CS 5330 Pattern Recognition and Computer Vision, Spring 2021

Dr. Bruce A. Maxwell
Khoury College of Computer Science
Roux Institute
Northeastern University

Course Description

Introduces the fundamentals of extracting 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, stereo, optical flow, motion estimation, and object detection and recognition. The course will cover both classical and modern computer vision techniques built on deep networks.

Students will learn by developing small and medium-scale vision systems to solve practical problems such as image filtering, content-based image retrieval, image stitching, augmented reality, and object recognition.

Prerequisites: Linear algebra or equivalent experience

> *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, including camera calibration.

B. Students understand and can implement image processing algorithms such as filtering, morphological operations, connected components, and feature detection.

C. Students understand and can implement algorithms for segmentation, detection, tracking, and classification of objects.

D. Students understand and can implement systems using deep networks to solve computer vision tasks such as object recognition and localization.

E. Students work in a group to design and develop a medium-sized image analysis and computer vision application.

F. Students present algorithms and results in an organized and competent manner, both written and orally.

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

  A line scan camera captures a single row of pixels. Because there are so few pixels, and each one can be connected to its own digitizer circuit without sacrificing sensor space, the pixels can be both very sensitive and fast.

- 2-D: line scan camera over time

  There are two different concepts here. First, if you put a line scan camera on a plane and fly it in a straight line you can generate a 2-D image over time, combining each row captured by the camera into a 2-D spatial image. Second, if you put a line scan camera over a moving belt, you can generate 2-D images of the items moving past the camera. You could also use the camera to view things like colored paper moving under the camera, checking for quality issues in the material.

- 2-D: line scan camera over frequency

  Light is made up of many frequencies, not just red, green, and blue. For some applications, subdividing the visual frequencies of light into small bins is important for differentiating different materials such as similarly colored paints. One way to capture find-grained spectral information is to use a regular 2-D camera sensor, but allow only a single bar of light through the aperture. By appropriately inserting a prism between the aperture and the sensor will split the light by frequency in the orientation perpendicular to the spatial dimension. This generates a 2-D image, with one dimension being spatial and one dimension being frequency.

- 2-D: image

  A regular 2-D image is captured by using a 2-D grid of pixels. A single pixel will generally have a small filter over it, controlling which wavelengths of light are incident on the silicon sensor. A greyscale image will have only a single value at each location. Technically, even a color image will have only a single measurement at each pixel location as standard sensor designs have only a single color filter for each physical pixel. Generating the RGB values at each pixel requires a process called Bayer interpolation. Technically, a color image is a 3-D matrix of information: two spatial and one frequency dimension.

- 3-D: images over time (video)

  Capturing a series of greyscale 2-D images over time generates a 3-D space of data with two spatial dimensions and one time dimension. A color video will have five dimensions with the addition of a frequency dimension.

- 4-D: image plus depth

  A device like the Kinect sensor is able to capture both a 2-D image and generate a depth image using a pair of cameras and stereo vision. A single image, therefore, has both intensity or color and depth at each pixel, creating data with two spatial dimensions, a frequency dimension, and a depth dimension.

- 5-D: video plus depth

Capture a video sequence using a camera with depth capability generates a five dimensional signal: two spatial, one frequency, one depth, and one time dimension.

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$\geq$ 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).

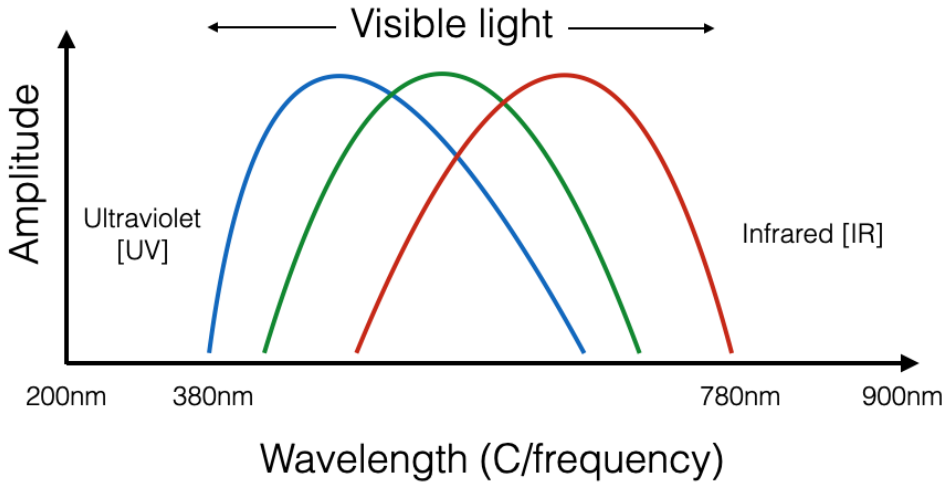


Figure 1: Visible light is part of the electro-magnetic [EM] spectrum. The figure shows approximations of the sensitivities of the red, green, and blue sensors in a camera or the human eye.

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, object detection
- 2000's: object recognition, SLAM, interactive vision systems, tracking, surveillance, image manipulation, large scale data collection, effective machine learning [ML]
- 2010's: explosion of deep networks and ML, vehicle navigation and robotics, object recognition, biometrics, face recognition, augmented reality, consumer applications

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 (focus): gathers light and focuses part of the world into a sharp image.
- Iris (aperture): 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: rods are greyscale sensors; standard cones are distributed between red, green, and blue sensitivities; special cones exist that are sensitive to narrow bands of blue light.
- 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 concepts develop.

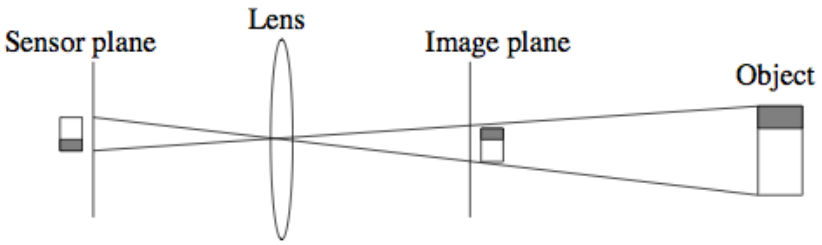
The computer vision community used to (a long time ago) believe that algorithms needed to have a biological basis in order to be valid. But computers are not human brains, and the algorithms that are appropriate for computers to implement–and which can successfully accomplish many tasks in computer vision–are 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.

With the advent of deep networks, inspired by neurons, vision systems are becoming more analogous to the human visual system in the sense that there are many layers of simple processing units connected in a large network in order to solve a problem. There are even researchers trying to build networks that can learn to

understand the visual simply by watching and interacting with it. Even the largest networks, however, don't yet have the capacity of the human brain (though that may change over the next few years).

Deep networks have also changed the nature of computer vision by focusing researchers on developing large data sets to feed the machine learning algorithms. These algorithms have solved a number of challenging problems, such as face detection and object recognition, to the degree they are commercially viable. The value of good data has skyrocketed. But it's important to define good data carefully. Biased data produces biased computer vision algorithms.

2 Imaging



Hardware

The simplest camera is a pinhole camera. You can make a pinhole camera with a piece of paper and a pin to make a small hole in the paper.

- 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. The image plane is a virtual sensor that sits in front of the lens rather than behind it. The projection of the scene onto the image plane is aligned with the orientation of the world.

All cameras must have a sensor:

- Silver plates: thin silver plate with photosensitive chemicals on it
- Film: a transparent thin film with photosensitive chemicals on it
- Photomultiplier tubes: generate electrical signals when struck with a photon
- Greyscale CCD: 2-D grid of sensors
- Single CCD color cameras (Bayer, Foveon, multi-spectral filters)
- Multi-CCD color cameras
- CMOS sensors: 2-D grid of sensors with logic associated with each pixel

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. A CMOS sensor is possible to fabricate using the same processes used for standard silicon computing chips.

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. CMOS sensors currently dominate most of the market because of the lower power and lower fabrication costs.

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 individual photons. A continuous world contains frequencies at all scales: sharp lines, small features, and big, slow changes.

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. A common example of this is watching turning wagon wheels in old western films. As the wagon starts to move, the wheels appear to be moving correctly. However, at some point during the acceleration the wheels appear to slow down and then start rotating backwards. This is an example of time aliasing.

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

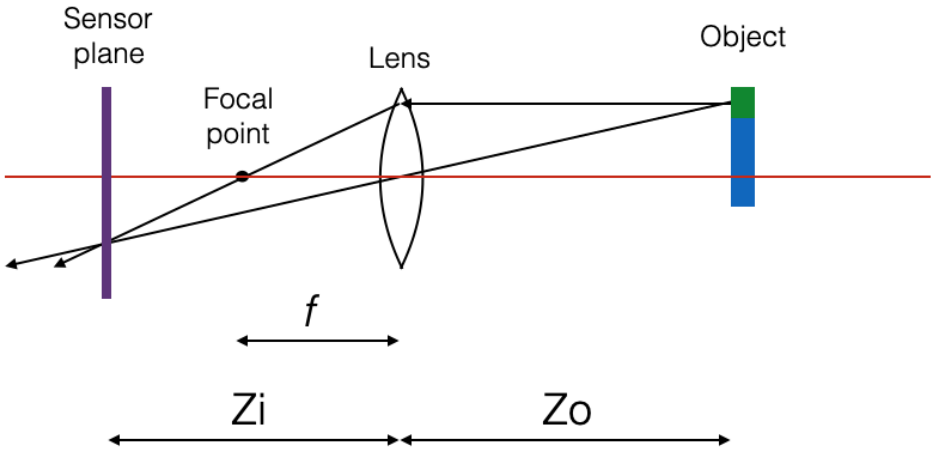


Figure 2: Basic geometry of how a lens focuses rays coming from an object. Rays parallel to the lens all converge on the focal point of the lens.

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)

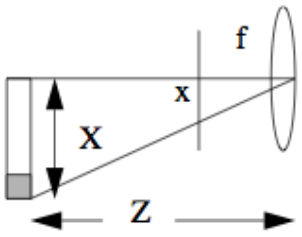
Thin lens law:

$$\frac{1}{Z_i} + \frac{1}{Z_o} = \frac{1}{f} \tag{1}$$

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:

$$x = \frac{-fX}{Z} \tag{2}$$

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.

$$\begin{bmatrix} Y \\ I \\ Q \end{bmatrix} = \begin{bmatrix} 0.30 & 0.59 & 0.11 \\ 0.60 & -0.28 & -0.32 \\ 0.21 & -0.52 & 0.31 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix} \tag{3}$$

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.

$$\begin{bmatrix} Y \\ U \\ V \end{bmatrix} = \begin{bmatrix} 0.30 & 0.59 & 0.11 \\ -0.30 & -0.59 & 0.38 \\ 0.58 & -0.59 & -0.11 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix} \tag{4}$$

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.

$$R = LM\cos(\theta) \tag{5}$$

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 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.

$$D_E(x_1, x_2) = \sum_c (x_{1c} - x_{2c})^2 \tag{6}$$

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.

$$D_C(x_1, x_2) = 1.0 - \frac{1}{||x_1|| ||x_2||} \sum_c (x_{1c} * x_{2c}) \tag{7}$$

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 $(r, g, b) = (\frac{R}{R+G+B}, \frac{G}{R+G+B}, \frac{G}{R+G+B})$

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.

$$R = B \oplus S = \bigcup_{b \in B} S_b \tag{8}$$

- 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.

$$R = B \otimes S = \left\{ \; b | b + s \in B \quad \forall s \in S \; \right\} \tag{9}$$

- 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.

$$R = B \bullet S = (B \oplus S) \otimes S \tag{10}$$

- 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.

$$R = B \cdot S = (B \otimes S) \oplus S \tag{11}$$

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° and repeat for each orientation.

$$\begin{bmatrix} 0 & 0 & 0 \\ d & 1 & d \\ 1 & 1 & 1 \end{bmatrix} \quad \begin{bmatrix} d & 0 & 0 \\ 1 & 1 & 0 \\ d & 1 & d \end{bmatrix} \tag{12}$$

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

$$M_{pq} = \sum_X \sum_Y x^p y^q f(x,y) \tag{13}$$

- Size, or Area (M_00)

$$M_{00} = \sum_X \sum_Y f(x,y) \tag{14}$$

- Centroid

$$x_c = \frac{M_{10}}{M_{00}} \quad y_c = \frac{M_{01}}{M_{00}} \tag{15}$$

- Central Moments: moments defined relative to the centroid (x_c, y_c)

$$\mu_{pq} = \sum_X \sum_Y (x - x_c)^p (y - y_c)^q f(x,y) \tag{16}$$

- 2nd order row and column moments: central moments about the x and y axes

$$\mu_{02} = \frac{1}{M_{00}} \sum_Y (y - y_c)^2 f(x,y) \tag{17}$$

$$\mu_{20} = \frac{1}{M_{00}} \sum_X (x - x_c)^2 f(x,y) \tag{18}$$

- Central axis angle: the angle of the central axis, which is the angle with the least central moment.

$$\alpha = \frac{1}{2} \tan^{-1} \left(\frac{2\mu_{11}}{\mu_{20} - \mu_{02}} \right) \tag{19}$$

- Second moment about the central axis: orientation independent characteristic, where $\beta = \alpha + \frac{\pi}{2}$

$$\mu_{22\alpha} = \frac{1}{M_{00}} \sum_X \sum_Y \left[(y - y_c) \cos\beta + (x - x_c) \sin\beta \right]^2 \tag{20}$$

- 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, α, can be found using the following.

$$\tan 2\alpha = \frac{2\sum(x-\bar{x})(y-\bar{y})}{\sum(x-\bar{x})(x-\bar{x}) - \sum(y-\bar{y})(y-\bar{y})} = \frac{2\mu_{xy}}{\mu_{rr} - \mu_{cc}} \tag{21}$$

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.

$$u_{rc} = M_{11}/M_{00} - \bar{x}\bar{y} \tag{22}$$

$$u_{rr} = M_{02}/M_{00} - \bar{y}^2 \tag{23}$$

$$u_{cc} = M_{20}/M_{00} - \bar{x}^2 \tag{24}$$

The eigenvalues of the covariance matrix are:

$$\lambda_i = \frac{\mu_{cc} + \mu_{rr}}{2} \pm \frac{\sqrt{4\mu_{rc}^2 + (\mu_{cc} - \mu_{rr})^2}}{2} \tag{25}$$

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

$$E = \sqrt{1 - \frac{\lambda_1}{\lambda_0}} \tag{26}$$

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

$$G(\sigma_x, \sigma_y) = \frac{1}{2\pi\sigma_x\sigma_y} e^{-\left(\frac{(x-x_0)^2}{2\sigma_x^2} + \frac{(y-y_0)^2}{2\sigma_y^2}\right)} \tag{27}$$

Note that the 2-D Gaussian distribution is simply the multiplication of two 1-D Gaussians.

$$G(\sigma) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(x-x_0)^2}{2\sigma^2}} \tag{28}$$

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

$$\tilde{G}_{3\times3} = \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix} \tag{29}$$

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.

$$\tilde{S}_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \tag{30}$$

$$\tilde{S}_y = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} \tag{31}$$

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.

$$L(\sigma) = -\frac{1}{\pi\sigma^4}\left[1 - \frac{x^2+y^2}{2\sigma^2}\right] e^{-\frac{(x^2+y^2)}{2\sigma^2}} \tag{32}$$

A reasonable approximation to the Laplacian is to use:

$$\tilde{L}_{3\times3} = \begin{bmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{bmatrix} \tag{33}$$

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 NxN mask, the central pixel gets replaced by the median value of the NxN 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.

$$\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 0 \\ 1 & 0 & 0 \end{bmatrix} \quad \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 0 & 0 & 0 \end{bmatrix} \quad \begin{bmatrix} 1 & 2 & 1 \\ 0 & 4 & 2 \\ 0 & 0 & 1 \end{bmatrix} \quad \begin{bmatrix} 0 & 2 & 1 \\ 0 & 4 & 2 \\ 0 & 2 & 1 \end{bmatrix}$$
$$\begin{bmatrix} 0 & 0 & 1 \\ 0 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix} \quad \begin{bmatrix} 0 & 0 & 0 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix} \quad \begin{bmatrix} 1 & 0 & 0 \\ 2 & 4 & 0 \\ 1 & 2 & 1 \end{bmatrix} \quad \begin{bmatrix} 1 & 0 & 0 \\ 2 & 4 & 0 \\ 1 & 2 & 1 \end{bmatrix} \tag{34}$$

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.

$$G(\sigma_x, \sigma_y, \sigma_c) = \frac{1}{(2\pi)^{\frac{3}{2}}\sigma_x\sigma_y\sigma_c} e^{-\left(\frac{(x-x_0)^2}{2\sigma_x^2} + \frac{(y-y_0)^2}{2\sigma_y^2} + \frac{(c-c_0)^2}{2\sigma_c^2}\right)} \tag{35}$$

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 Filer 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 Mean Shift

The mean shift algorithm is a method of identifying local peaks in a distribution. The concept is similar to K-means clustering.

- Given: a standard deviation, an initial location for the mean
- Using an N-D Gaussian centered on the mean, calculate a weighted average of the data within a radius
- Update the location of the mean
- Repeat until the movement of the mean is below a threshold
- Identify this average as a local peak, remove it and the data within a certain radius from consideration
- Pick a new mean location and repeat (if you want more than one local maxima in the distribution)

6.2.4 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

RANSAC is a heavily-used general purpose algorithm for identifying models in noisy data. It is useful in many situations.

6.2.5 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.

$$\rho = x \cos \theta y \sin \theta \tag{36}$$

Divide θ into 180 or 360 buckets and ρ 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 (θ) by 200 (ρ).

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 θ required by the accumulator and calculate ρ 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).

7 Texture

What is texture and how do we describe it?

- Spatial patterns
- Color patterns
- Repeated pattern / stochastic pattern
- Coarse / fine
- High energy / low energy
- Frequency spectrum: what bands have high energy / low energy

Most importantly, texture is not a pixel-wise concept. Color is a pixel-wise concept, texture requires an area, which makes it much more difficult to describe.

