CS1114: Study Guide 2

This document covers the topics we've covered in the second part of the course. Please refer to the class slides for more details.

1 Polygons and convex hulls

A polygon is a set of 2D points (called vertices, as in a graph) connected by edges in a given sequence. In this class, we've been especially interested in a particular kind of polygon: *convex* polygons. A convex polygon is a polygon that has the property that for any two points A and B inside or on the polygon, the line segment connecting A and B is entirely inside the polygon. Figure **??** shows some examples of convex (and non-convex) polygons.

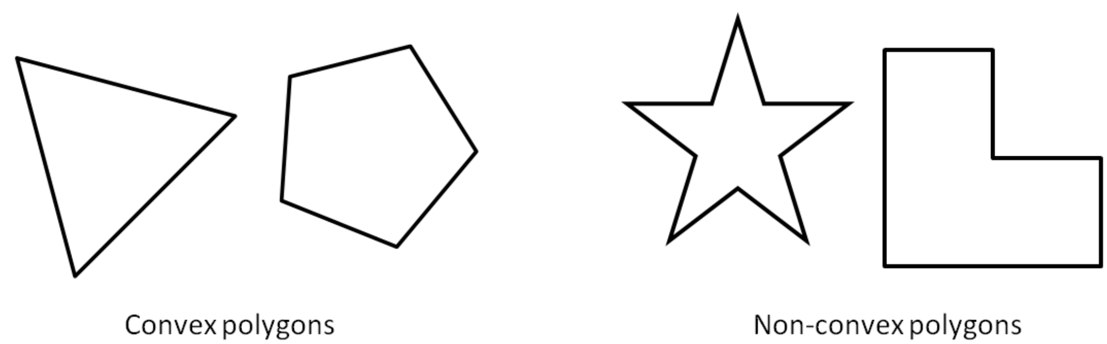


Figure 1: Some examples of convex and non-convex polygons.

This leads to the definition of a special kind of polygon called the *convex hull*. The convex hull of a polygon G is the smallest convex polygon that completely contains G. The convex hull will be made up of vertices of P (but not necessarily all vertices of G will be on the hull). Intuitively, the convex hull is what you would get if you took a stretchy rubber band, stretched it out to contain the polygon, then let it snap to its smallest possible shape. An example of a convex hull of a polygon is shown in Figure **??**. Naturally, the convex hull of a convex polygon is the polygon itself.

We can also define the convex hull on a point set (instead of a polygon). The convex hull of a point set P is the smallest convex polygon that contains all of the points. Again, the vertices of the convex hull will all be points in P. An example of a convex hull of a point set is shown in Figure **??**.

Note that an edge of the convex hull has the property that *all* points are on one side of the line passing through that edge. The converse of this is true as well: if a pair of points

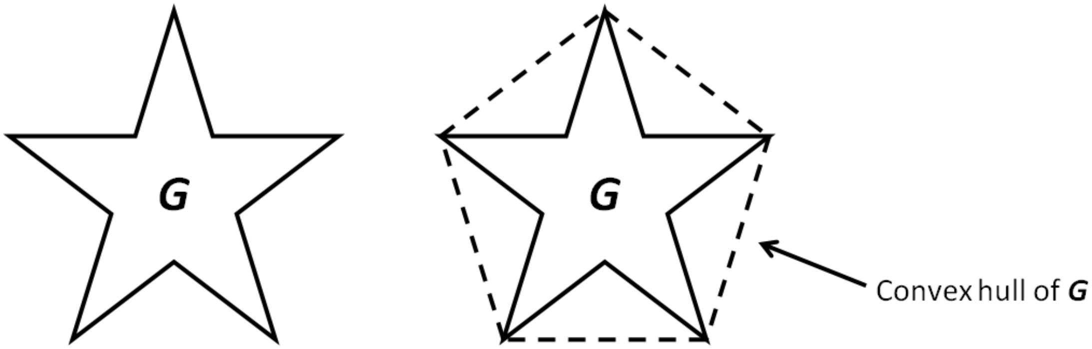


Figure 2: Convex hull of a polygon.

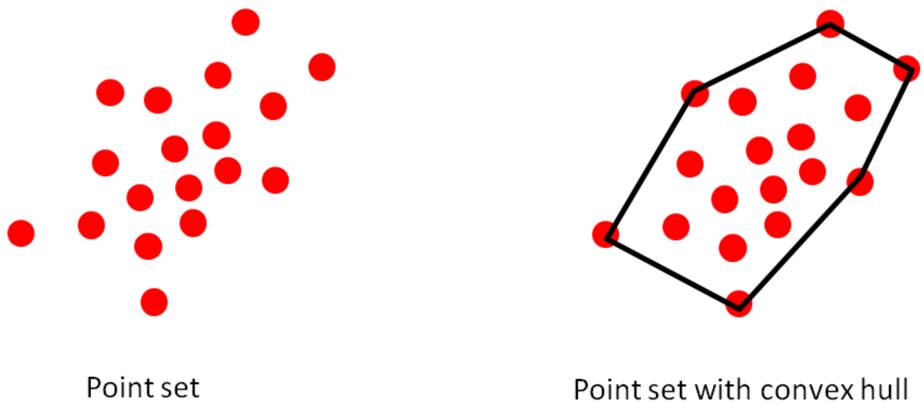


Figure 3: Convex hull of a point set.

