CMSC 754: Lecture 2
Geometric Basics and Fixed-Radius Near Neighbors
Tuesday, Sep 6, 2005

The material on affine and Euclidean geometry will not be covered in lecture, but is presented here just in case you are interested in refreshing your knowledge on how basic geometric entities are represented and manipulated.

Reading: The material on the Fixed-Radius Near Neighbor problem is taken from the paper: "The complexity of finding fixed-radius near neighbors," by J. L. Bentley, D. F. Stanat, and E. H. Williams, *Information Processing Letters*, 6(6), 1977, 209–212.

Geometry Basics: As we go through the semester, we will introduce much of the geometric facts and computational primitives that we will be needing. For the most part, we will assume that any geometric primitive involving a constant number of elements of constant complexity can be computed in $O(1)$ time, and we will not concern ourselves with how this computation is done. (For example, given three non-collinear points in the plane, compute the unique circle passing through these points.) Nonetheless, for a bit of completeness, let us begin with a quick review of the basic elements of affine and Euclidean geometry.

There are a number of different geometric systems that can be used to express geometric algorithms: affine geometry, Euclidean geometry, and projective geometry, for example. This semester we will be working almost exclusively with affine and Euclidean geometry. Before getting to Euclidean geometry we will first define a somewhat more basic geometry called affine geometry. Later we will add one operation, called an inner product, which extends affine geometry to Euclidean geometry.

Affine Geometry: An affine geometry consists of a set of *scalars* (the real numbers), a set of *points*, and a set of *free vectors* (or simply *vectors*). Points are used to specify position. Free vectors are used to specify direction and magnitude, but have no fixed position in space. (This is in contrast to linear algebra where there is no real distinction between points and vectors. However this distinction is useful, since the two are conceptually quite different.)

The following are the operations that can be performed on scalars, points, and vectors. Vector operations are just the familiar ones from linear algebra. It is possible to subtract two points. The difference $p - q$ of two points results in a free vector directed from q to p. It is also possible to add a point to a vector. In point-vector addition $p + v$ results in the point which is translated by v from p. Letting S denote an generic scalar, V a generic vector and P a generic point, the following are the legal operations in affine geometry:

$S \cdot V$	$\rightarrow$	V	scalar-vector multiplication
$V + V$	$\rightarrow$	V	vector addition
$P - P$	$\rightarrow$	V	point subtraction
$P + V$	$\rightarrow$	P	point-vector addition

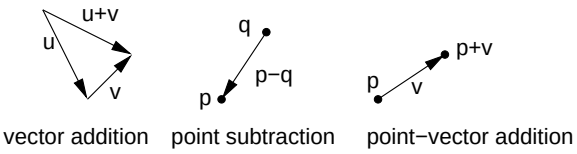

Figure 1: Affine operations.

A number of operations can be derived from these. For example, we can define the subtraction of two vectors $\vec{u} - \vec{v}$ as $\vec{u} + (-1) \cdot \vec{v}$ or scalar-vector division $\vec{v}/\alpha$ as $(1/\alpha) \cdot \vec{v}$ provided $\alpha \neq 0$. There is one special vector, called the *zero vector*, $\vec{0}$, which has no magnitude, such that $\vec{v} + \vec{0} = \vec{v}$.

Note that it is *not* possible to multiply a point times a scalar or to add two points together. However there is a special operation that combines these two elements, called an *affine combination*. Given two points p_0 and p_1 and two scalars α_0 and α_1, such that $\alpha_0 + \alpha_1 = 1$, we define the affine combination

$$\text{aff}(p_0, p_1; \alpha_0, \alpha_1) \;=\; \alpha_0 p_0 + \alpha_1 p_1 \;=\; p_0 + \alpha_1(p_1 - p_0).$$

Note that the middle term of the above equation is not legal given our list of operations. But this is how the affine combination is typically expressed, namely as the weighted average of two points. The right-hand side (which is easily seen to be algebraically equivalent) is legal. An important observation is that, if $p_0 \neq p_1$, then the point $\text{aff}(p_0, p_1; \alpha_0, \alpha_1)$ lies on the line joining p_0 and p_1. As α_1 varies from $-\infty$ to $+\infty$ it traces out all the points on this line.

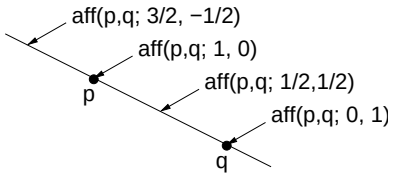

Figure 2: Affine combination.

