of the body reflection cylinder, so the appearance of a surface with both body and surface reflection will tend to form a skewed T, with the top bar of the T being the body reflection that goes from the origin to the fully lit body reflection color and the vertical bar of the T being the surface reflection, which goes from some point on the body cylinder towards the light source color.

### 4.2.2 Bi-Illuminant Dichromatic Reflection Model

The real world is not quite a simple as the dichromatic reflection model would have you believe, however, because in the real world the shadows do not go to zero. There is reflected and ambient light all around us that provides light in the shadows. It is useful to separate out direct light, the light coming directly from a light source, and ambient light, which is all other light in the scene. For example, in an open field on a sunny day, the sun is the direct light source and the blue sky is the ambient light source. Therefore, shadows tend to be more blue than lit areas, because they are lit primarily by the blue sky.

The primary way that the BIDR model affects appearance is by moving the body cylinders away from the origin. The dark point on each cylinder is a multiplication (in color space) of the body color and the ambient light color. But the direction of the cylinder as the direct light source gets stronger is defined by the multiplication of the body color and the direct light color. Therefore, the cylinder is offset from the origin, but generally in a way such that it no longer points at the origin, since the ambient and direct light source colors are rarely the same.

A second observation is that, because the ambient reflection is material dependent, there is no single offset that could be subtracted from the image that would cause all material cylinders to intersect the origin. Third, BIDR cylinders representing two different materials with differing amounts of direct illumination can intersect without being collinear. For example, a material in 20% direct illumination can be identical to a different material in 80% direct illumination.
One implication of the above three observations is that traditional measures of color such as hue-saturation and normalized color are not invariant to changes in illumination intensity. A surface with a uniform body reflection will change color as the intensity of the direct illuminant changes. A neutral surface of constant material reflection can change its hue from red to blue as the amount of direct illuminant varies under natural illumination conditions.

### 4.2.3 Relevance to Thresholding

Color spaces such as chromaticity and hue-saturation were originally defined, in part, to enable the segregation of colors in an illumination invariant manner. Thresholding in chromaticity, for example, is like selecting a cone in RGB color space with its point at the origin. It’s a natural way to define a body reflection cylinder. In a lab environment with black curtains and a single light source, this type of method works quite well, as the situation matches the dichromatic reflection model well and the shadows go to black.

Likewise, hue, by itself, tends to be invariant to body body reflection and surface reflection, as the surface reflection tends to reduce the saturation of the color but not change the overall hue.

However, in real-world environments, chromaticity or hue-saturation are no longer illumination invariant because the chromaticities are changing as the illumination changes intensity.

In order to create an illumination invariant color space, it is necessary to follow the procedures outlined by Maxwell, Friedhoff, and Smith in their 2008 paper. They demonstrated that the log RGB space representations of the body reflection cylinders are all roughly parallel. Therefore, there exists a plane in log RGB space that is approximately perpendicular to the log space body reflection cylinders, meaning it can be used as a log space chromaticity plane for segregating different colors.
4.3 Color Space Distances

When comparing whether two colors are similar, it is important to consider how to compute distances in color spaces that are functionally or perceptually meaningful. For example, using Euclidean distance $D_E = \sqrt{\Delta R^2 + \Delta G^2 + \Delta B^2}$ to measure color differences in RGB space creates some strange results. The Euclidean distance between two points that we perceive as bright yellow may be much larger that the Euclidean distance between a dark red and a dark blue. The reason is that the linear distance between colors gets compressed towards the origin of the RGB color cube, while it gets expanded as colors get brighter. Likewise, Euclidean distance in HSI space makes very little perceptual sense. Conversely, the ab channels of the CIE-Lab color space are designed such that Euclidean distances on the ab manifold match perceptual distances. The same holds true for the uv channels of the CIE-Luv color space.

The following are some common distance metrics and the color spaces within which they make the most sense.

4.3.1 Euclidean Distance

Euclidean distance measures the linear distance between two points, using the sum of the squares of the differences in each channel $c$.

$$D_E(x_1, x_2) = \sum_c (x_{1c} - x_{2c})^2$$

(6)

Euclidean distance makes sense in the following color spaces:

- Standard chromaticity: $(r, g) = (\frac{R}{R+G+B}, \frac{G}{R+G+B})$
- The ab chromaticity channels of CIE-Lab space (reflected light)
- The uv chromaticity channels of the CIE-Luv space (emitted light)
- Log-space chromaticity as per the BIDR model

4.3.2 Cosine Distance / Angular Distance

Cosine distance measures the angular difference between two vectors in a color space. It uses the fact that the dot product of two vectors is proportional to the cosine of the angle between them. Two vectors that have an identical orientation will have a cosine of 1.0. As the angle between the vectors increases, the value falls off to zero as they become orthogonal. In the RGB cube, it is not possible for two color vectors to be more than 90 degrees apart.

$$D_C(x_1, x_2) = 1.0 - \frac{1}{||x_1|| ||x_2||} \sum_c (x_{1c} * x_{2c})$$

(7)

It is also possible to use the angle between the two vectors as a distance by taking the arccosine of the normalized dot product, but it is a more expensive computational operation.

Cosine distance makes sense in the following color spaces, with the caveat that as the magnitude of the color vectors approaches zero, the angular estimates get progressively less reliable.
• Linear RGB
• sRGB, though the nonlinear transformation modifies the angles slightly
• Hue-saturation [HS]
• Standard chromaticity considered as a 3-value vector \((r, g, b) = \left( \frac{R}{R+G+B}, \frac{G}{R+G+B}, \frac{G}{R+G+B} \right)\)

### 4.4 Histogram Distances

Another common task when comparing colors is to look at distributions of colors in an image or image crop. One method of representing a distribution of colors is to use a histogram. A histogram is the result of dividing a color space into buckets and then counting how many pixels in the image fall into each bucket. The resulting histogram captures the relative distribution of colors in the image.

#### 4.4.1 Euclidean Distance

Given two normalized histograms A and B, treat the histogram as a vector and compute Euclidean distance. Not usually a great idea.

