High-Dimensional Linear Data Interpolation

Russell Pflughaupt

May 19, 1993

MS report under direction of
Prof. Carlo H. Séquin

Abstract

This report explores methods for interpolating across high-dimensional data sets. We describe and evaluate algorithms designed for problems with 100 to 10,000 points in dimensions 2 through 40. Specialized algorithms that attempt to take advantage of properties of locality are shown to be less efficient than generalized algorithm such as gift-wrapping and linear programming. In addition, this report contains an accumulation of information on the properties of high-dimensional spaces.

Table of Contents

1 Introduction

There are situations where one would like to have a real-time simulation model of a complicated system such as a high-performance airplane or rocket engine for 'what-if' type experiments before actual changes are induced in the operational state of the engine. A few thousand test points describing the state of the engine in a high-dimensional data space corresponding to the various variables measured (gas mixture, speed, pressure, temperature at various points) may have been gathered by actual measurements or by lengthy in-depth systems simulations on a high-powered computer. For actual use, one would like to interpolate (and perhaps extrapolate) the given data points in a robust manner. Although interpolation of data sets is a well known problem that has largely been explored in great detail, an area that has not been sufficiently explored is how dimensionality affects this problem.

A conceptually straightforward approach to interpolation in 2D is to simply triangulate the data and use the values at a triangle's vertices to interpolate over its interior. However, a triangulation scheme must be found that is both consistent and "well-behaved". Consistency is required so that a given data location is always interpolated in the same triangle with the same vertices. Being "well-behaved" means that every vertex of a triangle is in some sense "near" to every point enclosed by that triangle. It is easy to imagine an inconsistent triangulation which depends on the order in which the data points were originally presented. Likewise, it is easy to imagine a poorly behaved triangulation which produces long, skinny triangles. This would result in interpolations that use far away data points rather than closer ones.

Two methods of triangulation are possible: In the first, some reasonably good global triangulation would be computed, written to disk, and provided with an efficient index for access. In the second, the data set would be kept in memory and a canonical triangulation is computed on demand in the neighborhood of each query point. Given an efficient indexing scheme, access to a fixed triangulation on disk may have response times measured in milliseconds. However, constructing the complete global triangulation may take weeks.

This paper investigates local, on-the-fly triangulations. One would hope that an on-the-fly triangulation could be found that was both efficient in running time and storage requirements. This method has no initial cost for computing the global triangulation, but the response time might be much slower than that of the precalculated triangulation. Keeping a cache of most recently used triangles is one method to speed up nearby queries.

In choosing a local triangulation scheme, particular care must be taken to guarantee consistency. The obvious choice is the Delaunay triangulation. Ignoring the special case of co-circular points, the Delaunay triangulation will always produce a canonical triangulation that is based solely on the data points. The Delaunay triangulation can be defined as "the unique triangulation such that the circumcircle of each triangle does not contain any other point in its interior." [PREP85]

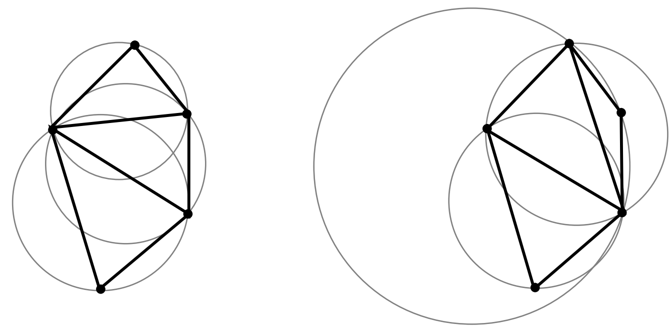


Figure 1: Example of Delaunay triangulation (left) and non-Delaunay (right)

This implies a very desirable property. Given a Delaunay triangle on a set of points, new points can be added and the triangle will remain Delaunay so long as no new point falls within its circumcircle. Stated another way, a triangle can be verified to be a Delaunay if its circumcircle contains no points other than its own vertices. This property allows local triangulations to be created that can be guaranteed to be Delaunay by checking for the absence of any other points within each circumcircle.

Up until now, we have limited ourselves to a language that can only describe objects in two dimensions. A more general language needs to be adopted that can refer to higher dimensional objects. We could let triangles become simplices, triangulations become hyper-tessellations, lines become hyperplanes, circles become hyperspheres, etc. However, much of this tends to obscure the underlying concepts. Unless explicitly stated otherwise, throughout the rest of this report we will assume simplices, triangulations, planes and spheres to all refer to their D-dimensional counterparts. Notice that the definition of a Delaunay triangulation remains essentially unchanged: the unique triangulation such that the circumsphere of each simplex does not contain any other point in its interior. In dimension D, a simplex is uniquely determined by its D+1 vertices. Likewise, a sphere in dimension D is uniquely determined by D+1 points that lie on its surface. This extension allows the Delaunay triangulation to be used in problems of arbitrary dimension.

2 Why the Problem is Difficult[1]

The purpose of this chapter is to expose the reader to aspects of high dimensional geometry. Much of what is presented in this chapter was developed and/or researched after our algorithms for finding a Delaunay simplex enclosing a query point had been completed. It is presented now to provide the reader with a context from which she may better understand the failings of the algorithms in the next chapter. It is also hoped that this information will prove useful to researchers in related problem areas.

We start by examining the D-dimensional simplex and calculate the number of nearest neighbors a vertex has, how many simplices are attached to each vertex, the total number of expected simplices in D dimensions with N data points, and the volumes of high dimensional objects.

The D Dimensional Simplex

Certainly a good place to start in the tour of high dimensionality is with an examination of the D dimensional simplex. A simplex can be thought of as a D dimensional "triangle". In 0D this would be a point, in 1D an edge, in 2D a triangle, in 3D a tetrahedron or cell, etc. A simplex in D dimensions requires D+1 vertices. Anything less would produce a simplex of less than D dimensions (i.e. 3 vertices in 3D just produces a triangle oriented in 3-space).

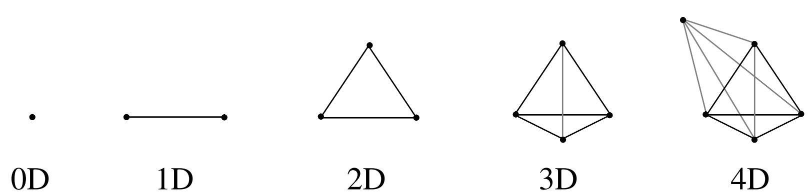


Figure 2: Simplices in D dimensions

The number of 1D edges for a D dimensional simplex is

$$\sum_{i=1}^{D} i \qquad \text{or} \qquad \frac{(D+1)\,D}{2}$$

This formula can readily be derived from the recursion that if a D+1'st vertex is added to a D-1 dimensional simplex, new 1D edges will be added to the new vertex from all the original D vertices (adding D more 1D edges).

A similar recursion gives the number of 2D faces. If a D+1'st vertex is added to a D-1 dimensional simplex, new 2D faces will be added to the new vertex from the original edges (add-

1. This Chapter might more aptly be called The Curse of Dimensionality [EUBA88].

ing D(D-1)/2 more 2D faces). In general, the number of 2D faces in a D dimensional simplex is the number of 1D edges in a D-1 dimensional simplex plus the number of 2D faces in a D-1 dimensional simplex. This is

$$\frac{(D+1)\,D\,(D-1)}{6}$$

3D cells are like everything else. Each time a vertex is added, the number of 3D cells increases by the number of 2D faces on the original simplex. In general, the number of 3D cells in a D dimensional simplex is the number of 2D faces in a D-1 dimensional simplex plus the number of 3D cells in a D-1 dimensional simplex. This is

$$\frac{(D+1)\,D\,(D-1)\,(D-2)}{24}$$

A pattern begins to emerge. Every D dimensional simplex is made up of lower dimensional simplices. Let S represent the dimension of the simplices you wish to count on a D dimensional simplex. Remembering that vertices are 0 dimensional simplices, the number of S dimensional simplices on a D dimensional simplex is

$$\frac{1}{S+1\,!}\cdot\frac{(D+1)\,!}{(D-S)\,!} \qquad \text{or} \qquad \binom{D+1}{S+1}$$

“Typical” Number of Simplices Attached to a Vertex in D Dimensions

An interesting aspect of dimensionality is the number of simplices a “typical” vertex in a data set would be attached to. Given an actual set of vertices, this can be computed exactly using a Vornoi diagram. However, we are more interested in general guidelines for high dimensions. Therefore we’ll derive the typical number using an infinitely large regular tessellation, since the average number must be exactly the same as if the tessellation were irregular.

We’ll start by taking a 3 dimensional cube and triangulating its 6 surface faces into 12 triangles. If you were to take a point inside the cube and construct simplices using this point and the triangles on the surface of the cube, there would be 12 simplices. However, moving this point to one of the cube’s corners forces the 6 simplices adjacent to this corner to collapse. This leaves half of the triangulated cube faces as “bottom” faces for simplices with their tops at that corner. With 2 simplices per face on the cube, there are 3*2 simplices. We can now recurse using this 3 dimensional cube. The 4 dimensional cube has 8 boundary cubes. Break half of these into the previously defined 6 3D simplices which then act as the bottoms for 24 4D simplices (with a shared vertex at one corner). The 5 dimensional cube has 10 boundary hypercubes resulting in 5*24=120 5D simplices. In general, the number of simplices in a cube is D!. Since each simplex is attached to D+1 vertices and since each cube accounts for 1 vertex in the grid, the typical number of simplices attached to a vertex in D dimensions is

$$(D+1)D! \quad \text{or} \quad (D+1)!$$

"Typical" Number of Nearest Neighbors in D Dimensions

To calculate the typical number of nearest neighbors in D dimensions we'll use the same tessellation described in the previous section, **"Typical" Number of Simplices Attached to a Vertex in D Dimensions**. We count the number of edges in a single cube and weigh them by the amount of sharing they experience with adjoining cubes. For example, a 3D cube has 12 boundary edges that are shared with 3 other cubes each, 6 edges embedded in the 6 faces of the cube that are shared with 1 other cube each, and 1 internal edge which is shared with no other cubes. Since there are 2 vertices for every edge, the typical number of nearest neighbors in 3 dimensions is

$$2 \times \left(\frac{12}{4} + \frac{6}{2} + 1\right) = 14$$

In general, the edges of a D dimensional cube get shared 2^{D-1} fold and the edges embedded in every face of dimension J get shared 2^{D-J} fold. Since a D dimensional cube has $2^{D-J}\binom{D}{J}$ features of dimension J, we need only sum from J=1 to J=D and multiply by 2.

$$2 \times \left[\binom{D}{1} + \binom{D}{2} + \ldots + \binom{D}{D-1} + \binom{D}{D} \right] = 2^{D+1} - 2$$

This number is considerably larger than the equivalent number for densest sphere packing because we are not constrained to have all nearest neighbors to be equidistant from each other[1].

Maximum Number of Simplices in D Dimensions with N Vertices

Paschinger [PASC82] and Seidel [SEID82], [SEID87] independently proved that the upper bounds for the number of *i*-dimensional faces μ_i in a D dimensional Vornoi diagram with N vertices is:

$$\mu_i(N, D) = \begin{cases} C_{D-i}(N, D+1) - 1 & i = 0 \\ C_{D-i}(N, D+1) & 0 < i \le D \end{cases}$$

1. [ODLY79] provides an upper bound on the maximum number of unit spheres that can touch a unit sphere in *n* dimensions. As an example, this upper bound for dimension 24 is 196,560 spheres.

Where $C_j(N, D+1)$ represents the maximal number of *j*-dimensional faces of a (*D*+1) polytope with *N* vertices:

$$C_j(N, D+1) = \begin{cases} \sum_{i=1}^{s} \frac{N}{i} \binom{N-i-1}{i-1} \binom{i}{j-i+1} & D = 2s-1 \\ \sum_{i=0}^{s} \frac{j+2}{i+1} \binom{N-i-1}{i} \binom{i+1}{j-i+1} & D = 2s \end{cases}$$

Since the Vornoi diagram and the Delaunay tessellation are duals of each other, a simple transformation will allow the use of the above equations:

$$\text{number of } i \text{ dimensional Vornoi faces} = \text{number of } D-i \text{ dimensional Delaunay faces}$$

The table below evaluates the equations from the preceding subsections to provide a hands-on feeling for the behavior of these equations. The columns Typical # of simplices attached to a vertex and Typical # of nearest neighbors come directly from the subsections in this report. The column # of “kissing” spheres is from [ODLY79]. The columns Maximum # of simplices using Typical # of nearest neighbors + 1 vertices and Maximum # of simplices using # of “kissing” spheres + 1 vertices evaluate the Paschinger/Seidel equation for the typical number of nearest neighbors and for the number of “kissing” spheres plus one extra vertex.

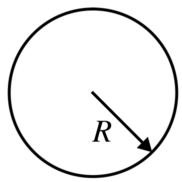
Table 1: Evaluation of Equations

Dim	Typical # of simplices attached to a vertex	Typical # of nearest neighbors	Maximum # of simplices using Typical # of nearest neighbors + 1 vertices	# of “kissing” spheres	Maximum # of simplices using # of “kissing” spheres + 1 vertices
2	6	6	9	6	9
3	24	14	89	12	64
4	120	30	755	25	505
5	720	62	35930	46	14146
6	5040	126	605241	82	158157
7	40320	254	164028749	140	14453909
8	362880	510	5398326779	240	251982509
9	3628800	1022	9065944285397	380	61787179574
10	39916800	2046	587400559978595	595	1171464411235

Volumes of High Dimensional Objects

This section presents a list of equations for the volumes of simple D dimensional objects. The objects examined are the sphere, cube, right simplex and equilateral simplex (graphs are at the end of the section). Ratios of these volumes become significant for estimating statistical likelihoods of finding points in certain regions.

Sphere of radius R:

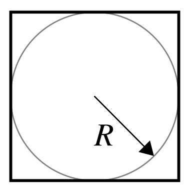


$$\text{Volume} = \frac{\pi^{D/2}}{\text{Gamma}\ (D/2+1)} \cdot R^D$$

where Gamma (0.5) $= \sqrt{\pi}$
Gamma (1) $= 1$
Gamma (n) $=$ (n-1) Gamma (n-1)

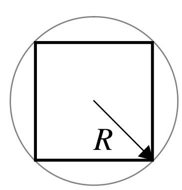
This equation appears in a slightly different form in [SOMM58].

Cube circumscribing Sphere of radius R:



$$\text{Volume} = (2 \cdot R)^D$$

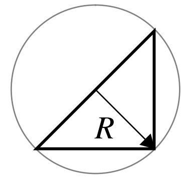
Cube circumscribed by Sphere of radius R:



The main diagonal of the cube will always be 2R regardless of its dimension. This dictates that the length of a side of the cube must be $\frac{2}{\sqrt{D}} \cdot R$

$$\text{Volume} = \left(\frac{2}{\sqrt{D}} \cdot R\right)^D$$

Right Simplex circumscribed by Sphere of radius R:



The volume of a D dimensional simplex is given by

$$\text{volume} = \frac{\left|\det\left[\text{vector spans}_{D\times D}\right]\right|}{D!}$$

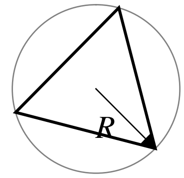
Treating the "base" vertex (the vertex where all the right angles exist) as the origin from which to compute the spans, yields

$$\text{volume} = \frac{\left|\det\begin{bmatrix} 0-\frac{2}{\sqrt{D}} & \frac{2}{\sqrt{D}}-\frac{2}{\sqrt{D}} & \dots & \frac{2}{\sqrt{D}}-\frac{2}{\sqrt{D}} \\ \frac{2}{\sqrt{D}}-\frac{2}{\sqrt{D}} & 0-\frac{2}{\sqrt{D}} & \dots & \frac{2}{\sqrt{D}}-\frac{2}{\sqrt{D}} \\ \dots & \dots & \dots & \dots \\ \frac{2}{\sqrt{D}}-\frac{2}{\sqrt{D}} & \frac{2}{\sqrt{D}}-\frac{2}{\sqrt{D}} & \dots & 0-\frac{2}{\sqrt{D}} \end{bmatrix}\right|}{D!}$$

producing the result

$$\text{Volume} = \frac{\left|\det\begin{bmatrix} -\frac{2}{\sqrt{D}} & 0 & \dots & 0 \\ 0 & -\frac{2}{\sqrt{D}} & \dots & 0 \\ \dots & \dots & \dots & \dots \\ 0 & 0 & \dots & -\frac{2}{\sqrt{D}} \end{bmatrix}\right|}{D!} = \frac{\left(\frac{2}{\sqrt{D}}\cdot R\right)^{D}}{D!}$$

Equilateral Simplex circumscribed by Sphere of radius R:



The equilateral simplex's volume promises to be the most difficult to determine. The derivation will proceed in three steps. First, the coordinates of an equilateral simplex will be found. Second, the volume of this simplex will be computed. And third, the radius of the enclosing sphere will be calculated.

Given the unit vectors on the coordinate axes of a D dimensional space, their endpoints collectively form a D-1 dimensional equilateral simplex[1]. We can create a D dimensional simplex by finding a new vertex which is a distance of $\sqrt{2}$ from the endpoints of all the unit vectors. By symmetry, this new vertex must have the form (a, a, a, ..., a).

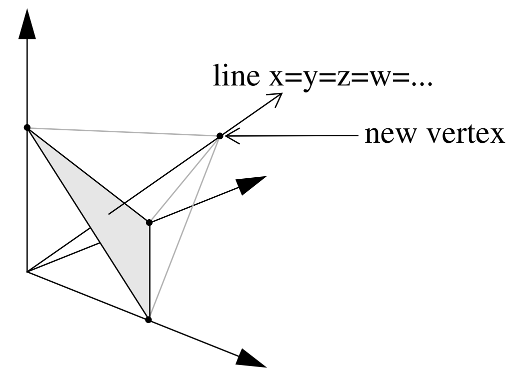


Figure 3: Geometric construction of 3D equilateral simplex

The distance between the new vertex and any of the endpoints of the unit vectors will be

$$\sqrt{(D-1)\ (a-0)^2 + (a-1)^2}$$

Setting this equal to $\sqrt{2}$ and solving for a yields

$$a = \frac{1 \pm \sqrt{D+1}}{D}$$

We are given two choices for a, and we'll chose one arbitrarily:

$$\text{new vertex} = \left(\frac{\sqrt{D+1}+1}{D},\ \frac{\sqrt{D+1}+1}{D},\ \ldots,\ \frac{\sqrt{D+1}+1}{D}\right)$$

The new vertex can quickly be verified by computing the distance between it and all other vertices. Noting that all the other vertices are vectors made up of "0"s except for a single "1" demonstrates that these distances must be equal. Solving for them shows that they are in fact $\sqrt{2}$.

