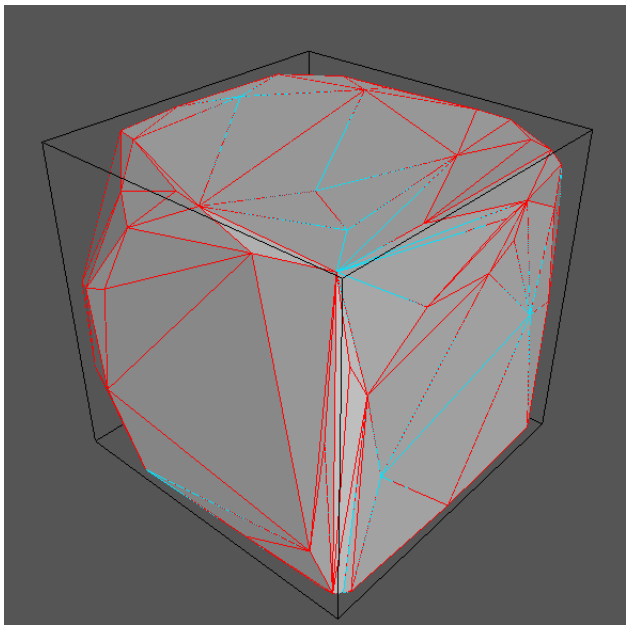
Finding and Visualizing Higher-Dimensional Convex Hulls With Quickhull

Ruby Malusa
CMSC 310

May 9, 2019



1 Introduction

The problem of computing the convex hull of a set of points is fundamental to computational geometry. In two or three dimensions, the convex hull of a set of n points has combinatorial complexity $O(n)$, and a lower bound of $\Omega(nlog(n))$ has been established by reducing the problem to sorting[1]. However, in dimension four or higher, the complexity of the hull becomes an overshadowing issue. A d-dimensional polytope, with m vertices, can have $O(m^{\lfloor d/2 \rfloor})$ facets ($n-1$-dimensional components, like the edges of a polygon or the faces of a polyhedron)[1]. This means that, for example, any algorithm that enumerates the facets of a 4D convex hull cannot do better than $O(n^2)$ in all cases.

In light of this, a great deal of effort has been devoted to output-sensitive algorithms, whose complexity depends on both the number of input points and the number of facets, and which perform better in the frequent case where there are many less facets than the maximum possible. One of the most popular of these is the higher-dimensional Quickhull algorithm, designed by Barber, Dobkin and Huhdanpaa in 1996[2], which forms the backbone of the software tool Qhull. Quickhull is an iterative algorithm, but rather than picking points randomly, it selects at each step the point that maximizes the distance to the hull facet that point is above. Ideally, this eliminates points from consideration by placing them inside the iterated hull at that step, reducing the total number of points that are processed when many are interior to the final hull. Quickhull has worse worst-case complexity than some other algorithms at the forefront of the output-sensitive arms

race, but it is fast for this common family of cases, and has the advantage of being (relatively) conceptually simple.

Visualizing the results of these algorithms in 4- or higher-dimensional space can be difficult. Helpfully, a variety of tools exist to create 3D projections of 4D polytopes, although these still require interpretation. Another, less-used method of visualization is cross-sectional slicing. Series of cross-sections can be unenlightening as they can fail to make it clear where the vertices and other characteristics of the original polytope are located. But hopefully, by combining cross-sections with projections and highlighting the locations of vertices as the hull cross-sections grow to meet them, some understanding can be communicated.

2 Related Work

In 1970, Chand and Kapur wrote that "practically no effort" had been made to discover algorithms for nonplanar convex hulls that were more efficient than the brute-force method, which has $O(n^3)$ complexity even in 2D. They introduced the gift-wrapping algorithm, in which an $n-1$ dimensional hyperplane is rotated over $n-2$-dimensional ridges of the eventual hull to uncover adjacent facets[3]. For n input points whose hull has F facets, gift-wrapping has complexity $O(nF)$[4].

Many subsequent attempts take a different approach. The "beneath-beyond" algorithm, as described by Seidel in 1981, is an extension of the planar incremental convex hull algorithm into higher dimensions. In dimension d, with n input points, it has time complexity $O(n^{\lfloor (d+1)/2 \rfloor})$[5]. It is not output sensitive, but when complexity is measured only in terms of the input size, it is worst-case optimal for even dimensions greater than four. In 1989, Clarkson and Shor created a faster, randomized incremental algorithm, with expected complexity $O(n^{\lfloor d/2 \rfloor})$ in dimension $d \geq 3$. It works in a dual space of intersecting halfplanes, and maintains a "conflict graph" which reduces the necessary processing by tracking (the dual equivalent of) which points can see which facets of the hull and vice versa[6].