In the special case where $0 \leq \alpha_0, \alpha_1 \leq 1$, $\text{aff}(p_0, p_1; \alpha_0, \alpha_1)$ is a point that subdivides the line segment $\overline{p_0 p_1}$ into two subsegments of relative sizes α_1 to α_0. The resulting operation is called a *convex combination*, and the set of all convex combinations traces out the line segment $\overline{p_0 p_1}$.

It is easy to extend both types of combinations to more than two points, by adding the condition that the sum $\alpha_0 + \alpha_1 + \alpha_2 = 1$.

$$\text{aff}(p_0, p_1, p_2; \alpha_0, \alpha_1, \alpha_2) \;=\; \alpha_0 p_0 + \alpha_1 p_1 + \alpha_2 p_2 \;=\; p_0 + \alpha_1(p_1 - p_0) + \alpha_2(p_2 - p_0).$$

The set of all affine combinations of three (non-collinear) points generates a plane. The set of all convex combinations of three points generates all the points of the triangle defined by the points. These shapes are called the *affine span* or *affine closure*, and *convex closure* of the points, respectively.

Euclidean Geometry: In affine geometry we have provided no way to talk about angles or distances. Euclidean geometry is an extension of affine geometry which includes one additional operation, called the *inner product*, which maps two real vectors (not points) into a nonnegative real. One important example of an inner product is the *dot product*, defined as follows. Suppose that the d-dimensional

vectors $\vec{u}$ and $\vec{v}$ are represented by the (nonhomogeneous) coordinate vectors $(u_1, u_2, \ldots, u_d)$ and $(v_1, v_2, \ldots, v_d)$. Define

$$\vec{u} \cdot \vec{v} = \sum_{i=1}^{d} u_i v_i,$$

The dot product is useful in computing the following entities.

Length: of a vector $\vec{v}$ is defined to be $\|\vec{v}\| = \sqrt{\vec{v} \cdot \vec{v}}$.

Normalization: Given any nonzero vector $\vec{v}$, define the *normalization* to be a vector of unit length that points in the same direction as $\vec{v}$. We will denote this by $\hat{v}$:

$$\hat{v} = \frac{\vec{v}}{\|\vec{v}\|}.$$

Distance between points: Denoted either $\text{dist}(p, q)$ or $\|pq\|$ is the length of the vector between them, $\|p - q\|$.

Angle: between two nonzero vectors $\vec{u}$ and $\vec{v}$ (ranging from 0 to π) is

$$\text{ang}(\vec{u}, \vec{v}) = \cos^{-1}\left(\frac{\vec{u} \cdot \vec{v}}{\|\vec{u}\|\|\vec{v}\|}\right) = \cos^{-1}(\hat{u} \cdot \hat{v}).$$

This is easy to derive from the law of cosines.

Orientation of Points: In order to make discrete decisions, we would like a geometric operation that operates on points in a manner that is analogous to the relational operations $(<, =, >)$ with numbers. There does not seem to be any natural intrinsic way to compare two points in d-dimensional space, but there is a natural relation between ordered $(d+1)$-tuples of points in d-space, which extends the notion of binary relations in 1-space, called *orientation*.

Given an ordered triple of points $\langle p, q, r \rangle$ in the plane, we say that they have *positive orientation* if they define a counterclockwise oriented triangle, *negative orientation* if they define a clockwise oriented triangle, and *zero orientation* if they are collinear (which includes as well the case where two or more of the points are identical). Note that orientation depends on the order in which the points are given.

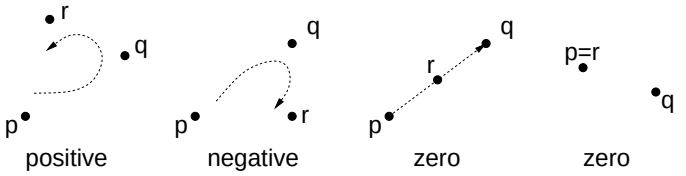

Figure 3: Orientations of the ordered triple (p, q, r).

Orientation is formally defined as the sign of the determinant of the points given in homogeneous coordinates, that is, by prepending a 1 to each coordinate. For example, in the plane, we define

$$\text{Orient}(p, q, r) = \det \begin{pmatrix} 1 & p_x & p_y \\ 1 & q_x & q_y \\ 1 & r_x & r_y \end{pmatrix}.$$

Observe that in the 1-dimensional case, Orient(p, q) is just $q - p$. Hence it is positive if $p < q$, zero if $p = q$, and negative if $p > q$. Thus orientation generalizes $<, =, >$ in 1-dimensional space. Also note that the sign of the orientation of an ordered triple is unchanged if the points are translated, rotated, or scaled (by a positive scale factor). A reflection transformation, e.g., $f(x, y) = (-x, y)$, reverses the sign of the orientation. In general, applying any affine transformation to the point alters the sign of the orientation according to the sign of the matrix used in the transformation.