1. This can be verified by noting that the distance between any two of these endpoints is $\sqrt{2}$.

Now that we have all the vertices of the D dimensional equilateral simplex, the volume of the simplex will be computed using the following formula

$$\text{volume} = \frac{\left|\det\left[\text{vector spans}_{D \times D}\right]\right|}{D!}$$

Treating vertex (1, 0, 0, ..., 0) as the origin from which to compute the spans yields

$$\text{volume} = \frac{\left|\det\begin{bmatrix} \frac{\sqrt{D+1}+1}{D} - 1 & \frac{\sqrt{D+1}+1}{D} & \frac{\sqrt{D+1}+1}{D} & \dots & \frac{\sqrt{D+1}+1}{D} \\ -1 & 1 & 0 & \dots & 0 \\ -1 & 0 & 1 & \dots & 0 \\ \dots & \dots & \dots & \dots & \dots \\ -1 & 0 & 0 & \dots & 1 \end{bmatrix}\right|}{D!}$$

producing the result

$$\text{volume} = \frac{\sqrt{D+1}}{D!}$$

Now that we have found the volume of the equilateral simplex of edge length $\sqrt{2}$, we must find the radius of its circumsphere. This can be accomplished by using the function **MakeSphere-FromPoints()**[1] or it can be constructed directly from the geometry. The center of the circumsphere must be an equal distance from the new vertex and the endpoints of the unit vectors. Symmetry again requires this point to have the form (b, b, b, ..., b).

$$\sqrt{D\left(\frac{\sqrt{D+1}+1}{D} - b\right)^2} = \sqrt{(1-b)^2 + (D-1)\,b^2}$$

Symbolic manipulation eventually reduces to

$$b = \frac{1+\sqrt{D+1}}{D\sqrt{D+1}}$$

1. This function is described in Appendix A, Solving Equations for High Dimensional Objects.

Knowing the coordinates of the center of the circumsphere allows the radius can be calculated in a straight forward manner

$$\text{radius}^2 = D\left(\frac{\sqrt{D+1}+1}{D} - \frac{1+D\sqrt{D+1}}{D\sqrt{D+1}}\right)^2$$

$$\text{radius} = \sqrt{\frac{D}{D+1}}$$

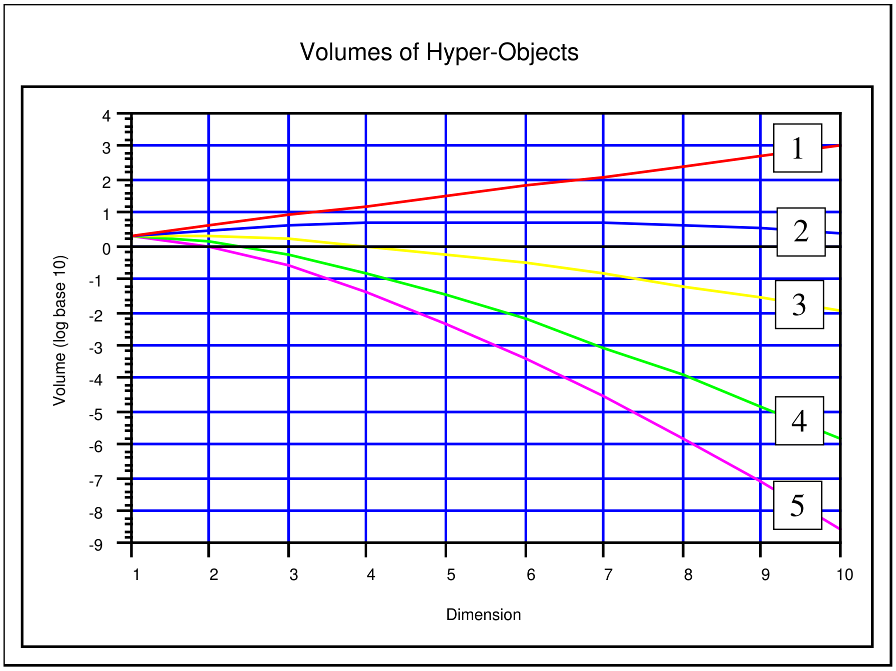
We now know that given a sphere of radius $\sqrt{\frac{D}{D+1}}$, an equilateral simplex inscribed by it will have a volume of $\frac{\sqrt{D+1}}{D!}$. To know the volume of an equilateral simplex inscribed by a sphere of radius R requires a normalization. Scaling all dimensions by $\sqrt{\frac{D+1}{D}} \cdot R$ normalizes to a sphere of radius R.

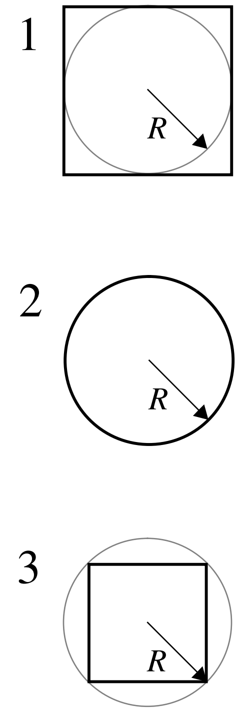
The volume of an equilateral simplex inscribed a sphere of radius R is

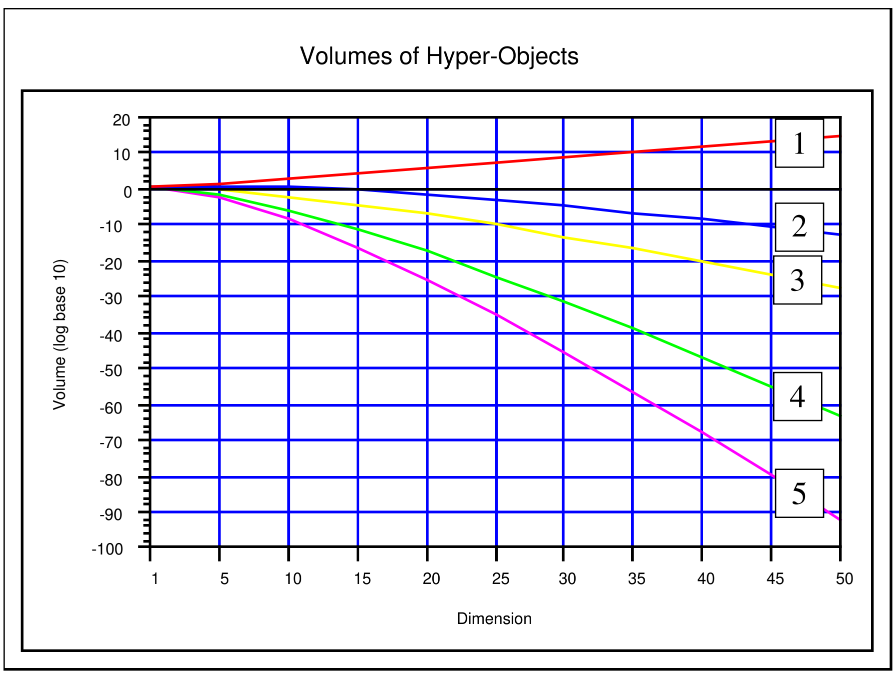
$$\text{Volume} = \frac{\sqrt{D+1}}{D!}\left(\sqrt{\frac{D+1}{D}} \cdot R\right)^D$$

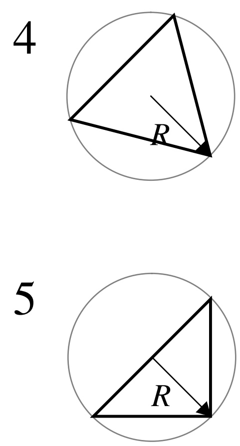
Table 2: Summary of Volumes of High Dimensional Objects

Object	Volume
Sphere of radius R	$\frac{\pi^{D/2}}{\text{Gamma}\ (D/2+1)} \cdot R^D$
Cube circumscribing Sphere	$2^D \cdot R^D$
Cube circumscribed by Sphere	$\frac{2^D}{D^{D/2}} \cdot R^D$
Right Simplex circumscribed by Sphere	$\frac{2^D}{D! \cdot D^{D/2}} \cdot R^D$
Equilateral Simplex circumscribed by Sphere	$\frac{(D+1)^{(D+1)/2}}{D! \cdot D^{D/2}} \cdot R^D$


Volumes of Hyper-Objects
Volume (log base 10)
Dimension
1
2
3
4
5


1
R
2
R
3
R


Volumes of Hyper-Objects
Volume (log base 10)
Dimension
1
2
3
4
5


4
R
5
R

Conclusions

These pages have shown how significant a role dimensionality plays in determining the properties of a data set. With increasing dimensionality, the volumes of simplices relative to their enclosing spheres and cubes decrease dramatically. Data sets begin to lie entirely on or near the surface of their convex hull. Query points become less and less likely to lie within this convex hull. When query points are within the convex hull, interpolations are ill-defined due to the inclusion of far away points in the interpolation. The picture becomes even bleaker when the degree of interconnect and the sharing of features is considered. Exploring these spaces efficiently appears to be intractable.

Fortunately the difficulties are surmountable. With sufficient care, algorithms can be developed to operate efficiently on high dimensional data sets. The lesson was a painful one to learn and our hope in making this lesson as painless as possible for other researchers is the reason for its inclusion in this report.

3 Finding the Enclosing Delaunay Simplex

This chapter describes four algorithms employed to find a D-dimensional Delaunay simplex which encloses a D-dimensional query point. If all D-dimensional data points are assumed to have scalar values associated with them, then these values could be linearly interpolated to yield the scalar value for the query point. It is proved in [OMOH90] that "the Delaunay triangulation gives rise to a piecewise linear approximation with the smallest maximum error at each point over" functions with bounded curvature.

The four algorithms are presented in the chronological order in which we thought of them. We believe that our flawed intuition, or even misconceptions, about high dimensions are relatively common, and that the reader may learn something worthwhile by following our mental struggles. Frequently we succumbed to the hope that our intuition with 2 and 3 dimensional spaces would aid us in high-dimensional spaces. In truth, our intuition led us astray at every opportunity. We were further hampered because one tends to think two dimensionally when one starts with a pen and paper. While the first two algorithms (and possibly even the third) may seem somewhat naive, they are presented for didactical reasons. Readers just interested in the "correct" approach can skip Sections 3.1 through 3.3.

All query points are assumed to lie within the convex hull of the data set. Handling query points outside the convex hull is discussed in Chapter 4.

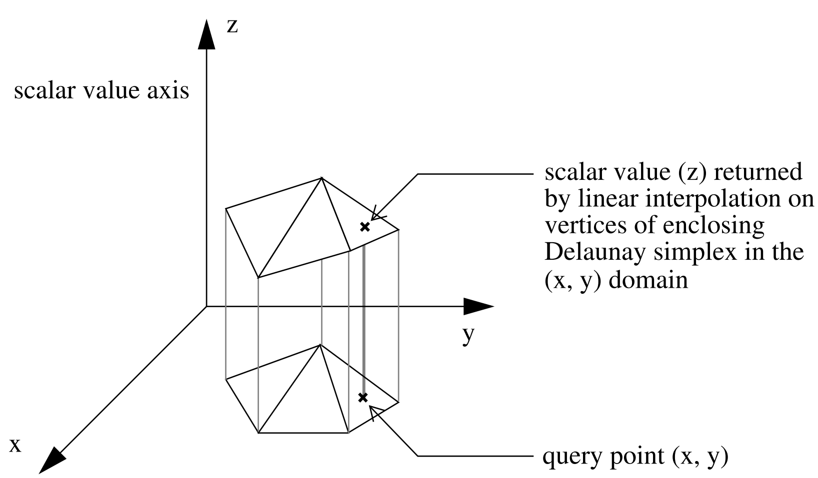


Figure 4: Example of Linear Interpolation

Appendix A describes in detail each of the mathematical functions used by the algorithms. Briefly, these are:

MakePlaneFromPoints() -- takes D D-dimensional points and returns the coefficients of the D-dimensional plane that passes through them.

MakePlaneFromNormal() -- takes a D-dimensional point and a D-dimensional normal (not necessarily of unit length) and returns the coefficients of the D-dimensional plane that passes through the point and is perpendicular to the normal.

TestPointVsPlane() -- tests an D-dimensional point against a D-dimensional plane and returns a measure of its distance from the plane.

BarycentricCoords() -- returns the D+1 barycentric coordinates of a D-dimensional point relative to D+1 D-dimensional points.

MakeSphereFromPoints() -- takes D+1 D-dimensional points and returns the center and radius of the D-dimensional sphere that passes through them.

Notice that the barycentric coordinates of a query point contained within a simplex can be used as the weights for a linear interpolation of the scalar values of the vertices. If a query point exactly matches one of the data points of an enclosing simplex, then its barycentric coordinates will all be 0 except for a single 1 where it matches the data point. In this case, the scalar value returned will be exactly the scalar value present at that data point.

All the algorithms presented below require a method to find the M closest data points to a query point. Using brute force, this can be accomplished by computing the distance (really distance squared) from the query point to every data point and keeping track of the M closest. We can, however, do better. A Kd-Tree is a data structure that partitions data points of arbitrary dimensionality into cells. By exploiting this spatial partitioning, finding the M closest points does not require computing the distance from the query point to every data point. Kd-Trees are discussed in Appendix B along with measurements demonstrating their effectiveness. Regardless of how it is implemented, the following function will be used by all the algorithms:

GetClosest(query_pt, M, point_list) -- returns a list of the M closest data points to *query_pt* sorted by distance (the closest data point is at the head of the list). GetClosest() can be called repeatedly with a higher value of M to increase the number of points in *point_list*.

3.1 A Backtracking Simplex Finder

In our first attempt at finding a Delaunay simplex enclosing a query point, we felt that it would be easier to find any local simplex enclosing the query point, and then "convert" this simplex into a Delaunay simplex. By looking at data points in order of increasing distance from the query point, we hoped to quickly arrive at a simplex enclosing the query point.

Since we are not concentrating on finding the "best" enclosing simplex (the Delaunay simplex), we felt that virtually any local enclosing simplex would do just fine. More specifically, it is not necessary to immediately detect an enclosing simplex when one exists. By having the algorithm backtrack on its (hopefully) infrequent failures, we hoped to exploit a simple and fast algorithm for most cases. These savings in time could then be used in the conversion of the simplex to Delaunay. We were wrong.

The Backtracking Simplex Finder:

Fetch the M nearest points to the query point Q into *point_list* using GetClosest(). The value of M should be set to optimize performance. It certainly must be a function of dimension, number of data points, or both. A discussion of how to choose a value for M is in Section 3.3, for reasons that will later become apparent.

Take the first D points from *point_list*. Let *mid_point* be the center of gravity of these D points.

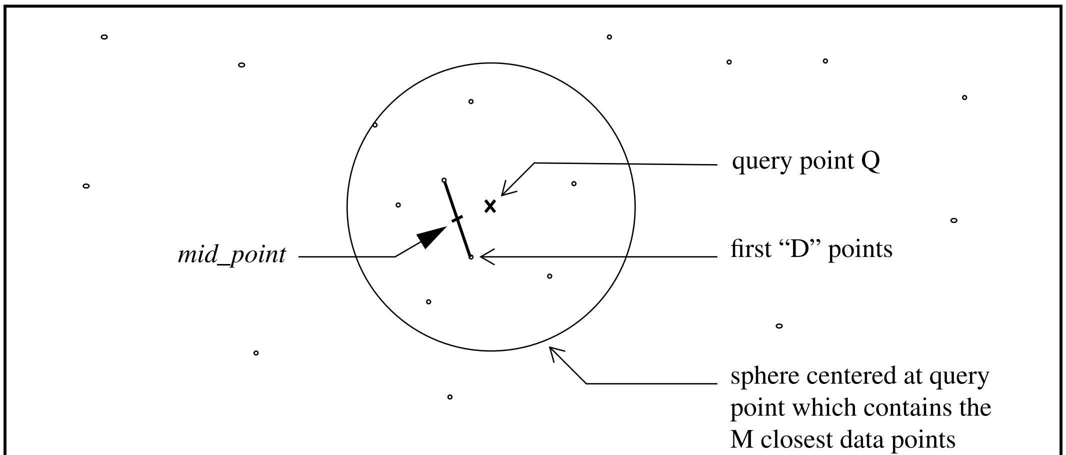


Figure 5: First step in Backtracking Simplex Finder

Solve for the coefficients of the D D-dimensional planes that pass through Q and each set of D-1 points using MakePlaneFromPoints(). These D planes form a double "cone". Any subsequent points in the frustum "below" the D-1 points can be ignored as they cannot possibly form a simplex which encloses Q. Any subsequent point in the cone "above" Q, together with the D-1 points will form a simplex which encloses Q. If a point is in neither of these two spaces, the double cone will be reevaluated.

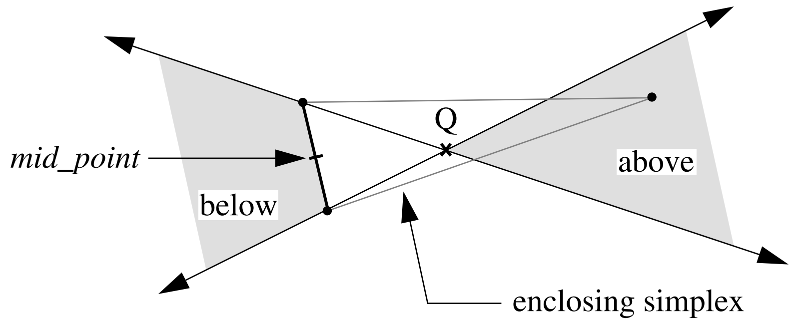


Figure 6: Examples of “above” and “below” cones

Test *mid_point* versus the D plane equations using TestPointVsPlane() and store the sign of each of the resulting values.

Main Loop:

Get the next closest point P from *point_list*. If there are no more points to take from *point_list* then we have examined all M points without finding an enclosing simplex. We can call GetClosest() repeatedly to fetch the next M closest points to Q. If we have fetched all the data points and still have not found an enclosing simplex, we will have to test for Q being outside the convex hull (this is not a trivial prospect). If it is found that Q is not outside the convex hull, then backtracking will be required to find a solution (see end of this algorithm).

Test the point P versus the D plane equations. If the signs are all *opposite* to the signs from the *mid_point* test, then P lies within the cone “above” Q. We have found a simplex enclosing Q and the algorithm is done. If the signs from this test are all the *same* as the signs from the *mid_point* test, then P lies within the cone “below” the D-1 points and can be ignored. Go back to the **Main Loop**.

If the signs are mixed, we need to adjust the D planes so that they take into account the new point P. We need to select one of the D points to be replaced by P so that the new cones formed by the new planes will increase in volume and retain as much of the volume they used to span as possible. This “best” point can be determined by the results of the plane equation tests; it will be the one with the highest number of opposite signs. After replacing a point with P it will be necessary to recalculate the plane equations (notice that only one plane equation will remain the same) and test the new *mid_point* versus these planes again. Go back to the **Main Loop**.

Analysis:

A nice feature of this algorithm (which will persist throughout all of our algorithms) is that its space requirements are static. No matter how many points are looked at, space is only needed for a few vectors to hold the sign tests and the coefficients of the D plane equations.

Unfortunately, this algorithm is poor in two respects. First, its execution time. The basic unit of work is solving D D-dimensional plane equations taking $O(D^4)$ time. Clearly, this algorithm cannot hope to be better than $O(D^4)$, and will probably be closer to $O(D^5)$ or $O(D^6)$. Second, it will not always find a solution when a solution is possible. When the D planes are adjusted to account for the new point P, the size of the cone defined by Q and the D planes can shrink along some dimensions. In cases such as this, it is possible for an enclosing simplex to exist using some of the previously examined points. The situation will not be detected because the point P does not fall within the current cone. It is this situation that must be checked for in the backtracking stage mentioned. Preliminary tests showed that as the dimensionality of the problem increases, backtracking is required more and more frequently. It quickly became apparent that there was no frequently used common case worth optimizing for speed. We did not try to implement the backtracking algorithm.

The failure of this algorithm was primarily due to our own two dimensional thinking (the pen and paper syndrome). In 2D, solving 2 2-dimensional plane equations is cheap. In dimension D, solving D D-dimensional plane equations is expensive. In 2D backtracking is never required. Each new point P can be used to increase the “volume” of the cones in such a way that previous “volumes” are totally enclosed. However, this is not the case in dimension D. We thus realized that such a backtracking algorithm is unacceptable. The next section describes a non-backtracking simplex finder that is better suited to high dimensions.