The authors of Quickhull credit Clarkson and Shor's algorithm as the basis for their work. Quickhull is an incremental algorithm, and while it does not work in dual space, it maintains "outside sets" of the points visible to each facet, which are equivalent to the conflict graph. Its innovation is in choosing the furthest point in each facet's outside set, rather than choosing randomly. Provided that certain balance conditions defined by the authors are met (so that choosing the furthest point is actually likely to eliminate interior points, if they exist), this change improves performance when there are interior points and maintains the same worst-case performance when all points are on the hull[2].

This figure from the 1996 paper describes the structure of Quickhull:

```
create a simplex of d + 1 points
for each facet F
    for each unassigned point p
        if p is above F
            assign p to F's outside set
for each facet F with a non-empty outside set
    select the furthest point p of F's outside set
    initialize the visible set V to F
    for all unvisited neighbors N of facets in V
        if p is above N
            add N to V
    the boundary of V is the set of horizon ridges H
    for each ridge R in H
        create a new facet from R and p
        link the new facet to its neighbors
    for each new facet F'
        for each unassigned point q in an outside set of a facet in V
            if q is above F'
                assign q to F''s outside set
    delete the facets in V
```

Fig. 1. Quickhull Algorithm for the convex hull in $\mathbf{R}^d$.

First, a simplex is constructed from the input points, ideally from ones with extreme coordinates. Then the rest of the input is sorted into outside sets. This is done by finding the oriented hyperplane through each facet and placing each point into the outside set of the facet whose hyperplane it is above (that is, has positive signed distance to). Each subsequent step involves picking a facet with a non-empty outside set, finding the furthest point in that outside set, and connecting it to the existing hull with a cone of new facets, deleting the facet(s) underneath. If the initial simplex has extreme coordinates, nondegenerate points will only be above one facet and only that facet will be deleted, but in special cases an additional step is needed

to find the set of visible facets to delete. Then the points in the outside sets of the deleted facets are re-sorted into the outside sets of the new facets, or discarded if they are inside the new hull. This process repeats until all facets have empty outside sets. The paper contains a proof that this does produce the convex hull of the input set, no matter which of the facets it is above a point is assigned to.

3 Work

The Quickhull algorithm was implemented for 3D and 4D point sets, using C++ and CGAL. The results were tested against those of the Qhull implementation to verify their correctness, and Geomview was used for visualizations.

CGAL is the geometric backbone of the project[7]. CGAL provides a 2D/3D and a dD geometry kernel, each of which contains representations of points, vectors, planes and so on[8][9]. These kernels can use either Cartesian or homogeneous coordinate representations; for this project, the homogeneous kernel was chosen for its robustness. Each kernel is parameterized to use a number type. As the homogeneous coordinates allow for the use of a common denominator to avoid division in most operations, the homogeneous kernels require only a field number type FT, which may not have a defined division operation. However, division is still necessary to return Cartesian coordinates or distances. In these cases, the function returns an object of CGAL's Quotient⟨FT⟩ type, representing one FT divided by another, which can be converted to a double or printed as necessary.

The 3D Quickhull implementation in qhull_3.cpp uses the number type Gmpzf, an exact arbitrary-precision floating-point type provided by the GMP library[10][11]. Coordinates are stored homogeneously as Gmpzfs, and conversion to double is used when comparing (squared) distances or printing off Cartesian coordinates. However, difficulty was encountered when attempting to parameterize the dD Homogeneous_d kernel with Gmpzf, so the 4D implementation in qhull_4.cpp uses a homogeneous kernel parameterized with doubles instead. This did not cause problems when finding the convex hull of the vertices of a hypercube, but there is a risk of imprecision errors in other cases involving co-hyper-planar points.

CGAL's Combinatorial Maps package, and its Linear Cell Complex extension, were used to store the structure of the hulls[12][13]. Combinatorial maps are a general-dimension equivalent of half-edge representations, in which polytopes are stored as collections of darts with pointers corresponding to each dimension. The Combinatorial Maps package provides various iterators over ranges of darts, allowing access to, for example, one dart per i-cell. It also provides modification operations, such as adding, removing, sewing and unsewing cells. A set of boolean marks is used for these operations, and is also available to the user for searches and similar tasks. Unfortunately, the iteration and sewing operations seem to use the same marks and tread on each other's toes at times. Sewing causes CGAL assertion violations (the condition "all darts of orbit unmarked" is not satisfied) and crashes if it is called within two nested loops iterating over darts, but works as expected if it is called afterwards. This is not mentioned in the documentation, which was useful but very terse in describing how to use the package.

The Combinatorial Maps package also provides the ability to associate arbitrary-type attributes with cells of some dimension, so that the same i-attribute is assigned to all darts belonging to a given i-cell. The Linear Cell Complex extension adds spatial structure to the combinatorial map by automatically associating a point in space with each 0-cell, and provides functions for operating on these points. The iterative hulls are stored in linear cell complexes; to each 0-cell is associated a point as well as an integer used for indexing when writing output to a file, and to each dim-1-cell is associated a tuple, which contains a supporting hyperplane, a list of points containing the outside set of that facet, and a boolean indicating whether the facet has already been processed as part of the visible set during a quickhull step and should be deleted.