#### 4.4.2 Intersection Distance

Given two normalized histograms (sum of the bucket values is 1) A and B, compare each bucket and sum the minimum of A and B across all buckets. Two perfectly matching histograms will have a value of 1. As the histograms differ more, the value will move towards zero.

#### 4.4.3 Earth-Mover’s Distance

Given two normalized histograms, figure out the minimum amount of editing required to convert one histogram to the other. This metric is somewhat slow to compute, but it provides a metric that does a good job of matching intuition about similar color distributions.
Content-based Image Retrieval

CBIR is an application of computer vision that attempts to search image databases by content rather than by associated text or other non-image information. Of course, supplementing searches with that information is very useful.

CBIR, arguably, as progressed to the best it can be short of being able to do generic object recognition on a huge data set. Certain types of searches have been improved by the creation of effective detectors for things like faces.

Process:
1. Select a query image
2. Calculate features of the query image
3. Compare the features to those extracted from images in the database

Image matching:
- Whole-image color features: work well because they don’t require spatial similarity between images
  - Histograms in various color spaces (e.g. CIE-Lab seems to work well here)
  - Distance metrics are important: L1, L2, intersection, EMD, DTW per channel
  - Percents of each color in an image
  - Primary color of each image (largest histogram bin, or mean-shift result)
- Localized image color features
  - User might select a region of interest on which features get calculated
  - Run fixed sized windows over images in the DB at many scales (pyramid processing)
  - Calculate histograms for the image in different locations and match those with DB images
- Recognized objects or object classes
  - Face detection
  - Sky detection
  - Outdoor v. Indoor identification
- Spatial color distributions
  - Have the user draw colors in a small image
  - Match by similarity to the colored sketches
- Variance histograms
  - Calculate the primary colors in the image (dense locations in the histogram).
  - Calculate the spatial variance of the primary colors, which indicates their spread in the image.
  - Use a distance measure that includes the spatial variance.
  - Captures brightly colored small objects very well (bananas).
• Texture features (see texture analysis section)

• Shape
  – Have the person draw a rough outline of a shape
  – Generate a low resolution version of the outline and blur it
  – Compare the outline with smoothed gradient images at a low resolution

• Blobs
  – Cluster the image using both spatial proximity, color, and texture to generate a set of blobs.
  – Match with images that have similar blobs (spatial location can be preserved or not)
  – Alternatively, let the user pick which parts of the image are important and use those blobs.
5 Binary Image Processing

Thresholding produces a binary mask, where 0 pixels form the background and 1 pixels form the foreground. Often, the binarization process does not produce a perfect result and further processing is necessary.

Pixel Connectedness

- Which pixels are neighbors?
  - 4-connected: neighbors share a pixel boundary
  - 8-connected: neighbors touch on a boundary or a corner

5.1 Growing and Shrinking

Growing a region, also called a closing operation, is intended to fill small holes or gaps in a region caused by imperfections in the binarization process.

Growing: turn on any background pixel with a foreground pixel neighbor (4 or 8-connected)

- Useful for closing holes in a region or strengthening region connections

Shrinking: turn off any foreground pixel with a background pixel neighbor (4 or 8-connected)

- Useful for getting rid of small bits of noise in the binary image and separating regions that bleed into one another.

Median filter: count the number of foreground pixels in a neighborhood. If the number of foreground pixels is greater than or equal to half the neighborhood size, set the pixel to a foreground pixel, otherwise set it to a background pixel.

- Useful for getting rid of small noise particles and internal holes without modifying the overall shape or extent of large regions.

5.2 Morphological Operators

Morphological operators apply masks to ‘on’ pixels in the image to turn on or off other pixels. The mask, or structural element $S$ is applied to each pixel in the binary source image $B$, and the results get collected in a destination image $R$. Morphological operators can also be applied directly to greyscale imagery by extending the definitions to continuous rather than binary variables.

Note that the growing and shrinking operators described above can both be expressed (in both connectedness forms) as morphological operators.

- Dilation: the result is the union of the structural element $S$ placed on each foreground pixel $b \in B$.
  Dilation, or growing, produces a larger foreground region than the original, and small holes in the area will be filled in by the process.

$$R = B \oplus S = \bigcup_{b \in B} S_b$$ (8)
• Erosion: the result is the union of all pixels where the structural element centered at that pixel covers only foreground pixels. Erosion, or shrinking, produces a smaller foreground region than the original, and small protrusions or thin links between different parts of the region will be eliminated by the process.

\[ R = B \otimes S = \{ b|b + s \in B \; \forall s \in S \} \]  
\hspace{1cm} (9)

• Closing: dilation, followed by erosion. The process generally results in a more uniformly shaped region, approximately the same size as the original, with small holes filled in.

\[ R = B \circ S = (B \oplus S) \otimes S \]  
\hspace{1cm} (10)

• Opening: erosion, followed by dilation. The process generally results in a more uniformly shaped region, approximately the same size as the original, with thin links between sub-regions eliminated.

\[ R = B \cdot S = (B \otimes S) \oplus S \]  
\hspace{1cm} (11)

In some cases, it is useful to perform N growing steps, followed by N shrinking steps in order to close larger holes. Likewise, the reverse is true for opening up areas between regions connected by thicker bridges. Rather than execute multiple dilation or erosion steps, however, it is possible to explicitly calculate the distance of each pixel in a region from its border, enabling shrinking or growing by multiple steps in a single process. The two methods of calculating distances are the grassfire transform, which produces Manhattan distances, and the distance transform, which produces Euclidean distances.

Grassfire transform: Calculates the Manhattan distance from each pixel in the foreground region \( F \) to the closest pixel in the background region \( B \). The algorithm uses two-passes to calculate the distances. Prior to executing the algorithm, decide whether outside the image is part of \( F \) or part of \( B \). If outside the image is part of \( F \), it should have a large value.

• Pass 1: initial labeling, traverse in row-major order from upper left to lower right.
  1. For each pixel in the set to be labeled, look up and left.
  2. If both pixels are part of \( F \), assign the pixel as the lower distance plus 1.
  3. Otherwise, assign the pixel a value of 1.