7.1 Statistical texture description

One way to think about textures is as random processes with certain statistical qualities. Statistical texture descriptors calculate features that try to capture these qualities in a way that differentiates different materials and patterns.

Local spatial frequency

- Autocorrelation: take a piece of the image and compare it to itself shifted spatially
- Take the Fourier transform [FFT] of a small area
- Take the Discrete Cosine Transform (DCT) of small areas (JPEG compression)
- Calculate the wavelet transform of the image, which gives similar information
- Large wavelengths (low frequencies) have poor spatial localization
- Small wavelengths (high frequencies) have good spatial localization
- For a given pixel, you get a feature vector with varying spatial coverage

7.1.1 Laws Filters

The Laws filters are derived from a 1×3 Gassian and its first and second derivatives. He argued that texture is change in an image, and that the first and second derivatives capture the basic characteristics of change (magnitude of change and how the magnitude is changing).

$$G_{1\times 3} = \begin{bmatrix} 1 & 2 & 1 \end{bmatrix} \tag{37}$$

$$G'_{1\times 3} = \begin{bmatrix} -1 & 0 & 1 \end{bmatrix} \tag{38}$$

$$G''_{1\times 3} = \begin{bmatrix} -1 & 2 & -1 \end{bmatrix} \tag{39}$$

Convolving these three matrices with one another creates a set of five unique 1×5 matrices.

$$\text{Gaussian (L5): } G \otimes G = \begin{bmatrix} 1 & 4 & 6 & 4 & 1 \end{bmatrix}^t \tag{40}$$

$$\text{Derivative (E5): } G \otimes G' = \begin{bmatrix} 1 & 2 & 0 & -2 & -1 \end{bmatrix}^t \tag{41}$$

$$\text{Spot (S5): } G \otimes G'' = \begin{bmatrix} -1 & 0 & 2 & 0 & -1 \end{bmatrix}^t \tag{42}$$

$$\text{Wave (W5): } G' \otimes G'' = \begin{bmatrix} 1 & -2 & 0 & 2 & -1 \end{bmatrix}^t \tag{43}$$

$$\text{Ripple (R5): } G'' \otimes G'' = \begin{bmatrix} 1 & -4 & 6 & -4 & 1 \end{bmatrix}^t \tag{44}$$

The above five filters can then be combined to create fourteen unique filters plus an averaging filter (5×5 Gaussian). For example, combining L5 with S5 produces the following, which reacts strongly to bright vertical lines.

$$L5 \times S5^t = \begin{bmatrix} 1 \\ 4 \\ 6 \\ 4 \\ 1 \end{bmatrix} \times \begin{bmatrix} -1 & 0 & 2 & 0 & -1 \end{bmatrix} = \begin{bmatrix} -1 & 0 & 2 & 0 & -1 \\ -4 & 0 & 8 & 0 & -4 \\ -6 & 0 & 12 & 0 & -6 \\ -4 & 0 & 8 & 0 & -4 \\ -1 & 0 & 2 & 0 & -1 \end{bmatrix} \tag{45}$$

Generally the output of the horizontal and vertical versions of each filter are combined together to produce a set of responses at each pixel that are orientation independent (but they can also be used individually to differentiate textures at different orientations.

There are 25 unique combinations. Combining the responses of rotated versions of each filter produces 14 unique combinations plus the $L5 \times L5$ filter.

Dividing each filter output by the output of the $L5 \times L5$ filter makes the filter responses somewhat independent of overall illumination/intensity levels, and produces 14 intensity normalized, orientation independent responses.

To further aggregate the results, a texture energy measure is generally defined for each pixel that is the sum of the absolute values of the filter responses over a local area. For example, the results may be summed over a 5×5 box or 7×7 box.

7.1.2 Co-occurence matrices

- Co-occurrence matrix is a histogram of what greyscale values occur together
- In general, a co-occurrence matrix is directional in specifying "togetherness"

Example: : generate $0°, 1$ matrix for the following image

- Pick a direction, e.g. $0°$
- Pick a distance, e.g. 1 pixel
- For each pixel, look in the direction to the selected distance and see what the neighbor is.
- Increment the entry in a 2D histogram for the pixel and its selected neighbor.

$$\begin{bmatrix} 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 2 & 2 & 2 \\ 2 & 2 & 3 & 3 \end{bmatrix} \rightarrow \begin{bmatrix} 2 & 2 & 1 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 3 & 2 \\ 0 & 0 & 0 & 1 \end{bmatrix}_{0^\circ,1}$$

A normalized co-occurrence matrix $N_{\alpha,d}$ is the result of dividing each entry of a co-occurrence histogram $C_{\alpha,d}$ by the sum of the histogram, producing probabilities.

$$N_d[i,j] = \frac{C_d[i,j]}{\sum_p \sum_q C_d[p,q]} \tag{46}$$

A normalized co-occurrence matrix can provide the following features:

Energy: the sum of the elements of the squared normalized co-occurrence matrix.

$$\text{Energy} = \sum_p \sum_q N_{\alpha,d}^2[p,q] \tag{47}$$

Entropy: the coherence of the co-occurrence matrix; maximum entropy is all of the values being in a single bin.

$$\text{Entropy} = -\sum_p \sum_q N_{\alpha,d}[p,q] \log N_{\alpha,d}[p,q] \tag{48}$$

Contrast: a weighted sum that emphasizes off-diagonals in the co-occurrence matrix.

$$\text{Contrast} = \sum_p \sum_q (i-j)^2 N_{\alpha,d}[p,q] \tag{49}$$

Homogeneity: a weighted sum that emphasizes the diagonal entries.

$$\text{Homogeneity} = \sum_p \sum_q \frac{1}{1+|i-j|} N_{\alpha,d}[p,q] \tag{50}$$

Maximum probability: the maximum value in the co-occurrence matrix.

7.1.3 Gabor Filters

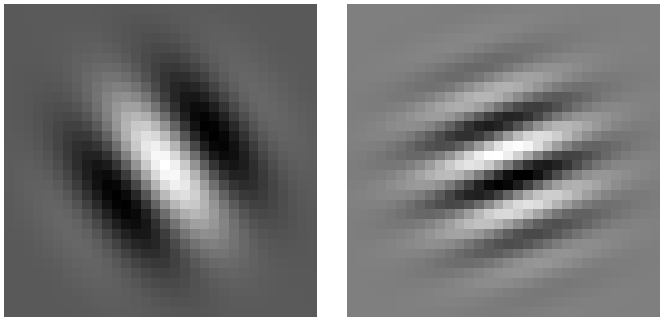
Gabor filters are defined as sine or cosine functions multiplied by a Gaussian. A particular Gabor filter responds to a band of spatial frequencies at a particular orientation.

$$\text{Gabor}_{\text{symmetric}}(x,y) = \cos(k_x x + k_y y) e^{-\left(\frac{x^2+y^2}{2\sigma^2}\right)} \tag{51}$$

$$\text{Gabor}_{\text{anti-symmetric}}(x,y) = \sin(k_x x + k_y y) e^{-\left(\frac{x^2+y^2}{2\sigma^2}\right)} \tag{52}$$

Visually, a Gabor filter looks like the images in figure 2.

As with other filter-based approaches, the algorithm for using Gabor filters is as follows:



(a) (b)

Figure 3: Gabor filters showing two different orientations and frequencies. (a) is a symmetric filter, while (b) is an asymmetric filter.

- Select a range of orientations and frequencies. Sometimes researchers use a large number of features to try and capture specific texture details. An alternative approach is to select a minimal set that forms an approximate basis space for the image.
- Apply the filters to the image, producing a stack of responses at each pixel.
- Aggregate the responses from a single filter in a local region to estimate the filter energy at each pixel. The result is still a stack of responses at each pixel. Sometimes this is called a Gabor jet.

7.1.4 Building Textons

Generating a large feature vector of responses at each pixel still may not capture what we think of as the quality of a texture. Some filters are extremely sensitive to location, and even a pixel or two difference can cause a significantly different response. In a very real sense, this variation over a small area in the filter response is a characteristic of the texture. Just calculating the average energy of the filter over a small area does not differentiate a rapidly changing response from a flat response.

One approach is to build a small histogram for each filter output, computing the histogram over a local area. Given 14 Laws filter responses, for example, one could build a local histogram for each filter that divides the filter's output range into a small set of bins (e.g. 8).

Such an explosion of features, however, can cause other problems, because high-dimensional feature vectors don't always behave well.

One approach to compressing the high dimensional feature vectors is to A) compress the individual histograms, or B) compress the whole large feature vector.

> Example: Compressing 1D histograms
>
> Given one or more training images of different textures, consider a single filter output, such as one of the Laws texture filters. If we apply the filter to each of the training images, we get one response at each pixel. If we then build an 8-bin histogram of all of the filter responses around each pixel within a 7x7 box, we end up with 8 values at each pixels. The 8 values indicate the distribution of responses around each pixel location.
>
> Given enough training data, we may have millions of examples of 8-bin histograms. There are a large number of possible histograms, because with a 7x7 box, the 49 pixels can be distributed

among the 8 locations in a large number of configurations. What we would like to do is create symbolic representations of the different, common, histogram forms.

A common approach to encoding complex data is to use the K-means algorithm. Select the number of symbols desired and then execute K-means clustering on the histograms, where K is the number of desired symbols.

If the number of features is large (e.g. > 20 or 30), then it is sometimes useful to execute an eigenanalysis first, before executing K-means. The eigenanalysis indicates the patterns of variation that exist in the data. For most vision data sets, the degree of correlation between features tends to be high, which means that there are well defined patterns of variation (if one feature goes up in value, several others tend to do the same). By finding those primary degrees of variation, which are the eigenvectors, we can project the data onto the eigenvectors to create a lower-dimensional space. Then we can execute K-means clustering on the resulting projected points.

Once the high-dimensional feature vector has been compressed to a smaller number of symbols, each pixel gets a single symbol. The symbol itself, however, may still be insufficient as a descriptor, because a texture has many scales to it, and not every filter captures the right scale. Thus, a texture may actually be described by the collection of symbols in a local area.

One more pass at histogram building and clustering produces a symbol for each pixel that represents a distribution of symbols around it. That technique has been reasonably successful for applications such as texture segmentation and content-based image retrieval using texture features.

The definition of texture, however, is still not clear. Symbols representing histograms of symbols that represent high dimensional feature vectors (possibly after eigenspace compression) that are histograms of feature responses is probably not the final representation of texture.

The approach does have value, however. Varma and Zisserman applied this concept of PCA to clustering to histograms of clusters in a local area to PCA to obtain feature vectors on a per pixel basis that performed quite well on real texture tasks.

7.1.5 Codebook Generation

A variation on K-means clustering is codebook generation, which is useful when you want to generate a large number (e.g. 256, 512, or 1024) of well-space clusters. The concept is the following.

- Identify the mean of the data. Initialize two clusters as the mean value plus a random offset.
- Run K-means clustering
- Divide each mean into two clusters by adding a random offsets to each cluster mean
- Repeat step 2 until you have a codebook of sufficient size

8 Calibration

While much of computer vision is interested in understanding the materials, objects, and semantic meaning of a scene, another important aspect of computer vision is understanding the shape and structure of a scene. If we can identify the spatial structure of the world around us we can use vision to navigate, build maps. create models, and guide robots or cars. A critical piece of understanding structure is understanding the mapping from 3D points in the world to 2D pixels in the image, and also understanding the reverse mapping that describes the 3D ray in the world corresponding to each pixel in the camera's sensor.

What makes calibration difficult is that every camera is unique. Every camera is slightly different in the size of its sensor element, the size of individual sensor pixels, the placement of its sensor element, and, most importantly, in the lensing system that focuses light onto the sensor. The combination of sensor, sensor size, sensor pixel size, focal length, and focal distortion constitute the **intrinsic parameters** of a camera: where the camera is located in the world does not affect these parameters. Note that the intrinsic parameters change, such as when the focus or zoom is modified, but they are still intrinsic to the camera, not to its location and orientation in the world.

The intrinsic parameters are not sufficient, however, to map a 3D point in some global coordinate system onto a pixel in an image. In addition to the intrinsic parameters, we also need the current position and orientation of the camera itself, **the extrinsic parameters**. A total of six additional parameters, three position parameters and three orientation parameters, are necessary in order to define the camera's position and orientation. The combination of the intrinsic parameters and extrinsic parameters gives us sufficient information to map a 3D point in the world into an image. Identifying those parameters can be challenging, but it is a critical first step to enabling accurate stereo, visual map building, localization, and augmented reality.

8.1 Projective Geometry and Transformations

One aspect of working with cameras is that they implicitly include a projective transformation from 3D to 2D. In other words, they map 3D points onto the 2D image place, dropping the z-coordinate information in the process. Mathematically, it is useful to use projective geometry when working with cameras. The basis of projective geometry is an equivalent relation between $n+1$ dimensional vectors where two vectors, as defined in (53), are considered equal if and only the left vector is a scaled version of the right one.

$$(x_1^a, \ldots, x_{n+1}^a) \equiv (x_1^b, \ldots, x_{n+1}^b) \tag{53}$$

A projective space consists of $(n+1)$ dimensional vectors with a canonical representation for sets of equivalent vectors. For example, projective points are typically written with a 1 in the last position. Any projective vector with a non-zero final element can be written in such a canonical form by scaling each element by the multiplicative inverse of the final element. Projective spaces are useful because it is straightforward to apply a projective transformation to projective points using a transformation matrix.

Key observations:

- The canonical form has a 1 in the last position.
- Each projective point written in canonical form represents a set of similar points.
- The point $(x, y, z, 0)$ represents a point at infinity in the direction (x, y, z).

Transformations are the tools we use to define translations, scales, and rotations of rigid objects. When a transformation is applied to a point, you can think of it as moving the point relative to a fixed origin, or as moving the origin relative to a fixed point. Either is valid.

There are three basic transformations forms:

$$T(t_x, t_y, t_z) = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_y \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{54}$$

$$S(s_x, s_y, s_z) = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{55}$$

$$R_x(\theta) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{56}$$

$$R_y(\theta) = \begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{57}$$

$$R_z(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{58}$$

So long as the 3-space coordinates are provided in projective form, the above equations will execute the proper transformation. If a transformation requires multiple steps, simply multiplying the individual transformation matrices together will generate a single complex transformation.

It is also possible to represent a perspective projection using a matrix.

$$P(f) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & \frac{1}{f} & 0 \end{bmatrix} \tag{59}$$

Division by the homogeneous coordinate resets the vector to the proper projection on the image plane.

$$\begin{bmatrix} \frac{fx}{z} \\ \frac{fy}{z} \\ f \\ 1 \end{bmatrix} = \begin{bmatrix} x \\ y \\ z \\ \frac{z}{f} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & \frac{1}{f} & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \tag{60}$$

8.2 Coordinate Systems

There are a number of different conceptual coordinate systems in between image coordinates and world coordinates, (even more if the camera is situated on a robot).

- World coordinates: $p_w = (X, Y, Z, 1)$
  - Origin O_w is at an arbitrarily selected point.
  - Axis orientations are also arbitrary, but generally selected intuitively.
- Camera Euclidean coordinates: $p_c = (X, Y, Z, 1)$
  - The focal point of the conceptual pinhole camera model O_c is the origin.
  - The z-axis, Z_c, is aligned with the optical axis and points away from the image plane.
  - All of the points you see in an image have positive z values.
  - A translation and rotation transforms the World coordinates to Camera Euclidean coordinates.
  - The translation and rotation may be due, in part, to a pan-tilt camera mount, for example.
  - Camera Euclidean X_c and Y_c coordinates are aligned with the physical pixels on the sensor plane.
  - Sometimes, it is useful to insert an intermediate coordinate system aligned with the camera body, or with the base of a pan-tilt-zoom camera.
- Image Euclidean coordinates: $p_i = (X, Y, 1)$
  - Camera and image Euclidean coordinates are related through a projective transformation.
  - Origin O_i is on the image plane and at the projective center of the image (the location to where the projective transformation maps Camera Euclidean points of the form $(0, 0, Z, 1)$.
  - X_i and Y_i axes are aligned with the Camera Euclidean coordinates.
- Image affine coordinates: $p_i = (u, v, 1)$
  - Normalized pixel coordinates, origin may be in the lower left or upper right corner (preference).
  - Center of the image (origin of Image Euclidean coordinates) is at $(u_0, v_0, 1)$.
  - v axis is aligned with the Image Euclidean Y_i axis.
  - u axis may have a different orientation and scaling (not generally on modern cameras)

8.3 Extrinsic Parameters

To move from world coordinates to Camera Euclidean coordinates, a point must undergo a rotation R and a translation t. It is possible to represent any rigid rotation in an N-dimensional space as an NxN matrix. If the camera coordinate axes are represented as normal vectors $\hat{X}_c, \hat{Y}_c, \hat{Z}_c$, then the rotation matrix that transforms world coordinates to camera coordinates is given in (61).

$$R_{w\rightarrow c} = \begin{bmatrix} X_{cx} & X_{cy} & X_{cz} \\ Y_{cx} & Y_{cy} & Y_{cz} \\ Z_{cx} & Z_{cy} & Z_{cz} \end{bmatrix} \tag{61}$$

The complete transformation for moving a point p_w World coordinates to a point p_c in Camera Euclidean coordinates is given in (62), where O_c is the origin of the camera coordinate system expressed in world coordinates.

$$\begin{aligned} p_c = \begin{bmatrix} x_c \\ y_c \\ z_c \end{bmatrix} &= R_{w\rightarrow c}(p_w - t_{w\rightarrow c}) \\ t_{w\rightarrow c} &= \begin{bmatrix} O_{cx} & O_{cy} & O_{cz} \end{bmatrix}' \end{aligned} \tag{62}$$

The parameters of the specific transformation required to convert World coordinates to Camera Euclidean coordinates are defined as the *extrinsic parameters* of the camera. The extrinsic parameters are required, for example, to know the baseline value for determining depth from disparity given stereo imagery. For a moving camera, the extrinsic parameters define the motion of the camera over time.