This generalizes readily to higher dimensions. For example, given an ordered 4-tuple points in 3-space, we can define their orientation as being either positive (forming a right-handed screw), negative (a left-handed screw), or zero (coplanar). It can be computed as the sign of the determinant of an appropriate 4×4 generalization of the above determinant. This can be generalized to any ordered $(d + 1)$-tuple of points in d-space.

Areas and Angles: The orientation determinant, together with the Euclidean norm can be used to compute angles in the plane. This determinant Orient(p, q, r) is equal to twice the signed area of the triangle $\triangle pqr$ (positive if CCW and negative otherwise). Thus the area of the triangle can be determined by dividing this quantity by 2. In general in dimension d the area of the simplex spanned by $d + 1$ points can be determined by taking this determinant and dividing by $d! = d \cdot (d - 1) \cdots 2 \cdot 1$. Given the capability to compute the area of any triangle (or simplex in higher dimensions), it is possible to compute the volume of any polygon (or polyhedron), given an appropriate subdivision into these basic elements. (Such a subdivision does not need to be disjoint. The simplest methods that I know of use a subdivision into overlapping positively and negatively oriented shapes, such that the signed contribution of the volumes of regions outside the object cancel each other out.)

Recall that the dot product returns the cosine of an angle. However, this is not helpful for distinguishing positive from negative angles. The sine of the angle $\theta = \angle pqr$ (the signed angled from vector $p - q$ to vector $r - q$) can be computed as

$$\sin\theta = \frac{\text{Orient}(q, p, r)}{\|p - q\| \cdot \|r - q\|}.$$

(Notice the order of the parameters.) By knowing both the sine and cosine of an angle we can unambiguously determine the angle.

Fixed-Radius Near Neighbor Problem: As a warm-up exercise for the course, we begin by considering one of the oldest results in computational geometry. This problem was considered back in the mid 70's, and is a fundamental problem involving a set of points in dimension d. We will consider the problem in the plane, but the generalization to higher dimensions will be straightforward. The solution also illustrates a common class of algorithms in CG, which are based on grouping objects into buckets that are arranged in a square grid.

We are given a set P of n points in the plane. It will be our practice throughout the course to assume that each point p is represented by its (x, y) coordinates, denoted (p_x, p_y). Recall that the Euclidean distance between two points p and q, denoted $\|pq\|$, is

$$\|pq\| = \sqrt{(p_x - q_x)^2 + (p_y - q_y)^2}.$$

Given the set P and a distance $r > 0$, our goal is to report all pairs of distinct points $p, q \in P$ such that $\|pq\| \leq r$. This is called the *fixed-radius near neighbor (reporting) problem*.

Reporting versus Counting: We note that this is a *reporting* problem, which means that our objective is to report all such pairs. This is in contrast to the corresponding *counting* problem, in which the objective is to return a count of the number of pairs satisfying the distance condition.

It is usually easier to solve reporting problems optimally than counting problems. This may seem counterintuitive at first (after all, if you can report, then you can certainly count). The reason is that we know that any algorithm that reports some number k of pairs must take at least $\Omega(k)$ time. Thus if k is large, a reporting algorithm has the luxury of being able to run for a longer time and still claim to be optimal. In contrast, we cannot apply such a lower bound to a counting algorithm.

The approach described here seems to work only for the reporting case. There is a more efficient solution to the counting problem, but this requires more sophisticated methods.

Simple Observations: To begin, let us make a few simple observations. This problem can easily be solved in $O(n^2)$ time, by simply enumerating all pairs of distinct points and computing the distance between each pair. The number of distinct pairs of n points is

$$\binom{n}{2} = \frac{n(n-1)}{2}.$$

Letting k denote the number of pairs that reported, our goal will be to find an algorithm whose running time is (nearly) linear in n and k, ideally $O(n+k)$. This will be optimal, since any algorithm must take the time to read all the input and print all the results. (This assumes a naive representation of the output. Perhaps there are more clever ways in which to encode the output, which would require less than $O(k)$ space.)