In the execution of either implementation, a simplex of linearly independent, ideally extreme points is constructed. (If d+1 linearly independent points cannot be found, the program prints an error message and exits.) Each dart is set up with appropriate attributes, and one dart handle on each facet is added to a list of facets to be processed. The program iterates through that list until it is empty, processing each facet as specified by the Quickhull algorithm. If the facet's outside set is not empty, a breadth-first search over adjacent facets is used to find both the set of facets visible to the furthest point and the boundary ridges of that set. Then, a cone of new facets is created. These facets have attributes associated and are sewn along matching ridges, to each other and to the boundary, before being added to the list of facets to process.

All the facets in the visible set are not deleted immediately, as they may still be present later in the list of facets to process. Instead, they are unsewn from their neighbors, so that they are unconnected and do not appear in searches, and their boolean dim-1-attribute is set to false, so that if the algorithm attempts to process them later, they are deleted instead. Those that are not in the list of facets to be processed will still be present at the end of execution, and so, when the hull output is written to a file, facets with false dim-1-attributes are ignored.

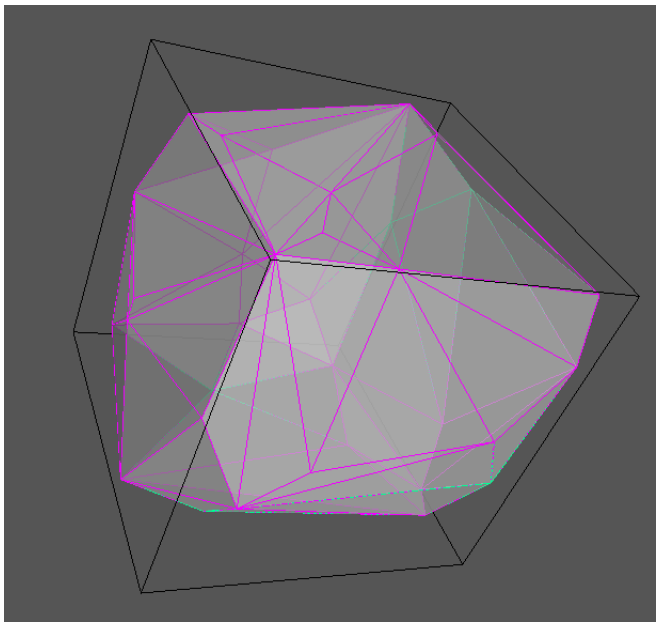
Output is written as an OFF file, hull_output.off. The 3D implementation writes a normal 3D OFF file listing each point and the triangular faces connecting them. The 4D implementation writes an nOFF file with dimension four, specifying the 4D coordinates of the points and the triangular ridges connecting them. (Qhull writes the 3-facets of the polytope rather than the 2-ridges, but this format was useful for visualization with Geomview; it would be simple to modify the output to match Qhull's format.) There is no facet merging, so the hull of a 3d cube, for example, has two triangular faces in place of each square face of the cube. However, if facet merging was desired, an additional step could be written to merge coplanar facets before exiting.

The program (in either dimension) can run in four modes, depending on which flag is passed to its executable. Passing "c" runs an example that calculates the example of an d-cube. "r" followed by an integer n generates n random points in a 200-by-200 d-cube centered on the origin, then calculates their hull. "f" followed by a file path reads in points from a file. "h", or no flags, provides information on the different modes and on the file format for input. The point set processed is written to input_pts.txt, so it can be examined or processed again, and the 4D implementation also writes hypercubes.off, which contains one hyper-rectangular prism centered on each input point, to aid in visualization.

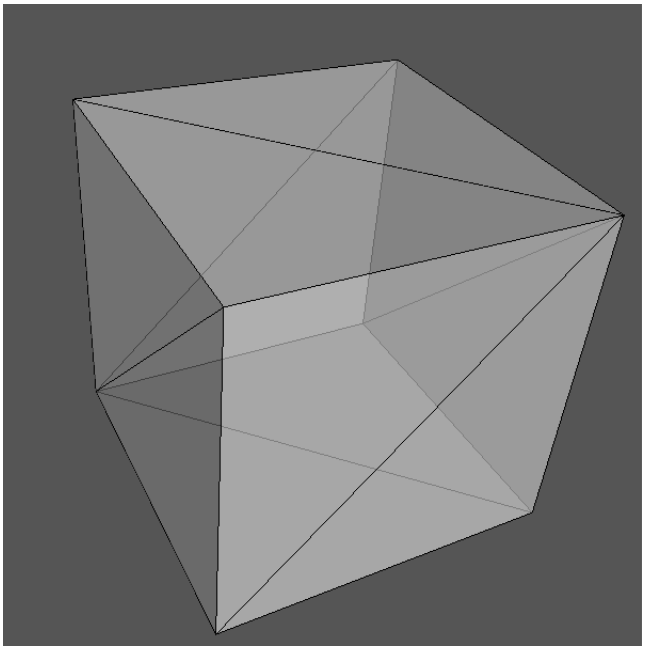
Please consult the README for information on compiling and running the program.