8.4 Intrinsic parameters

Converting from Camera Euclidean coordinates to Image Euclidean coordinates requires a projective transformation. In particular, a perspective transformation is required to fit the pinhole camera model. If the world coordinates are considered to be projective coordinates, then putting the projective vectors in canonical form divides the x and y coordinates by the z coordinate. The pinhole camera model is just a scale factor, $-f$ times the canonical x and y projective coordinates.

Because we are projecting world coordinates onto pixel coordinates, we also need to know the relative pixel width a (px/mm) and height c (px/mm) as well as the location of the center of the image (u_0, v_0). Furthermore, the rows and columns of the sensor may not be at exactly right angles, so a shear parameter b allows for some skew. The complete relationship is given by a matrix.

$$\vec{u} = \begin{bmatrix} u \\ v \\ w \end{bmatrix} = \begin{bmatrix} a & b & -u_0 \\ 0 & c & -v_0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \frac{-fx_c}{z_c} \\ \frac{-fy_c}{z_c} \\ 1 \end{bmatrix} = \begin{bmatrix} -fa & -fb & -u_0 \\ 0 & -fc & -v_0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \frac{-fx_c}{z_c} \\ \frac{-fy_c}{z_c} \\ 1 \end{bmatrix} \tag{63}$$

The matrix on the right is often called the calibration matrix K

- fa: the focal length defined in terms of number of pixels horizontally
- fc: the focal length defined in terms of the number of pixels vertically
- fb: the shear relative to the horizontal axis defined in pixels

- (u_0, v_0): center point, or origin of the image in pixels

The complete relationship between world Euclidean and image affine coordinates is given as a single 3x4 matrix M.

$$M = [KR| - KRt] \tag{64}$$

The transformation of a point in world space $(X, Y, Z, 1)$ to a point in camera space $(u, v, 1)$ is given by:

$$\begin{bmatrix} u' \\ v' \\ 1 \end{bmatrix} = \begin{bmatrix} u/w \\ v/w \\ 1 \end{bmatrix} = \begin{bmatrix} u \\ v \\ w \end{bmatrix} = [KR| - KRt] \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix} = M \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix} = \tag{65}$$

8.5 Radial Distortion

Ideal lenses have no distortion, but real lenses do. To account for distortion, which is a non-linear effect, a common model is to use a radially symmetric polynomial distortion model. The model implemented in OpenCV (Zhang, 2000), (Bouguet, MCT) is given in (67). It sits between the extrinsic transformation to Euclidean image coordinates and the intrinsic transformation to pixel coordinates.

Given image Euclidean coordinates (x', y'), as calculated in (66), the corrected image Euclidean coordinates will be (x'', y''), and the pixel coordinates will be (u, v).

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} x/z \\ y/z \\ 1 \end{bmatrix} = \begin{bmatrix} x \\ y \\ z \end{bmatrix} = R \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} + t \tag{66}$$

$$\begin{aligned} x'' &= x' \frac{1+k_1 r^2 + k_2 r^4 + k_3 r^6}{1+k_4 r^2 + k_5 r^4 + k_6 r^6} + 2p_1 x' y' + p_2(r^2 + 2x'^2) \\ y'' &= y' \frac{1+k_1 r^2 + k_2 r^4 + k_3 r^6}{1+k_4 r^2 + k_5 r^4 + k_6 r^6} + p_1(r^2 + 2y'^2) + 2p_2 x' y' \end{aligned} \tag{67}$$

$$\begin{bmatrix} u' \\ v' \\ 1 \end{bmatrix} = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x'' \\ y'' \\ 1 \end{bmatrix} \tag{68}$$

8.6 Single Camera Calibration

Generally, calibration is accomplished using a target with known properties. For example, a checkerboard with a known square size makes a good target. Given an arbitrarily selected origin and orientation for the checkerboard, each corner of each square represents a known point in 3D space. The projection of each of these points into image space provides an instance of (65). There are six unknowns for the extrinsic parameters and six for the intrinsic parameters, so at least twelve target points are required to estimate the full calibration matrix. Usually, many, many more are used in order to reduce the calibration error. Errors on the order of tenths of pixels are desirable.

As the relationship is non-linear–involves a division by Z–a non-linear optimization method must be used to solve for the parameters. The Caltech Matlab Calibration Toolbox is a convenient tool to use for calibration. The same technique is also implemented as part of the OpenCV computer vision code.

Given a calibration, there are two very useful things you can do.

- Calculate the 3D ray corresponding to any pixel in the scene
- Calculate the projection of any point in the scene into the image

One fun use of a calibration is to place virtual objects into a real scene. If the desired 3D location is known, the object can be mapped properly into the image. This has applications in user interfaces and in special effects for feature films.

9 Stereo

Stereo vision uses two or more cameras to take simultaneous images of a scene from different viewpoints. By identifying where in each image a particular scene point maps, the depth of the scene point can be calculated using triangulation.

In a canonical stereo setup, the two camera optical axes are parallel and separated by a distance b, called the **baseline**. In addition, the camera sensor rows are aligned and the focal lengths of the cameras are identical. Given that situation, if a scene point projects to row y_k in one image, it also projects to row y_k in the other image. Therefore, the only difference between the two images is the column (x value) onto which a particular scene point projects.

If we let the origin of the world be at the sensor origin of the right camera, then for an image point at location (X, Y, Z), where Z is distance from the camera, the perspective projection equation tells us that the ratio of X/Z is equal to x/f, where f is the focal length of the camera.

$$\frac{X}{Z} = \frac{x_R}{f} \tag{69}$$

Using the left camera origin, we also can describe the same similar triangles for the second camera.

$$\frac{b - X}{Z} = \frac{-x_L}{f} \tag{70}$$

Using both equations, we can get an expression in terms of the disparity $d = x_R - x_L$.

$$\begin{aligned} \frac{X}{Z} + \frac{(b - X)}{Z} &= \frac{x_R - x_L}{f} \\ \frac{b}{Z} &= \frac{x_R - x_L}{f} \\ \frac{1}{Z} &= \frac{x_R - x_L}{fb} \\ Z &= \frac{fb}{x_R - x_L} \\ Z &= \frac{fb}{d} \end{aligned} \tag{71}$$

The disparity equation (71) tells us a lot about designing a stereo camera setup. First, distance from the camera is inversely related to the disparity. The farther away an object is, the more similar is its projection into the left and right cameras. That makes sense: an object infinitely far away projects to the same pixel in both cameras. The closer an object is to the stereo pair, the more disparity there is in its projection onto the two cameras. This tells us three things about measuring depth.

- There is a near bound in measuring depth based on how big a disparity is allowed/feasible. This is, in part, a computational limit because trying to match image points with large disparities is computationally costly and challenging.
- There is a far bound in measuring depth when the disparity gets smaller than a pixel.

- The precision of distance measurements improves as the object gets closer to the camera. At the extreme, a single pixel difference in disparity is the difference between infinity and the distance corresponding to a disparity of 1.

The second thing the disparity equation tells us is that having a larger baseline extends the distance at which we can reasonably calculate measurements. However, a wider baseline also makes it more likely that the two cameras don't see the same aspect of a surface, which makes matching harder. A smaller baseline makes it more likely for the two cameras to see the same aspects of a surface and makes matching easier.

In the general case of two cameras that are not perfectly aligned, it is useful to specify the relationship of how a pixel in one image corresponds to one or more pixels in the other. As noted in the prior section on calibration, a pixel in an image corresponds to a ray in 3D space. Given a situation with a left camera and a right camera, consider a ray going from the optical center of the left camera through some pixel in its image plane. That ray, combined with the line segment connecting the two optical centers forms a plane: the epipolar plane. The intersection of the epipolar plane with the right camera image plane contains the set of all possible locations where the pixel in the left image might match up with its corresponding pixel in the right image. For any fixed point in space, the triangle formed by the optical centers and the point tells us how a set of pixels along a line segment in one image correspond to a set of pixels in the other image.

The epipoles e_l and e_r are the points in the corresponding image plane where the other camera's optical center maps. In the case of parallel image axes, the epipoles are lines.

Rectification is the process of warping one image to make the rows in one image (horizontal line segments) correspond to the same row in the other image. Conceptually, the process involves three steps. First, rotate the cameras so they are perpendicular to the axis connecting the two camera centers. Second, rotate the cameras around the optical axes so the epipolar lines are horizontal. Finally, re-scale the images, if necessary, to account for different focal lengths, magnifying the smaller image.

The process for generating the transformation is straightforward if we have the orientation and position of each camera (extrinsic parameters) and their intrinsic parameters. (See Trucco and Veri., pp 160).

The key is to understand that it is important to do the transformation backwards. Using the output image that you want, figure out which pixel to grab.

9.1 Essential and Fundamental Matrices

There are two key matrices that tells us about the relationship between two images. One is the Essential Matrix, which is a relationship that maps pixel locations in camera Euclidean coordinates (not pixels coordinates) from one image to the other. It is based on the fact that we can transform a pixel in one frame into the other frame by doing a translation by the optical axes, followed by a rotation to align the rays. The rotation necessary to align the two pixels has to be around the axis perpendicular to the epipolar plane, since both points are on that plane to begin with.

$$E = RS = R \begin{bmatrix} 0 & -T_z & T_y \\ T_z & 0 & -T_x \\ -T_y & T_x & 0 \end{bmatrix} \tag{72}$$

A ray in the left image defined by a point on the image plane p_l maps to the line segment connecting the corresponding p_r and the epipole e_r using the Essential matrix.

$$u_r = Ep_l \tag{73}$$

Because we have pixel coordinates, not camera Euclidean coordinates, we want to find the relationship between pixel coordinates. A point in camera Euclidean space maps to a pixel using the camera matrix M as defined in the calibration section. Therefore, given the calibration matrices for the left and right images, M_l and M_r, we can write the Fundamental matrix as the following.

$$F = M_r^{-T} E M_l^{-1} \tag{74}$$

The expression M_r^{-T} is the transpose of the inverse of M_r. The fundamental matrix maps a point in the left image to a line segment in the right. Given any two pixel locations p_l' and p_r', they have to satisfy the relationship $p_r'^T F p_l' = 0$. You can use this relationship to estimate the Fundamental matrix of any two image pairs if you know at least eight corresponding points in the scene that do not form a degenerate configuration.

9.2 Stereo Implementation

The process of implementing a stereo system has four steps.

- Simultaneous image capture: most cameras have a trigger mechanism
- Rectification: align the rows of the two images
- Registration: figure out the disparity at each pixel (dense), or at feature points (sparse)
- Depth calculation: use the disparity equation to get depth, but you might have to clean it up a bit

Rectification is the process of modifying one image of a stereo pair so that a ray corresponding to a row of one image maps to the same row in the other image.

Registration methods

- feature-based
- gradient matching (DTW)
- SSD (on raw data, on gradients, on 2nd derivative gradients)
- frequency-space

Registration in areas with no texture is hard/impossible to do. One solution is to project texture onto a scene.

- A standard projector can project a series of bit patterns onto a static scene. This makes it easy to match points with the same bit pattern over time.
- An IR projector can generate a grid on a scene, putting texture onto surfaces that have little or none. This is what the Kinect does.

10 Motion

Working with video adds a third dimension to the data: time. It also adds a tremendous amount of data, so many algorithms for working with video use a small time slice, a small image, or both.

10.1 Motion Detection

One of the more important characteristics of things in the world is whether they move or not. For example, it is exceptionally difficult for a person to maintain a rigid stance with undetectable motion for any significant length of time. Therefore, a straightforward method of differentiating between a face in a poster on a wall and a real person is that the real person moves. **Differential motion analysis** looks at the changes over time in a static scene where the camera does not move.

Motion detection also includes cues that indicate how a robot is moving through the world. Egomotion, or self-motion, causes features in the world to move in a geometrically determined manner that the robot can use to estimate its own motion relative to those objects. By tracking features over time, it is also possible to estimate the shape and depth of the environment using the series of images. **Optical flow** is the estimation of the motion field of the image over time, and **shape from motion** and **visual simultaneous localization and mapping** use features tracked over time to build models of the environment, estimate the egomotion of the camera, and localize the camera within the built environment.

10.2 Time to Collision

One interesting fact about motion is that we can estimate the time to impact of an object even without knowing any metric information about the object. The concept is simple: the perspective projection matrix tells us that an object will appear twice the size if it moves half the distance to the camera. Therefore, if we can track an object and calculate the time it takes to double in size, we know that the object will collide with the camera when an equal amount of time has passed. More generally, we can compute the time to collision if we calculate the derivative of the motion spatially.

Let the time to collision be $\tau = D/V$, where V is the unknown velocity (m/s) of the object and D (m) is the unknown distance to the object. The apparent size of an object of length L at time t as mapped onto the image plane by a pinhole camera with focal length f, is given by (75).

$$l(t) = \frac{fL}{D} \tag{75}$$

The time derivative of the change in the bar's size is given by (76).

$$l'(t) = \frac{dl(t)}{dt} = -\frac{fL}{D^2}\frac{dD}{dt} = \left(\frac{fL}{D}\right)\left(\frac{V}{D}\right) \tag{76}$$

Therefore, the ratio $l(t)/l'(t) = \tau$ is the time to collision.

$$\frac{l(t)}{l'(t)} = \frac{\left(\frac{fL}{D}\right)}{\left(\frac{fL}{D}\right)\left(\frac{V}{D}\right)} = \frac{D}{V} = \tau \tag{77}$$

10.3 Differential Motion Analysis

Differential motion analysis examines the difference between images of a scene taken at two different times. Typically, the frames are adjacent in a time series, but not necessarily. Some methods use the difference between the current frame and a *background frame* which can be either an image of the scene taken specifically for that purpose or a constructed image built over time out of a series of images.

Generally, differential motion analysis uses the absolute difference between the images. On modern processors with a saturating subtraction operator it is possible to generate the absolute values without a conditional operator. Saturated subtraction pegs a result to zero if the subtrahend is larger than the minuend. (78) shows the method, where $S(a, b)$ is saturated subtraction of b from a. The result is a greyscale or color difference image.

$$d_c(i,j) = S(I_A(i,j), I_B(i,j)) + S(I_B(i,j), I_A(i,j)) \tag{78}$$

A common second step is to generate a binary image out of the difference image using a threshold. Thresholding helps to reduce the effects of camera noise or apparent differences caused by small shifts in the camera location. A linear noise model with a term that scales with intensity is appropriate for thresholding difference images.

If we have two image, I_A and I_B, there are five possible explanations for differences between them.

- The pixel is on a moving object I_A and part of the background in I_B.
- The pixel is on part of the background in I_A and on a moving object in I_B.
- The pixel is on a moving object in I_A and part of a different object in I_B.
- The pixel is on a moving object in I_A and on a different part of the same object in I_B.
- The difference is due to noise or slight camera motion.

It is also possible to use a weighted sum of multiple difference images in order to generate the motion mask. While it will generally reduce the effects of noise or slight camera motion, averaging multiple difference images will blur the motion profile over time and insert a small delay into the system.

A cumulative difference mask based on N images with weights over time of $(a_{t_1}, a_{t_2}, a_{t_N-1})$ can be generated using (79).

$$d_c(i,j) = \sum_{k=1}^{N} -1 a_k \left| I_{k+1}(i,j) - I_k(i,j) \right| \tag{79}$$

Differential edge motion analysis is sometimes more robust than using difference images for the raw pixel values. For example, light sources will generally pulsate (60Hz sampled at 30Hz) and holes will appear in uniformly colored surfaces.

- Calculate the Sobel gradient magnitude for each image.
- Threshold the gradient.
- Calculate the difference between the edge pixel images.

Another simple approach to motion analysis is to use SSD, or another feature detector, to identify where small blocks of the image have moved.

- Select a block from the image at time $t-1$ and calculate a feature vector.
- Compare the feature vector to feature vectors from areas close to the original location in the image at time t
- The location with the closest match is the new location of that feature

Forward References The same process also works with SIFT features, Harris corners, to other features that are simple to track. They provide information about good places to track from frame to frame, reduce the size of the search space, and generally improve the robustness of the motion estimation. However, they only provide sparse information about motion in the image.

10.3.1 Background Images

Background images are models for how the scene appears with no unknown objects in it.

- Background images can be static
- Background images can evolve over time
- Background images can contain models of appearance at each pixel

A simple way to build a background image is to grab a static image of the scene with no foreign objects in it.

- User could select when to capture a background image
- System could pick a time to capture the image when the contents of the scene are known
- System could watch the scene for a time and pick an image when there has been no significant changes for a period of time

Problem: things change over time. Cameras get jostled, shadows move, new objects get inserted into the scene, and in some cases, background elements constantly move (leaves, waterfalls, etc).

A scene normally changes over time, so a background image must evolve in order to differentiate background from foreground objects. There is no reason we can't look at each pixel separately when building a background image.

Filter-based Background Images

One approach to building a background image is to treat the problem as one of filtering out the background color at each element. The basic idea is that the most common pixel value over a long enough period of time is probably the background color.

Build the background scene over time by looking at d_c (difference value) for each pixel over time. Pixels for which d_c is not changing significantly are likely background pixels. Another way to measure whether a pixel is changing a lot is to examine its standard deviation over time. There will be natural variation in pixel values over time, but it is generally much smaller than variation due to motion in the scene. Therefore, a threshold on standard deviation can be used to identify when the part of the scene corresponding to a pixel is stationary.

Probabilistic Background Images

A more rigorous approach to differentiating important from unimportant variation is to use a probabilistic model of appearance. For example, we could try to make decisions about where foreground objects are

located by examining the intensity and change in intensity at each pixel. If we decide that the noise in a pixel is Gaussian, then a Kalman filter is a natural way to model each pixel. We can then make decisions based on the Kalman state at each pixel.

One problem with filter-based background images is that naturally occurring repetitive motion such as trees waving in the breeze, ripples on the water, or waterfalls will cause those parts of the scene to constantly be novel. As these are high frequency events, a Kalman filter tracking intensity and its derivative won't really help us differentiate them from moving objects.