3.2 A Non-Backtracking Simplex Finder

In our second attempt at finding the canonical Delaunay simplex enclosing a query point, we still felt that it would be easier to first search for any local simplex enclosing the query point, and then "convert" this simplex into a Delaunay simplex. However, the results from the first algorithm dictated that backtracking must be avoided. By looking at points in order of increasing distance from the query point and guaranteeing that an enclosing simplex be found as soon as possible, the simplices returned should be very close to (and in some cases may actually be) the enclosing Delaunay simplex. We assumed that the conversion process would be fairly straightforward. Again, we were wrong.

The Non-backtracking Simplex Finder:

Fetch the M nearest points to the query point Q into *point_list* using GetClosest(). A discussion of how to choose a value for M is in Section 3.3.

Take the first D points from *point_list*.

Solve for the coefficients of the D-dimensional plane passing through these D points using MakePlaneFromPoints(). This plane, P_{cut}, partitions space into an interesting half and an uninteresting half. Any subsequent point on the same side of P_{cut} as Q has the possibility of enclosing Q by forming a simplex with some of the points previously examined. Any subsequent point on the other side of P_{cut} cannot possibly enclose Q by forming a simplex with any of the previously examined points. The heart of this algorithm is in the positioning of P_{cut} such that no previously seen points lie on the same side of P_{cut} as Q. When this is not possible, we will have found a simplex enclosing Q.

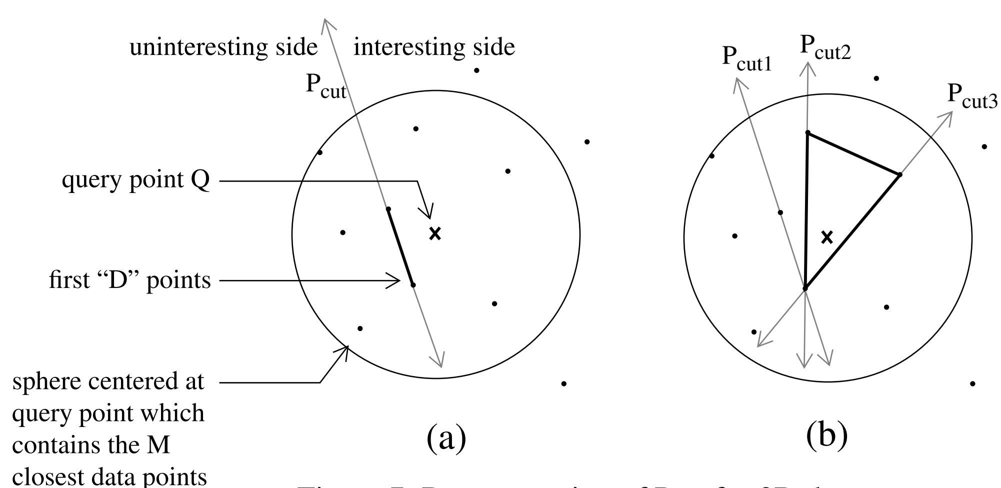


Figure 7: Demonstration of P_{cut} for 2D data set

Main Loop:

Get the next closest point P from *point_list*. If there are no more points to take from *point_list*, then we have examined all M points without finding an enclosing simplex. If M was chosen properly, then it is likely that Q is outside the convex hull of the entire data set. Instead of incurring the cost of repeated calls to GetClosest(), we immediately test all remaining points in the data set against P_{cut} using TestPointVsPlane(). If no points lie on the same side of P_{cut} as Q, then Q is outside the convex hull. If any points are on the same side of P_{cut} as Q, then we can use a modified GetClosest() to return all points within the radius from Q to the data point that had the largest value returned from its plane test with P_{cut}.

Test the point P versus P_{cut} using TestPointVsPlane(). If P is not on the same side of P_{cut} as Q then it is uninteresting (for the moment) and we go back to the **Main Loop**.

Get the barycentric coordinates of P in terms of Q and the D points that define P_{cut} using BarycentricCoords(). If the barycentric coordinate corresponding to Q is greater than 1 and all the other barycentric coordinates are less than 0, then P and the D points form a simplex that encloses Q.

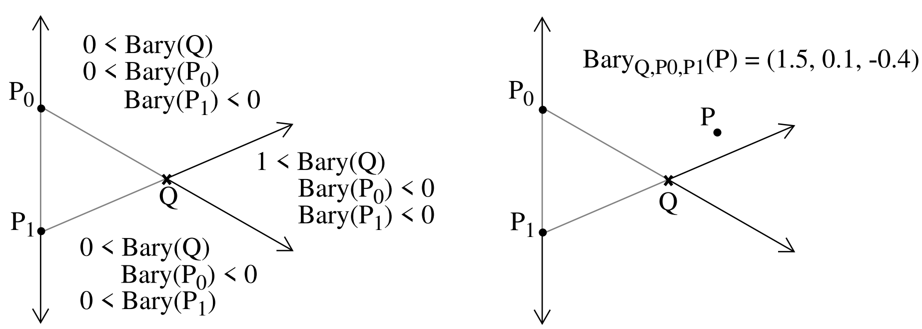


Figure 8: Example of use of barycentric coordinates

If P and the D points do not form a simplex enclosing Q, then we need to find a new P_{cut}. Use P to replace the point that corresponds to the highest barycentric coordinate and recalculate P_{cut}. Notice that P_{cut} may no longer partition space such that Q and all previously seen data points are on opposite sides.

Inner Loop:

Test all previously seen data points against the new P_{cut}. If the new P_{cut} partitions space such that all the previously seen points are on one side and Q is on the other side then go back to the **Main Loop**.

When all the previously seen points were being tested versus the new P_{cut}, the point with the worst failure (the one farthest into the "wrong" side) needs to be remembered. Call this point *pt_worst*. Calculate the barycentric coordinates of *pt_worst* in terms of Q and the current D points. Check the barycentric coordinates to see if *pt_worst* forms a simplex enclosing Q. If not, use *pt_worst* to replace the point that corresponds to the highest barycentric coordinate. Recalculate P_{cut} and go back to the **Inner Loop**.

Analysis:

Overall, we felt that this algorithm performed quite satisfactorily. Like the backtracking algorithm, its space requirements are static. Unlike the backtracking algorithm, the plane P_{cut} gives us a simple and direct test for Q being outside the convex hull. How to implement this essential test was conveniently overlooked in the backtracking algorithm.

An important observation was made between the algorithm in section 3.1 and this one. Our test for a point being inside or outside of a simplex previously involved solving D+1 D-dimensional plane equations and then testing a point versus these D+1 planes. This made the complexity of the algorithm's unit of work $O(D^4)$. By using barycentric coordinates, we can solve a single D+1-dimensional "plane equation" and arrive at the same result. This reduces the complexity to $O(D^3)$.

Statistics were gathered by running this algorithm over 100 randomly chosen query points. The query points were derived from the same distribution as the original data points were (Uniform over a cube and Gaussian distributions[1]). Although the statistics were not all compiled at the same time, the data points and query points were exactly the same over all runs[2].

The following two tables and graph show the average number of plane equations solved as a function of dimensionality and number of data points. Since the query points were drawn from a random distribution, these numbers reflect how long it takes to either find an enclosing simplex or determine that the query point is outside the convex hull.

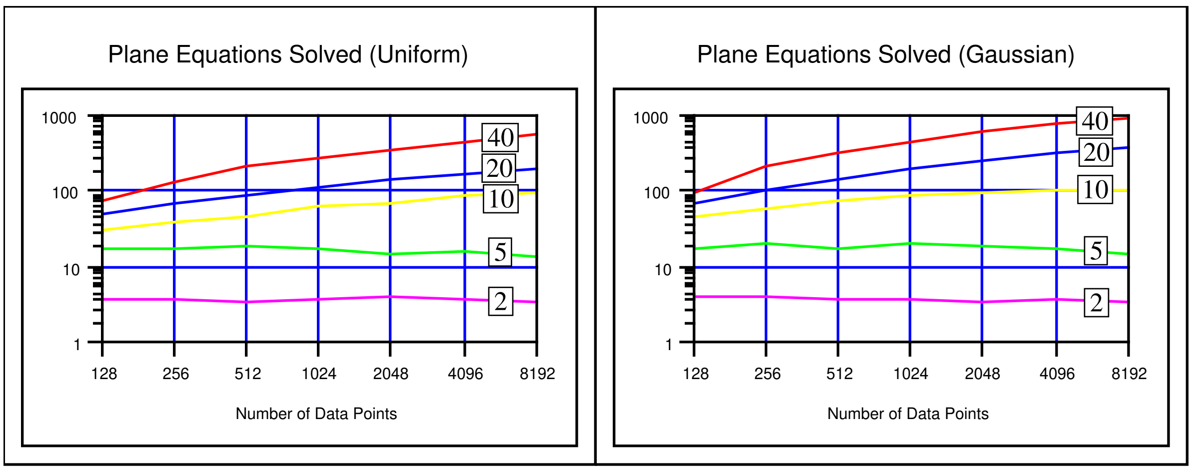
Table 3: Number of Plane Equations Solved (Uniform)

Dim	128 pts	256 pts	512 pts	1024 pts	2048 pts	4096 pts	8192 pts
2	3.64	3.90	3.62	3.70	3.98	3.94	3.54
5	17.28	17.86	18.36	17.56	14.94	16.18	14.28
10	31.18	37.96	47.62	62.44	69.80	85.14	97.18
20	50.74	70.38	87.44	115.74	142.32	171.24	193.84
40	73.38	135.86	208.14	282.32	352.16	435.68	554.16

1. Because the query points and the data points are drawn from the same distributions, the actual parameters of these distributions are not important.
2. By using the same initial random number seed, a random series is exactly reproducible.

Table 4: Number of Plane Equations Solved (Gaussian)

Dim	128 pts	256 pts	512 pts	1024 pts	2048 pts	4096 pts	8192 pts
2	4.12	4.18	3.76	3.72	3.54	3.88	3.38
5	18.28	21.42	18.22	20.86	19.26	17.34	15.22
10	44.90	59.16	75.06	86.88	97.74	101.10	104.00
20	69.58	103.96	145.64	201.38	253.48	314.16	381.42
40	92.16	210.26	323.26	460.74	628.96	754.96	909.80



It should be noted that it is possible to reformulate this algorithm to solve slightly fewer plane equations. By translating P_{cut} so that it passes through the query point instead of the D points, one can minimize the number of data points that actually cause P_{cut} to be recalculated. However, it was found that this did not speed up the algorithm appreciably (especially in higher dimensions) and only served to obscure the algorithm description. This method was therefore not used.

The next two tables and graph give the average execution time of this algorithm on 100 query points. Single precision floating point math was used on a SPARCstation2 with 16MB of real memory[1].

The reported times do not include the time to fetch the M nearest neighbors! We wanted to focus on the running time of the algorithm as opposed to the running time of our Kd-Trees (Kd-Trees are really just a sorting algorithm). For the purposes of these runs, a complete list of all data

1. Ignoring code size and data structure overhead, our largest data file only takes up 1.25MB (8192 points * 40 dimensions * 4 bytes/float) and so paging does not enter into these execution times.

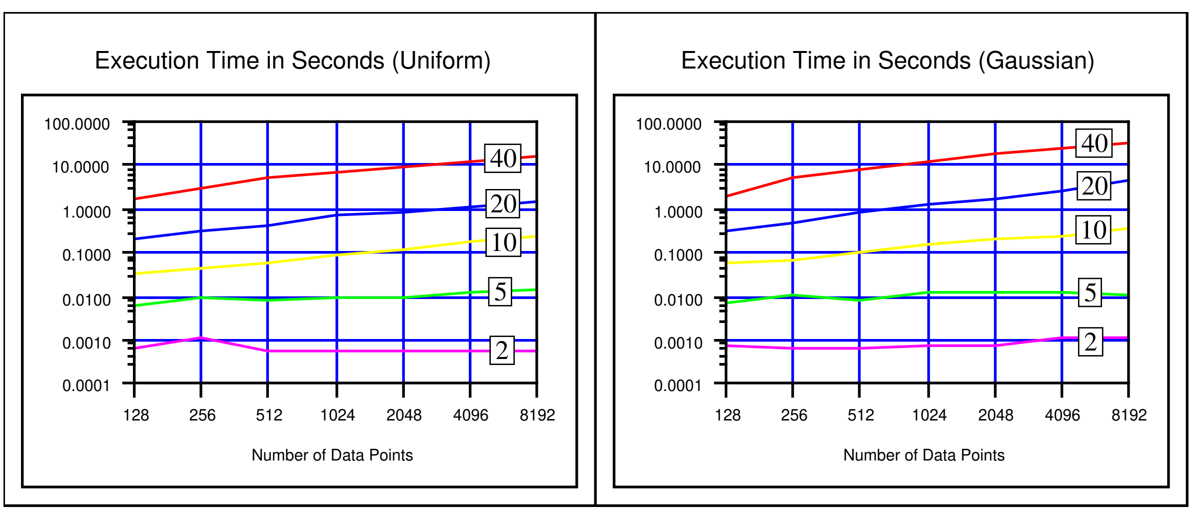
points sorted by distance to the query point was built outside of the timing loop (in the worst case this would take O(n logn) time).

Table 5: Execution Time in Seconds (Uniform)

Dim	128 pts	256 pts	512 pts	1024 pts	2048 pts	4096 pts	8192 pts
2	0.0007	0.0011	0.0006	0.0006	0.0006	0.0006	0.0006
5	0.0067	0.0091	0.0083	0.0095	0.0098	0.0124	0.0143
10	0.0340	0.0454	0.0617	0.0926	0.1205	0.1867	0.2440
20	0.2221	0.3152	0.4109	0.7288	0.8334	1.0736	1.4222
40	1.6560	3.1778	4.9421	7.0130	9.4124	12.1792	17.1561

Table 6: Execution Time in Seconds (Gaussian)

Dim	128 pts	256 pts	512 pts	1024 pts	2048 pts	4096 pts	8192 pts
2	0.0008	0.0007	0.0007	0.0008	0.0008	0.0012	0.0012
5	0.0074	0.0106	0.0082	0.0122	0.0118	0.0126	0.0113
10	0.0575	0.0724	0.1096	0.1524	0.2141	0.2514	0.3491
20	0.3029	0.4750	0.8499	1.2073	1.7598	2.7572	4.6088
40	2.1044	5.0583	7.9070	11.7839	17.5984	23.6056	32.4771



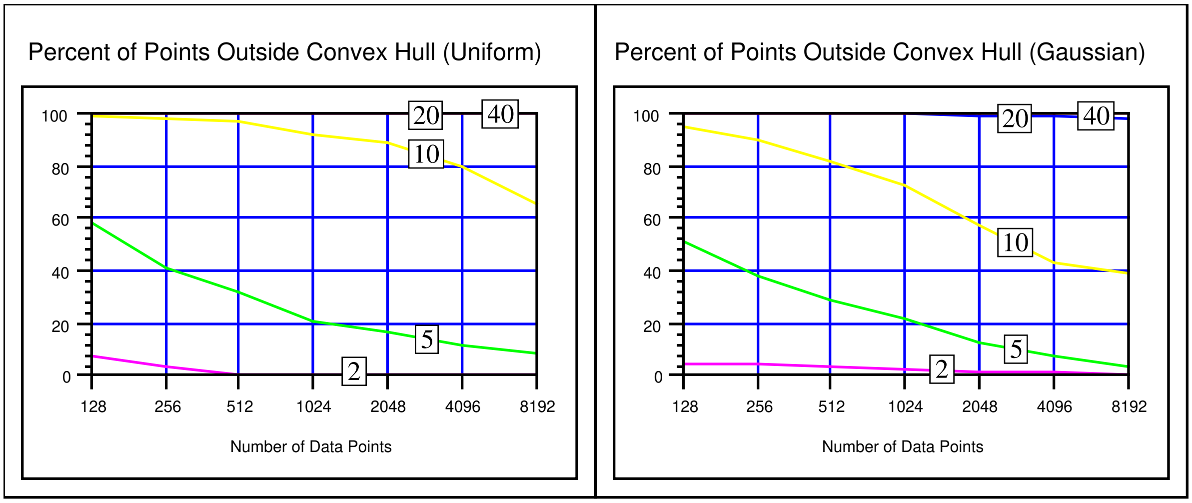
The next two tables and graph provide some additional insight into the previous statistics. Given that the 100 query points were drawn from a random distribution, it would be very interesting to know for how many points was an enclosing simplex found versus being outside the convex hull.

Table 7: Percent of Points Outside Convex Hull (Uniform)

Dim	128 pts	256 pts	512 pts	1024 pts	2048 pts	4096 pts	8192 pts
2	8	4	0	0	0	0	0
5	58	41	32	21	17	12	9
10	99	98	97	92	89	80	65
20	100	100	100	100	100	100	100
40	100	100	100	100	100	100	100

Table 8: Percent of Points Outside Convex Hull (Gaussian)

Dim	128 pts	256 pts	512 pts	1024 pts	2048 pts	4096 pts	8192 pts
2	5	5	4	3	2	2	1
5	51	38	29	22	13	8	4
10	95	90	82	73	57	43	39
20	100	100	100	100	99	99	98
40	100	100	100	100	100	100	100



In dimension 40 none of the query points are within the convex hull of the data set! [This is also nearly true in dimension 20.] This tells us that we really don't know how the algorithm will perform in D40 with "meaningful" query points. It also suggests that our problem is not really a 40 dimensional one. Problems of this type are likely to be lower dimensional manifolds embedded in a higher dimensional space. Our random data sets are definitely not indicative of real-world high dimensional problems. First of all, one would expect there to be a certain amount of correlation between a problem's variables (i.e. the pressure and volume of an ideal gas or temperatures taken at nearby points). Second, there should also be regions where taking small steps along one dimension produces very little change in the other dimensions (i.e. walking down the length of a pipe). For subsequent tests we will confine the query points to lie strictly within the convex hull of the data set.

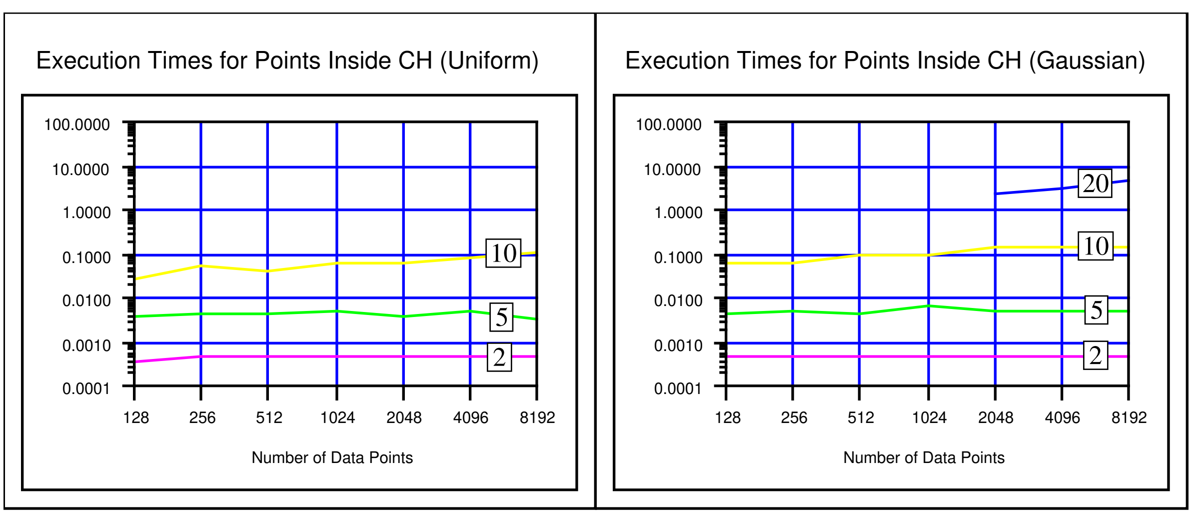
The next two tables and graph give the average execution times in seconds of this algorithm only for those query points that were contained within the convex hull of the data set. Special care must be taken when looking at these tables. The number of samples for each cell can range from 100 to only 1 and therefore the statistical accuracy of each cell can vary widely. For the reader's convenience, each statistic is subscripted by the number of samples that contributed to it.