• Pass 2: Correction labeling, traverse in backwards row-major order from lower right to upper left.
  1. For each pixel in the set to be labeled, look down and right.
  2. If both pixels are part of \( F \), assign the pixel the min of its current value or the lower distance of its neighbors plus 1.
  3. Otherwise, assign the pixel a value of 1.

Distance transform (Felzenszwalb and Huttenlocher, 2004): Euclidean metric distance from a foreground region to any pixel in the background region. Uses propagating wavefronts to calculate Euclidean distances efficiently (\( O(N) \)). Code is available online.

Both the grassfire transform and the distance transform are useful as morphological operators because they permit multiple growing or shrinking steps without requiring repeated use of a single structural element.
**Thinning**: used to reduce the thickness of regions without modifying connectedness. There are many thinning algorithms, sometimes called skeletonization algorithms. The concept is to reduce the region to a set of connected curve segments that preserve the region's shape and connectivity.

A simple pixel-based approach is to erode regions a pixel at a time, avoiding removing pixels that are required to preserve continuity. To avoid biasing the position of the skeleton, the process erodes from each of the four cardinal directions in turn.

It is possible to implement thinning to produce an 8-connected skeleton using a structural element that contains both ones and zeros, and which removes a point from the foreground set if the specified values of the structural element match the image exactly (http://www.cee.hw.ac.uk/hipr/html/thin.html). Using the masks given in (12) iterate the following process.

- Apply the left operator then the right.
- Rotate each operator by $90^\circ$ and repeat for each orientation.

\[
\begin{bmatrix}
  0 & 0 & 0 \\
  d & 1 & d \\
  1 & 1 & 1 \\
\end{bmatrix}
\quad
\begin{bmatrix}
  d & 0 & 0 \\
  1 & 1 & 0 \\
  d & 1 & d \\
\end{bmatrix}
\]  

(12)
5.3 Identifying and Characterizing Regions

Once we have thresholded and processing an image, we often want to identify connected regions within the binary image and calculate characteristics of the regions. There are two common algorithms for identifying connected components.

Two-pass connected component labeling

Set region counter to 0
Pass 1: look up and back
   If the pixel is not touching anything
      Assign it the value of the counter
      Increment the counter
   Else
      Assign it the region ID of the lowest valued neighbor
      If there are two neighbors with differing region IDs
         record the relationship in a list (union find data structure)
Pass 2: fix the region ID values and give each pixel its true region ID

Region Growing

Initialize a region map to -1
RegionID = 0

While there are still potential seed pixels
   Find a seed pixel
      If there is a seed pixel
         Label it with the region ID
         Push the seed pixel on the stack
         While the stack is not empty
            Pop the top pixel off the stack
            For each neighbor
               If the pixel is in the region
                  label it with the region ID
                  push it on the stack
      Increment RegionID

Return the region map

Both algorithms will return a region map where each pixel is labeled with its region ID.
5.4 Region Properties

Once we have regions, there are many characteristics we can use to describe them.

- **Moments**: define characteristics of the shape of the region
  \[ M_{pq} = \sum_x \sum_y x^p y^q f(x, y) \]  
  (13)

- **Size, or Area** \( M_{00} \)
  \[ M_{00} = \sum_x \sum_y f(x, y) \]  
  (14)

- **Centroid**
  \[ x_c = \frac{M_{10}}{M_{00}} \quad y_c = \frac{M_{01}}{M_{00}} \]  
  (15)

- **Central Moments**: moments defined relative to the centroid \((x_c, y_c)\)
  \[ \mu_{pq} = \sum_x \sum_y (x - x_c)^p (y - y_c)^q f(x, y) \]  
  (16)

- **2nd order row and column moments**: central moments about the \(x\) and \(y\) axes
  \[ \mu_{02} = \frac{1}{M_{00}} \sum_y (y - y_c)^2 f(x, y) \]  
  (17)

  \[ \mu_{20} = \frac{1}{M_{00}} \sum_x (x - x_c)^2 f(x, y) \]  
  (18)

- **Central axis angle**: the angle of the central axis, which is the angle with the least central moment.
  \[ \alpha = \frac{1}{2} \tan^{-1} \left( \frac{2\mu_{11}}{\mu_{02} - \mu_{20}} \right) \]  
  (19)

- **Second moment about the central axis**: orientation independent characteristic, where \( \beta = \alpha + \frac{\pi}{2} \)
  \[ \mu_{22\alpha} = \frac{1}{M_{00}} \sum_x \sum_y [(y - y_c) \cos \beta + (x - x_c) \sin \beta]^2 \]  
  (20)

- **Bounding box size**: maximum extent of the region in all directions.
- **Oriented bounding box**: maximum extent of the region parallel and perpendicular to the central axix.
- **Percent filled**: percent of the bounding box filled by the region.
Projections are histograms collected along an axis.

- X-projection
- Y-projection
- Central-axis projection
- Radial projection

The orientation of the axis with the least central moment, $\alpha$, can be found using the following.

$$
\tan 2\alpha = \frac{2 \sum (x - \bar{x})(y - \bar{y})}{\sum (x - \bar{x})(x - \bar{x}) - \sum (y - \bar{y})(y - \bar{y})} = \frac{2\mu_{xy}}{\mu_{rr} - \mu_{cc}}
$$

(21)

There are a few special cases in the computation.

- If $\mu_{rr} = 0$ and $\mu_{cc} = 0$ then the region is symmetric about its centroid (no dominant axis).
- If $\mu_{rc} = 0$ and $\mu_{rr} > \mu_{cc}$ then the major axis is vertical.
- If $\mu_{rc} = 0$ and $\mu_{rr} < \mu_{cc}$ then the major axis is horizontal.

You can compute the components of the central axis equation as follows.

$$
u_{rr} = M_{11}/M_{00} - \bar{x}\bar{y}
$$

(22)

$$
u_{cc} = M_{02}/M_{00} - \bar{y}^2
$$

(23)

$$
u_{cc} = M_{20}/M_{00} - \bar{x}^2
$$

(24)

The eigenvalues of the covariance matrix are:

$$
\lambda_i = \frac{\mu_{cc} + \mu_{rr}}{2} \pm \sqrt{4\mu_{rc}^2 + (\mu_{cc} - \mu_{rr})^2}
$$

(25)

The eccentricity $E$ of a region is given as a ratio of the eigenvalues.

$$
E = \sqrt{1 - \frac{\lambda_1}{\lambda_0}}
$$

(26)
6 Greyscale Image Processing

Binary images can make life easy because, if the foreground/background separation is good, then Binary images discard a lot of information in the thresholding process because they make hard decisions based on a few inputs. Often, we can achieve better results using greyscale or color images and looking for more sophisticated patterns.