One way to deal with such variation is to build explicit probabilistic models of appearance by watching the scene over a short time period in which there are no foreground objects. A simple model of appearance is to fit a Gaussian to the observed variation. A more complex model could use a sum of Gaussians to permit multi-modal distributions (such as shadows on the ground cast by leaves moving in the breeze). The models represent all of the expected appearances of a pixel given that it is part of the background. When a novel object appears in the scene, it will most likely cause pixels to move out of their observed model, pushing the novel object into the foreground.

We can also use models of reflection to create more accurate models of appearance for a surface. For example, surfaces that go in and out of shadow will traverse a cylinder in RGB space or log RGB space. By watching a scene with moving shadows for a while, it is possible to build estimates of the BIDR cylinder for each pixel, creating a more specific and accurate model of appearance.

Use simple texture characteristics can also help to indicate if something is occluding the background, even if the color of the foreground object is the same.

10.4 Optical Flow

The optical flow of an image is the velocity field that represents the motion of the data at each pixel in the image. Motion in the image can be caused by camera motion, object motion, illumination motion, or any combination of the three. The optical flow of an image, which basically tracks how pixel values move, is an approximation to knowing how the objects in the image are moving.

Because shading and appearance can change with motion (camera, object, or illumination), knowing that the pixel value C located at position $(x_{t-1}, y_{t-1}, t-1)$ moved to location (x_t, y_t, t) does not mean that an object necessarily changed its geometry relative to the camera.

Nevertheless, optical flow is used in many applications to estimate useful properties about the scene, including scene geometry.

Almost all optical flow techniques are based on the assumption that intensity is conserved and that motion is smooth

- Pixel values don't change, they just move around
- Pixel values don't move very far between frames

The basis for optical flow techniques is the Optical Flow Constraint Equation

$$\frac{d}{dt} g(x, y, t) = g_x u + g_y v + gt = 0 \tag{80}$$

$$-g_t = g_x u + g_y v = \nabla g \cdot \vec{v} \tag{81}$$

- The change in overall intensity in the image over time is zero (intensities are conserved)
- g_x is the image gradient in the x direction
- g_y is the image gradient in the y direction
- g_t is the image gradient in the t direction
- (u, v) is the motion vector of the pixel

All three of the gradients can be calculated using appropriately oriented Sobel operators

The OFCE provides one equation for each pixel in the image, but there are two unknowns at each pixel: the x and y velocities. Therefore, we need a second constraint.

- Smoothness in the velocity field can be defined as minimizing the derivative of the proposed velocity field over the image
- Smoothness can be defined in terms of the second derivative of the image (noisy) or in terms of the derivative of the velocity field solution, which builds it more cleanly into the estimation framework

Defining smoothness as the magnitude of the gradient of the velocity field over the image, we can define an error term that we want to drive to zero with our velocity field solution.

$$E = \int_X \int_Y \left[(\nabla g \cdot \vec{v} + g_t)^2 + \lambda^2 (\|\nabla u\|^2 + \|\nabla v\|^2) \right] dxdy \tag{82}$$

$$E \approx \sum_{i \in I} \left[(\nabla g_i \cdot \vec{v_i} + g_{ti})^2 + \lambda^2 (\|\nabla u_i\|^2 + \|\nabla v_i\|^2) \right] \tag{83}$$

- Trying to fit both the OFCE and minimize the magnitude of the velocity field gradient over the image
- λ indicates the importance of the smoothness term (a good value is 0.5)

10.4.1 Horn & Schunk

The baseline algorithm for calculating optical flow is the Horn and Schunk algorithm, which was one of the earliest proposed algorithms.

The algorithm calculates a dense optical flow field, trying to estimate the flow at each pixel.

1. Initialize all velocity vectors $c(i,j) = 0$ for all pixels (i,j)
2. Calculate the gradient images f_x, f_y, and f_t
3. Over the whole image:
   - Calculate $\bar{u}^{k-1}$ as the average u velocity in an area around (i,j) at iteration $k-1$
   - Calculate $\bar{v}^{k-1}$ as the average v velocity in an area around (i,j) at iteration $k-1$
   - $P = f_x\bar{u}^{k-1} + f_y\bar{v}^{k-1} + f_t$
   - $D = \lambda^2 + f_x^2 + f_y^2$
   - $u^k(i,j) = \bar{u}^{k-1} - f_x(i,j)\frac{P(i,j)}{D(i,j)}$
   - $v^k(i,j) = \bar{v}^{k-1} - f_y(i,j)\frac{P(i,j)}{D(i,j)}$
4. Calculate E as in (83)
5. Repeat from step 3 unless E is small enough or the process has completed K iterations

Note that it is important to filter the resulting optical flow estimates using the gradient magnitude. If a pixel is in a large area of similar values, the estimate of its motion is likely to be somewhat random.

Horn & Schunk provides a dense optical flow field, as it provides a vector estimate at every pixel. Sparse optical flow fields provide estimates at a smaller set of locations, generally locations in the scene that contain features that are easy to track.

10.4.2 Handling larger motions

One problem with optical flow methods is that the motion of the pixels needs to be small, generally on the order of a pixel, in order for the OFCE to make sense. One way to handle larger motions is to use a pyramid approach to the optical flow calculation.

An image pyramid is, conceptually, the original image and then a series of downsampled versions of the image. Often, each step in the pyramid reduces the size of the image by half in each dimension. Downsampling an image by half in each dimension results in approximately 1.33 times the computation of working with just the original image.

The concept is to first compute the optical flow on the smallest image. Then expand the computed field to the next largest image in the pyramid and run the computation again. Repeat the process until the vector field is computed for the original image, or whatever size image is considered good enough for the purpose.

10.4.3 Using a velocity vector field

Given an optical flow vector field, you can discover many things.

- Identify the type of motion taking place
  - Translations forward or backward will put the center of motion in a specific location on the plane
  - Translations parallel to the image plane create parallel optical flow vectors with parallax
  - Pan and tilt rotations of the camera create parallel optical flow vectors with no parallax
  - Rotation about the optical axis (roll) creates circular optical flow vectors
- Center-of-expansion. When the camera moves forward or backward, the scene seems to grow out of a specific point. The velocity vectors will tend to point at the COE. Let each velocity vector vote for the set of possible centers of expansion for the camera (2-D locations). Collect the votes
  - Need to be able to handle general motion, so need to careful design accumulator
  - Some motions cause COEs at infinity (rotations and translations parallel to the image plane)
  - A logarithmic accumulator for areas outside the image may work well.
- Identifying moving objects: segment the optical flow field by similar velocities. Each region with a similar velocity is moving coherently.
- Shape from motion: it is possible to extract shape from optical flow fields and/or tracked features.

10.5 Good Features and Feature Tracking

Computing where something has gone from frame to frame is a key problem in computer vision. Sparse optical flow algorithms, for example, do not try to track all pixels, but instead track only a small number of features in the scene. The question becomes what are good features in the image to track?

The fundamental problem with identifying small features to track is the aperture problem. Consider a small image with a dark line going through the middle. If the line moves perpendicular to its orientation, you can see the motion. If the line moves parallel to its orientation, however, you cannot tell. This is the aperture problem: you can only see motion when it is perpendicular to the gradient of the image.

Good features to track should tell us something about motion in all directions. Therefore, a good feature should contain image gradients, strong image gradients, in at least two perpendicular orientations. For example, an object's corner will have two perpendicular edges, and any kind of motion is obvious. More formally, we want to characterize the distribution of gradients in the image patch and select patches with strong gradients in many directions. Consider the covariance matrix of the gradients in an image patch.

$$A = \begin{bmatrix} I_x^2 & I_x I_y \\ I_x I_y & I_y^2 \end{bmatrix} \tag{84}$$

The eigenvalues of the covariance matrix tell us a lot about the strength of the signal and distribution of the signal. In particular, the ratio of the smaller eigenvalue to the larger one tells us how much energy is in the second dimension. A good feature, should have almost as much energy in the second dimension as the first. A Harris feature is a window where the quantity H is large, given $\alpha = 0.06$.

$$H = \det(A) - \alpha \,\text{trace}(A)^2 = \lambda_0\lambda_1 - \alpha(\lambda_0 + \lambda_1)^2 \tag{85}$$

An alternative formulation (Brown, Szeliski, and Winder 2005) uses the ratio of the product and the sum.

$$H' = \frac{\det(A)}{\text{trace}(A)} = \frac{\lambda_0\lambda_1}{\lambda_0 + \lambda_1} \tag{86}$$

The GoodFeaturesToTrack function in OpenCV makes use of Harris corners and a procedure for avoiding too many features close to one another.

- Calculate the Harris corner quality measure at every source image pixel.
- Execute non-maxima suppression to identify local maxima in a 3x3 neighborhood
- Sort the remaining corners by the quality measure in descending order
- Throw away each corner for which there is a stronger corner at a distance less than maxDistance

The next step in feature tracking is to match features from one image to the next. The simplest possible method is to grab a small window around the keypoint and use SSD between the two image patches. To avoid a full N^2 matching process, each keypoint is generally matched to just the keypoints in the second image that are within a specified radius of the keypoint in the first image.

Often, however, using raw pixel values fails because of changes in lighting or changes in the orientation of the surface between the two images. Therefore, the field has developed a number of derived image features that are intended to be rotation, scale, and intensity invariant.

10.5.1 SIFT Features

Pretty classic feature used a lot for about a decade, then there was an explosion of other features.

Identifying Keypoints

- Calculate the Difference of Gaussians for an image pyramid
- Identify local maxima horizontally and vertically in the pyramid
- For each remaining point, use the ratio of eigenvalues to reject keypoints with poor localization properties

Orienting Keypoints

- Compute the gradient magnitude and orientation image
- Create an orientation histogram from an area around the keypoint
- Use the peak of the histogram to give an orientation to the keypoint
- Rotate the gradient magnitude and orientation image to align with the keypoint orientation

Building the Scale-Invariant Feature

- Built over a 16x16 window of the gradients around a pixel (i, j)
- The gradient magnitudes are down-weighted by a Gaussian function centered on (i, j)

- In each 4x4 area of the 16x16 window, build an 8-bucket histogram of gradient orientations, weighted by the magnitudes
- When filling the histograms, the weights are spread among the spatial and orientation neighbors
- Remember: the orientations and magnitudes have been rotated
- The result is a 128 element feature vector

Variations on SIFT

- PCA-SIFT: compute a 3042 dimensional vector of gradients over a 39x39 patch and use PCA to project it into a pre-computed 36 dimensional eigenspace. (Ke and Sukthankar, 2004)
- SURF: use box filters to approximate the derivatives and integrals used in SIFT. (Bay, Tuytelaars, and Van Gool, 2006)

Using features for OR

- Use training images to find sets of features on the object(s) of interest
- On new images, calculate the SIFT features
- Figure out how many of the new features match the object features in the DB

10.5.2 Other Features

There are a number of other features that have been developed, with minor variations in their performance. See the OpenCV documentation for a variety of them.

- FAST
- MSER
- ORB
- BRISK
- FREAK

The basic concept is the same: find points that don't suffer from the aperture problem, are fast to calculate, and are easy to compare.

11 Tracking and State Estimation

Given that we can find features in one image, how do we best track them over time? What if we want to estimate the velocity of the points? What if the points are occluded or not visible for some number of frames? How do we estimate where to look for the points at some later time?

What we are trying to do, when we track points, is estimate the state of those points, where the state includes not only the location of the points, but also their velocity and possibly acceleration. We might even want to estimate the 3D locations of the points if we have a calibrated camera.

11.1 Bayes Filter

It is often the case in tracking that we have an existing estimate of the state of relevant parts of the world.

For example, we may have an estimate of the location of an object in the scene and its velocity. If our camera is moving, we may also have information about the camera motion and a model for its impact on the appearance of the world. Using this information we can predict the new pose of the object a short time into the future using an appropriate motion equation. If all the information we need to make that prediction is encompassed by our state information, the process is called a **Markov chain**. In a Markov chain, the behavior of a system is a function of only the current state. No prior information is required to determine the next state.

It should be clear, however, that small errors and unpredicted external forces make the prediction of future location of an object uncertain, and the level of uncertainty increases over time in the absence of feedback, or measurements. Therefore, we need a process the incorporates both prediction based on the current state and current actions and correction based on new information.

The most general probabilistic method of prediction-correction is called a Bayes filter. A Bayes filter does not work on static state values, but on probability distributions. Every value of each state variable has a belief associated with it, and the collection of beliefs–which sums to one–constitutes a probability distribution function. The algorithm for a Bayes filter requires the belief state at time $t-1$, x_{t-1} and any known changes–like camera motion–or relevant measurements at time t, u_t and z_t. The output is the new belief state at time t, x_t. Therefore, the Bayes filter is a mapping from a probability distribution, inputs, and measurements to a new probability distribution.

`Given:` $pdf(\vec{x_{t-1}}), u_t, z_t$
`for all state variables` $x_t \in \vec{x_t}$

1. Prediction: $bel(x_t(-)) = \int p(x_t|u_t, x_{t-1}) pdf(x_{t-1}) dx_{t-1}$
2. Correction: $pdf(x_t(+)) = \eta p(z_t|x_t(-)) bel(x_t(-))$

`return` $pdf(\vec{x_t}) <= pdf(\vec{x_t(+)})$

In the prediction step, the *a priori* estimate of the new state belief distribution function, $bel(x_t(-))$ is generated as a function of the prior state probability $pdf(x_{t-1})$ and expected changes u_t. The belief value for any given $x_t(-)$ is the integral over the range of prior state values of the probability of the state value, $pdf(x_{t-1})$, and the probability of ending in x_t given the state value and the expected changes.

Another way of thinking about the computation, is as a set of paths ending at location $x_t(-)$. The probability of ending up at $x_t(-)$ is the sum of the probabilities of all the paths ending there. The probability of any one

path is the product of the probability of starting at x_{t-1}, given by $pdf(x_{t-1})$, and the probability of moving along a path from x_{t-1} to $x_t(-)$ given the expected changes u_t, represented by $p(x_t|u_t, x_{t-1})$.

In the correction step, the belief in each new state $x_t(-)$ is adjusted by the probability of seeing the measurement z_t in that new state. The normalization constant, η, converts the belief distribution back into a true probability distribution function by normalizing its integral to one.

The Bayes filter offers a general method of representing beliefs over time. However, it is not computationally tractable to compute in closed form, as the probability distribution functions quickly become complex for realistic models of motion and measurement noise. However, there are many approximations and simplifications to the method that are tractable without significant degradation of performance.

11.2 Kalman filters

One of the tractable versions of the Bayes filter is the Kalman filter.

The Kalman filter is an optimal Bayes filter algorithm for estimating state given the following conditions

- The system is linear (describable as a system of linear equations)
- The noise in the system has a Gaussian distribution
- The error criteria is expressed as a quadratic equation (e.g. sum-squared error)

Example of a linear system: Basic Newtonian physics

$$\begin{bmatrix} x_{i+1} \\ \dot{x}_{i+1} \\ \ddot{x}_{i+1} \end{bmatrix} = \begin{bmatrix} 1 & \Delta t & \frac{1}{2}\Delta t^2 \\ 0 & 1 & \Delta t \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_i \\ \dot{x}_i \\ \ddot{x}_i \end{bmatrix} \tag{87}$$

Setting up the Kalman filter

1. A measurement z_k is linearly related to a system state x_k as in (88), where the noise term v_k is a Gaussian (normal distribution) with zero mean and covariance R_k.

$$z_k = Hx_k + v_k \tag{88}$$

2. If variations in x and z are Gaussian (normally distributed), then the optimal estimator for the next state $\hat{x}_k(+)$ is also the optimal linear estimator, and the estimate of the next state is a linear combination of an a priori state estimate $\hat{x}_k(-)$ and the measurement z_k.

$$\hat{x}_k(+) = A\hat{x}_k(-) + Bz_k \tag{89}$$

3. The system to be measured is represented by a system dynamic model that says that the next state is a linear function of the current state plus some noise w_{k-1}. Note that, in the absense of any measurements, the error in the system state estimation will grow without bounds at a rate determined by Q_k, the process covariance matrix.

$$x_k = \Phi_{k-1}x_{k-1} + w_{k-1} \tag{90}$$

4. The system model, which we create for a specific system using our knowledge of its physics, tells us the a priori estimate of the next state (the estimate prior to any measurement information) based on the last a posteriori state estimate.

$$\hat{x}_k(-) = \Phi_{k-1}\hat{x}_{k-1}(+) \tag{91}$$

5. We also have an error model that tells us how much error currently exists in the system. The new covariances of the error are linear functions of the old error and the system update transformations in (91). The new covariance also increments by the process covariance matrix (the inherent error in the state update equation (90)).

$$P_k(-) = \Phi_{k-1}P_{k-1}(+)\Phi_{k-1}^T + Q_{k-1} \tag{92}$$

6. Now we have estimates of the new a priori state and a priori error based on the state and error update equations. These are open loop equations that do not use any new measurement information. Now we need to update the state based upon the measurements, which means we need to estimate the Kalman gain matrices for (89). The Kalman gain balances how much to trust the state update (using $P_k(-)$) and how much to trust the measurement (using R_k). The Kalman gain matrix equation is given in (93). The expression is the process noise projected into the measurement domain, so the Kalman gain matrix is the process covariance divided by the process covariance plus the measurement noise. The lower the measurement error relative to the process error, the higher the Kalman gain will be.

$$\bar{K}_k = P_k(-1)H_k^T\left[H_kP_k(-)H_k^T + R_k\right]^{-1} \tag{93}$$

7. We can now complete the estimation of the error in the new state, $P_k(+)$, using the Kalman gain matrix, which tells us how much each piece will be trusted.

$$P_k(+) = \left[I - \bar{K}_kH_k\right]P_k(-) \tag{94}$$

8. Finally, we can calculate the a posteriori state estimate using the Kalman gain, the a priori state estimate and the new measurement.

$$\hat{x}_k(+) = \hat{x}_k(-) + \bar{K}_k\left[z_k - H_k\hat{x}_k(-)\right] \tag{95}$$

11.2.1 Kalman Filter Summary

Kalman filter variables:

- Φ_k = process matrix at step k defining how the system changes
- Q_k = process noise covariance matrix at step k
- H_k = measurement matrix relating the state variable and measurement
- R_k = measurement noise covariance matrix at step k
- $\hat{x}_k(+)$ = system state estimate at step k
- P_k = system covariance matrix at step k, estimate of uncertainty in $\hat{x}_k(+)$
- K_k = Kalman gain estimate at step k, how much to trust the measurement

Kalman filter equations

$$\hat{x}_k(-) = \Phi_{k-1}\hat{x}_{k-1}(+) \tag{96}$$

$$P_k(-) = \Phi_{k-1}P_{k-1}(+)\Phi_{k-1}^T + Q_{k-1} \tag{97}$$

$$\bar{K}_k = P_k(-1)H_k^T\left[H_kP_k(-)H_k^T + R_k\right]^{-1} \tag{98}$$

$$P_k(+) = \left[I - \bar{K}_kH_k\right]P_k(-) \tag{99}$$

$$\hat{x}_k(+) = \hat{x}_k(-) + \bar{K}_k\left[z_k - H_k\hat{x}_k(-)\right] \tag{100}$$

Example: temperature estimation

Temperature is a slowly varying signal we can model as a constant value with a small system variance.