Table 9: Execution Times for Query Points Inside Convex Hull in Seconds (Uniform)

Dim	128 pts	256 pts	512 pts	1024 pts	2048 pts	4096 pts	8192 pts
2	0.0004_{92}	0.0005_{96}	0.0005_{100}	0.0005_{100}	0.0005_{100}	0.0005_{100}	0.0005_{100}
5	0.0042_{42}	0.0048_{59}	0.0047_{68}	0.0050_{79}	0.0041_{83}	0.0049_{88}	0.0035_{91}
10	0.0270_{1}	0.0556_{2}	0.0449_{3}	0.0635_{8}	0.0622_{11}	0.0833_{20}	0.1152_{35}
20	n/a	n/a	n/a	n/a	n/a	n/a	n/a
40	n/a	n/a	n/a	n/a	n/a	n/a	n/a

Table 10: Execution Times for Query Points Inside Convex Hull in Seconds (Gaussian)

Dim	128 pts	256 pts	512 pts	1024 pts	2048 pts	4096 pts	8192 pts
2	0.0005_{95}	0.0005_{95}	0.0005_{96}	0.0005_{97}	0.0005_{98}	0.0005_{98}	0.0005_{99}
5	0.0043_{49}	0.0055_{62}	0.0048_{71}	0.0066_{78}	0.0053_{87}	0.0053_{92}	0.0053_{96}
10	0.0639_{5}	0.0670_{10}	0.0911_{18}	0.0975_{27}	0.1446_{43}	0.1544_{57}	0.1471_{61}
20	n/a	n/a	n/a	n/a	2.3233_{1}	3.1602_{1}	4.6600_{2}
40	n/a	n/a	n/a	n/a	n/a	n/a	n/a



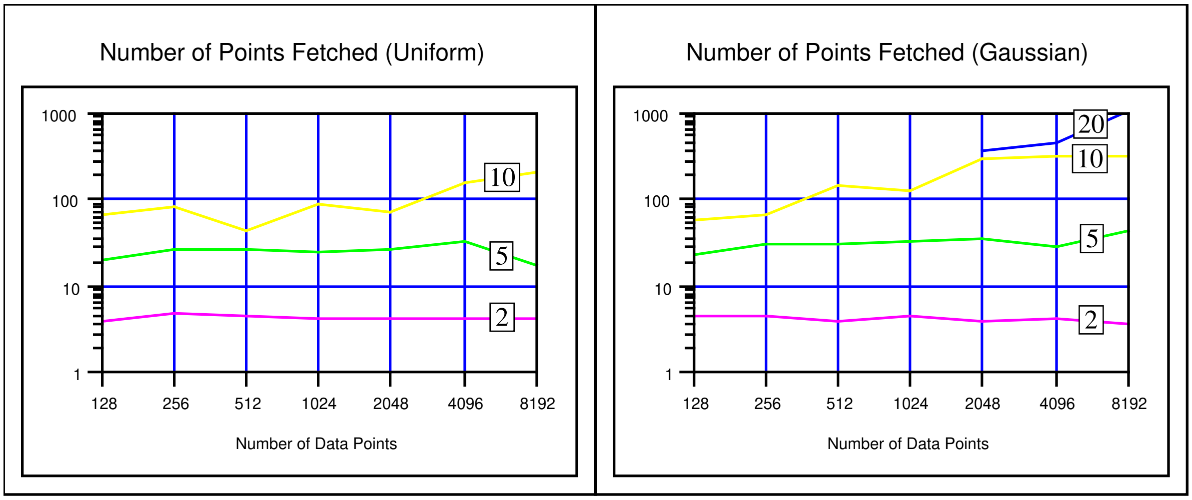
The next two tables and graph tell on average how many points the algorithm had to look at before it could find an enclosing simplex. This table does not include data on those query points which were outside the convex hull (by necessity, the algorithm must look at all the data points before it can determine that a query point is totally outside the convex hull).

Table 11: Number of Points Fetched (Uniform)

Dim	128 pts	256 pts	512 pts	1024 pts	2048 pts	4096 pts	8192 pts
2	3.93_{92}	4.77_{96}	4.53_{100}	4.19_{100}	4.40_{100}	4.23_{100}	4.35_{100}
5	20.83_{42}	26.63_{59}	26.21_{68}	24.53_{79}	27.78_{83}	32.20_{88}	17.73_{91}
10	67.00_{1}	86.50_{2}	43.00_{3}	90.00_{8}	71.82_{11}	156.95_{20}	213.37_{35}
20	n/a	n/a	n/a	n/a	n/a	n/a	n/a
40	n/a	n/a	n/a	n/a	n/a	n/a	n/a

Table 12: Number of Points Fetched (Gaussian)

Dim	128 pts	256 pts	512 pts	1024 pts	2048 pts	4096 pts	8192 pts
2	4.67_{95}	4.67_{95}	4.02_{96}	4.67_{97}	3.89_{98}	4.29_{98}	3.74_{99}
5	24.22_{49}	30.31_{62}	30.76_{71}	32.56_{78}	36.55_{87}	28.04_{92}	44.05_{96}
10	59.20_{5}	69.00_{10}	146.06_{18}	129.52_{27}	290.56_{43}	315.86_{57}	326.13_{61}
20	n/a	n/a	n/a	n/a	373.00_{1}	460.00_{1}	1063.00_{2}
40	n/a	n/a	n/a	n/a	n/a	n/a	n/a



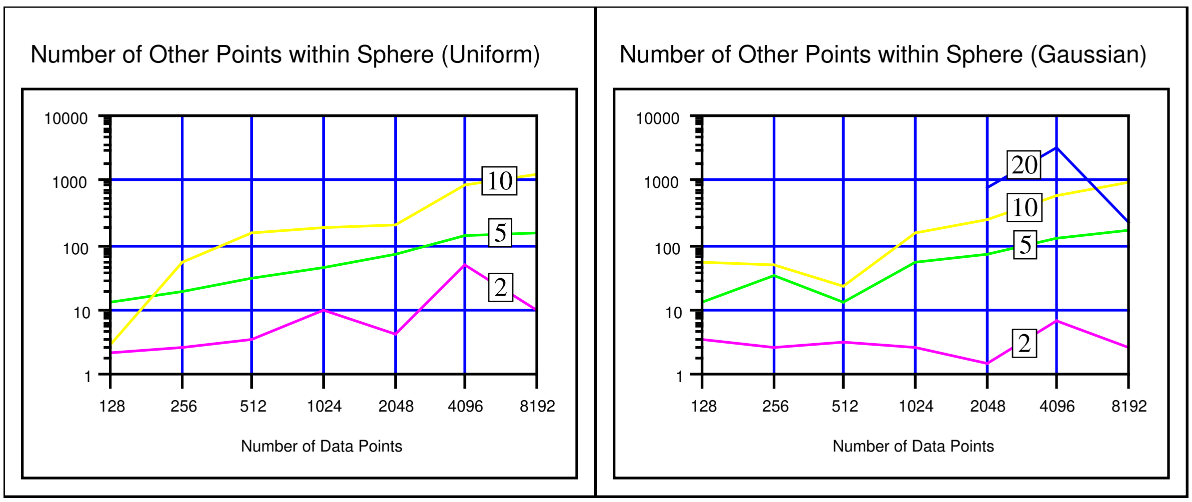
The last two tables and graph are perhaps the most important ones of all. They give us a measure of how hard it would be to turn an enclosing simplex into a Delaunay simplex. For each enclosing simplex, the sphere associated with its vertices was calculated. The average number of data points within these spheres are tabulated. Since these tables only include data for those points where an enclosing simplex was found, the same care must be taken in interpreting these tables as was done with the last two tables.

Table 13: Number of Other Points within HyperSphere (Uniform)

Dim	128 pts	256 pts	512 pts	1024 pts	2048 pts	4096 pts	8192 pts
2	2.28_{92}	2.77_{96}	3.59_{100}	10.47_{100}	4.23_{100}	51.00_{100}	10.22_{100}
5	13.48_{42}	19.03_{59}	31.66_{68}	43.78_{79}	74.81_{83}	139.35_{88}	157.40_{91}
10	3.00_{1}	53.50_{2}	158.33_{3}	184.75_{8}	204.82_{11}	821.40_{20}	1228.69_{35}
20	n/a	n/a	n/a	n/a	n/a	n/a	n/a
40	n/a	n/a	n/a	n/a	n/a	n/a	n/a

Table 14: Number of Other Points within HyperSphere (Gaussian)

Dim	128 pts	256 pts	512 pts	1024 pts	2048 pts	4096 pts	8192 pts
2	3.52 $_{95}$	2.69 $_{95}$	3.13 $_{96}$	2.69 $_{97}$	1.52 $_{98}$	7.09 $_{98}$	2.81 $_{99}$
5	12.90 $_{49}$	33.96 $_{62}$	13.68 $_{71}$	57.28 $_{78}$	76.92 $_{87}$	132.13 $_{92}$	178.52 $_{96}$
10	57.20 $_{5}$	49.50 $_{10}$	24.33 $_{18}$	163.44 $_{27}$	239.74 $_{43}$	594.35 $_{57}$	905.05 $_{61}$
20	n/a	n/a	n/a	n/a	749.00 $_{1}$	3369.00 $_{1}$	225.00 $_{2}$
40	n/a	n/a	n/a	n/a	n/a	n/a	n/a



We had hoped that this algorithm would produce simplices that would be easy to convert to Delaunay simplices. These last statistics force us to the conclusion that this is not true. It was shown in Chapter 2, Why the Problem is Difficult, that the volume of a sphere relative to the volume of an enclosed simplex is super-exponential. This directly relates to the algorithm because the circumsphere from each simplex encloses more and more volume in higher dimensions. We are doomed that as dimensionality increases, a simplex's sphere will enclose an ever higher percentage of the entire data set. This makes converting initial simplices to Delaunay simplices impractical (not to mention that the conversion process may not even directly yield a Delaunay simplex that still encloses the query point!). In some sense, we are no better off after the algorithm finishes than before it was started.

3.3 A Local Delaunay Simplex Finder

The previous algorithm demonstrated that converting a non-Delaunay simplex enclosing a query point into a Delaunay simplex enclosing the same query point is impractical. While finding a non-Delaunay simplex enclosing a query point is efficient, the conversion process is not. Clearly an algorithm is needed that starts out with a Delaunay simplex, and in its “walk” towards the query point only generates Delaunay simplices. Since the test for whether or not a simplex is a Delaunay simplex is local (see figure below) we still expected to fetch only a small subset of the entire data set for our calculations. Once again, we were wrong.

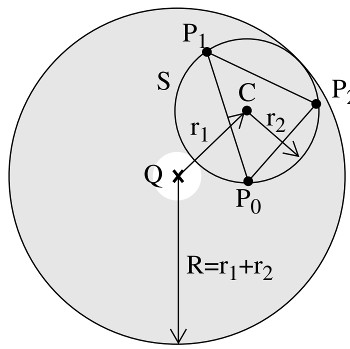


Figure 9: The simplex $P_0P_1P_2$ can be guaranteed to be a Delaunay simplex of the entire data set (a global Delaunay) in a straight forward manner. Calculate the center C and radius r_2 of the circumsphere S. Calculate the distance r_1 from Q to C. Since the M closest points to Q have been fetched into *point_list* in distance sorted order, we check those points whose distance to Q is between MAX(r_1-r_2, 0) and R to see if any lie within S. If no points do, the $P_0P_1P_2$ is a global Delaunay simplex. If the last point in *point_list* is closer than R from Q, then more points must be fetched to guarantee the global Delaunay status of $P_0P_1P_2$.

An efficient algorithm that only operates with Delaunay simplices is not immediately obvious. This is one of the main reasons we originally started with algorithms dealing with general simplices. We will now show how the gift-wrapping method [PREP85] used to solve convex hulls in general dimensions can also find Delaunay simplices. This was developed for 2D Delaunay triangulations in [EDEL87], and was extended to arbitrary dimensions in [OMOH89,1].

How Convex Hull Relates to Delaunay

By projecting a set of D-dimensional points up onto a D+1-dimensional paraboloid, facets of the convex hull of the D+1-dimensional paraboloid exactly correspond to D-dimensional Delaunay simplices. Suppose the paraboloid’s vertex is at the query point Q. For each data point, use the distance to Q squared as the value for the new D+1’st dimension.

$$P_{D+1} = (P_1 - Q_1)^2 + (P_2 - Q_2)^2 + \ldots + (P_D - Q_D)^2$$

Each facet of the convex hull defines a plane that cuts the paraboloid into two sections: the main "body" of the paraboloid, and the "cap" that was sliced off by the plane. The intersection of the plane and the paraboloid form an ellipsoid (the outline of the cap). When this ellipsoid is projected down onto D-dimensional space it forms a sphere[1]. Since there can be no points on the "cap" of the paraboloid (by nature of the convex hull), there can be no points within the sphere. This is exactly the condition needed for the D+1 points to form a Delaunay simplex.

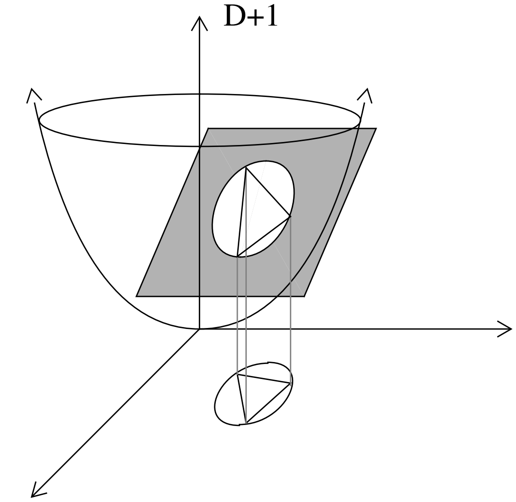


Figure 10: How Convex Hull relates to Delaunay triangulation

The Local Delaunay Simplex Finder:

The tricks have basically been given away by this point. All that remains is to work out the details. We assume that the reader is familiar with the gift-wrapping algorithm. [PREP85]

Fetch the M nearest points to the query point Q into *point_list* using GetClosest(). A discussion of how to choose a value for M is at the end of this section. Notice that in fetching these M points, GetClosest() orders the points in order of increasing distance. Rather than executing

1. This can be quickly verified by solving for the intersection of the paraboloid and the plane. A simple example is given below. Given

$$z = x^2 + y^2$$
$$z = ax + by + c$$

Solving for the intersection by eliminating z produces

$$x^2 + y^2 = ax + by + c$$
$$\left(x - \frac{a}{2}\right)^2 + \left(y - \frac{b}{2}\right)^2 = c + \frac{a^2}{4} + \frac{b^2}{4}$$

square roots to find the actual Euclidean distance to Q from a point, GetClosest() just uses the distance to Q squared. This has the effect of placing all of the points on a paraboloid whose vertex is Q -- exactly what is needed for gift-wrapping!

Take the first point (P_0) from *point_list*. Use this point to start the gift-wrapping process to find an initial convex hull facet on the D+1-dimensional paraboloid. [Notice that since every data point is on the convex hull, we could have chosen any point to start the gift-wrapping process from.] Find a plane that passes through P_0 and has all the D+1-dimensional data points on one side of it. Since we chose P_0 to be the first point from *point_list* (the lowest D+1-dimensional value), this plane is simply the one perpendicular to the D+1-dimensional axis.

To find an initial convex hull facet, we need to gift-wrap the plane D times, picking up a new data point at each iteration. The basic gift-wrapping process does not favor any particular direction when finding an initial facet (as it is not important which facet is found first). For our application, it is <u>very</u> important which facet is found first. We need to direct the gift-wrapping so that at each stage the probability that the first Delaunay simplex we find is the one that encloses Q is maximized.

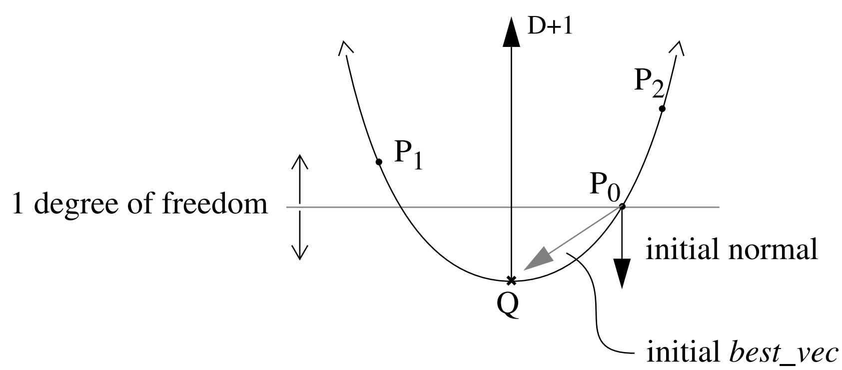


Figure 11: How to direct the gift-wrapping process

At the k-th gift-wrapping iteration to find an initial convex hull facet, there are exactly k points that already lie on the next gift-wrapping plane. For k=1, the single point can only constrain the next gift-wrapping plane to pass through it. For k>1, the additional points are used to force the next gift-wrapping plane to be coplanar with a set of edges from the partially built up simplex. Nevertheless, we are left with D+1-k degrees of freedom to choose. A heuristic is used to direct the gift-wrapping to swing towards the query point. We start by constructing the vector from the center of mass of the partially built up simplex to the query point. This vector is called *best_vec* because it represents the direction with the highest likelihood of forming a Delaunay simplex enclosing Q. Specifically, we use one degree of freedom to turn the normal vector of the current gift-wrapping plane towards *best_vec* (i.e. make sure that the sign of the dot product of the two vectors be positive). The remaining D-k constraints are formed using the relative magnitudes of the components of *best_vec*. If component *i* of *best_vec* has the largest absolute value and component *j* has the second largest absolute value, then we constrain

$$n_i = \frac{best\_vec_i}{best\_vec_j} \cdot n_j$$

For example, if *best_vec*=[9, -2, 5, -13, -3] and D-k=3, we constrain $n_4 = -\frac{13}{9} \cdot n_1$, $n_1 = \frac{9}{5} \cdot n_3$, and $n_3 = -\frac{5}{3} \cdot n_5$.

Main Loop:

Given a convex hull facet of D+1 points, project these points down to D dimensional space and use BarycentricCoords() to see if Q lies inside the Delaunay simplex defined by them. If Q is inside (by testing for all coordinates in [0, 1]) then we are done, otherwise we need to gift-wrap on the D+1 dimensional paraboloid about the sub-facet for which Q has the most negative barycentric coordinates (the corresponding antipodal point of the facet will be discarded). Go back to the **Main Loop**.

There is one problem in using the convex hull to solve for Delaunay simplices. Gift-wrapping solves for the entire convex hull of the paraboloid. The gift-wrapping plane can swing over the top of the paraboloid yielding simplices that are not Delaunay. This can only occur when an attempt is made to gift-wrap about a sub-facet of the D-dimensional convex hull of the points in *point_list* (i.e. the upper rim of the partial paraboloid). We require a method to detect when this has occurred, so that either more points can be fetched into *point_list* or a determination can be made that Q is outside the global convex hull. Fortunately, this test is fairly trivial. If a gift-wrapping plane partitions space such that the point (0, 0, ..., 0, +∞) is not on the same side as all the data points, then the plane has wrapped over the top of the paraboloid. This test is simply to check that the signs returned by TestPointVsPlane() for (0, 0, ..., 0, +∞) and for the center of mass of the paraboloid are the same.