6.1 Filters

Filters are used to modify the spatial frequency distribution of images. Any spatial signal—such as an image—can be represented as a sum of many different spatial frequencies with different amplitudes. High spatial frequencies are visual features that occur in a small spatial extent, such as lines or dots. Low spatial frequencies are visual features the occur over a large spatial extent, such as slow changes in intensity, perhaps caused by curvature on a smooth surface.

Common types of filters include: hi-pass, low-pass, or band-pass. A hi-pass filter will reduce low spatial frequencies in images and emphasize high spatial frequencies. Hi-pass filters are useful for emphasizing edges or texture in an image. A low-pass filter will do the opposite, and is useful for smoothing and reducing noise and small variations. A band-pass filter will emphasize a range of frequencies, but reduce spatial frequencies above and below the range. A band-pass filter is useful for identifying specific patterns that may exist within an image.

A filter can be N-dimensional, with 2-D filters being the most common in computer vision. A 2-D filter can be represented as a small image. Think of it as a pattern that is placed on the image at a specific pixel \((i, j)\), then a rule is applied that combines the image values and the filter values, and the output value for that pixel is the result. Because filters modify the image and generally take into account multiple pixels to determine the new value, it is necessary to put the results of a filter into a new image. Most filters do not work if the result is put back into the original image.

The rule used to apply filters is simple: multiply overlapping values from the filter and the image (one from each) and sum the results. The process is commonly called convolution. Formal convolution requires the filter to be placed upside down and backwards onto the image, although in computer vision that step is generally skipped.

Example: apply \([-1 \quad 0 \quad 1]\) to an image.

6.1.1 Gaussian

A Gaussian filter is a smoothing filter, or low-pass filter, that removes high frequencies from an image. It is effective at removing Gaussian noise, or noise with a zero mean, normal distribution.

The Gaussian filter gets its values from the 2-D Gaussian distribution:

\[
G(\sigma_x, \sigma_y) = \frac{1}{2\pi\sigma_x\sigma_y} e^{-\frac{(x-x_0)^2}{2\sigma_x^2} + \frac{(y-y_0)^2}{2\sigma_y^2}}
\]

Note that the 2-D Gaussian distribution is simply the multiplication of two 1-D Gaussians.
\[
G(\sigma) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-x_0)^2}{2\sigma^2}}
\]

A reasonable approximation to a 3x3 Gaussian filter is to use:

\[
\tilde{G}_{3x3} = \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}
\]

(29)

### 6.1.2 Sobel

The Sobel filter is a discrete approximation to the first derivative of a Gaussian. It is a high-pass filter and is often used to emphasize edge pixels in an image. The first derivative is not a symmetric operation, so there are two Sobel filters, one for the \( X \) direction and one for the \( Y \) direction.

\[
\tilde{S}_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}
\]

(30)

\[
\tilde{S}_y = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}
\]

(31)

### 6.1.3 Laplacian

The Laplacian filter is the 2nd derivative of a Gaussian distribution. It is often called the Mexican hat filter because of its shape. The Laplacian emphasizes where things are changing, and is sometimes used to emphasize edges. The Laplacian is a symmetric filter.

\[
L(\sigma) = -\frac{1}{\pi\sigma^4} \left[ 1 - \frac{x^2 + y^2}{2\sigma^2} \right] e^{-\frac{(x^2+y^2)}{2\sigma^2}}
\]

(32)

A reasonable approximation to the Laplacian is to use:

\[
\tilde{L}_{3x3} = \begin{bmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{bmatrix}
\]

(33)

### 6.1.4 Edge Preserving Filters

Often, we want to smooth an image but retain large edge information. A Gaussian filter smooths out Gaussian noise, but it also smooths across large edges, making them less precise and sometimes smoothing them so much they fall below whatever threshold is being used to detect them. A median filter is a common edge-preserving filter. Given an \( N \times N \) mask, the central pixel gets replaced by the median value of the \( N \times N \)
pixels under the mask. A median filter cannot, however, be expressed as a convolution or a frequency-based operation. It is an algorithmic process that requires an $N \log N$ process at every pixel. Furthermore, the process uses data dependent conditionals, which are about the worst possible algorithm for a modern pipelined, superscalar CPU.

An alternative is to apply multiple oriented box filters at each pixel and select the output with the smallest standard deviation under the active elements of the filter. For example, consider the following eight filter masks. One of the orientations will have a smaller standard deviation below the non-zero values. Another metric would be to choose the orientation that makes the smallest change to the central pixel. Unfortunately, this also tends to be computationally expensive.

$$
\begin{bmatrix}
1 & 2 & 1 \\
2 & 4 & 0 \\
1 & 0 & 0
\end{bmatrix} \quad \begin{bmatrix}
1 & 2 & 1 \\
2 & 4 & 2 \\
0 & 0 & 0
\end{bmatrix} \quad \begin{bmatrix}
1 & 2 & 1 \\
0 & 4 & 2 \\
0 & 0 & 1
\end{bmatrix} \quad \begin{bmatrix}
0 & 2 & 1 \\
0 & 4 & 2 \\
0 & 2 & 1
\end{bmatrix}
$$

$$
\begin{bmatrix}
0 & 0 & 1 \\
0 & 4 & 2 \\
1 & 2 & 1
\end{bmatrix} \quad \begin{bmatrix}
0 & 0 & 0 \\
2 & 4 & 0 \\
1 & 2 & 1
\end{bmatrix} \quad \begin{bmatrix}
1 & 0 & 0 \\
2 & 4 & 0 \\
1 & 2 & 1
\end{bmatrix}
$$

The most popular edge-preserving filter, and a filter with many other uses, is the bilateral filter. The bilateral filter concept is simple: we want to average together pixels that are similar in intensity/color, but we want to avoid including pixels that are very different in that average. The bilateral filter accomplishes this by using a Gaussian to average pixels spatially, but then it multiplies the weights of the spatial Gaussian by a second Gaussian that depends on the difference in the value between the central pixel and the pixel being averaged. So the weight of a pixel in the overall average depends not only on its spatial proximity to the central pixel, but also to its spectral proximity. A pixel that is close by may still have a very small weight if its value is very different.