Setup

- State is a single value: $[x]$ (temperature in C)
- Estimate a process variance Q (difficult to do, so probably just a small value)
- Estimate a measurement variance R by calculating the standard deviation for a large number of measurements of a constant temperature situation (or even just a constant voltage source). Want to separate the actual process noise and the measurement noise.
- Define a state estimation matrix

  Temperature is modeled as a constant value, so $\Phi = [1]$
- Relationship between measurements and the temperature H

  If we have C comes directly from the measurement process, then $H = [1]$

  If the measurement undergoes a transformation, put that in H
- Let the initial state be $[0]$
- Let the initial error be $[P]$

For each new measurement

- The a priori state is the same as the last a posteriori state (Φ is the identity)
- The a priori error is the same as the a posteriori error plus Q
- The Kalman gain is

$$K_k = \frac{P_k(-)}{P_k(-) + R} \tag{101}$$

- The new a posteriori state is

$$\hat{x}_k(+) = \hat{x}_k(-) + K_k \left(z_k - H x_k(-) \right) \tag{102}$$

- The new a posteriori error is

$$P_k(+) = \left(1 - K_k H \right) P_k(-) \tag{103}$$

After each iteration, the temperature is updated based on the new measurement. Note that the value of P will converge to a constant, as will K, so long as R and Q are constant. Both P and K can be precomputed using a differential equation, but it can be simpler to run the filter for a while and then preset P and K based on the stable values.

Example: 1-D motion estimation

Imagine a robot that can move in one direction estimating its distance from a wall as it moves. The robot is equipped with a noisy sensor, such as a sonar, and it also knows the velocity command last sent to the motors. We want to model not only the position of the robot, but also its current velocity.

Setup

- State is a pair of values, position and velocity: $\begin{bmatrix} x \\ \dot{x} \end{bmatrix}$
- Estimate a process covariance matrix Q by measuring variation in the velocity and position given a command and extending that to position by multiplying by the time step Δt. Note that small differences in Δt mean that the position error probably increases faster than the position error. The off-diagonal terms, in this case, will not be zero as errors in the velocity are strongly related to errors in the position.
- Estimate a measurement variance R by calculating the standard deviation for a large number of measurements of known distances.
- Define a state estimation matrix

  We are modeling the state as a constant velocity, so $\Phi = \begin{bmatrix} 1 & \Delta t \\ 0 & 1 \end{bmatrix}$
- Relationship between the state and the distance measurement is defined by H.

  We can assume we are measuring distance directly, and velocity as $\dot{x} = (z_k - z_{k-1})$ in which case $H = \begin{bmatrix} 1 & 0 \\ 0 & 1/\Delta t \end{bmatrix}$ and $\begin{bmatrix} z_k \\ z_k - z_{k-1} \end{bmatrix} = H \begin{bmatrix} x \\ \dot{x} \end{bmatrix}$.
- Let the initial state be $\begin{bmatrix} D \\ 0 \end{bmatrix}$ and let the initial error P be the identity I.

For each new measurement

- Update the a priori state using the last a posteriori state and the update matrix Φ.
- The a priori error is the same as the a posteriori error plus Q, which is a known covariance matrix.
- The Kalman gain is

$$K_k = \frac{P_k(-)H^t}{HP_k(i)H^t + R} = P_k(-)H^t[HP_k(-)H^t + R]^{-1} \tag{104}$$

- The new a posteriori state is

$$\hat{x}_k(+) = \hat{x}_k(-) + K_k\left(z_k - Hx_k(-)\right) \tag{105}$$

- The new a posteriori error is

$$P_k(+) = \left(1 - K_k H\right) P_k(-) \tag{106}$$

After each iteration, the velocity and position are updated based on the new measurement. Note that the value of P will converge to a constant, as will K, so long as R and Q are constant. Both P and K can be precomputed using a differential equation, but it can be simpler to run the filter for a while and then preset P and K based on the stable values.

11.3 Particle filters

In practice, Kalman filters can be used to estimate state for a wide variety of real systems. Even for non-linear systems, it is possible to calculate locally linear update matrices for the Kalman update equations. There are two limitations to Kalman filters, however, that restrict their application in some situations.

- The Gaussian noise requirement is not realistic for many real-world situations, which may be multi-modal.
- The need to invert a matrix during the calculation of the Kalman gain limits the number of possible state values, which can be large, especially for localization problems. (However, current computational capabilities make this less of an issue than it used to be).

One successful alternative to Kalman filters is the **particle filter**, which represents probability distributions using samples. The Kalman filter also represents probability distributions using samples, but since the distributions are required to be Gaussian, only the mean and standard deviation are required for the representation. Arbitrary probability distributions require many more samples in order to ensure an adequate representation.

11.3.1 Definition

A particle filter is a sample-based Bayes filter. Each sample represents the probability of a particular state vector given all previous measurements. The distribution of state vectors within the samples is representative of the probability distribution function for the state vector given all prior measurements.

- $Z_k = (z_1, z_2, \ldots, z_k)$ is the set of all measurements.
- X is a state vector
- $P(X|Z_t)$ is the probability of a state vector X given all prior measurements Z_t.

Knowing $P(X|Z_t)$, we could pick high probability states or evaluate possible states as to their fitness. However, it is difficult to generate the distribution directly. Using the Bayes filter approach, we can express the probability of any particular state at time t, x_t given the measurement sequence Z_t as in (107).

$$P(x_t|Z_t) = \frac{p(z_t|x_t)p(x_t|Z_{t-1})}{p(z_t|Z_{t-1})} \tag{107}$$

Therefore, the probability of a state x_t given all sensor readings is a function of three probabilities:

1. the probability of the state x_t given all but the last sensor reading,
2. the probability of the last sensor reading z_t given the state vector x_t, and
3. the probability of the last sensor reading z_t given all prior sensor readings Z_{t-1}.

These probabilities represent an open loop update, $p(x_t|Z_{t-1})$, a closed loop correction, $p(z_t|x_t)$, and a normalization $p(z_t|Z_{t-1})$.

A particle filter represents the probability distribution function $pdf(x_{t-1}|Z_{t-1})$ as a set of samples, where each sample includes a state estimate s^i_{t-1} and a state probability π^i_{t-1}.

$$pdf(x_{t-1}|Z_{t-1}) \approx \{(s^1_{t-1}, \pi^1_{t-1}), \ldots, (s^N_{t-1}, \pi^N_{t-1})\} \tag{108}$$

11.3.2 Particle Filter Update Algorithm

Initialization

The initial set of samples should be drawn from the initial distribution of state vectors. If the state vector is known at time zero, then all of the samples are assigned the known value with equal probability.

Prediction phase:

For each sample (s_{t-1}^i, π_{t-1}^i), calculate a new state s_t^{i-} using a motion model. A typical motion model uses the kinematic update equations plus additive Gaussian noise.

The end result of the prediction phase is the modification of the current particle set to represent the open loop update distribution.

Update phase:

The update phase weights the new samples according to the current measurement z_t.

$$\pi_t^{i+} = p(z_t|x_t) \tag{109}$$

Re-sample phase:

If the process stopped there, the samples would generally get less and less likely, because they would drift away from the high probability locations of the probability distributions. To correct for the drift, the next generation of samples get selected stochastically from the current set according to the relative weights; high likelihood samples are more likely to be selected, while low likelihood samples are likely to be dropped.

Resample process

1. Stochastically select N samples from the set of existing samples according to the weights π_t^{i+}.
2. Renormalize the weights to sum to one: $\sum_N \pi_t^{i+} = 1$

11.3.3 Basic Process Summary

1. Given: a set of samples $\{(s_{t-1}^1, \pi_{t-1}^1), \ldots, (s_{t-1}^N, \pi_{t-1}^N)\}$
2. Prediction: For each sample, (s_{t-1}^i, π_{t-1}^i), generate a new state s_k^{i-}.
3. Correction: For each sample, generate a new weight $\pi_k = p(s_k^{i-}|z_t)$.
4. Resample: Select a new set of samples based on the weights.
5. Normalize: Normalize the weights to one.
6. Repeat steps 2 through 5 to maintain the state probability distribution function over time.

11.3.4 Particle Filter Modifications

Order of operation

It turns out that you can sample at the end of the process based on the π_t^{i+}, or you can sample at the beginning of the process based on the π_{t-1}^{i+}. Either method seems to work, but drawing from the prior distribution lets you draw samples from different distributions.

- Sometimes you just want to insert random particles into the system drawn from $p(x_t)$. Inserting random particles lets the filter locate new high probability areas and recover from drift.
- Sometimes you know something about where particles ought to be.

Sampling algorithm

The simple way of selecting a next generation is $O(N^2)$. Let the sum of the weights be Π.

1. Pick a random number between 0 and Π.
2. Sum the weights of the particles until you reach the number.
3. Repeat from step one N times.

The simple algorithm requires N summations of $N/2$ (on average) particles to select the next generation.

A more clever method of sampling is only $O(N)$.

1. Pick a random number between 0 and $\frac{1}{N}\Pi$.
2. Sum the weights of the particles until you reach the number
3. Select that particle
4. Add $\frac{1}{N}\Pi$ to the random number
5. Repeat from step 2

The algorithm samples from the weight space uniformly and regularly, but by aligning the grid at a random location it avoids aliasing. Due to the random alignment, any particle has a probability of landing on the grid that is proportional to its weight. The algorithm is order N because it passes through the particles only once. Avoiding the $O(N^2)$ sampling time makes each iteration of the entire particle filter linear in the number of particles.

11.3.5 Example: Localization Using Particle Filters

Particle system localization has worked well in practice on numerous systems. They have proven robust to noise, robust to divergence in the state tracks (unlike a Kalman Filter), and they can be fast.

Source: Dellaert, Fox, Burgard, Thrun, "Monte Carlo Localization for Robots", ICRA'99, May 1999.

Video: http://www.cs.washington.edu/ai/Mobile_Robotics/projects/mcl/animations/global-vision.gif

Process:

1. Generate a visual map of the ceiling
    - From the robot, take pictures of the ceiling at regular intervals
    - Stitch the pictures together into a single mosaic
2. While the robot is moving
    - Look up with the camera and continuously watch the ceiling
    - Consider a single pixel in the middle of the image
    - Update a particle filter with the single pixel measurement

Results: The robot moved around the NMNH for several days

- After hours the robot moved fast (2m/s).
- System worked exceptionall well.
- Divergence would definitely have been an issue for a Kalman filter.

11.3.6 Example: Adaptive particle filters

One issue with particle filters is that as the number of dimensions grows, the number of particles required in the general case grows exponentially. For a 1-D system you may need only 40 particles, but for a 3-D system you may need $40^3 = 64000$ to ensure adequate representation of the probability distribution. That may stress the computational capabilities of the robot to the point where the sensor data is arriving faster than the particle system is being updated.

Fox (Fox, Adapting the sample size in particle filters through KLD-sampling, IJRR, 2003) proposed adaptive the number of particles used by the system based on the complexity of the probability distribution. When the distribution is unknown, it is necessary to use a large number of particles distributed around the space of possible locations. But when the distribution is strongly peaked, it is not necessary to represent the peak with a large number of points.

The key to making the system work properly is to have a useful estimate of the error in the representation of the probability distribution. When the error is large, the system needs to use more particles. When the error is small, the system can reduce the number of particles, speeding up the computation. In one localization example, the number of particles starts at 40,000 and then reduces to around 40 once the particles have condensed around the robot's true location.

Video:
http://www.cs.washington.edu/ai/Mobile_Robotics/projects/mcl/#_KLD-sampling:_Adaptive_particle_fil

11.4 Visual-SLAM

Visual Simultaneous Localization and Mapping is one of the killer apps of computer vision...

12 Pattern Recognition

Pattern recognition is one method of converting data into knowledge. The basic question in pattern recognition is to identify when a new object is sufficiently similar to a previously seen pattern that we can label the new pattern as being the same thing as the old pattern.

We can also think of the process as one of subdividing the world of data into a set of classes. In most cases, the classes have some semantic meaning, such as an object or a particular person's face.

The basic process for building a pattern recognition system is as follows:

- Design a process for extracting features from the raw data (e.g. images).
- Design a process for matching feature sets (a distance measure).
- Build a training set of example features from each class we wish to recognize.
- Build a classifier using the training data.
- Evaluate the classifier on a test set of labeled data not included in the training set.

The overall process may iterate many times using different design decisions, or more training data, in order to optimize performance on the test set.

In general, a pattern recognition system will have a set of N class models $C_1 \ldots C_N$ and a reject class C_r. The reject class is the class of inputs that do not match any model well enough to receive a label.

Classifiers have a hierarchy of complexity and take many different approaches to making decisions. One way to think about the complexity is by thinking of the feature space as an N-dimensional space with boundaries between the different classes. A more complex classifier permits more complex boundaries to define the difference between classes. In some cases this is a good thing, and permits carving out small, oddly shaped areas of the feature space. In other cases, we want smoother boundary shapes in order to avoid learning things that are too specific about a class.

12.1 Distance Measures

There are many ways to measure similarity between feature vectors. Some are more appropriate than others, and the particular distance measure often depends upon the type of data being compared.

One issue with distances is that we would like them to obey the *Triangle Inequality* for a metric space.

$$D(\vec{x}, \vec{z}) \leq d(\vec{x}, \vec{y}) + d(\vec{y}, \vec{z}) \tag{110}$$

The triangle inequality is important because it means that distances are meaningful, and there are no shortcuts or wormholes in the space.

Mean absolute distance (also called L1 distance): not as sensitive to outliers as higher order distances.

$$D_{\text{MAD}}(\vec{x}, \vec{y}) = \sum_{i=1}^{N} |x_i - y_i|^2 \tag{111}$$

Normalized mean absolute distance: takes into account the variances in each dimension

$$D_{\text{MAD}}(\vec{x}, \vec{y}) = \sum_{i=1}^{N} \frac{|x_i - y_i|^2}{\sigma_i} \tag{112}$$

Euclidean distance (also called L2 distance): optimal distance measure for Gaussian noise.

$$D_{\text{E}}(\vec{x}, \vec{y}) = \sum_{i=1}^{N} (x_i - y_i)^2 \tag{113}$$

Normalized Euclidean distance: takes into account the variances of the different dimensions.

$$D_{\text{ES}}(\vec{x}, \vec{y}) = \sum_{i=1}^{N} \frac{(x_i - y_i)^2}{\sigma_i^2} \tag{114}$$

Mahalanobis distance: takes into account the covariances of the different dimensions.

$$D_{\text{Maha}}(\vec{x}, \vec{y}) = (\vec{x} - \vec{y})^T \Sigma^{-1} (\vec{x} - \vec{y}) \tag{115}$$

Hausdorff distance: defined as the maximum distance between corresponding elements of a set. Hausdorff distance has been shown to be useful in matching geometric shapes.

Intersection distance: defined as the sum of the minimum value of corresponding elements. Useful for matching histograms.

Earth Mover's Distance: defined as the minimum amount of change required to convert one set into another. EMD is often used as a robust distance measure for histograms.

Binary distance/Hamming distance: Count how many corresponding elements of two sets are the same. Similarity may be defined using a threshold and any of the above distances.

Cosine distance: Compute the cosine of the angle between the two vectors.

Dynamic Time Warping: An algorithm for measuring the similarity between vectors of potentially differing length, where each element of the vector represents a location in time or space.

DTW is one method of defining a framework that provides a rigorous definitions of symbols and how they can match up with an exemplar symbol. While dynamic time warping was originally designed to compare one-dimensional time signals, such as digitized speech, it is directly applicable to multi-dimensional signals with spatial extent. The fundamental concept in DTW is that a symbol in a gallery pattern can match more than one symbol, or no symbols, in the probe pattern, but the ordering of the symbols must be preserved.

For example, given the gallery pattern RWG and the probe pattern $RRWWWGG$, the two patterns would have a low error in DTW matching because each duplicated symbol in the probe would match up with the same symbol in the gallery. However, if the probe pattern looked like $RWRWGWG$, the match would be poor, because at least two the symbols would match incorrectly. There is no way to stretch RWG to match perfectly a pattern with interleaved symbols. The benefit of DTW matching over FSAs is that the match score can be continuous, and reflects the quality of the match between the gallery and probe patterns.

To create a gallery in the example above, we might take multiple images of the red-white-green target, extract a number of image strips that cross all three colors, and then generate one or more gallery patterns, depending upon the variety that exists in the training set. The gallery patterns, for example, might be averages of multiple similar training strips.