To gain some insight to the problem, let us consider how to solve the 1-dimensional version, where we are just given a set of n points on the line, say, $x_1, x_2, \ldots, x_n$. In this case, one solution would be to first sort the values in increasing order. Let suppose we have already done this, and so:

$$x_1 < x_2 < \ldots < x_n.$$

Now, for i running from 1 to n, we consider the successive points x_{i+1}, x_{i+2}, x_{i+3}, and so on, until we first find a point whose distance exceeds r. We report x_i together with all succeeding points that are within distance r.

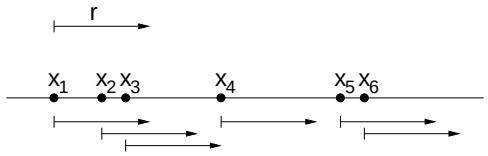

Figure 4: Fixed radius nearest neighbor on the line.

The running time of this algorithm involves the $O(n \log n)$ time needed to sort the points and the time required for distance computations. Let k_i denote the number of pairs generated when we visit p_i. Observe that the processing of p_i involves $k_i + 1$ distance computations (one additional computation for the points whose distance exceeds r). Thus, up to constant factors, the total running time is:

$$\begin{aligned} T(n,k) &= n \log n + \sum_{i=1}^{n} (k_i + 1) \;=\; n \log n + n + \sum_{i=1}^{n} k_i \\ &= n \log n + n + k \;=\; O(k + n \log n). \end{aligned}$$

This is close to the $O(k+n)$ time we were hoping for. It seems that any approach based on sorting is doomed to take at least $\Omega(n \log n)$ time. So, if we are to improve upon this, we cannot sort. But is sorting really necessary? Let us consider an approach based on bucketing.

1-dimensional Solution with Bucketing: Rather than sorting the points, suppose that we subdivide the line into intervals of length r. In particular, we can take the line to be composed of an infinite collection of half-open intervals:

$$\ldots, [-3r, -2r),\ [-2r, -r),\ [-r, 0),\ [0, r),\ [r, 2r),\ [2r, 3r), \ldots$$

We refer to these disjoint intervals as *buckets*. Given the interval $[br, (b+1)r)$, its *bucket index* is the integer b. Given any point x, it is easy to see that the index of the containing bucket is just $b(x) = \lfloor x/r \rfloor$. Thus, in $O(n)$ time we can associate the n points of P with a set of n integer bucket indices, $b(x)$ for each $x \in P$. Although there are an infinite number of buckets, at most n will be *occupied*, meaning that they contain at least one point of P.

There are a number of ways to organize the occupied buckets. They could be sorted, but then we are back to $O(n \log n)$ time. Since bucket indices are integers, a better approach is to store the occupied bucket indices in a *hash table*. Recall from basic data structures that a hash table is a data structure that supports the following operations in $O(1)$ expected time:

insert(o, x): Insert object o with key value x. We allow multiple objects to be inserted with the same key.

L = find(x): Return a list L of references to objects having key value x. This operation takes $O(1 + |L|)$ expected time. If no keys have this value, then an empty list is returned.

remove(o, x): Remove the object indicated by reference o, having key value x from the table.

Each point is associated with the key value given by its bucket index $b = \lfloor x/r \rfloor$. Thus in $O(1)$ expected time, we can determine which bucket contains a given point and look this bucket up in the hash table.

The fact that the running time is in the expected case, rather than worst case is a bit unpleasant. However, it can be shown that by using a good randomized hash function, the probability that the total running time is worse than $O(n)$ can be made arbitrarily small. If the algorithm performs significantly more than the expected number of computations, we can simply chose a different random hash function and try again. This will lead to a very practical solution.

How does bucketing help? Observe that if point x lies in bucket b, then any successors that lie within distance r must lie either in bucket b or in $b+1$. This suggests the straightforward solution shown below.

Fixed-Radius Near Neighbor on the Line by Bucketing

(1) For each $x \in P$, insert x in the hash table with the key value $b(x)$.

(2) For each $x \in P$ do the following:

  (a) Let $b(x)$ be the bucket containing x.

  (b) Enumerate all the points of buckets $b(x)$ and $b(x)+1$, and for each point $x' \in b(x) \cup b(x)+1$ such that $0 < x' - x \leq r$, output the pair (x, x').

Note that, in order to avoid duplicates we only report pairs (x, x') where $x' > x$. The key question is determining the time complexity of this algorithm is how many distance computations are performed in step (2b). We compare each point in bucket b with all the points in buckets b and $b+1$. However, not all of these distance computations will result in a pair of points whose distance is within r. Might it be that we waste a great deal of time in performing computations for which we receive no benefit? The lemma below shows that we perform no more than a constant factor times as many distance computations and pairs that are produced.