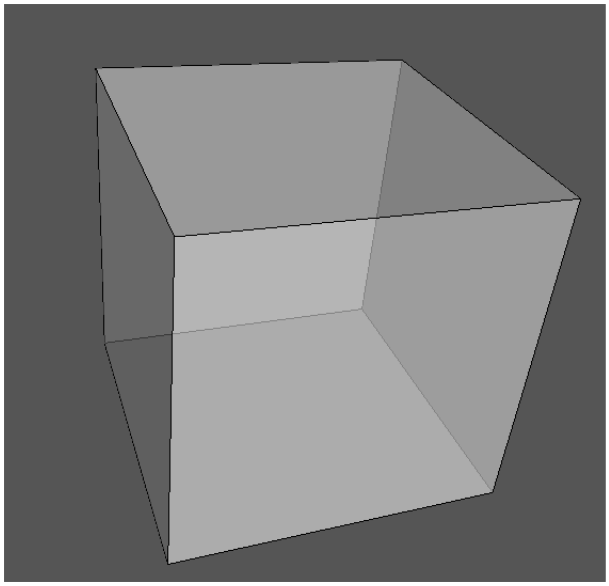
4 Results and Visualization

The results of running the program on 3D and 4D point sets, random, simplicial or d-cubical, are equivalent to the results of running QHull's implementation, differing only in their specific point indexing and in the lack of coplanar facet merging.

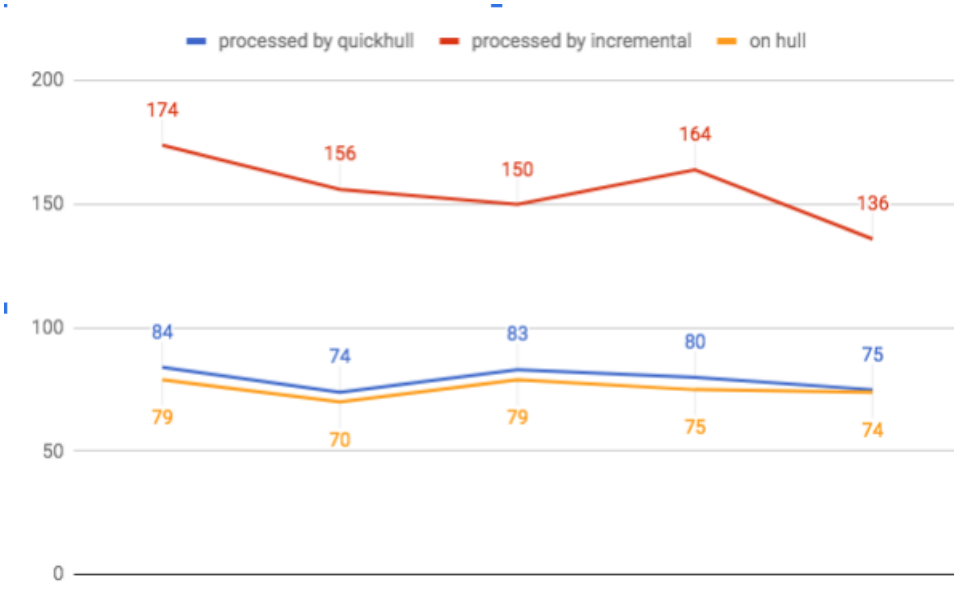


Above is pictured the convex hull of 100 points in 3D, rendered in Geomview (interior points are not shown). The hull produced by qhull_3 and Qhull's implementation are overlaid, with different edge colors, and it can be seen that they are the same (the placement of the pink and green sections is a consequence of the view angle, and changes as the shape is rotated). The figure that opens this report is similar, but with 1000 points instead.



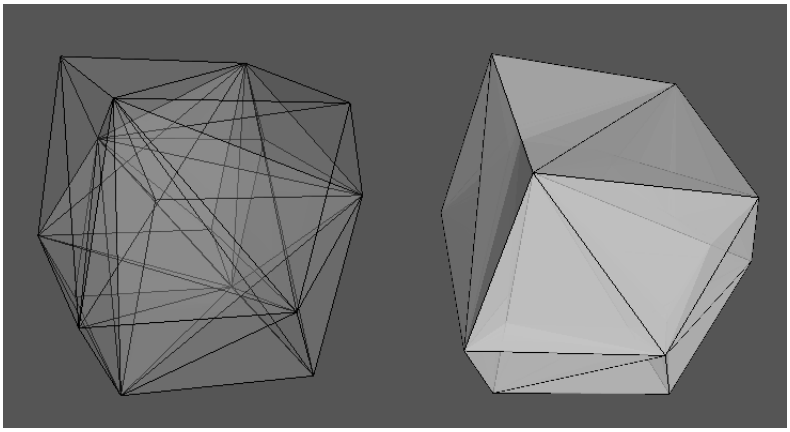


Here, the hulls of a a cube produced by qhull_3 and Qhull are compared to demonstrate the lack of facet merging.