Formally, dynamic time warping returns the minimum cost path that matches the gallery and probe images, starting at the first element of each pattern and ending at the final element of each pattern. Visually, if we lay

R	GR	GR	GW	GW	GW	GG	GG
W	WR	WR	WW	WW	WW	WG	WG
G	RR	RR	RW	RW	RW	RG	RG
	R	R	W	W	W	G	G

Figure 4: Dynamic time warping matrix for two patterns.

out one pattern along the X-axis and the second along the Y-axis, they form a matrix with each location in the matrix defining a potential match two symbols, as shown in figure 3. The minimum cost path minimizes both the error between the symbols and the cost of moving through the matrix.

A recursive definition of the DTW cost from the origin of the matrix to any other location in the pattern match matrix is given in (116).

$$D(i_x, i_y) = \min \left(D(i'_x, i'_y) + \zeta((i'_x, i'_y), (i_x, i_y)) \right) \tag{116}$$

Classifiers

Nearest neighbor: The simplest classifier is based upon a set of examples that come from a training set. The example patterns could be individual samples or the result of some aggregation operation, such as k-means. A new pattern is given the label of the example to which it is closest.

Example: color segmentation

Pick the number of segments desired to represent the training set. Execute a K-means algorithm on the training data colors. Each mean then represents one color class.

The distance measure used determines the shape of the class boundaries. A straight Euclidean distance corresponds to spheres or planes where the spheres intersect.

K-nearest neighbor: Similar to nearest neighbor, except each class has multiple exemplars, and the distance between a target and a class is the sum of the distances to the K nearest exemplars in that class. K is often 2 or 3.

The boundaries of classes using K-nearest neighbors can become very complex, even using a simple Euclidean distance. While each exemplar represents a sphere of influence, the multiple exemplars for a class interact because their distances to a target are combined.

Support Vector Machines [SVM] The SVM algorithm is a variation on nearest neighbor. SVM identifies the data points in the training set that sit along the boundary between classes and keeps track of only these points. SVM also incorporates the concept of transforming the data space to find a warped space within which the boundary between classes is representable as a plane. The combination makes SVMs both fast and able to represent complex boundaries between classes.

Cascade classifiers

A completely different approach to object detection is to use lots of very simple features with simple classifiers. A sufficient number of simple classifiers can perform as well or better than a single complex feature. In addition, since many instances can be rejected using a few simple features, the overall detection process may be faster.

The key to using lots and lots of simple classifiers in a cascade network is that each one has to be set up so that it almost never rejects real instances of the object. The false positive rate can be high, but the false negative rate must be close to zero. All of the real instances must make it all the way through the chain. If some of the false positives are discarded at each step, then the number of false positives at the end of the chain ought to be low.

As an example of simple features, consider boxes of various sizes that represent area sums. A simple feature is the difference of the area sums between adjacent boxes of different sizes and configurations. It turns out that such simple features can be computed very quickly using the **integral image**.

The **integral image** is generated from the original greyscale image I. The value at a pixel at (x, y) in the integral image B is the inclusive sum of all the pixels up and left of (x, y) in the greyscale image.

$$B(i,j) = \sum_{x_i \leq x, y_j \leq y} I(x_i, y_j) \tag{117}$$

The integral image can be calculated in a single pass, and it is useful for calculating pixel sums of rectangular areas (or averages). The sum of a box starting at pixel (i, j) of size (di, dj) is the result of four accesses and three operations.

$$S = B(i,j) + B(i+di, j+dj) - B(i+di, j) - B(i, j+dj) \tag{118}$$

Viola and Jones used simple box features of varying sizes in configurations that included: side-by-side, up-down, center subtract, and 2x2 checkerboard. The base resolution of the detector was 24x24, so the set of all box features of any size or location that fit within a 24x24 box was approximately 180,000.

The benefit of the Haar-like features is their ease of computation.

Viola-Jones Adaboost Variation

1. Givens:
   - Example images $(x_1, y_1, w_{t,1}), \ldots, (x_n, y_n, w_{t,n})$ where y_i is 0 for negative examples and 1 for positive examples and w_i is the weight of the training example at step t during the training process.
   - A feature set with a classifier h_j associated with each feature j.
2. Initialize training example weights $w_{1,i} = \frac{1}{2m}$ for the m negative examples and $w_{1,i} = \frac{1}{2l}$ for the l positive examples.
3. For $t = 1, \ldots, T$:
   (a) Normalize the weights so they sum to one.
   (b) For each classifier h_j, train it to minimize the weighted performance on the training set.
   $$e_j = \sum_i w_i \left| h_j(x_i) - y_i \right|$$
   (c) Let h_t be the classifier with the lowest error e_t.
   (d) Update the weights on the training set:
   $$w_{t+1,i} = w_{t,i} \left(\frac{e_t}{1-e_t} \right)^q$$
   where $q = 0$ if example x_i was correctly classified by h_t and $q = 1$ otherwise.
4. The final strong classifier H uses a weighted vote of the ensemble of weak classifiers.
   $$H(x) = \begin{cases} 1 & \sum_{h_j \in H} \alpha_t h_j(x) \geq \frac{1}{2} \sum_{t=1}^{T} \alpha_t \\ 0 & \text{otherwise} \end{cases}$$
   where $\alpha_t = \log \frac{1-e_t}{e_t}$.

The algorithm above generates a single classifier that combines a number of weak classifiers to generate a much stronger classifier. The single classifier is a set of individual features/tests represented by the h_j whose votes are weighted. Classifiers with better performance get weighted more heavily. The collection of features is often called an ensemble.

To obtain a cascade, Viola-Jones used a sequence of these ensemble classifiers until they achieved a sufficient false positive rate. A single ensemble did not have a sufficiently small false positive rate. After picking a stage of the cascade, only training samples that passed the first set of stages–plus new negative examples–were used to continue training.

Cascade Classifier Variation: Wu and Rehg

1. Givens: a training set with positive and negative examples, a minimum detection rate d (e.g. 0.999), and a maximum false positive rate f (e.g. 1×10^{-7}).
2. For each feature j, train a weak classifier h_j with a false positive rate of f.
3. Initialize an ensemble classifier $H \leftarrow$, step $t \leftarrow 0$, current detection rate $d_t = 0.0$, and current false positive rate $f_t = 1.0$.
4. while $d_t < d$ or $f_t > f$
   - if $d_t < d$, then find the feature k with classifier h_k such that, by adding it to H, the detection rate of the new ensemble of classifiers d_{t+1} is maximized.
   - else, find the feature k with classifier h_k such that, by adding to H, the false positive rate of the new ensemble of classifiers f_{t+1} is minimized.
   - $t \leftarrow t + 1$, $H \leftarrow H \cup \{h_k\}$.
5. The decision of the ensemble classifier is formed by a majority voting of weak classifiers in H.

$$H(x) = \begin{cases} 1 & \sum_{h_j \in H} h_j(x) \geq \theta \\ 0 & \text{otherwise} \end{cases}$$

   The threshold θ is initially set to half the number of classifiers in the ensemble, but may be decreased if necessary to achieve a different balance between d_T and f_T.

Note that all of these methods are simply classifier methods that work on a feature vector. All of the above examples use images as the basis for the feature vectors, but they also apply equally well to gesture and action recognition, speech recognition (where HMMs are one of the most successful methods), or any other situation where a classifier may be useful.

Mulitiple Classifier Boosting and Tree-Structured Classifiers

Cascade classifiers are degenerate decision trees. They tend to be fast because most inputs are discarded in the first few stages, and each stage considers a small number of simple features.

Boosted classifiers tend to be uni-modal, in the sense that each stage carves away big swaths of the input space, making it difficult to represent separated spaces. An alternative is to partition the input data and train separate classifiers on each partition. How to carve the input space is non-trivial, and it requires a clustering process in both the input and classifier space. The result is a set of classifiers, each with a different area of expertise.

It is also possible to use a variation on this method to convert a cascade classifier into a broad and shallow decision tree instead of a deep degenerate tree. This modification can generate significant speedups in run time.

12.2 Artificial Neural Networks

Neural networks are structures of interconnected nodes.

- Information flows between the nodes
- A weight is associated with each input to a node
- The output of a node is a function of its weighted inputs
- For general-purpose ANNs, the function at each node is nonlinear

A feedforward ANN is a subset of the possible ANN structures. A feedforward layer has three distinct types of nodes: input, hidden, and output.

- Information in a feedforward ANN starts from an input layer, to which an input sample is applied.
- The information then flows through one or more hidden layers
- The result of the computation is accessed in the output layer.

A feedforward ANN is like a combinational circuit in digital logic; its ouput is purely a function of its input. There are classes of ANNs, recurrent networks, that function as sequential circuits with the outputs being a function of both the input and the prior output of the ANN.

The function executed by a node to generate its output O is generally of the form

$$O = \Phi\left(\sum_N w_n I_n\right) \tag{119}$$

The nonlinear function $\Phi()$ is generally some kind of squashing function that produces a value within a fixed range from an arbitrary range input. The sigmoid function, for example, is commonly used.

$$\Phi(x) = \frac{1}{1 + e^{-x}} \tag{120}$$

The sigmoid function takes values on the real number line and modifies them to be in the range $(0, 1)$.

Example

Below is a simple example of a feedforward neural network

- Two input nodes accept the input pattern.
- Two hidden nodes are fully connected to both input nodes.
- One output node provides the result of the network

Supervised Training

While one could create an artificial neural network by hand, it is very difficult to set the weights of a network appropriately. Instead, we want to use a training method that learns the weights from a training set consisting of labeled examples.

- Training set: a set of labeled data containing examples of all classes the network needs to see.

- Testing set: a similarly labeled data set on which the network never trains, but which is used to evaluate the network's performance.

The process of training that is common used for feedforward neural networks is called backpropagation. It is a gradient descent algorithm that modifies the weights to improve the performance of the network on the training set.

The process of training using a single pattern is as follows:

- Apply the training pattern to the network
- Calculate the output of the network and compare it to the desired output
- Calculate how to change the weights going into the output layer to make the actual and desired outputs closer.
- Move back through the network, calculating how to change the weights at each layer.

Overall, a common version of backpropagation is:

- Apply all of the training patterns as above and store the sum of the weight changes
- Apply the weight changes to the network, modified by a learning rate
- Repeat thousands of times

Applying all of the training patterns to the network is called on epoch. Training usually takes thousands of epochs. There are many variations on backpropagation.

Algorithm for a single training pattern

1. Execute a forward pass on a training pattern to calculate the output of the network O.
2. Calculate the mean squared error between the output O and the desired output T.

$$E = \frac{1}{N} \sum_{i=1}^{N} (O_i - T_i)^2 \tag{121}$$

3. Calculate the weight changes required to move O closer to T
   - Weight changes are calculated using the **generalized delta rule**

$$\Delta w_{ji} = \eta \delta_j o_i \tag{122}$$

     - w_{ji} is the the weight going from node i to node j
     - Δw_{ji} is the change that should be made to w_{ji}
     - η is the learning constant, and governs the magnitude of the weight changes
     - δ_j relates the weight to its influence on the output
     - o_i is the output of node i
   - The weight change ought to be proportional to the amount of change in the overall error realized by changing the weight. In other words, we need to know the derivative of the error with respect to changes in the weight.

$$\Delta w_{ji} \propto -\frac{\partial E}{\partial w_{ji}} \tag{123}$$

- To figure out this derivative, we need to have a complete expression connecting the inputs and outputs of a node. The overall input to a node is a weighted sum of the outputs of all connected nodes.

$$I_j = \sum_i w_{ji} o_i \tag{124}$$

- Given the expression for the overall input to a node, the output of a node is simply

$$o_j = \Phi(I_j) = \frac{1}{1 + e^{-I}} \tag{125}$$

- The derivative of the sigmoid function is simple to calculate.

$$\Phi'(x) = \Phi(x)(1 - \Phi(x)) \tag{126}$$

- Now we can divide the derivative from above into two parts, reflecting the change in the error with respect to the input of a node and the change in the input to a node with respect to a single weight.

$$-\frac{\partial E}{\partial w_{ji}} = -\frac{\partial E}{\partial I_j}\frac{\partial I_j}{w_{ji}} \tag{127}$$

- The second term in (127) is the derivative of the input equation (124)

$$\frac{\partial I_j}{\partial w_{ji}} = \frac{\partial}{\partial w_{ji}} \sum_i w_{ji} o_i = o_i \tag{128}$$

  and explains the o_i in the generalized delta rule.

- The δ term in the delta rule corresponds to the remaining partial derivative, which we can again expand.

$$-\frac{\partial E}{\partial I_j} = -\frac{\partial E}{\partial o_j}\frac{\partial o_j}{\partial I_j} = -\frac{\partial E}{\partial o_j}\Phi'(I_j) \tag{129}$$

- The second term of (129) is given by $\Phi'(I_j)$. The first term is different for output units and hidden units.

- For an output node, the differential of the error term with respect to its output is given by (130).

$$\frac{\partial E}{\partial o_j} = \frac{\partial}{\partial o_j}\frac{1}{N}\sum_{i=1}^{N}(O_i - T_i)^2 = -(t_j - o_j) \tag{130}$$

  In other words, the weight change for a link coming into an output node depends upon how different the output is from the target.

  Combining (130) and (129), the delta value for an output node is given by

$$\delta_j = (t_j - o_j)\Phi'(I_j) \tag{131}$$

- Computing the delta value for a hidden node is more challenging, because it must take into account all of the ways in which a single weight can affect the different between the network outputs and the target.

Variations on Backpropagation

Methods of improving training:

- Boltzmann training: similar to simulated annealing. Add a random offset to the weights that decreases over time as training progresses.
- Momentum: add a fraction of the weight change in the prior step to the current step

Alternative squashing function, the hyperbolic tangent with a range $(-1, 1)$

$$\text{atanh}(bv) = a\left[\frac{1 - e^{-bv}}{1 + e^{-bv}}\right] = \frac{2a}{1 + e^{-bv}} - a \tag{132}$$

12.2.1 Example: ALVINN

Key factors in ANN design

- Architecture complexity: big enough to learn the task, small enough to train and run
- Creating adequate training and testing sets: both quantity and coverage of likely inputs
- Confidence measures

ALVINN: steering direction from images

Architecture

- 30x32 input image, scaled from the original image
- Four, fully connected hidden units
- 30 output units arranged linearly, trained to output a Gaussian

Training

- Person drives for 5 minutes at speed
- Take each image and rotate and translate the image using knowledge of the ground plane
- Varying the input images creates a sufficient training set for the task

Performance

- ALVINN drove at up to 55mph
- ALVINN drove 98% of the way across the country

Confidence

- Shape of the output indicated whether the network was confident: did it look like a Gaussian?
- Also tried training an output retina to mirror the input despite going through four hidden nodes. When the output didn't look like the input, the network didn't understand the input

Hidden Unit Sensitivity Analysis

Variation: use confidence metrics and multiple networks trained to different situations/roads

12.2.2 Face Detection

Rowley, Baluja, and Kanade, "Neural-Network Based Face Detection", CVPR 1996.
Rowley, Baluja, and Kanade, "Rotation Invariant Neural-Network Based Face Detection", CVPR 1998.

Architecture

- 20x20 input image (slightly smaller than Viola-Jones, but not much)
- Input was applied to multiple different receptive fields
  - Four 10x10 block receptive fields
  - Sixteen 5x5 block receptive fields
  - Six horizontal bar receptive fields
- 26 hidden units
- one output unit: face = 1, no-face = -1
- Hyperbolic tangent squashing function

$$\text{atanh}(bv) = a\left[\frac{1-e^{-bv}}{1+e^{-bv}}\right] = \frac{2a}{1+e^{-bv}} - a \tag{133}$$

Training

- 1050 hand selected and labeled greyscale face examples
  - Eyes and upper lip manually located
  - Image rectified (geometrically transformed) so all faces had the same orientation
  - Image intensity balanced and histogram equalized
  - Training image converted to a 20x20 block
- Each training example generated 15 variations by rotating and shifting each original image
- 1000 non-face images created using random pixel intensities
- Additional non-face images created during training
  - After N epochs, the network is run on a new batch of random non-face images
  - Add up to 250 new incorrectly labeled non-face images to the negative training set

Additional architecture characteristics

- For actual classification, the system used two separately trained networks
- The networks were applied to the image at different scales (pyramid processing)
  - Networks trained on same positive data
  - Since networks started with different weights and different negative examples, they learned slightly different functions
- Combine the results of the networks using boolean rules plus a merging step

  - Merge step on the output of each network separately:
    * Note where multiple detections occur
    * Count how many detections there are for a particular region (V-J method does the same)
    * If the number is above a threshold, detect a face
    * Collapse all of the detections to a single point (centroid)
    * Remove all overlapping detections
  - OR step: combine the results of the networks using an OR relationship (combine)
  - Merge step on the combined outputs using a different threshold for how many detections are required.

Performance

- Approximately a 90% recognition rate (missed only 1 in 10 faces, on average)
- 1 in 200,000 false positive rate (fairly high by current standards)

The Rowley-Baluja network was the first face detector to run in real time with an accuracy above 90%. Almost all face detectors prior to their work used skin color as the primary factor in detecting faces.

12.2.3 Autoencoding

An old idea, and previously a primarily theoretical usage of ANNs, is to build a network to encode its input as a smaller representation and then decode it back to its original form. The basic form of an autoencoding network is an input layer, one or more hidden layers that are smaller than the input layer, and then an output layer that is identical in size to the input layer.

An autoencoding network is trained to reproduce the input signal at the output. Since at least one hidden layer is smaller than the input–has fewer nodes–that means the network has to learn a compressed encoding of the input in order to properly recreate the output. Once trained, it is possible to split the network in half. The input layer and the first half of the hidden nodes are an encoding network. The second half of the hidden nodes and the output nodes are a decoding network.

One way to think of the first half of an autoencoding network is as a feature detector. In order to successfully train, the network has to learn something about the structure in the training data and use that structure to encode the input. Most natural images have a large amount of structure and redundant information that the network can use to encode the data. By training the network to learn that structure, we are, in effect, asking it to compute useful features of the training set.