It will simplify things considerably if, rather than counting distinct pairs of points, we simply count all (ordered) pairs of points that lie within distance r of each other. Thus each pair of points will be counted twice, (p, q) and (q, p). Note that this includes reporting each point as a pair (p, p) as well, since each point is within distance r of itself. This does not affect the asymptotic bounds, since the number of distinct pairs is smaller by a factor of roughly $1/2$.

Lemma: Let k denote the number of (not necessarily distinct) pairs of points of P that are within distance r of each other. Let D denote the total number distance computations made in step (2b) of the above algorithm. Then $D \leq 2k$.

Proof: We will make use of the following inequality in the proof:

$$xy \leq \frac{x^2 + y^2}{2}.$$

This follows by expanding the obvious inequality $(x - y)^2 \geq 0$.

Let B denote the (infinite) set of buckets. For any bucket $b \in B$, let $b+1$ denote its successor bucket on the line, and let n_b denote the number of points of P in b. Define

$$S = \sum_{b \in B} n_b^2.$$

First we bound the total number of distance computations D as a function of S. Each point in bucket b computes the distance to every other point in bucket b and every point in bucket $b+1$, and hence

$$\begin{aligned} D &= \sum_{b \in B} n_b(n_b + n_{b+1}) = \sum_{b \in B} n_b^2 + n_b n_{b+1} = \sum_{b \in B} n_b^2 + \sum_{b \in B} n_b n_{b+1} \\ &\leq \sum_{b \in B} n_b^2 + \sum_{b \in B} \frac{n_b^2 + n_{b+1}^2}{2} \\ &= \sum_{b \in B} n_b^2 + \sum_{b \in B} \frac{n_b^2}{2} + \sum_{b \in B} \frac{n_{b+1}^2}{2} = S + \frac{S}{2} + \frac{S}{2} = 2S. \end{aligned}$$

Next we bound the number of pairs reported from below as a function of S. Since each pair of points lying in bucket b is within distance r of every other, there are n_b^2 pairs in bucket b alone that are within distance r of each other, and hence (considering just the pairs generated within each bucket) we have $k \geq S$.

Therefore we have

$$D \leq 2S \leq 2k,$$

which completes the proof.

By combining this with the $O(n)$ expected time needed to bucket the points, it follows that the total expected running time is $O(n + k)$.

A worthwhile exercise to consider at this point is the issue of the bucket width r. How would changing the value of r affect the implementation of the algorithm and its efficiency? For example, if we used buckets of size $r/2$ or $2r$, would the above algorithm (after suitable modifications) have the same asymptotic running time? Would buckets of size any constant times r work?

Generalization to d dimensions: This bucketing algorithm is easy to extend to multiple dimensions. For example, in dimension 2, we bucket points into a square grid in which each grid square is of side length r. (As before, you might consider the question of what values of bucket sizes lead to a correct and efficient algorithm.) The bucket index of a point $p : (x, y)$ is a pair $b(p) = (b(x), b(y)) = (\lfloor x/r \rfloor, \lfloor y/r \rfloor)$. We apply a hash function that accepts two arguments. To generalize the algorithm, for each point we consider the points in its surrounding 3×3 subgrid of buckets. By generalizing the above arguments, it can be shown that the algorithm's expected running time is $O(n + k)$. The details are left as an exercise.

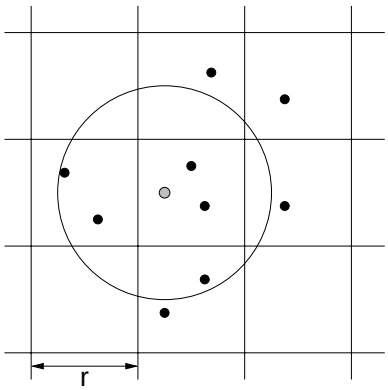

Figure 5: Fixed radius nearest neighbor on the plane.

This example problem serves to illustrate some of the typical elements of computational geometry. Geometry itself did not play a significant role in the problem, other than the relatively easy task of computing distances. We will see examples later this semester where geometry plays a much more important role. The major emphasis was on accounting for the algorithm's running time. Also note that, although we discussed the possibility of generalizing the algorithm to higher dimensions, we did not treat the dimension as an asymptotic quantity. In fact, a more careful analysis reveals that this algorithm's running time increases exponentially with the dimension. (Can you see why?)