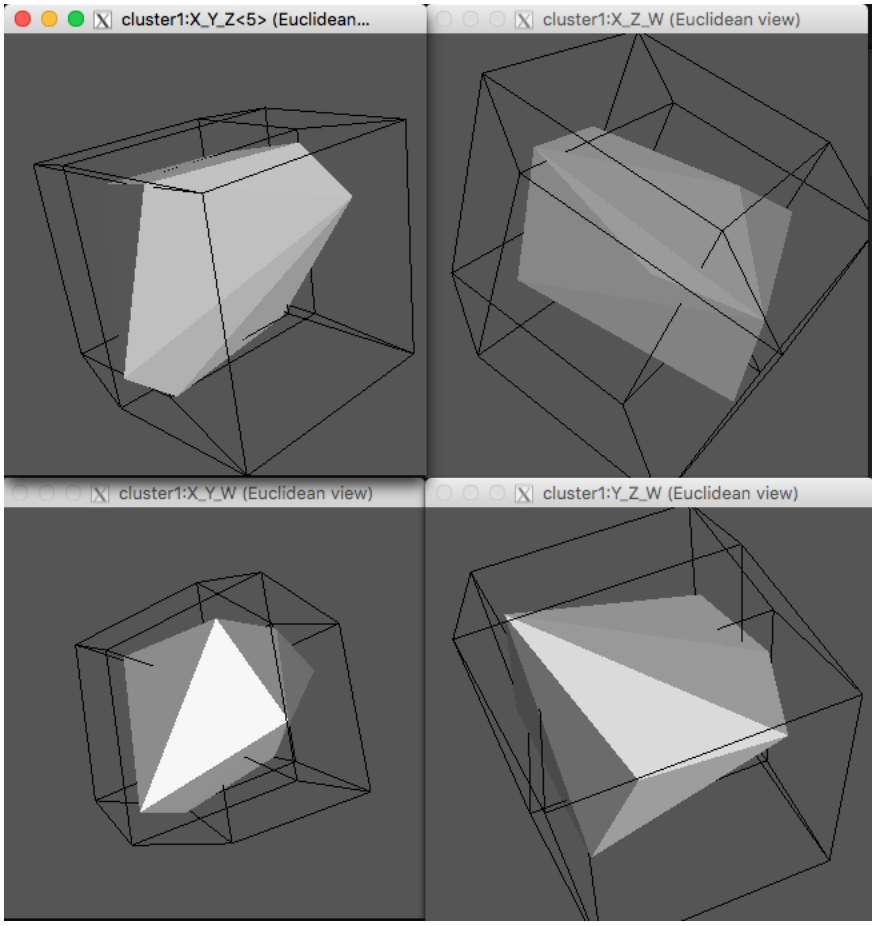


In the 1996 Quickhull paper, the authors provide a graph comparing points processed by Quickhull, points processed by a modified version with a random selection step, and points actually on the hull. Similar results were produced by this implementation, running on input sets of 1000 3D points scattered inside a cube.

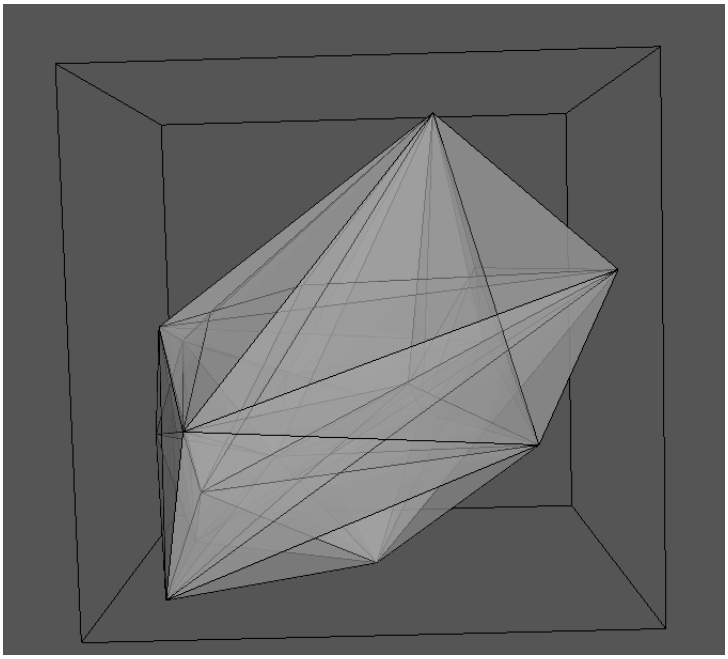
The more interesting visualization results were produced from the 4D hulls. Geomview's NDview and Slicer modules were used to create projections and cross-sections[14][15].