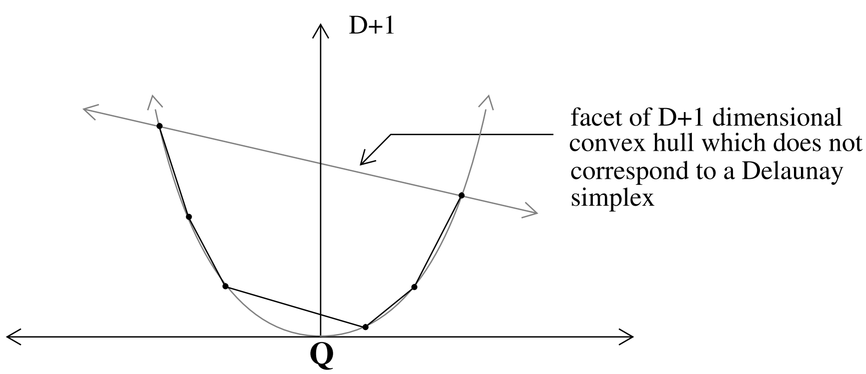


Figure 12: Example of Convex Hull facet which is not a Delaunay simplex

Analysis:

We initially had high hopes for this algorithm. The first two algorithms operated on general simplices. This algorithm operates on only Delaunay simplices. Surely the number of Delaunay simplices in the entire data set must be <u>much</u> less than the number of general simplices in the entire data set. This should greatly reduce the number of "walks" (or "wraps") required to eventually enclose the query point.

Unfortunately, as dimensionality increases, guaranteeing that a simplex is Delaunay requires looking at higher and higher percentages of the data set. As was mentioned in Section 2.2, the ratio of the volume of a sphere relative to the volume of an enclosed simplex is super-exponential. The circumsphere from each Delaunay simplex encloses more and more volume in higher dimensions. We are required to fetch a higher percentage of points from the data set, so that those points that could possibly lie within a simplex's circumsphere are contained in *point_list*.

Statistics were gathered by running this algorithm over 100 randomly chosen query points. It was shown in Section 3.2 that as dimensionality increases, it is less and less likely for a query point derived from the same random distribution as the data set to actually lie inside the convex hull of the data set. For this algorithm, the query point was derived using the data set in the following manner: chose a single random point *from the data set* and use GetClosest() to fetch the D+2 nearest neighbors to this point (note that this effectively returns the D+1 closest points and the point itself). The center of mass of these D+2 points will be used as a query point[1]. This construction guarantees that each query point will lie within the convex hull of the data set and it also produces query points that will tend to have a distribution similar to that of the data set.

Below are statistics measuring the average number of points that need to be fetched to guarantee that by the time an enclosing Delaunay simplex is found, every simplex in the "walk" towards Q was a Delaunay simplex. These statistics were gathered by keeping track of the largest r_1+r_2 (see Figure 9) encountered in the Delaunay "walk" towards each Q (with *point_list* containing all of the data points). It is then straight forward to count the number of data points within r_1+r_2 from Q. The first two tables show the average number of points needed to guarantee that every gift wrap produced Delaunay simplices. The next two tables and graph show this same data, except expressed as a percentage of the size of the entire data set.

1. The choice of using D+2 points is admittedly ad hoc. Using less than D+1 points would not allow the data space to be sufficiently explored. Using more than D+2 points would cause the constructed query point to migrate towards the center of mass of the data set.

Table 15: Average Number of Points Needed to Guarantee Delaunay (Uniform)

Dim	128	256	512	1024	2048	4096	8192
2	4.0	3.8	4.8	3.8	4.3	4.0	3.6
5	15.8	26.6	15.6	20.9	25.8	65.0	23.1
10	38.4	52.9	65.8	67.6	82.7	92.2	183.2
20	78.5	94.8	119.6	133.2	206.7	224.6	295.5
40	123.7	214.2	313.3	392.0	503.4	513.7	592.7

Table 16: Average Number of Points Needed to Guarantee Delaunay (Gaussian)

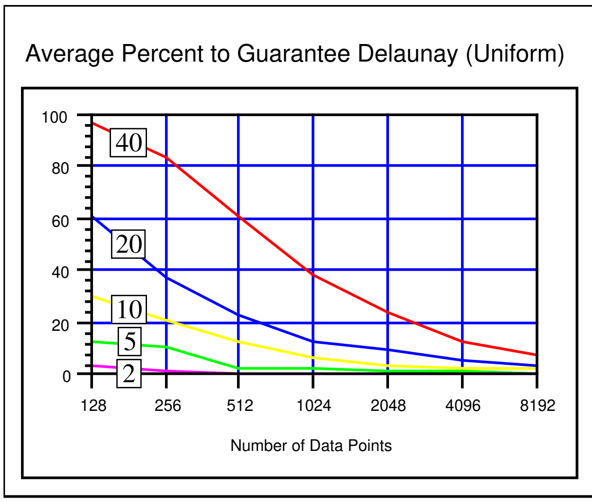
Dim	128	256	512	1024	2048	4096	8192
2	3.8	4.6	4.2	4.0	3.8	3.9	3.6
5	18.9	23.5	23.2	21.9	22.8	21.0	17.5
10	44.4	53.9	68.7	83.6	74.8	74.5	111.3
20	93.4	154.5	170.5	206.2	247.6	275.1	196.9
40	126.6	235.3	327.5	509.4	663.0	879.2	760.0

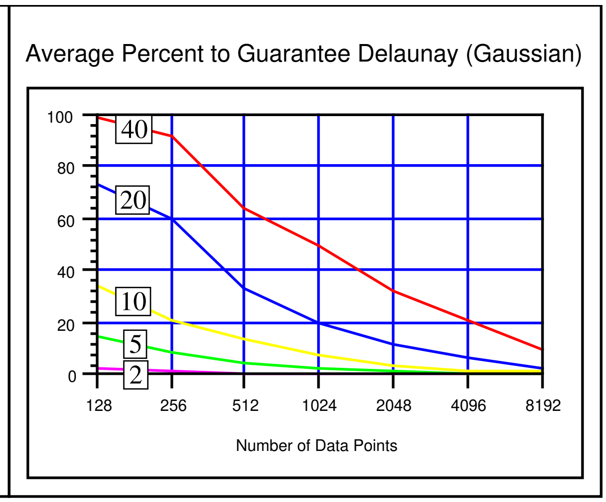
Table 17: Average Percent of Data Set Needed to Guarantee Delaunay (Uniform)

Dim	128	256	512	1024	2048	4096	8192
2	3.11	1.49	0.94	0.37	0.21	0.09	0.04
5	12.31	10.38	3.05	2.04	1.26	1.59	0.28
10	29.98	20.67	12.85	6.60	4.04	2.25	2.24
20	61.34	37.01	23.36	13.00	10.09	5.48	3.61
40	96.62	83.68	61.20	38.28	24.58	12.54	7.23

Table 18: Average Percent of Data Set Needed to Guarantee Delaunay (Gaussian)

Dim	128	256	512	1024	2048	4096	8192
2	2.99	1.78	0.82	0.39	0.19	0.09	0.04
5	14.76	9.19	4.53	2.13	1.11	0.51	0.21
10	34.65	21.07	13.43	8.16	3.65	1.82	1.36
20	72.96	60.37	33.29	20.13	12.09	6.72	2.40
40	98.87	91.92	63.96	49.74	32.37	21.46	9.28





The following two tables show the maximum observed number of points needed to guarantee that every gift wrap produced Delaunay simplices for our 100 random query points. The next two tables and graph show this same data, except expressed as a percentage of the size of the entire data set.

Table 19: Maximum Observed Number of Points Needed to Guarantee Delaunay (Uniform)

Dim	128	256	512	1024	2048	4096	8192
2	14	17	36	12	22	13	17
5	74	115	129	222	677	4087	309
10	115	250	424	405	913	1199	3705
20	128	255	473	762	2034	2581	5864
40	128	256	500	1024	2046	2696	5627

Table 20: Maximum Observed Number of Points Needed to Guarantee Delaunay (Gaussian)

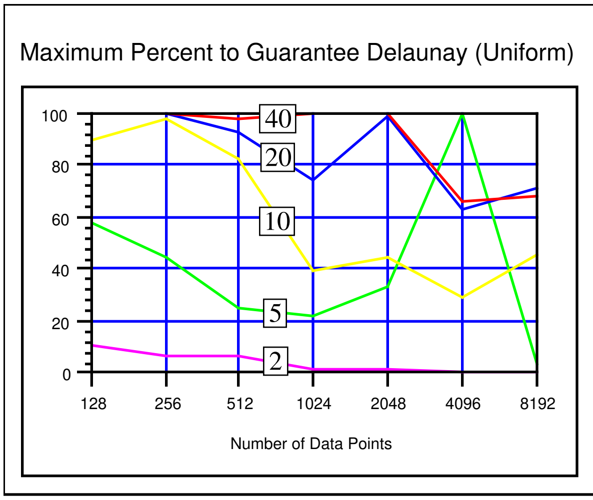
Dim	128	256	512	1024	2048	4096	8192
2	10	30	28	15	13	14	10
5	113	141	250	249	254	135	97
10	116	226	304	857	423	528	983
20	128	254	369	716	1228	1243	1103
40	128	256	510	968	1899	3711	3240

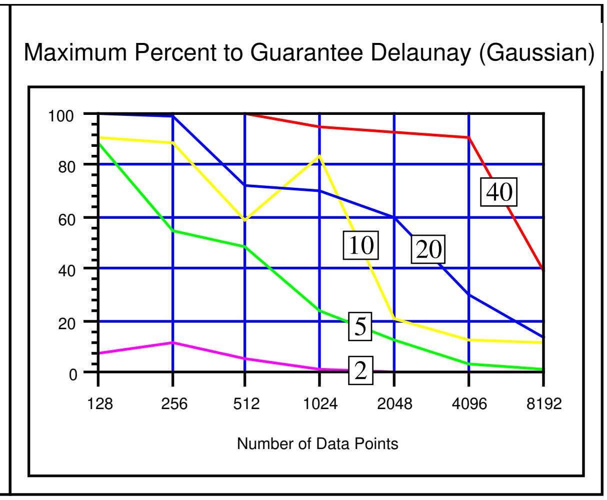
Table 21: Maximum Observed Percent of Points Needed to Guarantee Delaunay (Uniform)

Dim	128	256	512	1024	2048	4096	8192
2	10.94	6.64	7.03	1.17	1.07	0.32	0.21
5	57.81	44.92	25.20	21.68	33.06	99.78	3.77
10	89.84	97.66	82.81	39.55	44.58	29.27	45.23
20	100.00	99.61	92.38	74.41	99.32	63.01	71.58
40	100.00	100.00	97.66	100.00	99.90	65.82	68.69

Table 22: Maximum Observed Percent of Points Needed to Guarantee Delaunay (Gaussian)

Dim	128	256	512	1024	2048	4096	8192
2	7.81	11.72	5.47	1.46	0.63	0.34	0.12
5	88.28	55.08	48.83	24.32	12.40	3.30	1.18
10	90.62	88.28	59.38	83.69	20.65	12.89	12.00
20	100.00	99.22	72.07	69.92	59.96	30.35	13.46
40	100.00	100.00	99.61	94.53	92.72	90.60	39.55





We are forced to the conclusion that the "local" nature of a Delaunay triangulation does not lend itself to exploitation in higher dimensions. Global information must be used at every step to guarantee the Delaunay status of simplices. This is why we have waited so long in describing a method for choosing M. It is evident that in high dimensions M must be equal to the number of points in the entire data set (except for extremely large data sets). Even though the first two algorithms worked with general simplices, there still is the conversion process to Delaunay (which we never developed) which must use global information.

The current algorithm can be easily modified so that the paraboloid is computed just once at the center of mass of the data set. Instead of fetching M nearest neighbors to Q, just find the closest point P to Q. Use P as the initial point to gift-wrap about to find an initial convex hull facet. However, instead of starting with a plane perpendicular to the D+1'st dimension, use a plane tangent to the paraboloid at P (how to construct this plane is discussed in detail in the next section).

The remainder of statistics in this section were gathered by fetching all the data points into *point_list* for every query point (setting M=N) rather than changing the algorithm as described in the preceding paragraph. The results will be exactly the same except for a slight discrepancy in the execution time. Because we fetched the data points outside the timing loop, there is no cost for finding the closest point P to Q (it's just at the head of *point_list*). The changes described in the preceding paragraph require finding the closest point P to Q. Using a brute force linear search through the data set would take O(N·D) work. This is the same as one gift-wrap step.

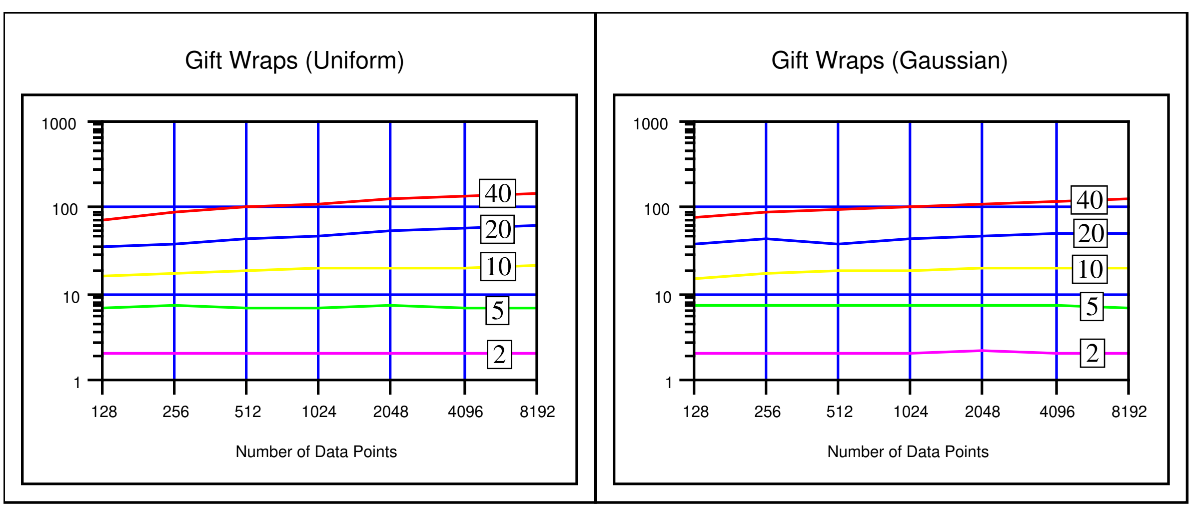
The following two tables and graphs show the average number of gift wraps used as a function of dimensionality and number of data points. Note that the minimum possible number of gift wraps is equal to D (this many is required to find and initial convex hull facet). Every gift wrap above D finds a new Delaunay simplex closer and closer to a query point. Each gift wrap requires O(N·D) work.

Table 23: Number of Gift Wraps (Uniform)

Dim	128	256	512	1024	2048	4096	8192
2	2.1	2.1	2.1	2.1	2.2	2.1	2.1
5	7.0	7.7	7.3	6.8	7.3	6.9	7.0
10	16.8	18.2	18.7	20.2	20.1	20.9	22.1
20	35.6	39.6	44.6	48.1	55.9	58.9	63.2
40	75.2	89.3	101.9	113.0	128.7	140.0	149.9

Table 24: Number of Gift Wraps (Gaussian)

Dim	128	256	512	1024	2048	4096	8192
2	2.1	2.1	2.1	2.2	2.2	2.2	2.2
5	7.6	7.5	7.4	7.5	7.4	7.7	7.2
10	15.7	17.3	18.5	18.9	19.8	20.6	20.2
20	37.3	42.8	38.8	43.9	46.3	51.1	50.2
40	76.1	88.4	93.3	102.0	108.6	116.9	123.6



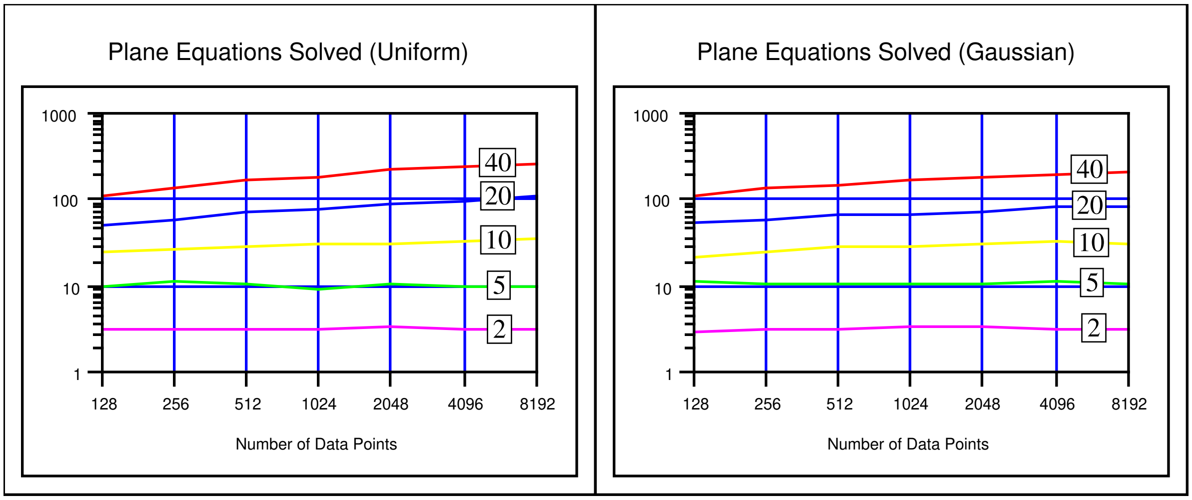
The following two tables and graphs show the average number of plane equations solved as a function of dimensionality and number of data points. Note that the minimum possible number of plane equations solved is equal to D (this many is required to find and initial convex hull facet). After an initial convex hull facet is found, two plane equations are solved for every gift wrap: one to compute a new normal, and one when using BarycentricCoords() to determine if the query point is inside the current Delaunay simplex. Each plane equation solved requires $O(D^3)$ work.

Table 25: Number of Plane Equations Solved (Uniform)

Dim	128	256	512	1024	2048	4096	8192
2	3.2	3.1	3.2	3.3	3.3	3.2	3.2
5	10.1	11.4	10.5	9.6	10.6	9.8	10.0
10	24.6	27.3	28.5	31.3	31.3	32.8	35.2
20	52.2	60.1	70.3	77.2	92.7	98.7	107.3
40	111.4	139.6	164.7	187.0	218.4	241.0	260.7

Table 26: Number of Plane Equations Solved (Gaussian)

Dim	128	256	512	1024	2048	4096	8192
2	3.1	3.2	3.3	3.3	3.4	3.3	3.3
5	11.1	11.0	10.8	10.9	10.8	11.5	10.5
10	22.5	25.6	27.9	28.8	30.6	32.2	31.5
20	55.5	58.6	66.6	68.9	73.6	83.1	81.5
40	113.3	137.9	147.6	165.1	178.2	194.9	208.2



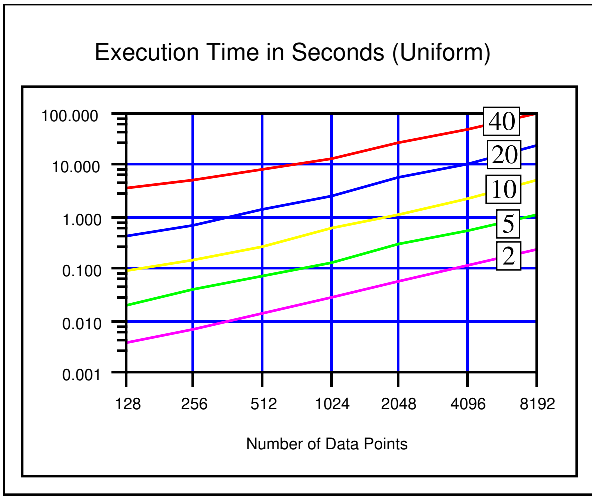
The following two tables and graphs show the average execution time in seconds as a function of dimensionality and number of data points.