A and B in the point set have the property that all other points are on one side of the line passing through AB, then the edge AB will be part of the convex hull.

Next, we consider how to find the convex hull of a set of points. Note that we can easily find a few points that are *definitely* on the convex hull: for instance, the leftmost point, the rightmost points, the uppermost point and the lowermost point. In fact, for any vector representing a direction, the most extreme point in that direction will be part of the convex hull.

1.1 Algorithms for convex hull

In class we looked at several different algorithms for computing the convex hull (and you implemented one of these).

Giftwrapping algorithm. The first was the giftwrapping algorithm. The idea behind the giftwrapping algorithm is to find a point p_1 in P that is *definitely* on the hull, then work our way around the hull by continually selecting the next point the makes the greatest "right-hand turn":

GiftWrappingAlgorithm(P):

1. Start with a point $p_1 \in P$ that is guaranteed to lie on the border of the hull (we chose the bottom-most point in class, but you can choose any extreme point). Add p_1 to the hull.

2. Find the point $p \in P$ that makes the largest "right-hand turn." That is, if p_{prev} is the previous point on the convex hull, and ℓ is the line between the two previous points in the convex hull (or a horizontal line, in case the previous point is the bottom-most point) then the p maximizes the angle θ between ℓ and the line through p_{prev} and ℓ. Add p to the hull.

3. Repeat step 2 until we get back to p_1.

Giftwrapping is an example of an *output-sensitive* algorithm: the running time depends on the size of the convex hull (by size, we usually mean the *number of vertices* on the convex hull). If n is the size of the point set P (again, the number of points in P) and h is the size of the convex hull, then the running time of the giftwrapping algorithm is $O(nh)$: for each vertex on the convex hull, we spend $O(n)$ time looking for the next vertex. In the worst case, when *all* the points are on the hull, then the running time will be $O(n^2)$. If the points are randomly distributed (inside a circle, for instance), then we expect that about $\sqrt{n}$ points will be on the convex hull, so the running time of giftwrapping will be $O(n\sqrt{n})$ on average for this kind of distribution.

Quickhull. We also learned about the quickhull algorithm. Quickhull is a divide-and-conquer algorithm that, like quicksort, finds the convex hull by dividing the point set P into two parts, then recursively running quickhull on each part. You won't need to know the details of the quickhull algorithm, but you should know that the running time is $O(n \log n)$. Thus, quickhull is *not* an output-sensitive algorithm; the running time depends only on the size of the input set.

Applications of the convex hull. In class, we used the convex hull to come up with a compact descriptor for the shape of the lightstick (we then found the major axis of this shape to find the direction the lightstick was pointed). However, there are other things the convex hull can do. In particular, in class we saw how the convex hull can be used for sorting. In particular, given a list of n numbers $x_1, x_2, \ldots, x_n$, suppose we create a set of 2D points $(x_1, x_1^2), (x_2, x_2^2), \ldots, (x_n, x_n^2)$, i.e., points on a parabola. If we find the convex hull of this point set, then, starting from the bottommost point (assuming all numbers x_i are non-negative) the order of the points along the convex hull will be the same as the sorted order of the original numbers. This type of approach, where we solve one problem (in this case, sorting) by converting it to a different problem (convex hull) and solving the new problem, is called a *reduction*: we are reducing the problem of sorting to that of computing the convex hull. This has implications for the best possible running time of the new algorithm. In this case, we've learned that the best possible running time of a sorting algorithm is $O(n \log n)$—in general, we can't come up with a faster sorting algorithm (though we might be able to for certain specific types of input). This means that we also can't find the convex hull in faster than $O(n \log n)$ time; otherwise, we could come up with a faster sorting algorithm by converting to convex hull.[1]

2 Interpolation

Consider the following problem: we're given the small image below, and want to blow it up to poster size. How can we do this?

Instead of blowing it up to poster size, let's just try to blow it up so that it is four times as wide and four times as tall. That means we need to replace each pixel with a 4×4 patch of pixels (i.e., each pixel needs to be replaced with 16 pixels). How do we decide what to color the new pixels that we're inserting in between the original pixels? This problem is called *interpolation*—we need to figure out the values of an image in between known values. The general form of interpolation is that we're given some values of function (not necessarily an image) at a few *sample points* in the domain, and we want to compute the value of the function for the *entire* domain. For an image, these sample points are the pixel locations of the original image.

The exact values we fill in depend on our assumptions of how the function behaves. There are many possible assumptions we could make.