Formally, the weight of a pixel is given by the following.

$$
G(\sigma_x, \sigma_y, \sigma_c) = \frac{1}{(2\pi)^{\frac{3}{2}}\sigma_x\sigma_y\sigma_c} e^{-\left(\frac{(x-x_0)^2}{2\sigma_x^2} + \frac{(y-y_0)^2}{2\sigma_y^2} + \frac{(c-c_0)^2}{2\sigma_c^2}\right)}
$$

As there is no free lunch, the bilateral filter can also be expensive to compute in its raw form. However, there are fast approximations to the bilateral filter that produce high quality results (Paris and Durand, "A Fast Approximation of the Bilateral Filter using a Signal Processing Approach", ECCV 2006).
6.2 Edge Detection Process

Canny Edge Detection Process

- Pre-process image (smoothing or median filter)
- Gradient operator
- Threshold in the gradient direction
- Thin the edges
- Chain code or Hough Transform to identify linked edges

6.2.1 Creating chain codes

After thresholding and thinning, each edge ought to be represented by a single pixel line in a binary image. The following algorithm returns a set of edge pixel sets, each represented by a seed pixel and a chain code.

- Find a seed pixel on a line (a good seed pixel has 1 or 2 neighbors)
- curLineID = 0;
- If 1 neighbor, label is with curLineID and push the seed pixel on the stack
- if 2 neighbors, push the seed and 1 neighbor on the stack, labeled with curLineID and curLineID+1
- While the stack is empty
  - Pop off the top seed pixel
  - Pick a direction and follow the line and build a chain code, labeling each with the ID of the seed
  - At any branching, terminate the current segment
  - Label the branches with new region IDs and increment curLineID
  - Push each potential seed onto the stack

Can also take a boundary following approach

Can also take a segmentation/connected components approach

6.2.2 Least Median of Squares

1. Initialize an MinError to a large value
   (a) Pick 2 points in the data set
   (b) Fit a line to the two points
   (c) Calculate the median squared error to the line of all points in the data set
   (d) If the error is less than MinError, set MinError and store the line parameters
   (e) Repeat from 1a for some number of iterations or until MinError is small enough
6.2.3 RANSAC

1. Initialize MaxSupport to 0
   
   (a) Pick enough data points to build a model (e.g. 2 for a line)
   
   (b) Build the model
   
   (c) Go through all of the data points and count how many are within a certain distance of the model (support)
   
   (d) If the support is greater than MaxSupport, set MaxSupport and store the parameters
   
   (e) Repeat from 1a for some number of iterations or until MaxSupport is big enough

2. Recalculate the model using a least-squared error method using all of the support for the model

3. Recalculate the inliers using the robust model
6.2.4 Hough Transform

1. Quantize the parameter space to generate an N-D histogram, or Hough accumulator.
2. Initialize the parameter histogram to zero.
3. For each point in the data set, let it vote for all parameter sets to which it could belong.
4. Identify maxima in the Hough accumulator as likely models.

Example: line detection

Line detection works best in a polar representation, with each line represented by its closest approach to the origin and the angle between the x-axis and a ray perpendicular to the line. In (36), \( x \) and \( y \) are any point on the line.

\[
\rho = x \cos \theta y \sin \theta
\]  

(36)

Divide \( \theta \) into 180 or 360 buckets and \( \rho \) into half the diagonal size of the image, using the center of the image as \((0, 0)\). For a 320x240 image, a reasonable Hough accumulator would be 180 (\( \theta \)) by 200 (\( \rho \)).

After identifying high gradient pixels in the image, as above, make one pass through the image. Each high gradient pixel votes for each line that could potentially pass through it. Using (36), step through all the values of \( \theta \) required by the accumulator and calculate \( \rho \) in order to identify the Hough accumulator locations. A simple improvement on this algorithm is to limit each pixel’s votes to only those lines that are appropriate given the local gradient direction.

After filling the Hough accumulator, the maxima in the accumulator should correspond to high probability models in the image. Note that, because of small roundoff errors, a single actual maximum will normally look like more than one high vote bins. Therefore, just selecting the top \( N \) bins in the Hough accumulator may generate the same line multiple times.

A reasonable process to follow is given below.

- Use a small box filter (e.g. 3x3) to identify the highest vote location in the accumulator.
- Store the line corresponding to that parameter bin.
- Suppress all of the accumulator values (set them to zero) in an area around the selected bin.
- Repeat \( N \) times if you want \( N \) lines, or until the best line fit has too few votes.

The same process also works for finding circles, boxes, and ellipses. Note that the Hough accumulator grows exponentially with dimension, so the method works efficiently only for small numbers of parameters (i.e. 2. or 3).
A Coding Guide

A.1 C/C++ Basics

C is a language not too far removed from assembly. It looks like Java (or Java was based on C syntax) but it’s not. The key difference between Java and C/C++ is that in C you have to manage memory yourself. In C you have access to any memory location, you can do math on memory locations, you can allocate memory, and you can free memory. This gives you tremendous power, but remember the Spidey rule: with great power comes great responsibility. The corollary of the Spidey rule is: with great power come great screw-ups.

C is a functional language, which means all code is organized into functions. All executable C programs must have one, and only one function called main. Execution of the program will begin at the start of the main function and terminate when it returns or exits. You can make lots of other functions and spread them around many other files, but your executable program will always start with main.

C++ is an object oriented language that allows you to design classes and organize methods the same way you do in Java. C++ still retains its functional roots, however, and your top level program still has to be main. You can’t make an executable function out of a class as you can in Java.

Code Organization

There are four types of files you will create: source files, header files, libraries, and object files.