Once we have a trained first half of the network, it is possible to make that network the input to a new network that is trying to learn a specific task, such as face recognition or object recognition. We can hold the encoding network fixed while the remaining network learns, then execute additional training that lets the whole network learn together. The idea and use of autoencoding networks has played a significant role in the rise of deep learning networks, in part because it offers a way to efficiently initialize and train very large networks.

12.3 Learning Eigenspace Manifolds for Pattern Recognition

Neural networks are one method of recognizing objects based on their overall appearance. The hidden nodes of a feedforward network distill the appearance of an object into a small number of variables (e.g. ALVINN used four hidden nodes to represent a 960 element input).

- Consider the weights of all of the inputs connected to a hidden node
- The weights multiply the input value, and the result is the weighted sum of the inputs
- Without the squashing function, the operation is functionally a dot product between the weight vector and the input vector
- The process of generating the input to each hidden node is identical to projecting the N-dimensional input onto an n-dimensional vector
- The output of each hidden node is an indicator of the strength of the signal along the vector
- The network learns which n-dimensional vectors are most important for capturing the input

By compressing the input in such an extreme manner, the network is clearly discarding lots of information. However, within the domain in which it was trained the network is able to function appropriately.

A fair question is whether we can directly calculate useful n-dimensional vectors for capturing the essence of a data set.

There are certainly methods for calculating efficient n-dimensional vectors for compressing a data set with minimal information loss. The question is whether these same vectors also function as an effective space in which to differentiate elements of the data set.

Principle components analysis [PCA] is the workhorse of appearance based recognition methods. PCA identifies the eigenvectors and eigenvalues of the covariance matrix of the data set. The eigenvectors represent the directions of primary variation in the data, and the eigenvalues indicate the importance of the corresponding eigenvectors. The eigenvector with the largest eigenvalue is the direction of the largest variation in the data.

When PCA is used to identify the vectors of primary variation, the expectation is that the directions of primary variation are also effective at differentiating members of the data set (i.e. recognition). This will not always be the case. For example, a person will often wear different clothes from day to day, creating significant variation in their appearance. Over the course of many days, therefore, clothes are not an effective method of identifying an individual despite the fact they are probably the visual factor that exhibits the most variation across individuals.

The key to making PCA produce vectors that are useful for the task, therefore, is ensuring that the visual input contains primarily features that are relevant to recognition. When considering something like face recognition, for example, overall intensity, the size of the face, and the orientation of the face are irrelevant to the recognition task.

Pre-processing

Common pre-processing steps include:

- Calculating some kind of oriented bounding box around the object that can be standardized across new images. For example, if the object can be segmented from the background, find the bounding box that is oriented along the axis of least second moment.
- Scale and orient the object into a canonical size and orientation: e.g. 20x20 axis-aligned
- Adjust the intensities to account for differences in overall intensity
  - One method is to normalize the image patch so that the length of the image is 1

$$L = \sqrt{\sum_{i=1}^{N} x_i^2} \tag{134}$$

$$\hat{x}_i = \frac{1}{L} x_i \tag{135}$$

  - A second method is to subtract off the mean and execute a histogram equalization.

    Histogram equalization uses the cumulative intensity distribution to convert the histogram into a uniform histogram. For example, consider an image patch with 400 pixels with each pixel having a range of 0..255. If we were to create an equalized histogram with 8 bins, then we would sort the 400 pixels in increasing order, put the first 50 in bin 1, the next 50 in bin 2, and so on. Then we would recolor the image so that all the pixels in each bin had equal intensities. The histogram may not be completely equalized, because all pixels of a certain original value should end up in the same bin.

Principle Components Analysis [PCA]

Given: A data set D consisting of N vectors $x_i, i \in \{0 \ldots N-1\}$ of length M

The process of generating the principle components of a data set are as follows:

1. Calculate the mean of the data set μ_d
2. Subtract the mean from each member of the data set

$$\hat{x_i} = x_i - \mu_d \tag{136}$$

3. Create an M x N matrix A, where each column of A is one of the examples from the data set

$$A = \begin{bmatrix} \uparrow & & \uparrow \\ \hat{x}_0 & \ldots & \hat{x}_{N-1} \\ \downarrow & & \downarrow \end{bmatrix} \tag{137}$$

4. Execute singular value decomposition on the matrix A.

$$A = UWV^t \tag{138}$$

   Properties of SVD:

   - Given: A is an NxM matrix.
   - U is an NxM matrix with N-element orthogonal columns.
   - W is an MxM diagonal matrix, generally represented as an M-element vector.
   - V is an MXM matrix with M-element orthogonal columns.

   The columns of U form an orthogonal basis for representing the columns of A. They are identical in direction to the eigenvectors of AA^t. The columns of V form an orthogonal basis for representing the rows of A, and are identical in direction to the eigenvectors of A^tA.

   The diagonal elements of the matrix W are called the singular values of A, and are related to the eigenvalues of AA^t (and A^tA). The largest singular value corresponds to the primary direction of variation in the data. The second largest singular value corresponds to the next largest, but orthogonal direction, and so on. Singular values at or near zero correspond to the null space of A. The null space is the set of solutions to the linear system that equal zero.

5. Identify the columns of U that correspond to the $N-1$ largest non-zero values (there may be fewer). These are all of the eigenvectors of the data set.
6. Select the K basis vectors $e_j, j \in \{0 \ldots K-1\}$ that correspond to the largest K singular values. The selection of K depends upon the values of the singular values.
   - Normalize the $N-1$ singular values so they sum to one
   - In descending order, calculate the number of singular values K that are required to reach a target percentage of 1 (e.g. 90% or 95%)
7. The K basis vectors form a reduced basis space for the data set. Generally, K is small relative to N or M.

The PCA process above provides the basis vectors for representing the data set. Now we have to map the training data into the reduced basis space. For each training vector x_i

1. Subtract the mean data vector μ_d, to get $\hat{x}_i = x_i - \mu_d$
2. Calculate the dot product of $\hat{x}_i$ and each basis vector e_j, producing a vector of K values that represent the location of the training sample in the reduced basis space. The space is generally called an eigenspace.

Any new unknown input goes through the same process:

1. pre-process the data
2. subtract off the data mean
3. project it into the eigenspace

Every training example is located at a point in the eigenspace. We can now apply whatever pattern recognition techniques we want to the data.

- Take all the training examples for a particular object and calculate a mean location in the eigenspace as an exemplar.
- Use all of the training examples as exemplars for something like K-nearest neighbor matching.
- Build a decision tree in eigenspace using the training data to build the tree.
- Train a neural network or other classifier to attach a label to the data.

In eigenspace analysis, there is also the concept of universal and specific eigenspaces.

- The universal space is the eigenspace built from all of the training examples of all objects in the database.

  Object recognition usually occurs in the universal space, as it represents the appearance of all of the objects relative to one another.

- A specific space is an eigenspace built from all of the training examples of a single object.

  If a specific space, or object space, is built from many views of an object from known orientations and illumination conditions, then the pose of the object can be determined in the object space.

Reprojection

Any point in an eigenspace can be converted back into the original space.

- The vector describing the location in eigenspace is $y = [y_0 \dots y_{k-1}]'$
- The reprojection is calculated as a weighted sum of the eigenvectors

$$\hat{R} = \sum_{i=1}^{K} y_k e_k \tag{139}$$

- The reprojection $\hat{R}$ is a difference vector, so to obtain the real image, add back in the data set mean μ_d

$$R = \hat{R} + \mu_d \tag{140}$$

- R may still be a normalized, or histogram equalized version of the input, so to visualize it you need to rescale it back to [0, 255] appropriately.

  Reprojection is useful for determining if the input image is a member of the class of objects to be recognized. For example, an eigenspace generated out of faces only captures the essential properties of face images.

  - Any input image will project into the face eigenspace and produce a set of K values
  - The values may be far away from any real faces, in which case it can be rejected
  - The values may be close to a real face, in which case it needs to be verified as a face
    - Reproject the projection of the input image back into the input space
    - Compare the reprojection to the original input
    - If they are similar, then the input was appropriately encoded
    - If they are different, then the input is not part of the face space

12.4 Decision Trees

Decision trees are a pretty standard machine learning method

- There are standard heuristic algorithms for building useful trees that generalize well
- You can also build random trees
- You can also build forests of random trees as an ensemble classifier

Microsoft Kinect

- Build very large trees: depth = 20
- Build a forest or 3 random trees
- Training at a node
  - Randomly propose a set of splitting candidates (feature/threshold)
  - Calculate the information gain of each splitting candidate
  - If the the largest information gain is sufficient, split into left and right and recurse for the left and right subsets.
- Features: difference of two depth image pixels relative to the pixel of interest
- Goal: pixel-wise classification of body parts
- Training Set: synthetic depth silhouettes

To get body part locations: use mean-shift to identify modes in the distribution, starting at all high probability body part pixels. The weight of the pixels reaching each mode is used to determine whether to incorporate the mode into a final estimate. They also found a bottom-up clustering algorithm worked almost as well and ran faster.

Interesting points on training

- The depth of the tree is an important factor in determining the size of the training set and accuracy of the system
- The amount of training data is important, especially for the larger trees
- The flexibility of the features was important to accuracy, but also required more training data

12.5 Deep Networks

Hinton: 5 ways to train an ANN

- Pretend you don't have to and use small ANNS where you can set the weights manually
- Search randomly
- Learn layers of feature detectors and classify later (how do we guide this?)
- Backprop
- Forward/Backward or Recognition/Generation or Wake/Sleep learning

The biggest problem with backprop is that it is really, really slow and the error space of a large network is so huge it's hard even to find a decent local minimum. Think about the amount of adjustment made to a 10-layer deep node with hundreds of nodes per layer between it and the output. The reflection of any single weight is so muddled that progress towards any optimum is very slow. You need some method of training large networks faster and more efficiently, dropping them more reliably into a decent starting location for further gradient descent optimization.

Common non-linear mappings.

- A sigmoid is a nice clean function, the most interesting part of a sigmoid is the linear section in the middle. Changes in the saturated sections are small and gradient descent works slowly there. Each node effectively becomes a logistic regression on its inputs $S(z) = S(\vec{w}^T\vec{x} + b)$

$$\text{Sigmoid}(x) = S(x) = \frac{1}{1 + e^{-x}} \tag{141}$$

$$S'(x) = S(x)(1 - s(x) \tag{142}$$

- The rectified linear function [ReLU] has recently been shown to produce better performance on large networks. As a piecewise linear function, it seems to be more friendly to gradient descent algorithms. There are also machine instructions that do saturated math (anything less than zero is zero). This is a slightly different result than a logistic regression $\text{ReLU}(z) = \text{ReLU}(\vec{w}^T\vec{x} + b)$.

$$\text{ReLU} = max(0, x) \tag{143}$$

$$\text{ReLU}'(x) = \begin{cases} 0 & x \leq 0 \\ 1 & x > 0 \end{cases} \tag{144}$$

Treat each node as implementing the following, where $\vec{x}$ is the set of inputs to the node, $\vec{w}$ is the weight attached to each input, and c is a learned offset parameter.

$$f(\vec{x}; \vec{w}, c) = \max(0, \vec{w}^T\vec{x} + c) \tag{145}$$

An intuitive way of thinking about the ReLU function is that the function maps a point in R^N, where N is the number of input weights, onto the closest point located on the vector defined by the weights $\vec{w}$, then moves any mapped point on the vector (number line) with a value < 0 to 0. So it is almost identical to

mapping a point onto a single eigenvector, except that there is an anchor for the eigenvector and no mapped points are allowed behind the anchor.

12.5.1 Common Cost Functions

- Want to maximize the log likelihood of the model, which turns out to be equivalent to minimizing the squared error, in theory.
- One problem with maximizing squared error is that the model can get saturated as it approaches a category value like 0 or 1. Many non-linear activation functions can't represent 0 or 1 but can get arbitrarily close. When a network is functioning in this saturated region, training and change is very slow and it can lead to a lack of generalization.
- Softmax is the term used for outputs that are trained to be 1 of N high in order to select between categories. Generally, the output nodes include a sigmoid function to convert the activation into something like a probability. One aspect of the Softmax is that it has an extra degree of freedom. For example, if you have two categories, training two output nodes contains no more classification information than training one for a two category problem. The issue is that Softmax will produce the same classification output if all output nodes are shifted by some constant. If this over-representation of the output is not taken into account in the network structure, then there are an infinite (theroetically) number of possible output functions the network could learn, each shifted by a constant.

12.5.2 Locally Connected Networks and Convolution

One issue with increasing network size to handle larger inputs (e.g. larger images) is that fully connected networks become very expensive. As Rowley showed, using smaller retina that cover a subset of the input space can be an effective method of learning features relevant to the problem without having the exponential growth in network weights.

In a locally connected network, each node of a hidden layer is connected to a subset of the nodes on the prior layer. One question to ask is whether the weights connecting each hidden layer node to its predecessors on the prior layer need to be different. Can the hidden nodes share the same input weight vectors, just be connected to different prior nodes? In other words, can we learn the weights for a single filter that convolves with all possible sub-windows of the input to produce the input values for hidden layer?

So how would we train the weights for such a layer? If we are trying to train the first hidden layer of a network, then the inputs are small windows of pixels values. One method of training the weights for the first layer is to train the convolution filter weights as an autoencoder. Create an autoencoder network with an 8x8 window as input, a set of K hidden nodes, and an 8x8 output node. Then train the network to recreate the input at the output. Once trained, you have K features, each generated by a specific convolution filter. One benefit of this approach is that a convolution filter is fast to implement in hardware (a bunch of product and sum operations over a regular pixel grid).

Once trained, you have K sets of hidden node values, with each set representing the result of applying a different learned convolution filter to the input.

12.5.3 Pooling

The issue with generating so many features is how to use them in classification. In classification–as opposed to feature generation–fully connected networks are useful. However, with so many features to choose from

we again run into the problem of an explosion of weights to train. Hence, we want to somehow pool the results of the convolution layer.

A pooling layer takes the result of some section of the prior layer and computes some statistic of that area such as the mean or max value. If the pooled area is contiguous and the pooling process includes only a single feature, then the pooling step is effectively a translation invariant calculation: the input can move around inside the contiguous area, but the mean or max of the filter responses in that area should stay constant.

There are other methods of pooling that might offer different forms of invariance, such as pooling values across features. If the features are rotated versions of one another, for example, then pooling creates a rotation invariant layer.

12.5.4 CNN Design

A Convolutional Neural Network is a series of convolution layers and pooling layers followed by one or more fully connected layers as in a traditional feed-forward network. The convolution and pooling layers provide a set of learned features that are used by the fully connected network to do discrimination/recognition. The design is intended to force the network to learn structures in the training data, generate invariance to particular types of modifications of the input, and then give the recognition section of the network features that have relevant interpretations given the input data.

Another way of thinking about the problem is as projection of local sections of the input onto learned basis vectors, then computation of local statistics of the data in the new basis space. In both cases, this process is generally repeated multiple times. In theory, this creates more and more abstract features that capture multiple layers of structure in the input.

Input: N x N x C image, where C is the number of channels (generally 1 or 3)

Convolution Layer: K filters of size M x M x Q, where M is the size of the filter and Q is the number of channels used ($Q \leq C$). If we assume the convolutions are applied only to pixels where the filter fits fully in the image, then the result of the convolution layer is a set of K x (N - M + 1) x (N - M + 1) x Q values. Each of the K sets is generally called a map.

Pooling Layer: Each map is typically subsampled over a P x P contiguous region. Common values for P are in the range of 2 to 5.

Nonlinearity: The nonlinearity function, including the bias term, can be applied either before or after the pooling layer.

Initialization: train the K filters of the convolution layer using an autoencoding network training process using standard backpropagation. This gives the convolution layers useful initial values so they are likely to place the entire network in a reasonable location for global gradient descent training.

Pooling Layer Training: Back propagation training through a pooling layer requires up-sampling of the delta function. For mean pooling, this involves sending the same value to all of the prior outputs from the convolution layer. For max pooling, this means sending the information only to the max node.

Convolution Layer Training: Since convolution is theoretically flipped and reversed, the back propagation also has to flip and reverse the information to feed the data to the weights. In order to keep the number of weights constant, the convolution weights are all tied. Therefore, we have to collect all of the proposed changes to the weights and make a single update after calculating all of the individual proposed changes.

12.6 Recognition/Generation Training

Using only back propagation to train a network only models the way the output depends on the input. It does not encourage modeling latent variables–variables that encode structure in the training data. As a result, it is easy for the network to key in on irrelevant features that accidentally relate the output to the input.

To initialize many layers of convolution/pooling, we can iteratively work up the chain of layers from the input layer to as many layers as we want to use. At each layer, we hold all prior layers constant, then train the new layer to be an autoencoder for the prior layer.

An alternative idea: training on residuals

12.6.1 Handwritten Digits Example

LeCun, "Gradient-Based Learning Applied to Document Recognition", Proc. of IEEE, 1998.

Task: given a 32x32 input image of a hand-written digit

Network Design

- C1: 6 28x28 feature maps using 5x5 kernels
- S2: Pool 2x2 blocks. Output is 6 14x14 maps
- C3: 16 10x10 feature maps using 5x5 kernels. Each kernel connects with several feature maps from S2.
- S4: Pool 2x2 blocks. Output is a 5x5 map with 16 feature maps.
- C5: 120 feature maps with the kernels fully connected to all 16 feature maps in S4.

Layer F6 has 84 units. Each unit uses the Radial Basis Function (not a sigmoid). An RBF works by comparing the actual output vector to a desired output vector, which can be either hand-specified or learned.

- The output layer was designed so the proper output for each character or digit was a stylized version of it on a 7x12 bitmap (84 nodes).
- Digits that are similar, like o, O, and 0, will get similar scores, which is a good property to have if the output is going to a higher level system identifying words.

12.6.2 Object Recognition Using CNN and SVM

Huang, LeCun, "Large-scale Learning with SVM and Convolutional Nets for Generic Object Recognition", CVPR 2006.