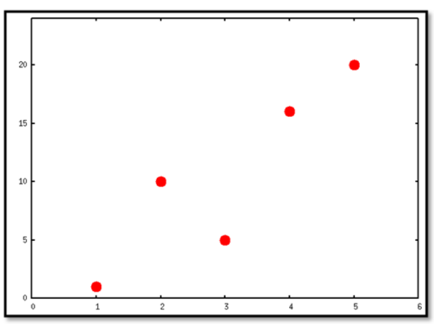
Nearest neighbor interpolation. One of the simplest interpolation schemes is called *nearest-neighbor interpolation*. With nearest-neighbor interpolation, we assume that the value of the function is the same as the value of the closest known sample. For our example of blowing up an image by a factor of four, this means we would replace each pixel with a 4×4 block with the same color as the original pixel. The resulting image would look like this:

[1] Two comments about this reduction.

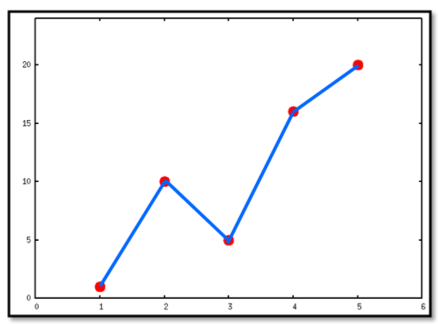
1. Strictly, it is only the case that the combined running time of converting a sorting problem to a convex hull problem **and** solving the convex hull problem cannot be less than $O(n \log n)$. In theory, if the conversion process itself took $O(n \log n)$ time, then there might still be a faster algorithm for convex hull. Note that the conversion process in this case takes just $O(n)$ time, however. We just need to scan through the list of numbers creating a 2D point for each one.
2. Actually, the best algorithms for convex hull work in $O(n \log h)$ time, which is faster than $O(n \log n)$ if h is small. However, this doesn't help us with the sorting problem, since we want to create a point set where *every* input point is on the convex hull.

The result is pretty blocky, and doesn't look that great. Our assumption that the values of the image should be equal to the closest known value is not a very good assumption, unfortunately. A somewhat better assumption is that the function behaves linearly between the known samples. This assumption leads to *linear interpolation*.

Linear interpolation Linear interpolation (or *lerp*, as it is known in computer graphics, where it is very commonly used), only really makes sense for one dimensional functions. In the figure below, filling in the values of the following function given the samples on the left using linear interpolation results in the function on the left:



Known samples of a function



Values interpolated with linear interpolation

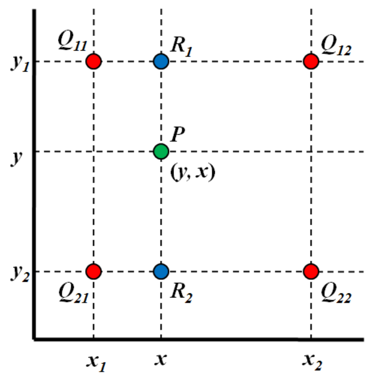
To compute the value of the function at a particular position x using linear interpolation, suppose the closest known samples on the left and right of x are $(x_{\text{left}}, y_{\text{left}})$ and $(x_{\text{right}}, y_{\text{right}})$. The value of at x is then computed as:

$$f(x) = \frac{x_{\text{right}} - x}{x_{\text{right}} - x_{\text{left}}} f(x_{\text{left}}) + \frac{x - x_{\text{left}}}{x_{\text{right}} - x_{\text{left}}} f(x_{\text{right}})$$

If x_{left} and x_{right} are at consecutive integer locations (e.g., $x_{\text{left}} = 1$ and $x_{\text{right}} = 2$, then this simplifies to:

$$f(x) = (x_{\text{right}} - x) f(x_{\text{left}}) + (x - x_{\text{left}}) f(x_{\text{right}})$$

Bilinear interpolation How do we apply linear interpolation in two dimensions? The answer is *bilinear interpolation*; this time, we define bilinear interpolation only for an integer grid of known sample points. Given a new point at which we'd like to compute the value of the function, we find the four nearest sample points, then use linear interpolation first along the y-direction, then along the x-direction (we would also get the same answer by interpolating along the x-direction first). This is demonstrated in the figure below:



To compute the value of the function at the green point P, we find the four surrounding points $Q_{11} = (y_1, x_1)$, $Q_{12} = (y_1, x_2)$, $Q_{21} = (y_2, x_1)$, and $Q_{22} = (y_2, x_2)$, then linearly interpolate along the y-direction to compute the function values at the two blue points R_1 and R_2. We then linearly interpolate R_1 and R_2 to compute the value at P. Again, if the four surrounding points on an integer grid, repectively, then the formula for the function value at P is:

$$f(P) = (x_2-x)(y_2-y)f(Q_{11})+(x-x_1)(y_2-y)f(Q_{12})+(x_2-x)(y-y_1)f(Q_{21})+(x-x_1)(y-y_1)f(Q_{22})$$

Using bilinear interpolation, our 4×4 image looks like this:

Compared to nearest neighbor interpolation, this looks much smoother, and not as blocky. However, we're not really resolving any more detail about the scene. There are better interpolation schemes than bilinear interpolation (such as bicubic interpolation), but no interpolation scheme will really be able to add detail that isn't present in the original image. The basic reason for this is that *many* different larger images, when shrunk down, would give the same small image. Thus, we can't expect to recover a larger image perfectly from a small image. (It's easier to see this when we take this to its logical conclusion—trying to blow up a full-resolution image from a single pixel. Clearly, this is impossible.)