- Source files: contain C/C++ code and end with a .c or .cpp suffix (.cc is also used for C++)
- Header files: contain type denitions, class declarations, prototypes, extern statements, and inline code. Header files should never declare variables or incorporate real code except in C++ class denitions.
- Libraries: contain pre-compiled routines in a compact form for linking with other code.
- Object files: object les are an intermediate step between source les and executables. When you build an executable from multiple source les, using object les is a way to speed up compilation.

The main function and command line arguments

One of the most common things we like to do with executable shell functions is give them arguments. C makes this easy to do. The main function should always be defined as below.

```c
int main(int argc, char *argv[]) {
    return(0);
}
```

The main function returns an int (0 for successful completion) and takes two arguments: arge and argv. The argument arge tells you how many strings are on the command line, including the name of the program itself. The argument argv is an array of character arrays (strings). Each separate string on the command line is one of the entries in argv. Between the two arguments you know how many strings were on the command line and what they were.
Data Types

The basic C data types are straightforward.

- **char / unsigned char**: 8 bits (1 byte) holding values in the range [-128, 127] or [0, 255]
- **short / unsigned short**: 16-bits (2 bytes) holding values in the range [-32768, 32767] or [0, 65535]
- **int / unsigned int**: 32 bits (4 bytes) holding signed or unsigned integers up to about 4 billion
- **long / unsigned long**: 32 (4 bytes) or 64 bits (8 bytes), depending on the processor type holding very large integers
- **float**: 32-bit (4 byte) IEEE floating point number
- **double**: 64-bit (8 byte) or longer IEEE floating point number

There aren’t any other basic data types. There are no native strings. You can create structures that are collections of basic data types (the data part of classes in Java). You can create union data structures where a single chunk of memory can be interpreted many different ways. You can also create arrays of any data type, including structures or unions.

```c
int a[50];
float b[20];
```

The best way to create a structure is to use the typedef statement. With the typedef you can create new names for specific data types, including arrays of a particular size. The following creates a data type Vector that is an array of four floats.

```c
typedef float Vector[4];
```

Example: Defining a structure

```c
typedef struct {
    short a;
    int b;
    float c;
} Fred;
```

The above defines Fred to be a structure that consists of three fields a, b, and c. The syntax for accessing the fields of Fred is dot-notation. The following declares two variables of type Fred. The first is initialized in the declaration, the second is initialized using three assignment statements.

```c
Fred tom = {3, 2, 1.0};
Fred f;

f.a = 6;
f.b = 3;
f.c = 2.0;
```

C does not pre-initialize variables for you (Java does). Whatever value a variable has at declaration is the result of random leftover bits sitting in memory and it has no meaning.
Strings

C does not have a built-in string type. Generally, strings are held in arrays of characters. Since an array does not know how big it is, C strings are null-terminated. That means the last character in a string must be the value 0 (not the digit 0, but the value 0). If you create a string without a terminator, something will go wrong. String constants in C, like “hello” will be null-terminated by default. But if you are manually creating a string, don’t forget to put a zero in the last place. The zero character is specified by the escape sequence ‘\0’.

Since strings in C are null-terminated, you always have to leave an extra character for the terminator. If you create a C array of 256 characters, you can put only 255 real characters in it.

Never allocate small strings. Filenames can be up to 255 characters, and pathnames to files can get large very quickly. Overwriting the end of small string arrays is one of the most common (and most difficult to debug) errors I’ve seen.

C does have a number of library functions for working with strings. Common ones include:

- `strcpy(char *dest, char *src)` - copies the source string to the destination string.
- `strcat(char *dest, char *src)` - concatenates src onto the end of the destination string.
- `strncpy(char *dest, char *src, size_t len)` - copies at most len characters from src into dst. If src is less than len characters long, the remainder of dst is filled with ‘\0’ characters. Otherwise, dst is not terminated. This is a safer function than strcpy because you can set len to the number of characters that can fit into the space allocated for dest.
- `strncat(char *dest, char *src, size_t count)` - appends not more than count characters from src onto the end of dest, and then adds a terminating ‘\0’. Set count appropriately so it does not overrun the end of dest. This is a safer function than strcat.

To find out about a C library function, you can always use the man pages. Typing `man strcpy`, for example, tells you all about it and related functions.

Header Files

You will want to create a number of different types for your graphics environment. In C the best way to put together new types is the typedef statement. In C++, use classes. Both types of declarations should be placed in header files. As an example, consider an Image data type. In C, we might declare the Image data type as below.

```c
typedef {
    Pixel *data;
    int rows;
    int cols;
} Image;
```

The difference with C++ and using a class is not significant, except that in C++ you can associate methods with the class.
class Image {
public:
    Pixel *data;
    int rows, cols;

    Image();
    Image(int r, int c);
};

Prototypes of functions also belong in header les. Prototypes describe the name and arguments of a function so that it is accessible in any source le and the compiler knows how to generate the code required to call the function.

Pixel *readPPM(int *rows, int *cols, int *colors, char *filename);

Extern statements are the appropriate method for advertising the existence of global variables to multiple source les. The global variable declarations themselves ought to be in source les. Initialization of the global variables also need to be in the source les. If the declaration itself is made in the header le, then multiple copies of the global variable may exist. Instead, an extern statement advertises the existence of the variable without actually instantiating it.

extern int myGlobalVariable;

Inline functions are small, often-used functions that help to speed up code by reducing the overhead of function calls. Rather than use a typical function call that requires pushing variables onto the stack, inline functions are copied into the function from which they were called. Because the functions are copied by the compiler, the compiler must have access to inline functions during compilation. Therefore, inline functions must be located in the header les. They are the only C code that belongs in a header le. In C++, methods dened within the class declaration are implicitly inline, but not necessarily. It is a good idea to only dene methods explicitly declared as inline in the header file, especially for large projects.

Useful include les

Standard include les for C provide denitions and prototypes for a number of useful functions such as printf(), provided by stdio.h, malloc, provided by stdlib.h, and strcpy(), provided by string.h. In addition, all math functions such as sqrt() are provided b math.h. A good template for include les for most C programs is given below.

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <string.h>

When using C++, if you want to use functions like printf(), you should use the new method of including these les, given below. In addition, the include le iostream is probably the most commonly used include le for C++.