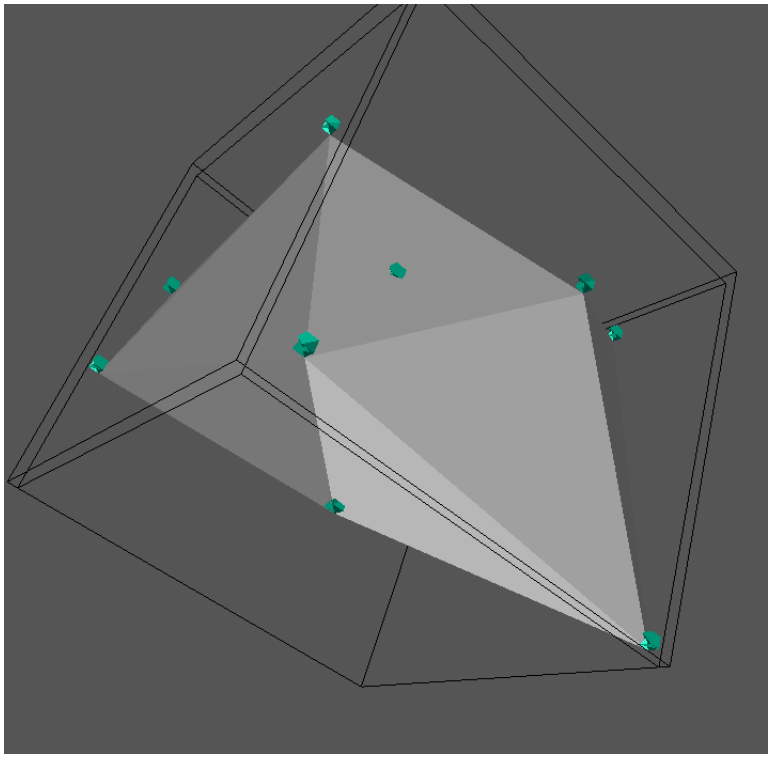
Here are two projected renderings of the hypercubical hull produced by qhull_4, at different transparencies, with divided square ridges visible. In order to generate these images, the nOFF files were opened in Geomview, and NDview was used to increase the dimension of the space and create projective cameras.



Here, NDview is used to create four views of the 4D hull of a set of points. Each omits one coordinate; the windows are labeled XYZ, YZW, etc. depending on which coordinates are used. These views are available as presets in NDview. When the polytope is rotated, all the views change accordingly.

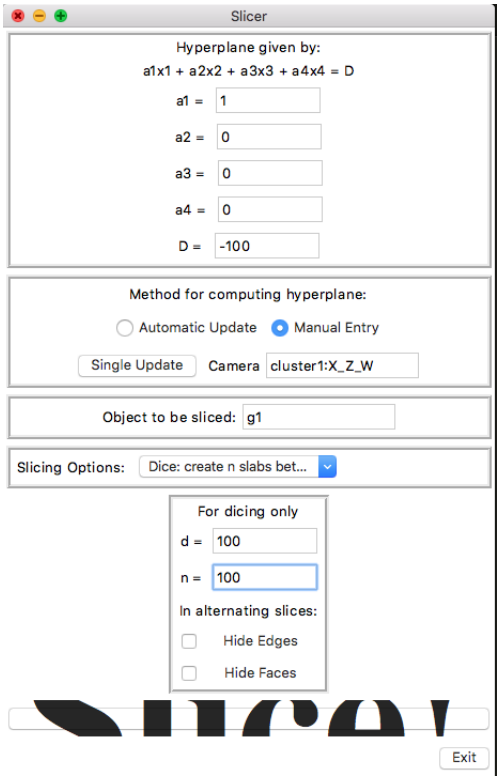


A projection of another random hull is shown here with transparent 2-cells, so that the interior cells are visible. The hull is composed of 4-simplices, so the projected 2-cells are all triangles, and the 3-cells are tetrahedra.