3 Aliasing

These kind of interpolation schemes break down even more badly when we try to *shrink* an image. Here's the result of using nearest-neighbor interpolation to shrink down an image by two times, three times, and four times:

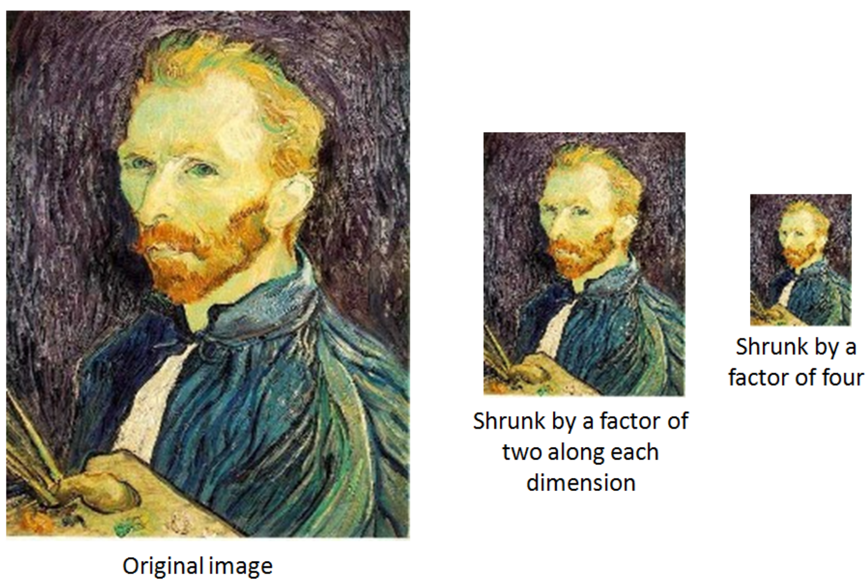
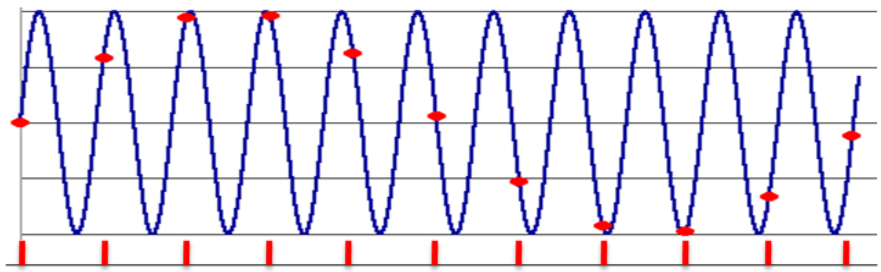


Figure 4: Downsampling with nearest-neighbor interpolation.

The results look very crufty. The problem here is *aliasing*, a very common phenomenon that you've probably experienced (possibly without knowing it). For instance, aliasing is what causes things that are rotating fast enough, such as wagon wheels in an old Western, to appear to start rotating backwards. The problem stems from sampling a high-frequency function too sparsely. For instance, consider the smooth 1D blue sine wave below. We can represent this function more compactly by sampling it only at a few locations (the red dots).



However, the resulting samples look like a sine wave with a completely different frequency. This is aliasing: sampling and getting a function that looks completely different than the original function. This happens when we downsample an image as well.

To understand what we need to do, suppose we downsample an image by a factor of 4 along each direction. The nearest-neighbor approach would essentially throw out three out of every four rows and columns of the image to produce the output image. Each pixel in the new image, corresponds to *16* pixels in the input image, however; what we should do it average these pixels, rather than throwing out all but one of them. Thus, by averaging over samples of the original image, we can *anti-alias* and get a smooth image back. Here's what the downsampled images of Van Gogh look like with this approach:

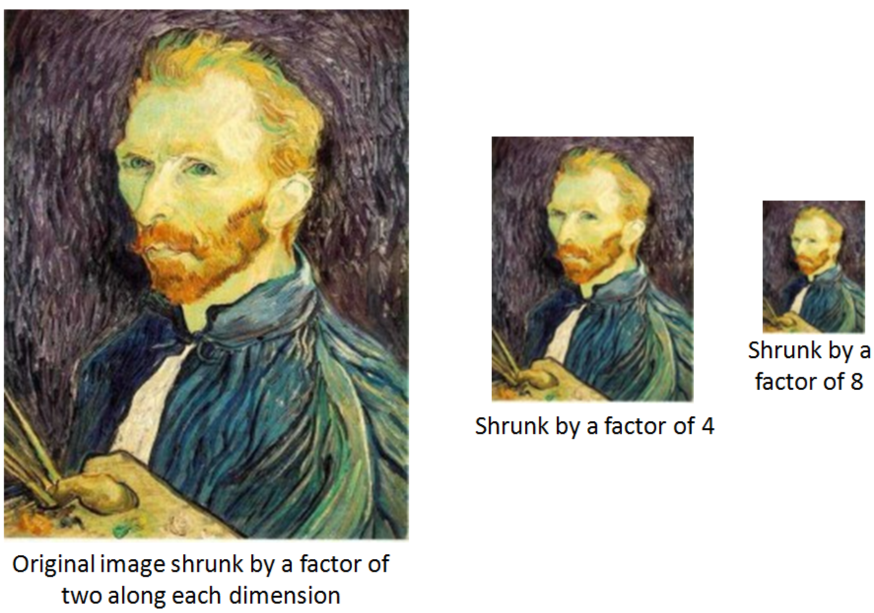


Figure 5: Downsampling with filtering.