#include <cstdio>
#include <cstdlib>
#include <cmath>
#include <cstring>
#include <iostream>
Pointers

Variables hold data. In particular, variables hold collections of ordered bits. All variables hold nothing but bits, and what makes them different is how we interpret the bits.

In C there are two major subdivisions in how we interpret the value in a variable. Some variables hold bits that we interpret as data; it has meaning all by itself. Some variables hold addresses: they point to other locations in memory. These are pointers.

When you declare a variable, it gives a label to some memory location and by using the variable’s name you access that memory location. If the variable is a simple data type (e.g. char, int, float, double) then the memory location addressed by the variable can hold enough bits for one of those types. If the variable is a pointer (e.g. char *, int *, float *, double *) then the memory location addressed by the variable can hold enough bits for the address of a memory location.

Until you allocate memory for the actual data and put the address of that allocated location into the pointer variable, the pointer variable’s address is not meaningful.

You can declare a pointer variable to any data type, including types that you make up like arrays, structures and unions. Adding a * after the data type means you are declaring a pointer to a data type, not the actual data type itself. That means you have created the space for an address that will hold the location of the specified type.

To allocate space for actual data, use the malloc function, which will allocate the amount of memory request and return a pointer to (the address) of the allocated memory. The sizeof function returns the number of bytes required to hold the specified data type.

Example: Declaring and allocating pointers

```c
int *a; // declare a pointer to an integer
a = malloc(sizeof(int)); // allocate memory for the integer
*a = 0; // assign a value to the integer
free(a); // free the allocated memory (the address in a is no longer valid)
```

The above first declares an int pointer and allocates space for it. The next line says to dereference the pointer (*a), which means to access the location addressed by a, and put the value 0 there. The final line frees the space allocated in the malloc statement.

Every malloc statement should be balanced by a free statement. Good coding practice is to put the free statement into your code when you make the malloc. The power of C is that you can, if you are sure about what you’re doing, access, write and manage memory directly.
Arrays

Arrays in C are nothing but pointers. If you declare a variable as `int a[50];` then the variable `a` holds the address of a memory location that has space for 50 ints. The difference between `int *a;` and `int a[50];` is that the former just allocates space for the address of one (or more) integers. The latter allocates space for 50 integers and space for their address and puts the address in `a`.

The benefit to making arrays using simple pointers is that you can set their size dynamically. Because arrays are pointers, you cannot copy their value from one array to another using a simple assignment. That just copies the address of one array into the variable holding the address of the second array (which is bad). You have to copy arrays element by element.

---

Example: creating an array

```c
int *a;
int size = 50;
int i;

a = malloc(sizeof(int) * size);

for(i=0;i<size;i++) {
    a[i] = 0;
}

free(a);
```

The above creates an array of 50 ints and puts the address of that memory in `a`. It then puts the value 0 in each of the memory locations and then frees the memory. If you want to create 500 ints, all you have to do is change the value of the variable `size`.

---

You can declare multi-dimensional arrays in C.

```c
int a[4][4]; // creates a 4x4 array
```

However, multi-dimensional arrays in C don’t act like true 2-D arrays. For a fixed-size data type, like a 4x4 matrix, they work fine. But you can’t directly allocate a multi-dimensional array using `malloc`. You have to build multi-dimensional arrays yourself, as shown in the next section.
A.2  Image Data Structures

Images are big 2D arrays. The common way to address a particular image location is (row, column) notation. This is different than traditional mathematical notation, which generally puts the horizontal axis (x-axis) first and accesses 2D locations using (x, y). The reason is that data on a screen is organized in row-major order. All of the pixels on the first row (top row) of an image come first in order from left to right. Just remember that x is a column and y is a row.

Any image can be represented as a single big array. Just take all the rows from top to bottom and concatenate them together. The index of pixel \((r, c)\) is then \([r * \text{cols} + c]\). To allocate an image as a big array, we can just figure out how many pixels there are an allocate enough for the image.

We can also create real 2-D arrays by first creating an array of row pointers and then setting their value to the proper addresses in a pixel array. The latter makes accessing random row-column locations easier to code in many situations. Allocating, setting up, and freeing 2-D arrays takes a bit more effort. The code below shows how to create both types of image arrays for a single channel (greyscale) floating point image.

A.2.1  OpenCV Image Data Structures

OpenCV uses the cv::Mat class to hold images and their associated information in memory. The cv::Mat class also acts as the interface to image and video IO. Most memory allocation and de-allocation is handled by the cv::Mat constructors and destructors, though the constructor does allow you to pass in a pointer to previously allocated memory in which to store the data. If you pass in a pointer to allocated data, you are responsible for de-allocating that data when you are done with the cv::Mat object.

The most common usages will likely be to read an image from a file, grab it from a video stream, or create a blank image of a particular size. In order to read an image, use the cv::imread function, which returns a cv::Mat object.

```cpp
cv::Mat src;
src = cv::imread( filename );
```

Note that in this case the cv::imread function allocates space for the image. It then copies a pointer to that memory space to the src.data field of the cv::Mat object. That memory space should be de-allocated by the cv::Mat destructor when it goes out of scope, so you do not need to explicitly free it. Likewise, the assignment operator (=) is smart in the sense that, if src already possessed allocated memory, it would appropriately release it before taking on the new data pointer. The assignment operator never allocates new memory, so the data pointer copied to src must have been allocated by the imread function.