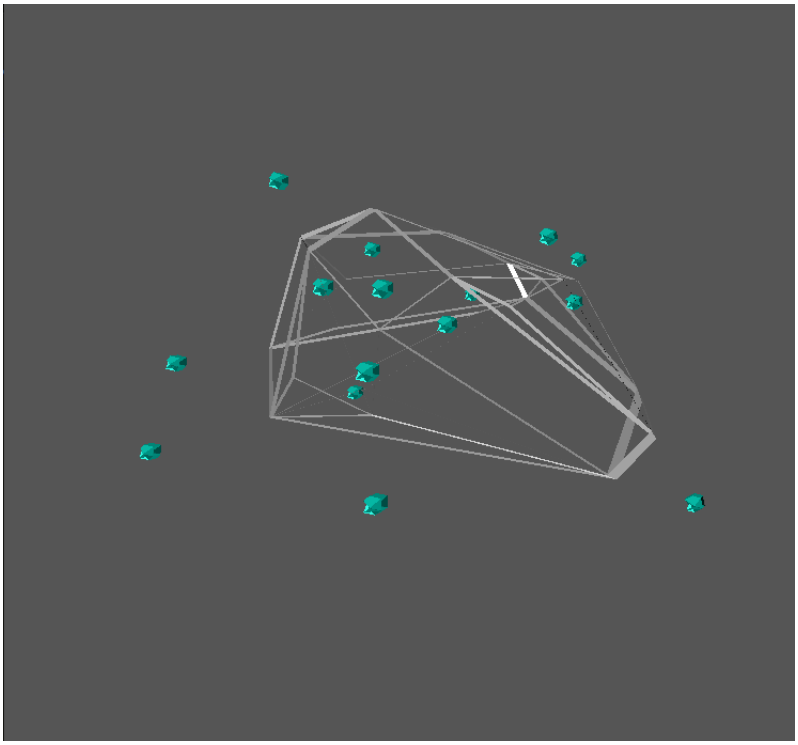


Here is a projection of another random hull, which will be used to demonstrate the slicing technique. hypercubes.off is also included and colored blue; in projection onto the yzw hyperplane, this produces a blue dot at the (y,z,w) location of each of the input points.

The Slicer module does not actually generate exact hyperplanar cross-sections. Instead, it cuts through an object with a plane, dividing it into sections on either side. There is also an option to "dice," by cutting along a series of evenly spaced planes. The interface for dicing is pictured below.

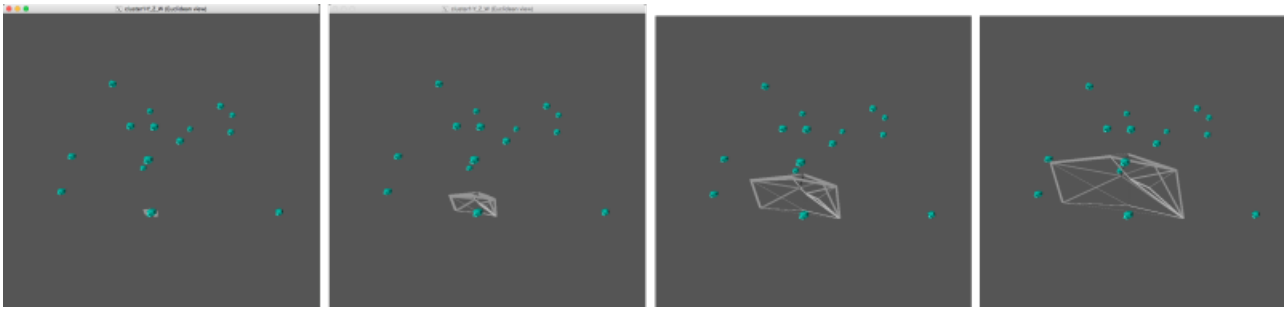


These settings dice the polytope into 100 sections with a series of hyperplanes spaced between $x = -100$ and $x = 100$. These sections have some width, but when they are projected one at a time onto the yzw-plane, they approximate the edges of successive 3d polyhedral cross-sections of the 4d hull.

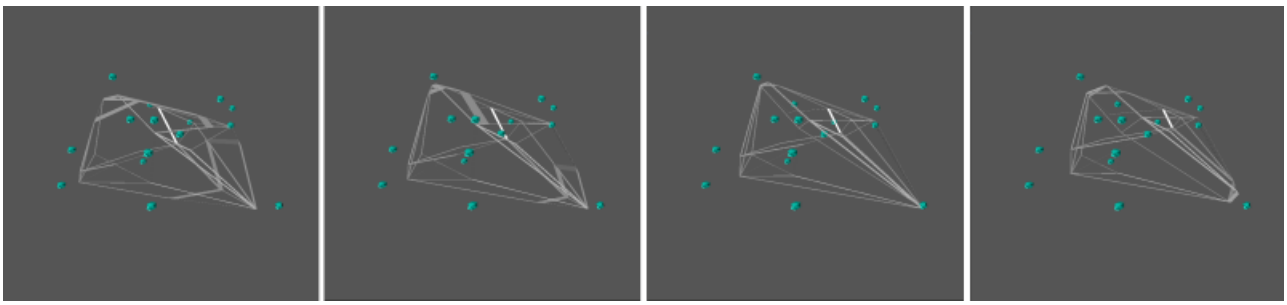


This is most striking when a set of cross-sections is made into an animation; as the x-coordinate of the slicing plane approaches the x-coordinate of one of the projected input points, one can see the hull grow towards,

reach, and recede away from that input point. Some such animations are included in the /animations folder.