One way to achieve this is to *filter* the image first to blur it, then downsample the blurred image using regular nearest-neighbor interpolation (this approach is called *pre-filtering*, because we filter before downsampling). Filtering is another word for *convolution*.

3.1 Convolution

Convolution is an image operator involving two images: a *target* image and a (usually quite small) image called a *kernel*. To convolve a image target image I with a kernel, we run the kernel across each pixel of I, multiply it by the pixels of I in a window of the same size of the kernel, then add up the results. For instance, if the target image and kernel are:

`image` =

0	1	0	1	0
1	2	3	4	0
1	1	1	1	0
0	2	4	5	7
1	1	3	4	5

`kernel` =

0	1	0
1	1	1
0	1	0

then the result of convolving `image` with `kernel` is:

`output` =

2	3	5	5	1
4	8	10	9	4
3	7	10	11	8
4	8	15	21	17
2	7	12	17	16

Often we use kernels whose values add up to 1—this results in an image that is no brighter or darker than the original. For instance, to blur an image, we might use the kernel:

`blur` =

$\frac{1}{9}$	$\frac{1}{9}$	$\frac{1}{9}$
$\frac{1}{9}$	$\frac{1}{9}$	$\frac{1}{9}$
$\frac{1}{9}$	$\frac{1}{9}$	$\frac{1}{9}$

This kernel averages every 3×3 window of an image.

4 Image transformations

What if we want to apply more complex transformations to an image than just simple scaling. We'll define a image *transformation* as a function that maps the 2D coordinates of an image to a new set of coordinates. We'll focus especially on *linear* transformations, i.e., transformations of the form:

$$f(x, y) = (ax + by, dx + ey)$$

for constant coefficients a, b, d, and e. Linear transformations can be represented by a 2×2 matrix

$$T = \begin{bmatrix} a & b \\ d & e \end{bmatrix}$$

and applying the transformation can be done with matrix multiplication, defined as:

$$\begin{bmatrix} a & b \\ d & e \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} ax + by \\ dx + ey \end{bmatrix}$$

Scaling

A scaling transformation can be represented with a matrix

$$S = \begin{bmatrix} s & 0 \\ 0 & s \end{bmatrix}.$$

To see this, let's see what happens when we multiple this matrix by a point $\begin{bmatrix} x \\ y \end{bmatrix}$:

$$\begin{bmatrix} s & 0 \\ 0 & s \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} sx \\ sy \end{bmatrix}$$

This multiplies points by a factor of s. If $s > 1$ we would be enlarging the image, and if $s < 1$, we would be shrinking the image.

Rotation

A rotation by an angle θ can be represented with a matrix

$$R = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}$$

Translation and homogeneous coordinates To represent a translation (moving the image left/right, up/down), we need more than just a linear transformation with a 2×2 matrix. For this, we need to introduce an idea known as *homogeneous coordinates*. We will play a trick by adding a 1 after the x, y coordinates of a vector (resulting in what we'll call a 2D homogeneous vector), and switch to using a 3×3 matrix (whose last row is always $\begin{bmatrix} 0 & 0 & 1 \end{bmatrix}$). The resulting matrix is an *affine* transformation with six parameters:

$$T = \begin{bmatrix} a & b & c \\ d & e & f \\ 0 & 0 & 1 \end{bmatrix}$$

Multiplying an affine transformation by a homogeneous vector gives:

$$\begin{bmatrix} a & b & c \\ d & e & f \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} ax + by + c \\ dx + ey + f \\ 1 \end{bmatrix}$$

We can now represent a translation by a vector (s, t) as:

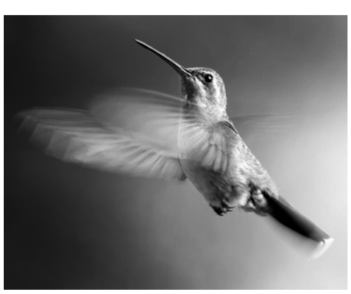
$$\begin{bmatrix} 1 & 0 & s \\ 0 & 1 & t \\ 0 & 0 & 1 \end{bmatrix}.$$