There are two ways of creating a cv::Mat of a given size. One uses a constructor variant, the other uses the create method. Note that the release call is not strictly necessary, since release is called by the object destructor.

```cpp
cv::Mat src(1024, 1024, cv::CV_8UC3 );
cv::Mat dest;

dest.create( 480, 640, cv::CV_8UC3 );
// do things here
dest.release();
```
Example: creating your own image arrays

The following creates an image as a 1-D array and initializes it to zeros.

```c
int rows = 50;
int cols = 50;
int size = rows * cols;
int i;
float *image;

image = malloc(sizeof(float) * size);

for(i=0;i<size;i++) {
    image[i] = 0.0;
}

free(image);
```

The following creates a 2-D array and initializes it to zeros.

```c
int rows = 50;
int cols = 50;
int size = rows * cols;
int i, j;
float **image; // note the image is a double pointer

image = malloc(sizeof( float *) * rows ); // allocate row pointers
image[0] = malloc(sizeof(float) * size); // allocate the pixels
for(int i=1;i<rows;i++) {
    image[i] = &( image[0][i*cols] ); // set the row pointers
}

for(i=0;i<rows;i++) {
    for(j=0;j<cols;j++) {
        image[i][j] = 0.0;
    }
}

free(image[0]); // free the pixel data
free(image); // free the row pointer data
```
Opening and Displaying an Image Using OpenCV

```c
#include <cstdio>
#include <cstring>
#include "opencv2/opencv.hpp"

int main(int argc, char *argv[]) {
    cv::Mat src;
    char filename[256];

    if(argc < 2) {
        printf("Usage %s <image filename>\n", argv[0]);
        exit(-1);
    }
    strcpy(filename, argv[1]);

    // read the file and make sure it was successful
    src = cv::imread(filename);
    if(src.data == NULL) {
        printf("Unable to read image %s\n", filename);
        exit(-1);
    }

    // Print out information about the image
    printf("filename: %s\n", filename);
    printf("Image size: %d rows x %d columns\n", (int)src.size().height,
            (int)src.size().width);
    printf("Image dimensions: %d\n", (int)src.channels());
    printf("Image depth: %d bytes/channel\n", (int)src.elemSize()/src.channels());

    // create a window, display the image, wait for a keypress
    cv::namedWindow(filename, 1);
    cv::imshow(filename, src);
    cv::waitKey(0);
    cv::destroyWindow(filename);

    return(0);
}
```

To compile the above, you will need to make sure OpenCV is installed and then include the proper OpenCV libraries in the compile command.
A.3 Compilation Basics

C/C++ files must be compiled before you can run your program. We will be using gcc/g++ as our compiler. As noted above, you can distribute your functions/classes among many files, but only one of the files you compile together into an executable can have a main function. It’s good coding style to distribute functions across a number of files in order to keep things modular and organized. As in Java, it’s common to use a separate file for each class.

You can always list all of the C/C++ files you want to compile together on the command line and tell gcc to build your executable. The example below compiles the three C files and links them together into a single executable called myprogram (the -o flag tells gcc what to call the output.

\[
gcc -o myprogram file1.c file2.c file3.c
\]

The problem with this approach is that every time you make a change to one of the files, all of the files get recompiled. With a large project, recompiling all of the files can be time consuming. The solution to this is to create an intermediate file type called an object file, or .o file. An object file is a precompiled version of the code ready to be linked together with other object files to create an executable. The -c flag for gcc/g++ tells it to create an object file instead of an executable. An object file is always indicated by a .o suffix. To build the same program as above, you would need to precompile each of the C files and then link the object files.

\[
gcc -c file1.c
\]
\[
gcc -c file2.c
\]
\[
gcc -c file3.c
\]
\[
gcc -o myprogram file1.o file2.o file3.o
\]

This seems like a lot of extra work, until you make a change to file1.c and then want to recompile. You can rebuild the executable using the commands

\[
gcc -c file1.c
\]
\[
gcc -o myprogram file1.o file2.o file3.o
\]

This can save you lots of time when you are working with a large project, because the files you didn’t touch don’t have to be recompiled. gcc/g++ just uses the object file to link things together.

Sometimes you also want to link in libraries such as the standard math library or libraries that you built yourself. The -l flag lets you link in existing libraries that are in the compiler’s search path. For example, to link in the math library, you would change the last line of the above example to be the following.

\[
gcc -o myprogram file1.o file2.o file3.o -lm
\]

When you want to link in a library you’ve built yourself, you need to not only link it with the -l flag, but also tell the compiler where to look with the -L flag. The example below tells the compiler to go up one directory and down into a subdirectory named lib to look for libraries. If the file libmylib.a is located there, it will successfully link.

\[
gcc -o myprogram file1.o file2.o file3.o -L../lib -lmylib -lm
\]

You may also need to tell the compiler where to find include files (.h files). If, for example, all of your include files in an include subdirectory, you will want to add a -I (dash capital I) option to the compile line when compiling the .c files. Linking is too late for includes, as they must be available during compilation. The following compiles the file file1.c into an object file file1.o and tells the compiler to look in a neighboring include subdirectory called include.
gcc -c file1.c -I../include

There are many other flags that can be used by gcc. Two you will see this semester are -Wall, which turns on all warnings, and -O2, which sets the optimization level to 2, which is fairly aggressive. It will likely make your code a bit faster, which is good.

Development Tools

The mac does not come with development tools in a standard install, although they are included on the installation disks. You need the Xcode package to get gcc/g++ and standard development tools like make. If you also want X-windows and other Unix tools and programs, then you will want to download XQuartz (version 2.3 as of September 2008) and MacPorts. MacPorts is a package manager and lets you install things like ImageMagick, gimp, and xv. Install Xcode, the XQuartz, the MacPorts. It will take a while, but none of it is particularly difficult.

Windows also does not come with development tools standard. The cygwin package is the most common way developers get an X-windows environment and all of the standard unix development tools. Note that there may be additional packages you need to download in addition to the standard install.

A.4 Makefiles

The make program is a way of automating the build process for programs. It’s most useful for large programs with many files, but it’s also useful for smaller programs. It can save you from typing a lot of stuff on the command line and keeps you from having to remember all the flags every time.

The simplest makefile is just two lines. The first line defines the name of the rule and its dependencies, the second line defines the rule’s action.

    myprogram: file1.c file2.c file3.c
    gcc -o myprogram file1.c file2.c file3.c -I../include -lm

The first line defines a rule called myprogram and lists the files upon which the rule depends. If any of those files change, the rule should be executed. The second line defines the action of the rule. If you type make myprogram the rule will execute. In fact, since make executes the first rule by default, all you need to type is make and the rule will execute.

When you’re just starting out, create simple makefiles where the rules are spelled out explicitly. There are many more things you can do with make, and it incorporates a complete scripting language for automating complex tasks. For more information, see the makefile tutorial linked to the course website.