Above, the cross-sectional polyhedron grows out from the initial point of contact with the slicing hyperplane in successive slices.



In these images, the cross-section comes towards, reaches, and then begins to recede from the lower rightmost point.

5 Conclusion and Future Work

Learning to use CGAL was fascinating and deeply stressful. It really is a very powerful library, modular and customizable in every way you can imagine, and that hugely raises the difficulty of getting started even with something basic. The documentation for the packages I was using (understandably) assume familiarity with the other CGAL components on which they are built, and (frustratingly) leave out a lot of helpful detail. Compiler errors are harder to understand when everything is wrapped in several layers of parameterization keywords, and the source of assertion violations at runtime was not always clear. I still do not know why the dD homogeneous kernel cannot be parametrized with Gmpzf; the compiler says a division operation is necessary, but it works fine for the 2/3D kernel, where division returns a Quotient. My level of understanding of the packages I used has improved, but I would like to get more familiar with the nuts and bolts of how their parameterization and kernels work so that learning other packages might be easier.

Apart from that, most of the problems I encountered were fairly low-level bugs in, ex., breadth-first search, or the functions used to set up planes or glue together matching ridges. I defined as many things as possible in a dimension-independent way in the 3D version, but did not foresee some changes that had to be made based on the differences in the dart structure in higher dimensions. The 4D version of the project is theoretically adaptable to arbitrary dimension. The main code that would have to be changed is the function for determining when two ridges match, which currently is only defined for 3- and 4D and takes advantage of the natural ordering of vertices in line segments and triangles. However, as dimension is defined as a const int in the header so that the properties of the kernel are known at compile time, and the dart attributes contain a tuple of length d that must also be set up at compile time, it would be difficult to make a version that took the dimension as input.

There is a lot more I would like to do with visualization. Geomview is designed to interface easily with other programs, so it would be nice to pipe output directly to Geomview without the intermediate step of

opening it by hand. It would also be great to write a program to generate cross-sections and projections, instead of picking through them by hand as I did. Geomview has a fairly powerful command language and is theoretically capable of these things.

I don't know how to end this conclusion ??

References

[1] S.L. Devadoss and J. O'Rourke. *Discrete and Computational Geometry*. Princeton University Press, 2011.

[2] C. Bradford Barber, David P. Dobkin, and Hannu Huhdanpaa. The quickhull algorithm for convex hulls. *ACM TRANSACTIONS ON MATHEMATICAL SOFTWARE*, 22(4):469–483, 1996.

[3] Donald R. Chand and Sham S. Kapur. An algorithm for convex polytopes. *J. ACM*, 17(1):78–86, January 1970.

[4] Joseph O'Rourke. *Computational Geometry in C*. Cambridge University Press, New York, NY, USA, 1994.

[5] Raimund Seidel. A convex hull algorithm optimal for point sets in even dimensions. Technical report, Vancouver, BC, Canada, Canada, 1981.

[6] Kenneth L. Clarkson and Peter W. Shor. Applications of random sampling in computational geometry, ii. *Discrete & Computational Geometry*, 4(5):387–421, Oct 1989.

[7] The CGAL Project. *CGAL User and Reference Manual*. CGAL Editorial Board, 4.14 edition, 2019.

[8] Hervé Brönnimann, Andreas Fabri, Geert-Jan Giezeman, Susan Hert, Michael Hoffmann, Lutz Kettner, Sylvain Pion, and Stefan Schirra. 2D and 3D linear geometry kernel. In *CGAL User and Reference Manual*. CGAL Editorial Board, 4.14 edition, 2019.

[9] Michael Seel. dD geometry kernel. In *CGAL User and Reference Manual*. CGAL Editorial Board, 4.14 edition, 2019.

[10] Michael Hemmer, Susan Hert, Sylvain Pion, and Stefan Schirra. Number types. In *CGAL User and Reference Manual*. CGAL Editorial Board, 4.14 edition, 2019.

[11] Torbjrn Granlund and the GMP development team. *GNU MP: The GNU Multiple Precision Arithmetic Library*, 6.1.2 edition, 2016.

[12] Guillaume Damiand. Combinatorial maps. In *CGAL User and Reference Manual*. CGAL Editorial Board, 4.14 edition, 2019.

[13] Guillaume Damiand. Linear cell complex. In *CGAL User and Reference Manual*. CGAL Editorial Board, 4.14 edition, 2019.

[14] Nina Amenta, Stuart Levy, Tamara Munzner, and Mark Phillips. *Geomview*, 1.9.5 edition, 2014.

[15] Olaf Holt. *NDView*, 1.0 edition, 2014.