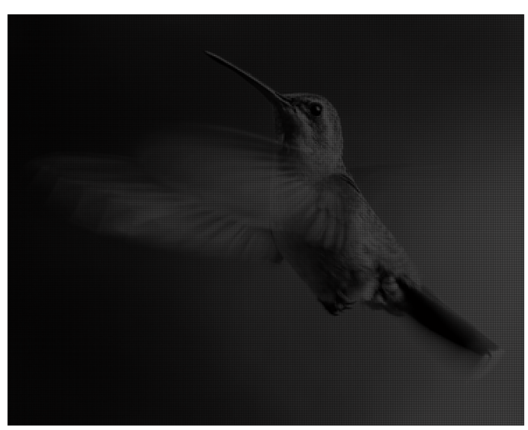
To see why, let's multiply by a homogeneous vector:

$$\begin{bmatrix} 1 & 0 & s \\ 0 & 1 & t \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} x + s \\ y + t \\ 1 \end{bmatrix}.$$

4.1 Inverse mapping

How do we actually apply a transformation to an image? One approach is to take each pixel in the input image, apply the transformation to its x, y position, then set the resulting output pixel to the same color as the input pixel. This approach is called *forward mapping*, and it has some weird results; for instance, for a rotation or upscaling, we might not "hit" every pixel in the output image. Thus, we might be left with gaps in our image when we're finished:





A better approach is called *inverse mapping*. With inverse mapping, we take each pixel of the *output* image, apply the *inverse* transformation to get the corresponding pixel location in the input image, then sample the color at that location to figure out what color to assign to the output pixel. By definition, this guarantees that we will fill in each pixel of the output image with a color (as long as the corresponding pixel of the input is in bounds). However, we might hit a fractional pixel in the input image. Thus, we need to use interpolation to get the value of that pixel. Often, a simple interpolation approach such as bilinear interpolation works; however, if we're scaling down the input image, we might want to use *trilinear interpolation* on an image stack, as you did in the assignment.

5 Feature-based object recognition

Now that we know a bit about image transformations, we can use them to help us do something interesting: object recognition. Suppose we have an image of an object we'd like to find (a template image), and an image that we'd like to find the object in (a search image):

Template image

Search image

There are many possible ways to try and find the search image in the template image. For instance, we could apply convolution (or, more accurately, normalized cross-correlation) to the two images; if we're lucky, the peak of the resulting image will be the location of the search object. However, this might won't work in several cases:

1. The object appears at a different scale or orientation in the search image.
2. The object is partially covered by other objects.
3. The lighting in the search image is different than in the template image.

In class, we talked about another approach: using local features.

5.1 Invariant local features

A local feature is a small, distinctive patch of an image—a fingerprint that is likely to be found in other images of the same object. For instance, "corner-like" features often make good candidates for such fingerprints. Finding such features in an image is known as *feature detection*. Feature detection involves finding the 2D positions of likely features, as well as coming up with a *descriptor vector* that describes the appearance of the image around that feature. Ideally, we want our detected features to be *invariant* to certain image transformations, such as rotation, scaling, and changes in brightness or contrast. Here, invariant means that the feature, and it's descriptor vector, remain unchanged under these transformations. If our features achieve invariance, then we can robustly compare fingerprints across very different images. Using local features for object recognition has several desirable properties:

1. **Locality**: Features are local, and therefore some of them can be covered up or not detected and we can still find the object.

2. **Distinctive**: Distinct features are detected, so we can potentially distinguish objects in a large database.

3. **Quantity**: We typically find lots of features in an image.

4. **Efficiency**: Compared to approaches which match the *entire* image, the local features approach can be very fast.

5.2 SIFT

SIFT (Scale-Invariant Feature Transform) is one of the best feature detectors around—it produces features that are invariant to scaling, rotation, certain changes in illumination, and small changes in viewpoint (how it does this is left to a computer vision course). It produces a 128-dimensional descriptor for each detected feature.

5.3 Feature matching

Once we detect features in the template and search images, we need to match features between them—that is, find similar-looking features in each image. We learned about one approach to this problem, *nearest-neighbor matching*. To do nearest-neighbor matching, we first need to define a distance function between two feature descriptors A and B. We will use the typical Euclidean distance:

$$\mathrm{d}ist(A, B) = \sqrt{\sum_{i=1}^{128} (A_i - B_i)^2}$$

where A_i is the i^{th} dimension of descriptor A (similarly with B_i). Once we have a distance function, nearest neighbor matching finds, for each feature in the template image, the nearest neighbor in the search image (i.e., the feature with the closest descriptor).

Unfortunately, there are many reasons why the nearest neighbor might be an incorrect match. For instance, the corresponding feature may be covered by another object, or it may simply have not have been detected by SIFT. Thus, we'll have many incorrect matches with this technique. To reduce the number of incorrect matches, we can, for instance, throw our nearest neighbors whose distances are larger than a threshold. However, it's very difficult to get rid of all the bad matches while keeping all the good matches. The techniques we'll look at next will have to work in spite of this fact.

5.4 Finding image transformations

The next step in our object detection framework is to find an image transformation between the template image and the search image that describes the position, orientation, and scale of the object in the search image. Again, we'll focus on affine transformations, which will explain many of the transformation we see in practice when doing object recognition.