Task: given left and right stereo images of a scene with objects of interest

Network design

- Five feature layers:
  - C1: 12 5x5 kernels to generate 8 feature maps; 4 feature maps use kernels in the left and right images. Output is eight 3-D 92x92 feature maps.
  - S1: Pooling layer subsamples 4x4 areas from C1, with 16 learned parameters to weight the subsampling. Output is 8 maps of size 23x23.
  - C3: 96 6x6 kernels to generate 24 feature maps. Each feature map integrates from two monocular inputs and two stereo inputs. Output is 24 18x18 feature maps.
  - S4: Pooling layer subsamples 3x3 ares from C3. Output is 24 feature maps of size 6x6.
  - C5: 80 or 100 6x6 kernels (whole of S4). The output is a single 100- or 80-dimensional vector.
- Output layer is a fully connected 5-node layer that executes a 1 of N classification function.

The layers C1-C5 form a feature detector. The final object recognition system applied all of the training data to C1-C5 and then used an SVM to train a classifier on the resulting encoded vectors.

On a more complex task, the CNN-SVM combination worked better than the CNN alone. On a simpler task, the CNN work about as well.

12.6.3 Residual Learning

He, Zhang, Ren, and Sun, "Deep Residual Learning for Image Recognition", LSVRC, 2015.

The fundamental idea of residual learning is that the function being learned is redefined so the desired output incorporates the input. In particular, given a desired input/output mapping $H(x)$, have the network instead learn the function $F(x) = H(x) - x$. Another way of thinking about it, is that the network is trained to generate $F(x)$ and then x is explicitly added to the layer output to obtain $H(x)$.

The hypothesis is that it is easier to learn a function of the form $H(x) - x$ than to learn $H(x)$ directly. Empirically, this seems to be true, as residual deep learning networks are currently outperforming all others and enable extremely deep networks with hundreds of layers. Intuition says that adding the x term both encourages the network to learn the structure of the input and anchors the function $H(x)$ relative to the input. From a control theory point of view, if the desired function $H(x)$ is the identity function, then proper training means driving the output $H(x) - x$ to zero within finite time.

Experiment 1: ImageNet

- 1000 categories; 1.28 million training images; 50k validation, 100k validation sets
- 18-layer plain net: no shortcuts
- 34-layer plain net: no shortcuts
- 34-layer Residual net: shortcut per layer
- Train all three from scratch
  - 34-layer plain network has worse validation error and training error than 18-layer plain network
  - 34-layer ResNet had better properties than 18-layer ResNet, both were better than plain networks

Experiment 2: identity shortcut v. others

- Identity shortcut is almost as good as adding weights to the shortcut path

Experiment 3: Deeper Bottleneck Architectures on ImageNet

- Have three layers of weights, with one convolution, in between the shortcuts.
- 50 layers
- 101 layers and 152 layers
- 50/101/152 layer networks are more accurate
- Ensemble of six networks of different depths (2 @ 50, 2 @ 100, 2 @ 152) won the competition

Even pushed their system to 1202 layers on a smaller data set; they think it overtrained, because it didn't generalize as well, though it learned the training set as well as other networks.

Why?

They analyzed the standard deviation of the weights at the various layers of the network. What they found was that, on average, the residual networks had smaller weights with a lower standard deviation. Their argument is that each layer of the network learned a little less, or was responsible for a simpler function that without the shortcuts. Therefore, the networks were able to train more effectively in the same number of epochs with the same training set.

12.6.4 Video/Image Labeling Using Deep Networks

Karpath, Toderici, Shetty, Leung, Sukthankar, Fei-Fei, "Large-scale Video Classification with Convolutional Neural Networks", CVPR 2014.

Database: Sports1M, 1 million YouTube videos belonging to a taxonomy of 487 classes of sport.

Issues with video

- Size of the data
- Presenting the input to the network
- Size of the network

Typical approach to video classification

- Local visual feature detection: dense or sparse features
- Combination into a fixed-size video-level description. As example process might be quantizing the features using k-means and then generating histograms over time and space to represent the video (not so different from Varma and Zisserman Texture analysis).
- Training a classifier, such as an SVM, to differentiate the video types.

How to configure a network for video analysis if we view the video as a bag of short, fixed-sized clips

- Single frame: produce features for single frames and collect them
- Late fusion: separate convolutional stacks analyzing 2 or more frames connected at the top layer
- Early fusion: a single convolutional stack analyzing 2 or more frames
- Slow fusion: merging convolutional stacks covering 2 or more frames

Unique aspects of the approach

- View the video as a bag of short, fixed-sized clips
- Found that using smaller images to improve run-time reduced performance
- Found that using smaller network layers to improve run-time reduced performance
- Split the input into **context**–a low resolution stream–and **fovea**–a higher resolution window in the middle of the current frame.
- Context stream receives the input at quarter resolution (half in each dimension)
- Fovea receives the center quarter of the image (same size as the context stream, but full resolution)
- Both the context and center streams end at a layer of size 7 x 7 x 256

Architecture: [CNP - CNP - CCC][CNP - CNP - CCC] - FF

Normalization layers: adjust the values of the outputs according to local responses

- Divide each kernel response in a convolutional layer by an expression proportional to the sum of the responses in a neighborhood
- Similar to lateral inhibition in a biological network
- Improves performance when using ReLU nodes for non-linearity

Preprocessing and augmentation

- Crop all images to a 200x200 center region
- Randomly sampling a 170x170 region
- Randomly flip images horizontally
- Subtract the mean intensity of all pixels (117)

Train using Downpour Stochastic Gradient Descent

Interesting results

- Networks do much better than feature-based systems
- Differences among networks are not very significant
- Slow fusion network did the best of the different configurations
- Context stream tends to learn color features
- Fovea stream tends to learn greyscale filters
- Motion aware-networks are more likely to underperform when there is camera motion

They also retrained the network on the UCF-101 data set and it works reasonably well.

13 Vision and Graphics

13.1 Video textures

An interesting application of computer vision is the creation of video textures: infinite length video sequences that are not simple loops, generated from short video sequences.

- Take a fixed length video sequence
- Find places where one location in the video looks like a previous frame
- Figure out where loops can exist
- Traverse the graph in a random way to generate changing sequences

Using video textures it is possible to:

- generate video sequences of any length
- modify different parts of the video differently
- create interactive animated sprites

Analyzing the video content

- Analyze the video clip to identify similar scenes
  - Want to find all the places in the sequence that are similar
  - Create a matrix showing the similarity of each frame to every other frame
  - The video can be divided into sections, and a matrix calculated for each section independently
- Distance measures: start with L_2 distance between frames (sum of squared differences)
  - Dynamics needs to be maintained in the image (think swinging pendulum)
  - Measure L_2 distance in surrounding frames as well
  - Overall distance is a weighted sum of the distances of the frame and its neighbors in time
- Distances also need to reflect whether transition to a frame would cause a dead end
  - The transition to a particular frame might be excellent, but there may be no way back
  - Consider dead ends to have high costs
  - Propagate those back through the matrix
  - Good transitions to a frame that leads to a dead end should have a high cost
  - Good transitions to frames that do not lead to dead ends should have a low cost
- Keep only transitions that are local minima
- Keep only transitions that are sufficiently good
- Finally, prune down to a small number of good transitions.

Q-learning to discover the cost of future decisions
(a form of reinforcement learning)

- Initialize the matrix of transition costs to the SSD between frame i and frame j with some weighted window around both frames.

$$D_{ij} = \sum_{k=-m}^{m-1} w_k D_{i+k,j+k} \tag{146}$$

- The initial probability of making a transition is proportional to the transition cost.

$$P_{ij} \propto e^{-D_{i+1,j}/\sigma} \tag{147}$$

- The denominator σ is set to a small multiple of the average non-zero D_{ij} so the likelihood of making a transition at any given frame is low.
- In order to take into account future decisions, we want to incorporate their cost. The coefficient p manages the relative weight of the current transition, and the coefficient α manages the importance of future decisions. In the paper, α is set to a value in the range [0.99, 0.999].

$$D'_{ij} = (D_{ij})^p + \alpha \sum_k P'_{jk} D'_{jk} \tag{148}$$

- While we can iteratively calculate a solution, convergence is slower than if we simply use the minimum future cost (best case transition).

$$D'_{ij} = (D_{ij})^p + \alpha \min_k D'_{jk} \tag{149}$$

- Algorithm
  - Initialize the transition matrix costs to D_{ij} without considering future decisions.
  - For some number of iterations
    - For each row, from last to first, update the cost of the row, holding the rest of the matrix constant.
    - Letting $m_j = \min_k D'_{jk}$, the update equation is:

$$D'_{ij} = (D_{ij})^p + \alpha m_j \tag{150}$$

  - Terminate when the matrix stabilizes.

Generating Sequences

Generating an infinite sequence

- Pick a starting point.
- Play the sequence until the next transition.
- Take the transition with some probability
- If the transition is the last backwards transition, take it with probability 1.

Generating a sequence of a particular length

- Use a search technique to find a sequence of transitions that fill up the time
- The section of video past the last backwards transition is potentially useful

Making smooth transitions

- Linearly combine frames from before the transition and after to generate a smooth sequence.
- Compute optical flow and try to morph the pre- and post- transition frames together
- For video that has been divided, use feathering to combine them back together.

Video (SIGGRAPH)

13.2 Stylizing Images and Video Sequences

Litwinowicz, "Processing Images and Video for an Impressionist Effect", SIGGRAPH 1997.

Non-photorealistic rendering of existing images and video streams requires analysis of the information in the image, often using standard computer vision techniques. Litwinowicz was one of the first to make use of CV analysis to generate coherent video streams in an impressionist style. In order to create convincing and smooth results, he had so solve a number of problems.

- How to place brush strokes
- How to align brush strokes
- How to move brush strokes over time
- How to create and destroy brush strokes over time

Initial algorithm

- Create a grid
- Jitter the grid to randomize stroke placement
- The color of the brush is chosen based on a smoothed version of the image
- The color is jittered in RGB and intensity
- The orientation is chosen and jittered
- The length is chosen and jittered
- When the stroke is drawn, it is clipped to strong edges (sobel magnitude)

Adjusting the orientation of edges

- Using a larger Sobel kernel (e.g. 7x7), calculate the orientation field
- Zero out orientations when the magnitude is small
- Interpolate gradient orientations into low magnitude areas
- Interpolate the gradient orientations around the brush center to get a direction
- Use the normal to the gradient direction as the brush stroke orientation

In all cases, store the brush stroke perturbations (not the actual colors) with the stroke to enable coherence from frame to frame. The brush strokes are kept sorted in drawing order from back to front. The ordering should start out random.

Smooth Video

- Triangulate the brush stroke centers (Delaunay Triangulation)
- Calculate optical flow to get the local motion of objects in the scene
- Use the optical flow field to calculate brush stroke center motions
- Use the triangulation to calculate where there are large triangles
- Subdivide triangles that are too large to create new vertices
- Place new brush strokes where there are new vertices
- If two brush strokes are too close, eliminate the one that is further behind
- Sort the new brush strokes from front to back
- Uniformly and randomly insert the new brush strokes into the old sorted list
- Recalculate brush stroke orientations according to the new gradient field
- Recalculate the base color according to the new smoothed image
- Re-apply the stored perturbations

13.3 Real-Time Video Abstraction

Winnemoller, Olsen, and Gooch, "Real-Time Video Abstraction", SIGGRAPH 2006.

This work seeks to create cartoon style versions of images and video sequences.

Algorithm

- Conversion to CIE-Lab space
- Bilaterial filtering, possibly multiple times
- Luminance quanitization into a fixed number of values
- Edge calculation
- Combine Luminance quanitzed image and Edge image
- Convert back to RGB
- Image warping to sharpen the image

Calculating gradients

- Use Difference of Gaussian [DOG] edges with a smooth threshold
- Take the difference of two Gaussians of different sizes, then threshold
- The two Gaussians are related by $\sigma_e = \sqrt{1.6}\sigma_r$

13.4 Converting an Image to Pen and Ink

Salisbury, Wong, Hughes, Salesin, "Orientable Textures for Image-Based Pen-and-Ink Illustration", SIGGRAPH '97

Inputs to the system:

- Tone image, a direction image, and a stroke example set
- Tone image can be a photograph
- Direction image is painted by the user, but could also come from gradient orientations
- Strokes can come from a library or be created by the user

Difference image: a blurred version of the rendered image subtracted from the tone image

- The strokes are placed into the image according to a difference image
- Importance image: current difference image divided by the original difference image value
- Draw strokes according to the importance image
- Stop drawing when the maximum value in the importance image is below a threshold

Drawing a stroke

- Random select a stroke from the library
- The stroke's angle at each point should match the direction field
- The control hull is modified to pass through the desired locations (e.g. a cubic spline)
- Pick a random control point and pin it, then grow the control points outwards
- Clip the strokes when the direction field changes quickly (e.g. edges)
- Clip or remove strokes when they would make an area too dark
- Update the difference and importance images after each stroke.
  - Use an approximation of the blurred stroke (a thick fuzzy line)
  - Don't add then blur, just subtract a heuristically modified kernel from the difference image

13.5 Cubist-Style Rendering

Collomosse and Hall, "Cubist-Style Rendering from Photographs", IEEE Trans. on Visualization and Computer Graphics, 2002.

Key issues

- Cubism focuses on salient features, how do we define salience?
- How should salient features be selected?
- How should the angular geometry be reproduced?
- How should the final composition be rendered?

Method input: one or more source images

- Register all source images so that objects of interest fall upon one another
- Threshold using color to partition foreground and background
- Translate images so the first moments of foreground are coincident
- Clip the images to a uniform width and height
- Result: pixels at the same (i, j) location in any image correspond.

Identifying salient features in registered images

- Define salient as pixels that are uncommon in an image
  - Convolve the image with 5 first and second order derivatives at multiple scales (1, 2, 4, 8)
  - Each pixel is represented with a 20 element vector
  - Assume the vectors form a 20-dimensional Gaussian with a specific mean
  - Use Mahalanobis distance to look for pixels sufficiently far from the mean
  - Final step is to let a person identify clusters of interest that become features

Geometric Distortion

- Construct a continuous vector field over each source image
  - Fit a superquadric to the source image (foreground)
  - Target superquadric is closer to a cube
  - Generate the vector field as the warp from source to the target
  - Use the vector field to warp the image
  - Non-linear optimization method to obtain the warp

Composition

- Salient features are distributed evenly and in proportion to the original image set
- Features to not overlap
- Space between is filled with suitable non-salient texture

- Non-salient regions are broken up

Weight features selection by the fractional size of the equivalence class, then stochastically sample

- If a selected feature overlaps other features, remove the other features from consideration

To fill in the uncovered space

- If an unchosen feature is intersected by only one chosen feature, extend the chosen feature over the area
- If an unchosen feature is intersected by multiple chosen features, choose one chosen feature and extend it, avoiding already filled areas
- If two multiply intersected unchosen features intersect, arbitrarily choose one
- Once the chosen feature areas are selected, use a distance transform to color in the regions

Remainder of the image

- Use the segment size of salient regions to break up the non-salient areas into similar sized regions
- Perform segmentation using Fuzzy C-means
- The center of each region is then used to compute a Voronoi diagram
- Each fragment is rendered with a modified intensity gradient across it based on distance from the fragment center
- The gradient directions are calculated using graph coloring
  - Each fragment is assigned an integer in the range [1, t], where t is the number of colors (gradient directions) to be used
  - The gradient direction of the fragment is determined by the color of the fragment
- Convolve the non-salient regions with a low-pass filter to reduce edge boundaries

Finally, boost the contrast within the foreground using a form of histogram equalization.

Image Rendering

- Color quantization
  - Quantize in three areas: distorted salient regions, foreground of distorted image, background of distorted images.
- Brush Stroke Generation
  - Generate 3D strokes and map them onto the image using a z-buffer.
  - A number of stroke parameters enable different effects
  - Stroke orientation using gradient orientation

13.6 Image and Video Based Painterly Animation

Hays and Essa, "Image and Video Based Painterly Animation", ?

A variation on Litwinowicz.

Multiple layers of brush strokes

- Each layer corresponds to a layer of refinement
- Think of higher layers as having higher frequency information

Output can be much larger than the input image

Brush strokes have opacity

Motion: Use Black and Anandan optical flow

Brush stroke generation/removal

- brush strokes can fall off layers
- higher frequency brush strokes can move away from high frequency areas of the image
- As brush strokes come closer to another stroke or move away from important areas their opacity decreases and they are removed

Brush stroke orientation is generated by moving out from high strength edges, not the gradient field

- New brush strokes that are on high gradient areas are strong
- Strong brush strokes are the kernel of a radial basis function that determines other strokes
- Brush strokes are clipped to high gradients

Color

- Average color of a smoothed version of the input
- Sometimes adding noise to achieve certain effects

Overall, some refinements to the Litwinowicz methods

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 definitions, class declarations, prototypes, extern statements, and inline code. Header files should never declare variables or incorporate real code except in C++ class definitions.
- Libraries: contain pre-compiled routines in a compact form for linking with other code.
- Object files: object files are an intermediate step between source files and executables. When you build an executable from multiple source files, using object files 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.

```
int main(int argc, char *argv[]) {

  return(0);
}
```

The main function returns an int (0 for successful completion) and takes two arguments: argc and argv. The argument argc 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.

```
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.

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

Example: Defining a structure

```
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.

```
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.

```
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 files. Prototypes describe the name and arguments of a function so that it is accessible in any source file 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 files. The global variable declarations themselves ought to be in source files. Initialization of the global variables also need to be in the source files. If the declaration itself is made in the header file, 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 files. They are the only C code that belongs in a header file. In C++, methods defined within the class declaration are implicitly inline, but not necessarily. It is a good idea to only define methods explicitly declared as inline in the header file, especially for large projects.

Useful include files

Standard include files for C provide definitions 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 files 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 files, given below. In addition, the include file iostream is probably the most commonly used include file 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

```
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

```
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.

```
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 * 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.

```
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.

```
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.

```
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.

```
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

```
#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 same 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.