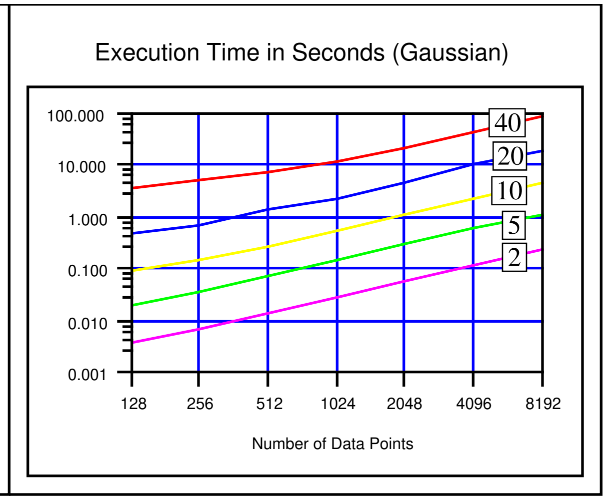
Table 27: Execution Time in Seconds (Uniform)

Dim	128	256	512	1024	2048	4096	8192
2	0.004	0.007	0.014	0.029	0.062	0.118	0.239
5	0.020	0.040	0.070	0.135	0.294	0.549	1.113
10	0.098	0.152	0.282	0.624	1.169	2.392	4.998
20	0.455	0.741	1.437	2.586	5.548	11.222	23.819
40	3.633	5.427	8.674	13.961	26.319	51.038	104.244

Table 28: Execution Time in Seconds (Gaussian)

Dim	128	256	512	1024	2048	4096	8192
2	0.004	0.007	0.014	0.029	0.061	0.121	0.242
5	0.021	0.039	0.071	0.148	0.295	0.615	1.146
10	0.091	0.144	0.277	0.581	1.141	2.353	4.555
20	0.477	0.725	1.361	2.342	4.575	10.083	18.989
40	3.674	5.334	7.837	12.435	22.226	45.258	85.831


Execution Time in Seconds (Uniform)
100.000
10.000
1.000
0.100
0.010
0.001
128
256
512
1024
2048
4096
8192
Number of Data Points
40
20
10
5
2


Execution Time in Seconds (Gaussian)
100.000
10.000
1.000
0.100
0.010
0.001
128
256
512
1024
2048
4096
8192
Number of Data Points
40
20
10
5
2

3.4 A Global Delaunay Simplex Finder

In the **Introduction** it was stated that this paper would investigate local, on-the-fly triangulations. However, the previous algorithm demonstrated the necessity of using global information at every step when computing Delaunay simplices in high dimensions. In light of this, we will show how a Linear Programming package can be used to solve for a Delaunay simplex enclosing a query point Q.

The advantages from using a Linear Program solver are many. Our work is simplified enormously in that we only need to specify an appropriate objective function and set of constraints. The LP solver frees us from the responsibility of finding a solution in the most efficient way. The authors also have much more faith in persons better versed in issues of mathematical software than themselves. These issues include efficiency, numerical stability, bug-free code and (perhaps) provably correct algorithms.

We used LSSOL version 1.02 from Stanford University as our LP solver. Although LSSOL is designed to solve linear least-squares and convex quadratic programming, it nicely handles linear programming as a special case of quadratic programming. LSSOL assumes that all matrices are dense (i.e. every matrix entry is assumed to be non-zero), which for our purposes will turn out to be ideal.

The reader will note that the formulation of this algorithm is very similar to that of the previous algorithm. The key idea is to place all the D-dimensional data points on a D+1-dimensional paraboloid, just as before (this need only be done once at the center of mass of the data set, as opposed to once per query point). Our objective function will be the equation of a D+1-dimensional plane. By constraining every data point to lie on one side of this plane, the plane is restricted to not pass through the convex hull of the paraboloid. The objective function is constructed so that minimizing its value clamps the plane to a facet of the convex hull -- specifically the facet that corresponds to the Delaunay simplex enclosing Q.

At this point, a short description of LSSOL is appropriate. For Linear Programming, LSSOL solves the following class of problems[1]:

$$\text{minimize:} \quad c_{1 \times n} \cdot x_{n \times 1}$$

$$\text{subject to: } l_{(n+m) \times 1} \le \left\{ \begin{matrix} x_{n \times 1} \\ C_{m \times n} \cdot x_{n \times 1} \end{matrix} \right\} \le u_{(n+m) \times 1}$$

LSSOL solves for the unrestricted[2] $x_{n \text{ x } 1}$. We must fill in $c_{1 \text{ x } n}$ (the coefficients of the objective function), $C_{m \text{ x } n}$ (the coefficients of the general constraints), $l_{(n+m) \text{ x } 1}$ (the lower

1. We adopt the C-like numbering convention where the vector $x_{n\text{x}1}$ is made up of elements x_0 through x_{n-1} and the array $C_{m\text{x}n}$ is made up of elements $C_{0,0}$ through $C_{m-1,n-1}$.
2. By "unrestricted", we mean that each x_i can be either positive or negative. Most LP solvers require every x_i to be strictly non-negative.

bounds) and $u_{(n+m) \times 1}$ (the upper bounds). Notice that the lower and upper bounds apply both to the variables LSSOL is solving for and to the constraints. As far as linear programming goes, LSSOL cannot solve a larger class of problems than another linear programming method (e.g. the simplex method). However, it does allow a more compact representation of the same problem. LSSOL's unrestricted variables can be replaced by the difference of two strictly positive variables (i.e. $x_i \to x_{i1} - x_{i2}$). Likewise, specifying two constraints for each LSSOL constraint mimics the effect of lower and upper bounds.

With that brief summary, we now proceed to show how LSSOL can find the enclosing Delaunay simplex. Deciding how to specify the plane equation is certainly the first step. A sample plane equation in three dimensions is:

$$Ax + By + Cz + D = 0 \qquad \text{or} \qquad Ax + By + D = -Cz$$

dividing by $-C$ yields

$$A'x + B'y + C' = z$$

We will use this same format but in a more general setting. The *A'*, *B'* and *C'* will be replaced by x_0 through x_D (the coefficients of the plane equation LSSOL will solve for). The *x*, *y* and *z* will be replaced by P_0 through P_D. The general plane equation becomes:

$$P_0 x_0 + P_1 x_1 + \ldots + P_{D-1} x_{D-1} + x_D = P_D$$

We want to maximize the P_D value of the query point Q projected onto this plane.

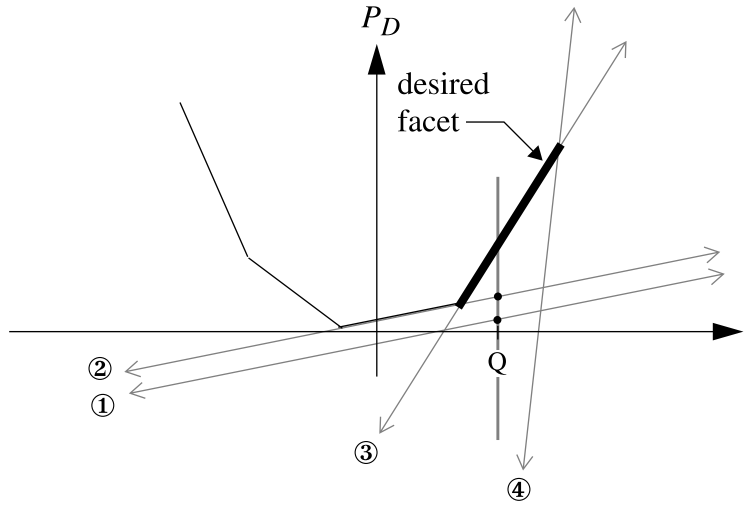


Figure 13: Assuming that a plane can be constrained not to pass through the convex hull of the paraboloid (which is centered at the center of mass of the data set), any plane which maximizes the P_D value must be clamped to the convex hull. This is demonstrated by plane ① having a lower P_D value than plane ②. We can further increase the P_D value by rotating plane ② until it matches plane ③. Rotating plane ③ in any direction causes the P_D value to decrease. Clearly plane ③ maximizes the P_D value at Q. Likewise plane ③ lies on the facet of the convex hull which corresponds to the Delaunay simplex enclosing Q.

Since LSSOL minimizes its objective function, we must minimize the negative of the P_D value of the plane at Q. This objective function becomes:

$$-Q_0x_0 - Q_1x_1 - \ldots - Q_{D-1}x_{D-1} - x_D$$

LSSOL expects the user to provide lower and upper bounds for each of the variables to be solved for (the x_i's). We place no restrictions on the values these variables can have as they are simply coefficients of a plane equation. Therefore, l_0 through l_D are set to $-\infty$ and u_0 through u_D are set to $+\infty$[1].

The constraints are formed by requiring every D+1 dimensional data point lie on the same side of the plane that LSSOL is solving for. If we were using TestPointVsPlane(), we would accomplish this by formulating an inequality like the one below for every point in the data set:

$$Ax + By + Cz + D \geq 0 \qquad \text{after division by -C,} \qquad A'x + B'y + C' \leq z$$

1. Really -10^15 and +10^15.

We will express exactly this, but formatted for Linear Programming:

$$P_0 x_0 + P_1 x_1 + \ldots + P_{D-1} x_{D-1} + x_D \le P_D$$

Those data points that satisfy the equation with an equality are tight constraints and together they define the plane. All the other data point are loose constraints and have no effect in determining the coefficients of the plane.

If the query point lies outside the convex hull of the data set, LSSOL will indicate that the solution is unbounded. This reflects that fact that a plane can be constructed which is parallel to the P_D axis.

A small sample problem is given for clarity: Given a set of six 2D points

$$T_0 = \begin{bmatrix} -1 \\ 2 \end{bmatrix} \quad T_1 = \begin{bmatrix} -1 \\ -2 \end{bmatrix} \quad T_2 = \begin{bmatrix} 1 \\ -3 \end{bmatrix} \quad T_3 = \begin{bmatrix} 2 \\ 1 \end{bmatrix} \quad T_4 = \begin{bmatrix} 2 \\ 0 \end{bmatrix} \quad T_5 = \begin{bmatrix} 3 \\ -1 \end{bmatrix}$$

we wish to find the Delaunay simplex enclosing the query point

$$Q = \begin{bmatrix} 1.5 \\ -1 \end{bmatrix}$$

The objective function is

$$\text{minimize:} \quad -(1.5)\, x_0 - (-1)\, x_1 - x_2$$

so the objective function matrix becomes

$$c_{1 \times n} = \begin{bmatrix} -1.5 & 1 & -1 \end{bmatrix}$$

We center the paraboloid at the center of mass of the data points

$$Tcom = \begin{bmatrix} 1 \\ -0.5 \end{bmatrix}$$

The lower and upper bounds and the matrix of constraints are presented below:

$$l_{(n+m) \times 1} = \begin{bmatrix} -\infty \\ -\infty \\ -\infty \\ -\infty \\ -\infty \\ -\infty \\ -\infty \\ -\infty \\ -\infty \end{bmatrix} \qquad C_{m \times n} = \begin{bmatrix} -1 & 2 & 1 \\ -1 & -2 & 1 \\ 1 & -3 & 1 \\ 2 & 1 & 1 \\ 2 & 0 & 1 \\ 3 & -1 & 1 \end{bmatrix} \qquad u_{(n+m) \times 1} = \begin{bmatrix} \infty \\ \infty \\ \infty \\ 10.25 \\ 6.25 \\ 6.25 \\ 3.25 \\ 1.25 \\ 4.25 \end{bmatrix}$$

Given the $c_{1 \times n}$, $l_{(n+m) \times 1}$, $C_{m \times n}$, and $u_{(n+m) \times 1}$, LSSOL returns (-0.714, -1.426, 2.679) as the equation of the plane that minimizes our objective function, -3.034 as the value of the objective function at the query point, and a table indicating that constraints 4, 5 and 7 were tight. Constraints 4, 5 and 7 correspond to the points

$$T_1 = \begin{bmatrix} -1 \\ -2 \end{bmatrix} \qquad T_2 = \begin{bmatrix} 1 \\ -3 \end{bmatrix} \qquad T_4 = \begin{bmatrix} 2 \\ 0 \end{bmatrix}$$

which are the vertices of the Delaunay simplex enclosing Q.

There is one further refinement that can aid LSSOL in solving our problem as efficiently as possible. We are allowed to supply an initial "guess" at what the $x_{n \text{ x } 1}$ solution vector might be (if a guess is not supplied, the x_i's are initialized with 0). By constructing the plane which is tangent to the paraboloid at the data point closest to Q, we can give LSSOL a head start in finding a solution.

Let T represent the data point which is closest to Q. The normal of the paraboloid at T is

$$n = (-2\,(T_0 - Tcom_0),\ -2\,(T_1 - Tcom_1),\ \ldots\ ,\ -2\,(T_D - Tcom_D),\ 1)$$

We can call MakePlaneFromNormal() with the normal n and the point T to generate the desired plane equation P_{guess}. All that remains is to convert P_{guess} into a format compatible with our objective function. This is accomplished in the following manner

$$x_i = -\frac{P_{guess_i}}{P_{guess_D}} \qquad 0 \leq i < \mathrm{D}$$

$$x_D = -\frac{1}{P_{guess_D}}$$

Analysis:

This algorithm is the culmination of our work. There are no remaining "shortcuts" for us to remove.

Statistics were gathered by running this algorithm over 100 randomly chosen query points. It was shown in Section 3.2 that as dimensionality increases, it is less and less likely for a query point derived from the same random distribution as the data set to actually lie inside the convex hull of the data set. For this algorithm, the query point was derived using the data set in the following manner: choose a single random point from the data set and use GetClosest() to fetch the D+2 nearest neighbors to this point (note that this effectively returns the D+1 closest points and the point itself). The center of mass of these D+2 points will be used as a query point. This construction guarantees that each query point will lie within the convex hull of the data set and it also produces query points that will tend to have a distribution similar to that of the data set.

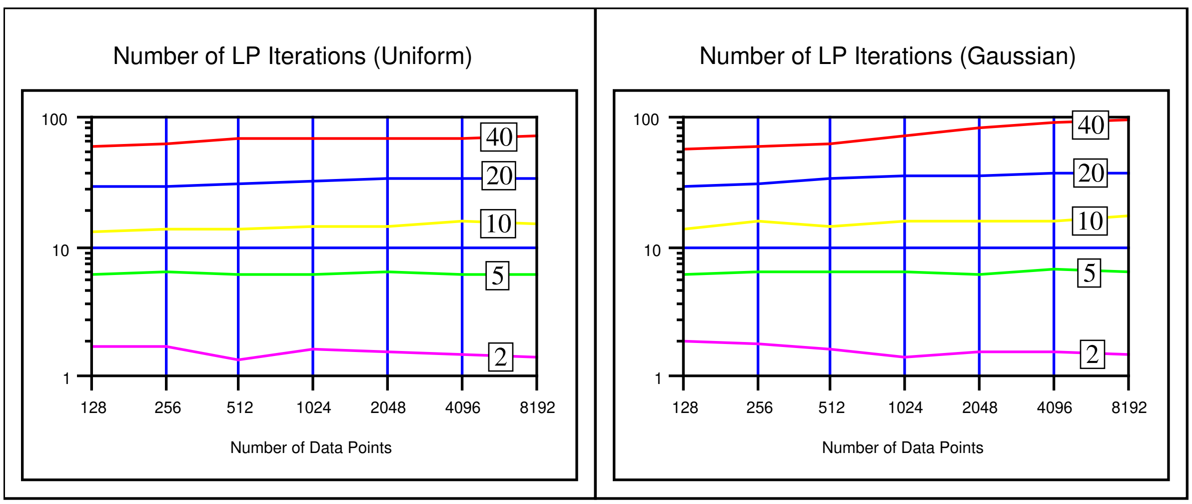
The following two tables and graph show the average number of iterations that LSSOL performed in finding a Delaunay simplex enclosing Q.

Table 29: Number of LSSOL Iterations (Uniform)

Dim	128	256	512	1024	2048	4096	8192
2	1.8	1.7	1.4	1.6	1.6	1.5	1.4
5	6.2	6.4	6.2	6.2	6.4	6.2	6.1
10	13.4	13.6	13.8	14.1	14.7	15.5	15.3
20	28.8	29.8	31.0	32.1	33.5	33.7	33.0
40	59.1	62.4	67.1	67.4	69.6	69.5	71.4

Table 30: Number of LSSOL Iterations (Gaussian)

Dim	128	256	512	1024	2048	4096	8192
2	1.9	1.8	1.6	1.4	1.6	1.6	1.5
5	6.1	6.3	6.4	6.4	6.2	6.6	6.5
10	13.6	15.6	14.6	15.8	16.1	16.1	17.5
20	29.7	31.3	33.9	36.1	36.0	36.6	37.5
40	55.7	60.5	62.9	70.4	83.0	89.3	95.8



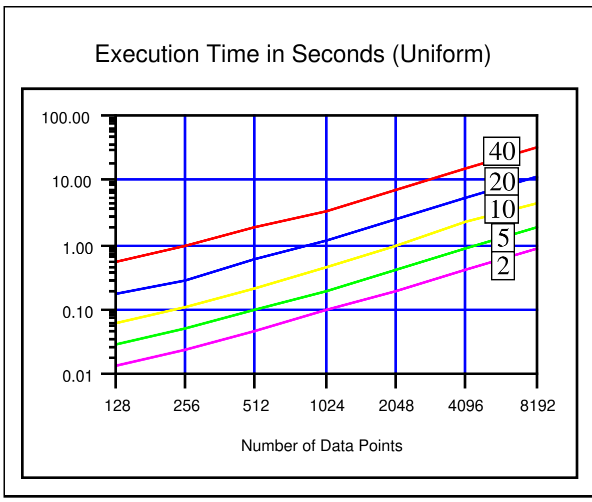
The following two tables and graphs show the average execution time of LSSOL in seconds as a function of dimensionality and number of data points.

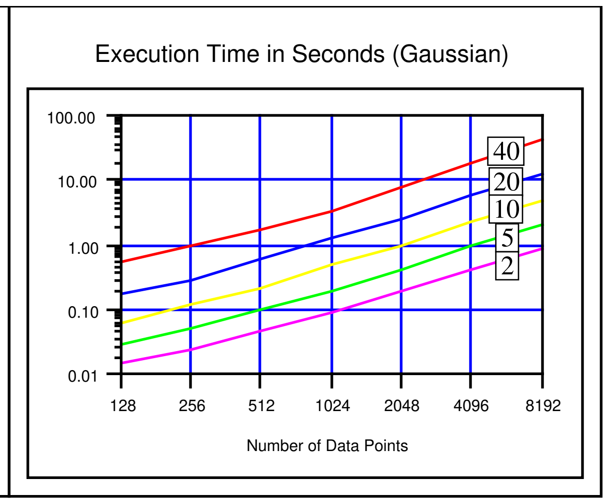
Table 31: Execution Time in Seconds (Uniform)

Dim	128	256	512	1024	2048	4096	8192
2	0.01	0.03	0.05	0.10	0.20	0.42	0.85
5	0.03	0.05	0.10	0.20	0.43	0.93	1.91
10	0.06	0.11	0.21	0.44	0.95	2.18	4.56
20	0.17	0.29	0.59	1.17	2.47	5.41	11.36
40	0.57	0.96	1.82	3.33	6.90	14.72	32.48

Table 32: Execution Time in Seconds (Gaussian)

Dim	128	256	512	1024	2048	4096	8192
2	0.01	0.03	0.05	0.09	0.20	0.43	0.87
5	0.03	0.05	0.10	0.20	0.42	0.98	2.00
10	0.06	0.13	0.22	0.48	1.02	2.25	5.07
20	0.18	0.30	0.63	1.29	2.62	5.81	12.72
40	0.54	0.94	1.73	3.49	8.13	18.59	42.79





4 Extrapolations

The previous sections assumed that any query point would lie within the convex hull of the data set. In a real application, this would be too large a limitation to impose on the user. An algorithm is needed that will provide the user with “guesses” based on extrapolations from her data set. This is not a trivial prospect. It can be easy to discover that a query point is outside the convex hull and still be difficult to know which simplex to extrapolate from. The figure below shows an example where just determining that a query point is outside the convex hull does not provide enough information to allow an extrapolation to proceed.