We now have the following problem: we're given a set of known matches between the two images (some of which are incorrect), and we want to find an affine transformation that explains these matches as well as possible. Let's refer to our set of matches as

$$\begin{gathered} x_1, y_1 \to x'_1, y'_1 \\ x_2, y_2 \to x'_2, y'_2 \\ \cdots \\ x_n, y_n \to x'_n, y'_n \end{gathered}$$

i.e., the feature at location (x_1, y_1) in image 1 matches the feature at location (x'_1, y'_1) in image 2. If all of our matches were exact, then we would want to find the transformation T such that

$$\begin{gathered} T \begin{bmatrix} x_1 \\ y_1 \\ 1 \end{bmatrix} = \begin{bmatrix} x'_1 \\ y'_1 \\ 1 \end{bmatrix} \\ T \begin{bmatrix} x_2 \\ y_2 \\ 1 \end{bmatrix} = \begin{bmatrix} x'_2 \\ y'_1 \\ 1 \end{bmatrix} \\ \cdots \\ T \begin{bmatrix} x_n \\ y_n \\ 1 \end{bmatrix} = \begin{bmatrix} x'_n \\ y'_n \\ 1 \end{bmatrix} \end{gathered}$$

However, our matches aren't exact. The resulting problem is a form of linear regression. The basic linear regression problem is fitting a line to a set of points (x, y)—fitting an affine transformation to pairs of 2D points is exactly the same problem in a higher dimension. We will consider this problem in the next section, but for now observer that—just as we can fit a line exactly to two (x, y) pairs—we can fit a 2D affine transformation to three $(x, y) \to (x', y')$ pairs. To see why, note that each match gives us two equations involving the parameters of T:

$$T \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} \tag{1}$$

$$\begin{bmatrix} a & b & c \\ d & e & f \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} \tag{2}$$

which corresponds to the equations

$$ax + by + c = x'$$
$$dx + ey + f = y'$$

Thus we have two linear equations in the six unknowns a, b, c, d, e, f. Given three matches, we can find six such equations, and can solve for the unknowns (remember that to solve a linear system, we need as many equations as unknowns). We can solve the linear system using a matrix inversion in Matlab.

We likely have many more than three matches, but this suggests a simple algorithm: select three matches at random and compute T.

RANSAC. There is a problem with this algorithm, however—if we choose a wrong match among the three, then we will compute a completely wrong transformation. And this is pretty likely to happen: if $\frac{1}{3}$ of the matches are wrong, then the changes of picking one wrong match among three random matches is almost $\frac{2}{3}$. The solution to this problem is an algorithm called RANSAC (RANdom SAmple Consensus). The key to RANSAC is that we have a way of testing how good a candidate transformation T is: we can see how many of the matches it agrees with (it will certainly agree with the three matches we chose, but a correct transformation will hopefully agree with most of the correct matches, as well). By "agree," we mean that the transformation maps a point (x_i, y_i) in the template image "close" to its matching point (x'_i, y'_i) in the target image (we don't expect it to map the point *exactly* to the target). To define this concretely, we'll define a threshold t (say 5 pixels), then say a match $(x_i, y_i) \rightarrow (x'_i, y'_i)$ agrees with T if

$$\mathrm{d}ist\left(T \begin{bmatrix} x_i \\ y_i \\ 1 \end{bmatrix}, \begin{bmatrix} x'_i \\ y'_i \\ 1 \end{bmatrix}\right) < t$$

(where d*ist* is the Euclidean distance defined above).

To tell how well a transformation T agrees with the set of matches, we can simply count the number of matches that agree (these are called *inliers* to the transformation—the matches that don't agree are called *outliers*). We then sample many random sets of three matches and find the resulting T that agrees with the largest number of matches:

RANSAC(input: set of matches):

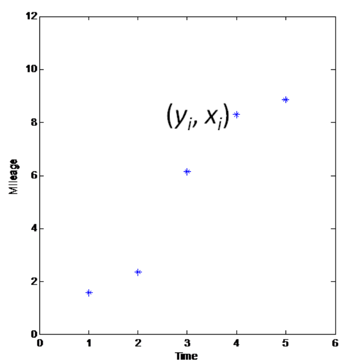
1. Randomly select three matches.
2. Solve for the corresponding transformation T.
3. Compute the number of inliers to T.
4. If T has the most inliers so far, save it.
5. Repeat steps 1-4 N times, then return the best T.

The RANSAC algorithm works well, and is used in many different computer vision applications. However, it relies on two parameters: t, the "closeness" threshold, and N, the number of rounds (though if we expect to have a certain percentage of inliers, and want to compute the right T with a given probability, e.g. 99.9999%, then we can compute the N we need).

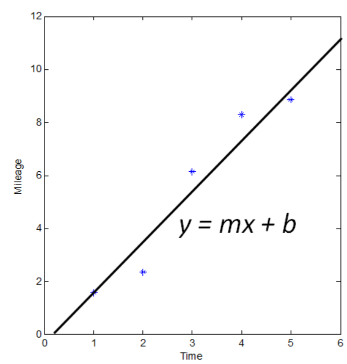
6 Optimization and least squares

Let's return to the linear regression problem. Suppose we want to fit an affine transformation to more than 3 matches (say, all of the inliers we find with RANSAC). Again, no affine transformation will perfectly fit all of the inliers. What we have is a linear system with many more equations than variables. For instance, if we have 100 inlier matches, then we have 200 equations, but still only 6 variables. What do we do now?