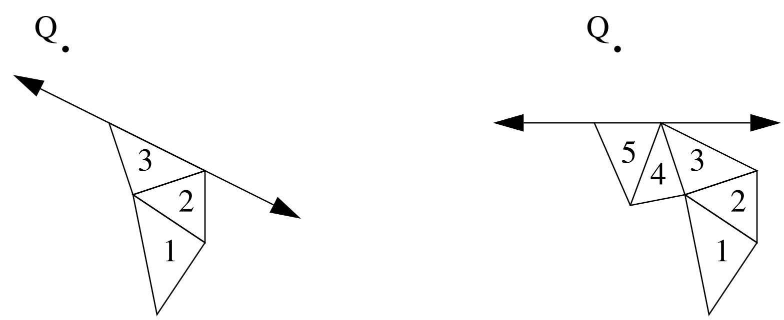


Figure 14: Given a query point Q, an algorithm initially finds simplex 1. In its walk towards Q, it visits simplices 2 and 3. At simplex 3, it discovers that Q is outside the convex hull of the data set. That does not necessarily mean that simplex 3 is the best candidate to extrapolate from. Simplex 5 appears the more natural selection.

In Chapter 1, it was declared that any triangulation scheme must be both consistent and well-behaved. An extrapolation scheme should also have these properties. It is difficult to define “well-behaved” in this context. First, a partitioning is needed that maps regions of space outside the convex hull to Delaunay simplices having a facet on the convex hull. Second, a method of extrapolation is needed which behaves in a manner similar to our interpolations.

Unfortunately the Voronoi diagram, the dual of the Delaunay triangulation, cannot help us because it partitions space along the perpendicular bisectors of the convex hull facets. This maps regions of space to a specific vertex. We are, however, interested in mapping regions of space to the convex hull facets of the data set. The figure below demonstrates this for a small section of a set of points.

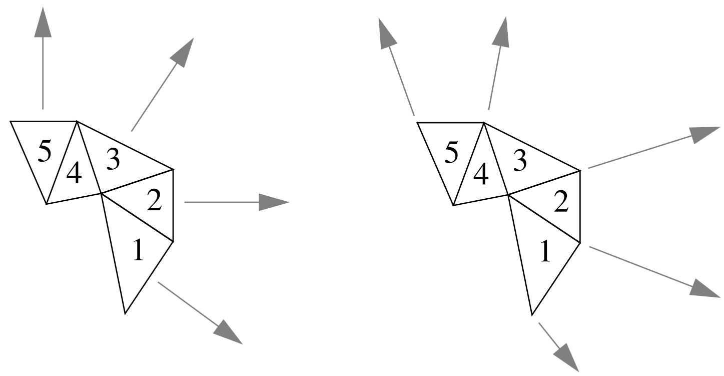


Figure 15: The Vornoi partitioning and a “well-behaved” partitioning

Our formulation of an extrapolation algorithm strives to fulfill the following constraints: consistency, continuity at region boundaries and local control[1] (it would be nice to add “reasonableness” to this list - however, it is left to the reader to decide if this was accomplished). Care was taken to continue in the spirit of Delaunay simplex interpolation: partition space based on the location of the data points and not the output values, then perform barycentric interpolations on the output values.

How to Find a Delaunay Simplex from which to Extrapolate:

The main idea is to find a Delaunay simplex on the convex hull that partitions space outside the convex hull into suitable frustums for extrapolation. The algorithm works by finding the convex hull facet whose plane has the greatest distance between itself and the query point while also partitioning space so that the data points and the query point lie on opposite sides of the plane. This construction is very similar to the one used in Section 3.4, **A Global Delaunay Simplex Finder**.

The **Main Loop** of this algorithm initializes a group of variables based on the geometry of the current candidate Delaunay simplex. **Inner Loop 1** iterates over the subfacets of the candidate Delaunay simplex to decide whether or not it could be the proper one to extrapolate from. **Inner Loop 2** then iterates over the vertices of the candidate Delaunay simplex to ensure that it is the best simplex to extrapolate from.

1. By “local control” it is meant that no properties of the global data set (like center of mass, bounding box, etc.) should be used in the algorithm. As was shown in the previous section, it may still be necessary in high dimensions to check all points to ensure that a simplex is a Delaunay simplex.

Main Loop:

Let: S be the current Delaunay simplex[1],
Let: F be a facet of S on the convex hull (other facets of S may also be on the convex hull),
Let: COG_F be the center of gravity of the vertices of F,
Let: SF_0 through SF_{D-1} be the subfacets of F,
Let: COM_{SF0} through COM_{SFD-1} be the center of masses of the vertices of the SF_i's,
Let: L be a list containing all the SF_i sorted by Euclidean distance between Q and COM_{SFi}.

Inner Loop 1:

Remove the subfacet SF_{cur} from the head of L. Gift-wrap about SF_{cur} to find the next convex hull facet F_{new}. Construct the vector $n_{bisector}$ by averaging the unit facet normals from F and F_{new}.

Use **MakePlaneFromPoints()** to construct the plane P made up of the vertices of SF_{cur} and the point formed by adding $n_{bisector}$ to any one vertex from SF_{cur}. Using **TestPointVsPlane()**, check to see if Q and COG_F are on the same side of P. If they are not, go back to the **Main Loop** replacing S with S_{new} and F with F_{new}.

If L is not empty at this point, continue on with **Inner Loop 1**, otherwise goto **Inner Loop 2**.

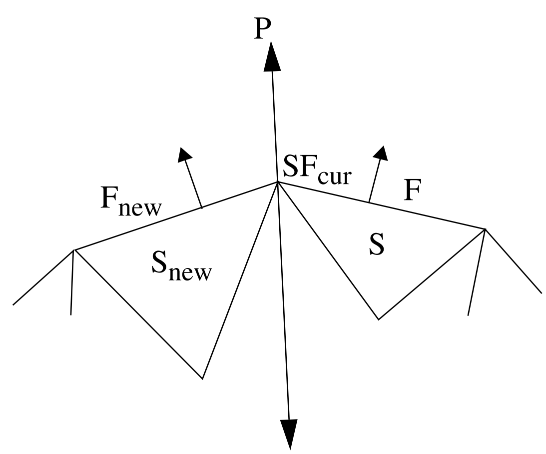


Figure 16: How the partitioning plane P is constructed

Inner Loop 2:

At this point we have a Delaunay simplex which is only likely to be the best simplex to extrapolate from. The problem we have is that the frustums determined by the

1. The first time through this loop, S will just be the last Delaunay simplex encountered when Q was determined to be outside the convex hull.

bisector planes in general will overlap. This can be shown by noting that the intersection line of two bisector planes passing through a vertex will not necessarily match all of the intersection lines coming out of that particular vertex. **Inner Loop 1**'s examination of Delaunay simplices only approximates finding the convex hull facet whose plane has the greatest distance between itself and the query point. To ensure this property, we must perform the expensive operation of examining all the convex hull facets which share a vertex with the current convex hull facet. If this were not done, consistency in extrapolation could not be guaranteed.

Continually giftwrap about each vertex of F until the giftwrap returns to F[1]. Check each facet encountered in this process to see if one has a larger distance between the query point and the plane of the facet than exists for the facet F. If this is true and the query point lies on the opposite side of this plane than the data points, then assign this facet to F and begin **Inner Loop 2** again. If no facet meets this criteria then find the simplex S attached to F. S is the proper Delaunay simplex to extrapolate from.

How to Extrapolate from a Delaunay Simplex:

The previous subsection, **How to Find a Delaunay Simplex from which to Extrapolate**, is likely to be usable over a wide range of data sets. However, actual extrapolation techniques will likely be very individualistic. With this in mind, three methods for extrapolation are presented which demonstrate some of the trade-offs that can be made.

1) The straightforward technique is to simply use barycentric coordinates from the simplex S to determine the output values at Q. This certainly produces consistent results and is computationally reasonable, but it has the disadvantage of producing severe discontinuities between adjoining regions.

2) Another approach designed to curtail the discontinuities at region boundaries requires much more computation. This algorithm uses the outer frustum formed by the D bisector planes as a basis for extrapolation. For each of the outgoing rays of the frustum, we try to devise an averaged linear function that best reflects the trends of the data values observed in the neighborhood of each convex hull vertex. An offset plane parallel to F which passes through Q is then constructed. The intersection of this plane with the rays of the frustum determine the vertices of a D-1 dimensional simplex. This simplex is interpolated across to calculate the output value for Q.

It is assumed that in addition to the Delaunay simplex S with facet F, the algorithm has at its disposal the planes P_0 through P_{D-1} (the D P-planes that form the boundaries between the outer frustums). These will need to be re-calculated if the **Inner Loop 2** from the preceding subsection assigned a new simplex to S.

Calculate the intersection point I of all the P_i planes by solving a set of linear equations. For each $v_{F,i}$, solve for the intersection line L_i of the D-1 P-planes which pass through $v_{F,i}$ (this is

---

1. It is worth noting that the number of convex hull facets touching a vertex is likely to be O(D!) (see **"Typical" Number of Simplices Attached to a Vertex in D Dimensions**).

straightforward given $v_{F,i}$ and I). For every 1D edge E_j connected to $v_{F,i}$, calculate the output value gradient for E_j by dividing the difference of the output values at the vertices of E_j by the length of E_j in the input space[1]. Assign the average output value gradient to L_i.

Construct the plane P_F which overlays F by calling **MakePlaneFromPoints()** with the vertices of F. Construct the plane P_Q which is parallel to P_F and passes through Q by calling **MakePlaneFromNormal()** with the normal from P_F and the point Q.

Let A be the minimum Euclidean distance between I and P_F (use **TestPointVsPlane()** and divide by the square root of the sum of P_F's coefficients squared). Likewise, let B be the minimum Euclidean distance between I and P_Q. The "facet" F_Q can now be constructed. If $v_{F,i}$ are the D vertices of F, then let $v_{FQ,i}$ be the corresponding vertex positions on F_Q:

$$v_{FQ,i} = I + \frac{B}{A} \cdot (v_{F,i} - I)$$

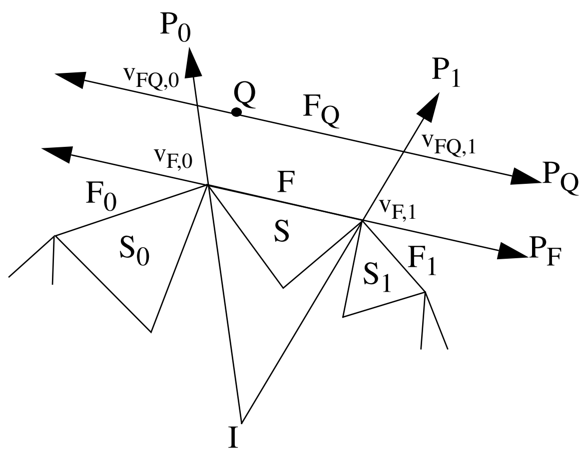


Figure 17: How to construct plane P_Q (parallel to P_F and passing through Q)

Given the output value at each $v_{F,i}$ and the output value gradient from each L_i, assigning an output value to each $v_{FQ,i}$ is trivial. The output values at Q are obtained by barycentric interpolation in F_Q. To reduce computation, project Q and the $v_{FQ,i}$ down onto the space formed by removing the dimension corresponding to the smallest absolute value in the normal of plane P_Q (this choice minimizes numerical inaccuracies).

The reader will notice that the output values returned throughout a region of space assigned to a particular Delaunay simplex do not form a plane. Instead, they are the result of bilinear interpolation along the boundaries between Delaunay simplices. Also we have not eliminated the discontinuities at region boundaries, we have only lessened them (it was shown in the preceding subsection that the frustums formed by the bisector planes overlap).

1. It is worth noting that the number of 1D edges attached to a vertex is likely to be $O(2^D)$ (see **"Typical" Number of Nearest Neighbors in D Dimensions**).

3) The third and most correct approach involves making frustums which do not overlap and therefore do not allow discontinuities. Only an outline for how this might be accomplished is given. The exact details appear to be quite involved and could be considered an interesting area of research in itself.

The D bisector planes from the preceding subsection correspond to the locus of points equidistant from the planes of two convex hull facets which share a common edge. The idea is to extend this concept by forming additional planes which truncate the frustums of facets sharing a common vertex. In the figure below, for the vertex shown, the frustums associated with the facets A and C need no truncation whereas the frustums associated with the facets B and D do. A plane needs to be constructed which passes through the convex hull vertex and partitions space so that the points which lie on this plane are equidistant from the planes corresponding to facets B and D. The result of this operation produces frustums which have a distinct resemblance to Vornoi cells.

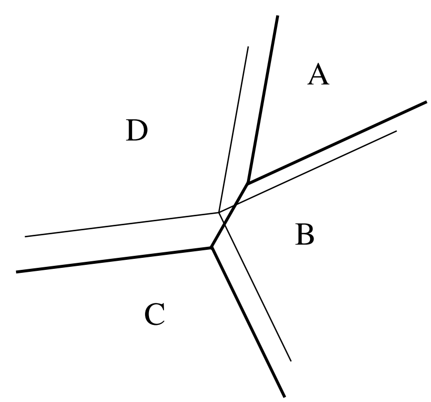


Figure 18: An example of truncated frustums

After the frustum is properly truncated by each of the facets it shares a vertex with, we still need to construct output values at the query point. Using a plane parallel to F which intersects rays of the frustum still seems like the appropriate thing to do. A complication arises in that the intersection of this plane with the truncated frustum is no longer a simplex. An interpolation scheme needs to be found which can interpolate over this region. To prevent discontinuities, this interpolation must have the property that at edges, the interpolation degenerates to only using the vertices of the edge that the query point lies in.

5 Outstanding Issues

The algorithms of the preceding sections make many assumptions about the data sets they expect to handle. The data sets generated to test the performance of these algorithms did not violate these assumptions. Real world data sets will not be so obliging. Three cases are presented to demonstrate the difficulties in dealing with real world data sets. This list is certainly not an exhaustive one; it contains only those situations recognized by two lone researchers working in an academic environment. It should be noted ahead of time that we have given only cursory thought to these problems. No solutions are forthcoming in this paper.

Input and Output Dimensions

The dimensions of data sets were assumed to be easily divisible into two categories: those dimensions which make up the input space and those dimensions which make up the output space. The abstract of this paper uses as an example the operational state of an airplane or rocket engine. One person may think of air speed as an output dimension: how fast is the plane moving given the current fuel mixture? Another may just as likely think of air speed as an input dimension: how hot is the engine given the current air speed?

Specification of input and output dimensions clearly require some guidance from the person requesting the information. However, more still needs to be done. In general, problem dimensions will not be independent. The potentially complex interactions between dimensions makes it that much more difficult when trying to decide which dimensions should be considered input and which should be considered output. This related problem is developed more fully in the next paragraph.

Low-D Surfaces in High-D Spaces

Data sets, particularly those in high dimensions, are quite unlikely to span the space they are imbedded in. It is far more likely that the data lies on a lower dimensional manifold within the problem space. Unless recognized and accounted for, it has the disturbing consequences of invalidating virtually all interpolations and extrapolations. Consider the following example:

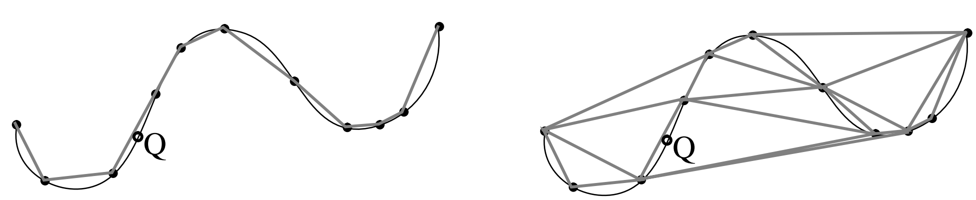


Figure 19: Example of a 1D manifold in a 2D space

In the right half of Figure 20, querying the point Q (which actually lies on the curve underlying the data samples) results in interpolations using data points that are clearly farther away than other points in the data set. These situations must be recognized and account for. If the requirement for consistency is to be maintained, local on-the-fly methods cannot be used. Only by solving for a global solution will consistency be guaranteed.

Self-intersecting Data Surfaces

It is possible for the data set to be self-intersecting. At intersection regions, it is quite reasonable for different output values to exist. The disparity between these output values is a measure of how much context was present when the original process was measured.

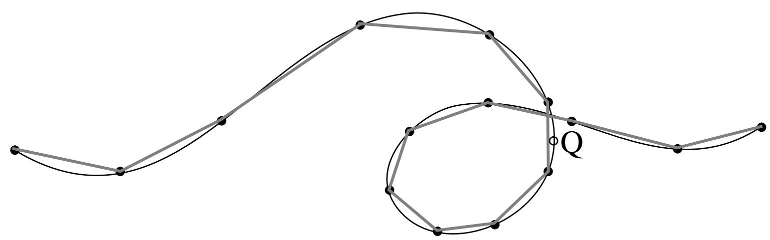


Figure 20: Example of self-intersecting 1D manifold in a 2D space

Ideally, one would like to express the data in terms of some underlying parametric functions. User's query points would be mapped to a parametric value. In situations where different parametric values are possible, the context of past queries could be used to resolve the "best" parametric value.

6 An Alternative Approximation Technique

In the early beginnings of this research, it appeared both correct and desirable that all interpolations should be consistent and continuous with respect to adjacent queries. In developing the algorithms in this paper, it became less and less clear that the requirements of consistency and continuity should be absolute. People who "just want to get work done" may find these ideals too confining. A fast, straightforward algorithm that produces "good enough" results may be preferable. One such method is the use of bumptrees [OMOH91]. This section is a brief discussion of bumptrees and how they might be used to solve the class of problems we are interested in.

"A bumptree is a new geometric data structure which is useful for efficiently learning, representing, and evaluating geometric relationships in a variety of contexts. They are a natural generalization of several hierarchical geometric data structures including oct-trees, k-d trees, balltrees and boxtrees. They are useful for many geometric learning tasks including approximating functions, constraint surfaces, classification regions, and probability densities from samples." [OMOH91]

For our purposes, a bumptree would contain all the points of a data set such that each leaf node corresponds to a single point in the data set[1]. An *influence function* is associated with the leaf nodes and internal nodes in the bumptree. The leaf node functions peak at their data points, vanish outside of a specified radius and are spherically symmetric (hence the name "bump"). For interior nodes, "the defining constraint is that each interior node's function must be everywhere larger than each of the functions associated with the leaves beneath it." [OMOH91] These interior node functions are used for branch-and-bound pruning.

Given this framework, it is now possible to use the bumptree to efficiently request all leaf functions "whose value at a specified point is within a specified factor (say 0.001) of the [leaf function] whose value is largest at that point." [OMOH91] By merely summing up the matching leaf functions, a reasonable approximation can be quickly arrived at.