Let's consider the slightly simpler problem of 1D linear regression: we have a set of points (x_i, y_i) and want to fit a line $y = mx + b$ to them:



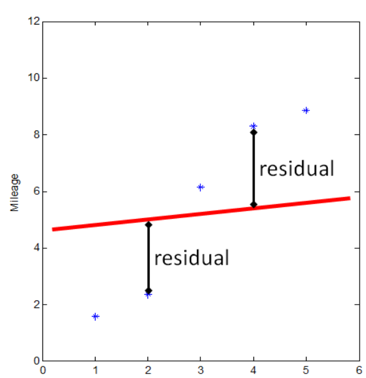
Input points



Points with the best-fit line

There is no perfect line, but we seek the *best* possible line. How do we define "best"?

The idea is that we'll come up with a cost (or objective) function $Cost(m, b)$ that measures how bad a line defined by m and b is, then find the line that minimizes this function. The most natural cost function is one that measures how much the line differs from the give points. We call these differences *residuals*; for linear regression, they can be visualized as vertical distances from each input point to the line:



The length of the residual for a point (x_i, y_i) is $|y_i - (mx_i + b_i)|$. Thus, one objective function could be the sum of residuals:

$$Cost(m, b) = \sum_{i=1}^{n} |y_i - (mx_i + b_i)|$$

For a variety of reasons, however, we prefer the sum of *squared* residuals (we'll explain a few reasons in a moment):

$$Cost(m, b) = \sum_{i=1}^{n} |y_i - (mx_i + b_i)|^2$$

When using a cost function that is sum of squared residuals, finding the minimum is known as the *least squares* problem. How do we find the line that minimizes $Cost(m, b)$? There are a few possible approaches:

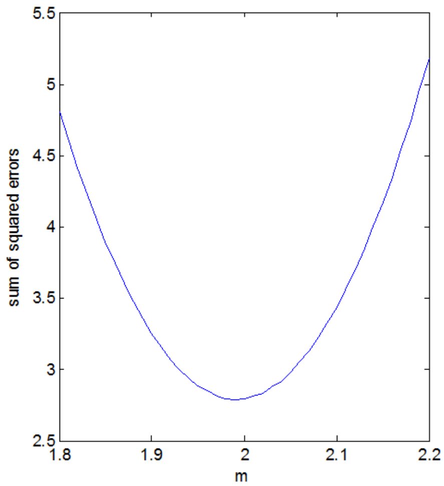
1. Guess an answer, check it, and repeat until we get a good answer (not such a good approach).
2. Guess an answer and improve it until we get a good answer (better).
3. Magically compute the right answer (best).

This process of finding the minimum of the function is known as *optimization*. We'll mostly discuss an approach that does a form of 2 above called *gradient descent*.

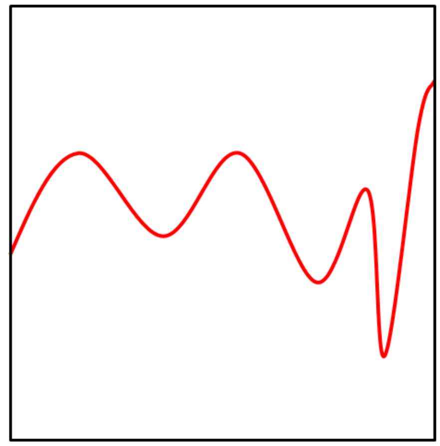
Gradient descent. Gradient descent can be visualized as finding the minimum by simulating a ball rolling down the objective function until it reaches the bottom. With gradient descent, we go through the following procedure:

1. Start at a guess for the right answer (which might be a completely random guess).
2. Next, compute the direction of steepest descent—the direction we should move that will take us furthest downhill (this direction is called the *gradient*). With a normal 1D function, this direction will either be "left" or "right"; in 2D it will be a direction in the plane.
3. Take a step in that direction, making sure that we end up below where we started.
4. Repeat 2-3 until we stop going downhill by very much. We've reached a minimum.

Does this approach get us to the right answer? For certain functions, yes, we will reach the correct answer eventually. This class of functions are called *convex*. Very similar to a convex polygon, a convex function is one where the area above the function forms a convex set—there are no nooks and crannies. Here's an example of 1D convex and non-convex function:



Convex



Non-convex

Convex functions are “nice”: they always have a single global minimum, and we will always find this minimum using gradient descent. A quadratic is an example of a convex function, and the sum of convex functions is always convex; thus, the sum of squared residuals (at least, of a linear function) is convex, and so we can successfully solve least squares using gradient descent (this is one reason why least squares is often used).

The other reason why least squares is so common is that it can actually be solved in closed form. We don’t need to use gradient descent at all: there is a formula for computing the minimum solution. However, this is outside the scope of this course.