For large data sets, bump trees have the distinct advantage of being able to prune away large section of the tree. It has been shown in earlier sections that this technique does not provide much help to solving for Delaunay simplices. Although the bumptrees have not been experimentally studied in the context of this investigation, it is felt that bumptrees can offer significant speed advantages over other "correct" methods. There is even the potential for the resulting interpolations to be "better" in the sense that they vary much more smoothly than linear interpolations using the vertices of a simplex.

1. Exactly how to construct the bumptree is described in [OMOH89,2]. As would be expected, there are many types of constructions that make trade offs between construction time and query time.

7 Conclusion

Four algorithms were designed to implement exact linear interpolation on high-dimensional data sets. It could be said that the development of these algorithms leaves one with a sense of futility in dealing with high dimensional data sets. The first algorithm tries to use backtracking in a small neighborhood of points to find a general simplex that could later be turned into a Delaunay simplex. The last algorithm uses a linear program which at each iteration must consider every single point in the data set. Each algorithm was forced to abandon one of the assumptions from the previous algorithm. In the end, we were forced to discard all that our intuition had told us.

A second significant realization that has come from this research is that high dimensional data sets are likely to be lower dimensional manifolds embedded in the higher dimensional spaces. Finding a method to reduce the apparent dimensionality of these types of problems appears to be an intractably hard problem. However, those are the best areas of research.

Special thanks to Stephen Omohundro and Raimund Seidel for their many helpful ideas and to Michael Saunders and Philip Gill for their assistance with LSSOL.

Appendix A: Solving Equations for High Dimensional Objects

This appendix shows how to implement the functions described in the beginning of Section 2.

MakePlaneFromPoints() -- given the D D-dimensional points $\tilde{v}_0$ through $\tilde{v}_{D-1}$, solving for $\tilde{P}$ in the following system of equations

$$\begin{bmatrix} v_{0,0} & v_{0,1} & v_{0,2} & \cdots & v_{0,D-1} \\ v_{1,0} & v_{1,1} & v_{1,2} & \cdots & v_{1,D-1} \\ v_{2,0} & v_{2,1} & v_{2,2} & \cdots & v_{2,D-1} \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ v_{D-1,0} & v_{D-1,1} & v_{D-1,2} & \cdots & v_{D-1,D-1} \end{bmatrix} \cdot \begin{bmatrix} P_0 \\ P_1 \\ P_2 \\ \cdots \\ P_{D-1} \end{bmatrix} = \begin{bmatrix} -1 \\ -1 \\ -1 \\ \cdots \\ -1 \end{bmatrix}$$

produces the coefficients of the plane

$$P_0 x_0 + P_1 x_1 + P_2 x_2 + \ldots + P_{D-1} x_{D-1} + 1 = 0$$

which passes through all the $\tilde{v}_i$'s. The complexity of this operation is $O(D^3)$. It should be noted that computing plane equations in this way does not allow all planes to be represented. If any of the $\tilde{v}_i$'s is on the origin, it is impossible to solve this set of equations. However, this method is more efficient than solving for a D+1 dimensional $\tilde{P}$ and one can always translate their data set to ensure that no $\tilde{v}_i$'s do lie on the origin.

MakePlaneFromNormal() -- given a D-dimensional point $\tilde{v}$ and a D-dimensional normal $\tilde{n}$ (not necessarily of unit length), solving for $\tilde{P}$ as

$$P_i = -\frac{n_i}{\tilde{v} \bullet \tilde{n}}$$

produces the coefficients of the plane

$$P_0 x_0 + P_1 x_1 + P_2 x_2 + \ldots + P_{D-1} x_{D-1} + 1 = 0$$

which passes through $\tilde{v}$ and is perpendicular to $\tilde{n}$. The complexity of this operation is $O(D)$.

TestPointVsPlane() -- given a D-dimensional point $\tilde{v}$ and a D-dimensional plane $\tilde{P}$, return

$$P_0 v_0 + P_1 v_1 + P_2 v_2 + \ldots + P_{D-1} v_{D-1} + 1$$

which is a measure of $\tilde{v}$'s distance to $\tilde{P}$. This does not return the actual minimum Euclidean distance from $\tilde{v}$ to $\tilde{P}$ but the distance scaled by $\sqrt{P_0{}^2 + P_1{}^2 + \ldots + P_{D-1}{}^2 + 1}$. The complexity of this operation is $O(D)$.

BarycentricCoords() -- given the D+1 D dimensional points $\tilde{v}_0$ through $\tilde{v}_D$ and a query point $\tilde{w}$, solving for $\tilde{B}$ in the following system of equations

$$\begin{bmatrix} v_{0,0} & v_{1,0} & v_{2,0} & \cdots & v_{D,0} \\ v_{0,1} & v_{1,1} & v_{2,1} & \cdots & v_{D,1} \\ \ldots & \ldots & \ldots & \ldots & \ldots \\ v_{0,D-1} & v_{1,D-1} & v_{2,D-1} & \cdots & v_{D,D-1} \\ 1 & 1 & 1 & \ldots & 1 \end{bmatrix} \cdot \begin{bmatrix} B_0 \\ B_1 \\ \ldots \\ B_{D-1} \\ B_D \end{bmatrix} = \begin{bmatrix} w_0 \\ w_1 \\ \ldots \\ w_{D-1} \\ 1 \end{bmatrix}$$

produces the barycentric coordinates of $\tilde{w}$ relative to all the $\tilde{v}_i$'s. If all the $\tilde{B}_i$'s are between 0 and 1, then $\tilde{w}$ lies inside the simplex defined by the $\tilde{v}_i$'s. The complexity of this operation is $O(D^3)$.

MakeSphereFromPoints() -- given the D+1 D-dimensional points $\tilde{v}_0$ through $\tilde{v}_D$, solving for $\tilde{C}$ in the following system of equations

$$\begin{bmatrix} v_{0,0} & v_{0,1} & \cdots & v_{0,D-1} & 1 \\ v_{1,0} & v_{1,1} & \cdots & v_{1,D-1} & 1 \\ \ldots & \ldots & \ldots & \ldots & \ldots \\ v_{D-1,0} & v_{D-1,1} & \cdots & v_{D-1,D-1} & 1 \\ v_{D,0} & v_{D,1} & \cdots & v_{D,D-1} & 1 \end{bmatrix} \cdot \begin{bmatrix} C_0 \\ C_1 \\ \ldots \\ C_{D-1} \\ C_D \end{bmatrix} = \begin{bmatrix} \sum v_{0,i}{}^2 \\ \sum v_{1,i}{}^2 \\ \ldots \\ \sum v_{D-1,i}{}^2 \\ \sum v_{D,i}{}^2 \end{bmatrix}$$

produces the center and radius

$$\text{center}_i = \frac{C_i}{2} \ ; \ 0 \le i < D \qquad \text{radius} = \sqrt{\frac{\sum_{i=0}^{D-1} (C_i)^2}{4} + C_D}$$

of the sphere which passes through all the $\tilde{v}_i$'s. The complexity of this operation is $O(D^3)$. Because of the unintuitive nature of this calculation, we will show its derivation for D=2. Extending the derivation to higher dimensions should not require an unreasonable leap of faith.

The general equation for a circle is

$$(x - x_0)^2 + (y - y_0)^2 = r^2$$

We need to solve for the center (x_0, y_0) and radius r of the circle. Expanding the above equation and rearranging terms produces

$$-2xx_0 - 2yy_0 + x_0^2 + y_0^2 - r^2 = -x^2 - y^2$$

First dividing by -1 and then replacing $2x_0$ with A, $2y_0$ with B and $-x_0^2 - y_0^2 + r^2$ with C, this can be re-written as

$$Ax + By + C = x^2 + y^2$$

Putting this into matrix form and plugging in the 3 data points P_0, P_1, and P_2 produces

$$\begin{bmatrix} P_{0,x} & P_{0,y} & 1 \\ P_{1,x} & P_{1,y} & 1 \\ P_{2,x} & P_{2,y} & 1 \end{bmatrix} \cdot \begin{bmatrix} A \\ B \\ C \end{bmatrix} = \begin{bmatrix} P_{0,x}^2 + P_{0,y}^2 \\ P_{1,x}^2 + P_{1,y}^2 \\ P_{2,x}^2 + P_{2,y}^2 \end{bmatrix}$$

After solving this set of equations for A, B and C, the center of the circle is calculated as

$$x_0 = \frac{A}{2}$$

$$y_0 = \frac{B}{2}$$

and its radius is

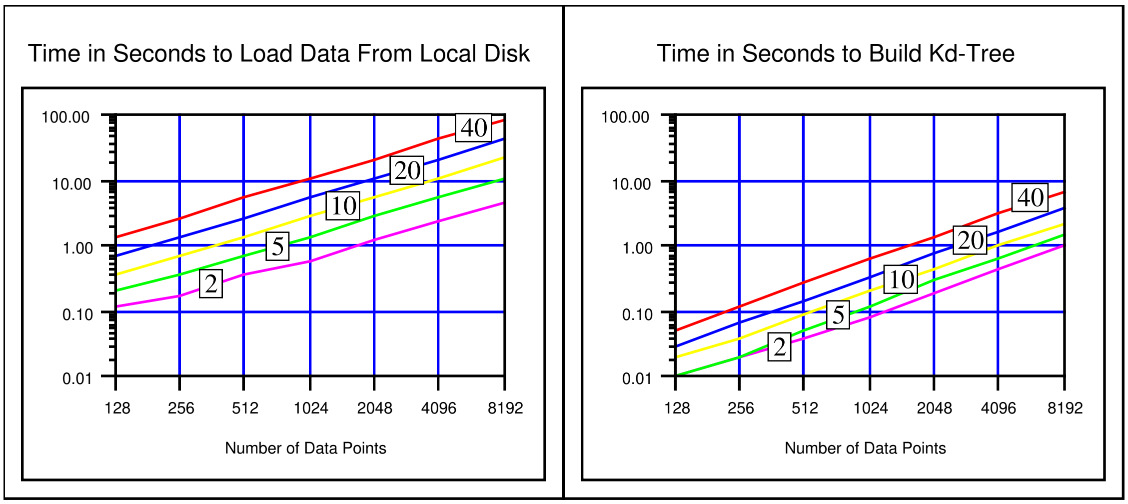
$$r^2 = x_0^2 + y_0^2 + C$$

$$r = \sqrt{\frac{A^2}{4} + \frac{B^2}{4} + C}$$

Appendix B: Kd-Trees in High Dimensions

The purpose of this appendix is to analyze the utility of Kd-Trees [BENT75] in high dimensional spaces[1].

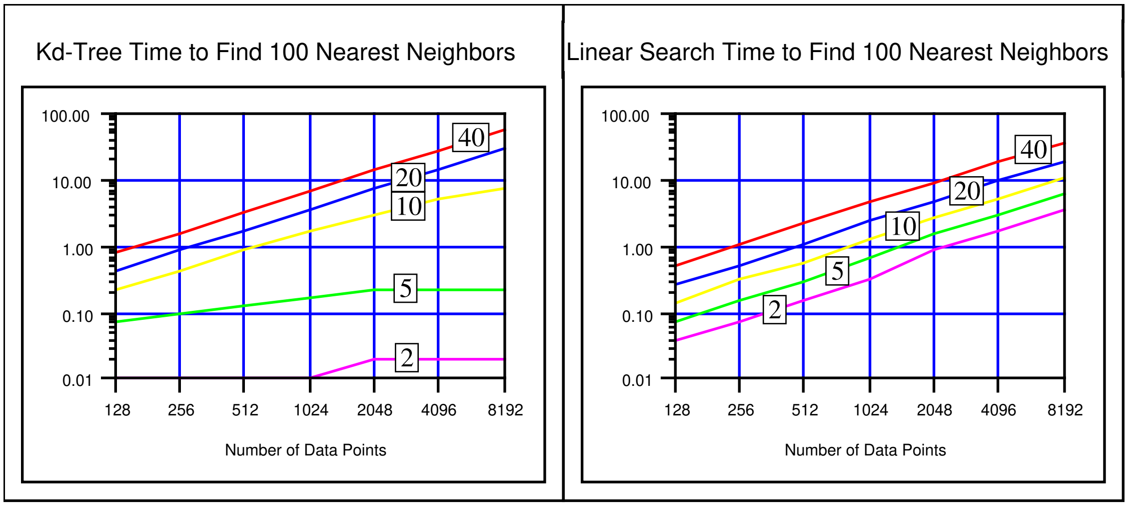
The first issue is how long it takes to build the Kd-Tree data structures. Since they need only be built once per data set, the build time will be compared against the time required to read the data set from a local disk.



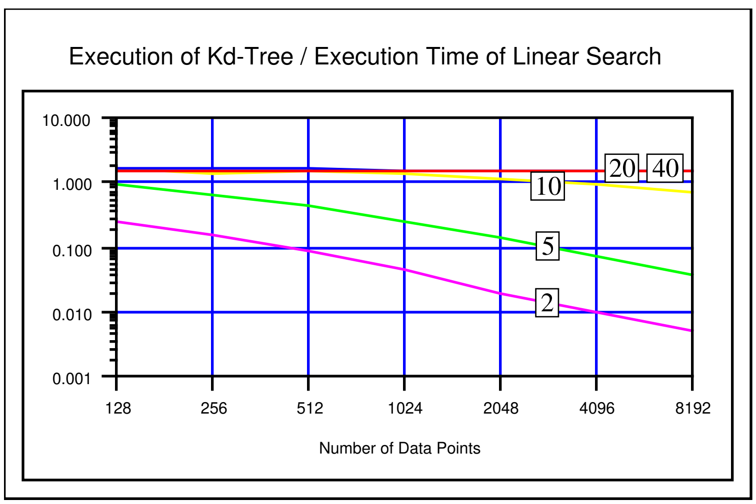
The above graphs show that a Kd-Tree can be built faster than the data for it can read from disk! In fact, it takes only about 10% of the time required to read the data set from a local disk. Assuming that CPU performances will continue to improve faster than I/O performances, this percentage can only decrease. It is clear from this that constructing a Kd-Tree will not present any kind of a bottleneck.

The next step of this analysis is to measure the performance of Kd-Trees. We chose as our performance metric <u>the time to find the nearest neighbor to a query point</u>. Given a random data set and 100 random query points drawn from the same Gaussian distribution, the execution times of the 100 Kd-Tree searches for the nearest neighbors were compared to the execution times of simple linear searches for the nearest neighbors.

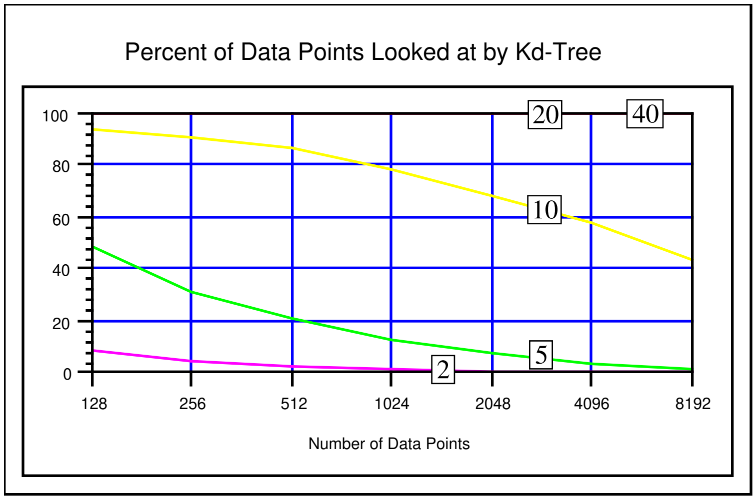
1. All measurements were made on a SPARCstation2 with 16MB RAM and a 600MB external SCSI disk. The Kd-Trees were constructed by splitting along the dimension with the highest variance (as opposed to the longest dimension of a bounding box).



A more natural way to look at this data is by dividing the execution time of the Kd-Tree searches by the execution time of the linear searches. When the result is less than 1, the Kd-Tree outperforms the linear search.



Whereas the linear search's performance is only affected by the number of points in the data set, the Kd-Tree's performance is closely tied to the percent of the data points it looks at.



The above graphs have answered two interesting questions. The first is, "How much slower are Kd-Trees than linear searches when all the data points are looked at?" The answer is that they are slower by approximately a factor of 1.5. The second question is, "Under what conditions are Kd-Trees slower than linear searches?" A rule-of-thumb answer is that Kd-Trees are slower when more than half of their data points are looked at. This appears to occur when the number of data points is $< 4 \times 2^D$. When there are not enough data points, a Kd-Tree cannot partition space effectively. The overhead in traversing the Kd-Tree makes it slower than the simple linear search. There is, however, a caveat to this. As was stated in Section 3.2, these high dimensional problems are likely to be lower dimensional manifolds embedded in a higher dimensional space. By modifying the Kd-Tree algorithm to split along arbitrary planes rather than only planes perpendicular the coordinate axes, one would expect to see the Kd-Tree performance behave as if the problem were really expressed in the dimensionality of the manifold. [SPRO90]

It has been shown that the cost for building a Kd-Tree is low. Assuming that an application will do a fair amount of computation between each Kd-Tree call, a worst case performance degradation of 50% is not unacceptable considering the potential performance gains.

Bibliography

[BENT75] J.L. Bentley, "Multidimensional Binary Search Trees Used for Associative Searching", Comm. ACM 18(9):509-517, September 1975.

[EDEL87] H. Edelsbrunner, "Algorithms in Computational Geometry", Heidelberg, Springer-Verlag, 1987.

[EUBA88] R.L. Eubank, "Spline Smoothing and Non-parametric Regression", New York, M. Dekker, 1988.

[ODLY89] A. M. Odlyzko and N. J. A. Sloane, "New bounds on the Number of Unit Spheres That Can Touch a Unit Sphere in n Dimensions", Journal of Combinatorial Theory, Series A, 26(2):210-214, March 1979.

[OMOH89,1] Stephen M. Omohundro, "Geometric Learning Algorithms", International Computer Science Institute Technical Report TR-89-041, 1989.

[OMOH89,2] Stephen M. Omohundro, "Five Balltree Construction Algorithms", International Computer Science Institute Technical Report TR-89-063, 1989.

[OMOH90] Stephen M. Omohundro, "The Delaunay Triangulation and Function Learning", International Computer Science Institute Technical Report TR-90-001, 1990.

[OMOH91] Stephen M. Omohundro, "Bumptrees for Efficient Function, Constraint and Classification Learning", In Lippmann, Moody, and Touretzky, (eds.) Advances in Neural Information Processing Systems 3. San Mateo, CA, Morgan Kaufmann Publishers, 1991.

[PASC82] I. Paschinger, "Konvexe Polytope und Dirichletsche Zellenkomplexe", Ph.D. Thesis, Math. Inst., Univ. Salzburg, 1982.

[PREP85] Franco P. Preperata and Michael Ian Shamos, "Computational Geometry, An Introduction", New York, Springer-Verlag, 1985.

[SEID82] R. Seidel, "The Complexity of Vornoi Diagrams in Higher Dimensions", Proc. of the 20th Allerton Conference on Communication, Control, and Computing, 1982.

[SEID87] R. Seidel, "On the Number of Faces in Higher Dimensional Voronoi Diagrams", Proc. of the Third Annual Symposium on Computational Geometry, 1987.

[SOMM58] D.M.Y. Sommerville, "An Introduction to the Geometry of N Dimensions", New York, Dover Publications, 1958.

[SPRO90] R.F. Sproull, "Refinements to Nearest-Neighbor Searching in k-d Trees", Sutherland, Sproull and Associates Technical Report SSAPP #184c.