Tetrahedral and Surface Meshing
Strategies for Signed Distance Functions

Stephen Kelly

Supervised by
Professor Hamid Ghadyani

December, 2019
Abstract

In this project the principles of solid modeling, surface meshing, and tetrahedral meshing using signed distance functions are discussed. Three libraries were published as open source projects in the Julia programming to facilitate each of these three aspects. The libraries are: Descartes.jl for signed distance function generation and composition, Meshing.jl for surface meshing, and DistMesh.jl for tetrahedral mesh generation. Both Descartes.jl and Meshing.jl were developed prior to the start of this project, however, several improvements and modifications were developed to facilitate performance and support for the DistMesh.jl library. The primary contribution is the extension and evaluation of the DistMesh algorithm first published by Per-Olof Persson, with a port to the Julia programming language. Throughout the study, methods were evaluated to improve the performance of the algorithm during runtime and improve the final output mesh quality. The Julia version supports additional features along with performance improvements that enable the algorithm to be applied to a broader set of problems. Similarly, the coupling of DistMesh.jl with Descartes.jl and Meshing.jl is the start of a comprehensive design and Finite Element pre-processing toolset for the Julia programming language.
Contents

1 Introduction .......................................................... 3
   1.1 Trends in Solid Modeling ......................................... 3
   1.2 Julia Programming Language .................................... 5

2 Functional Representation of Solid Geometry .................. 6
   2.1 Signed Distance Functions ...................................... 6
   2.2 Overview of Descartes.jl ...................................... 6

3 Surface Meshing Strategies ....................................... 9
   3.1 Marching Cubes .................................................. 9
   3.2 Performance Analysis and Direct Function Sampling ........ 10
   3.3 Multithreaded Marching Cubes ................................ 11

4 DistMesh ............................................................ 12
   4.1 Memory Management and Vectorization ...................... 14
   4.2 Hilbert Sorting .................................................. 15
   4.3 Nonlinear Edge Displacements ................................ 16
   4.4 Distance Caching ............................................... 18
   4.5 Analysis of Parameter Selection ................................ 19
   4.6 Output Quality Analysis ....................................... 19
   4.7 Quality-Based Retriangulation and Termination ............. 20

5 Conclusion .......................................................... 22

Appendices ................................................................ 26
   A Meshing.jl .................................................................. 26
   B Descartes.jl ................................................................ 28
1 Introduction

Modern manufacturing methods change the demands of computer graphics and computational geometry systems. The advent of 3D printing, multi-axis milling, and robotics is creating a need for alternative geometric constructions. Commonplace techniques largely center around the manipulation of surface topologies, through the use of structures such as splines, triangular meshes, and boundary representations. Primarily the concern of these structures are to represent a solid body. In mathematical terms, this is a representation of a manifold subset of a given space. With mathematical representations of solids, complexity can be arbitrarily high. The purpose of this discussion is to demonstrate the use of functional representations of solids, sometimes called implicit or signed distance fields/functions and develop a toolset to aid in computational numerics such as Finite Element Analysis.

1.1 Trends in Solid Modeling

Historically, the choice of representation for CAD models has been the boundary representation. The field of interactive CAD arguably began in 1963 when Ivan Sutherland demonstrated Sketchpad. This program featured an interface using a light pen which made it very familiar to those used to traditional drafting. There were several technological advancements made in Sutherland’s Sketchpad thesis, however potentially the most impactful was the notion of "constraints based modeling". In this system, common drafting tools such as protractors, squares, and rulers were introduced in the digital realm. This technique is now commonplace; a user sketches a drawing using primitives such as lines, arcs, and splines, and later adds dimensions and constraints to the sketch thereby dynamically changing the sketch to meet some design intent. This system was designed to emulate the protractors and drafting tables of the draftsmen of the day.

Marshall McLuhan’s phrase “the medium is the message” is particularly salient here. Today’s modern CAD tools are built around the representation of surfaces. Internal structure is ancillary, as most modern products are stamped, formed, injected, or milled. However, advanced manufacturing technologies requires techniques to allow for the representation of both surfaces and internal geometry of an object. Current CAD tools generally do not have well-formed facilities for the construction of internal structure. Recently, several companies have started to address this issue. In Figure 1 some examples of internal topology for a turbine rotor are constructed with nTopology’s software for additive manufacturing. Consequently, nTopology has published several articles about the use of signed distance field modeling their software, which further validates this trend in solid modeling. It is important to note that traditional mesh structures are still in use here, however they do not constitute the underlying representation of the geometry and rather are a by-product of the signed distance representation.
Two core ideas motivate this project:

- Advanced Manufacturing
- Simulation Aided-Design

One of the core ideas of 3D printing is the concept of infill. Infill is the structure of the material inside the component, similar to the gyroid structure in Figure 1. Several studies have shown that 3D printing achieves orders-of-magnitude reduction in energy use and cost savings when the complexity of the component is high. Infill is largely responsible for allowing the complexity of components to be high and low in material use. Similarly, methods such as topology optimization allow a design to be modified such that mechanical properties are preserved and excess material is removed. However, topology optimization techniques lead to complex components generally suitable for additive manufacturing, rather than traditional manufacturing techniques. The work in this paper covers some initial progress in the Julia programming language towards the development of an API for signed distance field modeling, the construction of surface meshes suitable for 3D printing and visualization, and the construction of tetrahedral meshes for numerical analysis. In Figure 2, it can be seen that these areas are referred to as “pre-processing” and are often a time consuming component of a simulation.
1.2 Julia Programming Language

Julia is a programming language first released in early 2012 by a group of developers from MIT. The language targets technical computing by providing a dynamic type system with near-native code performance. This is accomplished by using three concepts: a Just-In-Time (JIT) compiler to target the LLVM framework, a multiple dispatch system, and type inference. More simply, the language is designed to be dynamic in a way that allows rapid prototyping of code and understandable to a reader, yet provides a design amicable to performance optimizations and specialization. Dynamic type systems allow the programmer to ignore or selectively specify type information, such as the bytes in an integer or dimensions of an array, and allow the compiler to infer propagation of information based on the input types. JIT compilation means code is compiled during runtime which allows functions to be specialized and optimized for various data types given type-generic code.

The syntactical style is similar to MATLAB and Python. The language implementation
and many libraries are available under the permissive MIT license, making them widely used in industry. Benchmarks have shown the language can consistently perform within a factor of two of native C and FORTRAN code. This is enticing for a solid modeling application and for numerical analysis, as the code abstraction can grow organically without performance penalty. In fact, the authors of Julia call this balance a solution to the “two language problem”, also called Ousterhout’s Dichotomy. The problem is encountered when abstraction in a high-level language (Matlab, Python, Tcl, Ruby) will disproportionately affect performance unless implemented in a low-level language (C, Fortran).

2 Functional Representation of Solid Geometry

Solid geometry can be formulated using pure mathematical functions. These representations are sometimes called implicit functions or signed distance functions. The use of functions to represent geometry has seen somewhat spares treatment in computer graphics, as most systems prefer some form of discrete geometry, typically a triangular mesh. However, functional forms remain highly useful in areas such as computer generated imagery where image fidelity is of concern. Functional forms lead to natural formulations for path tracing, ray tracing, and ray marching when modeling the propagation of light in a realistic fashion. There have been some prior examples of functional solid modeling applications in the past, with different objectives in mind.

2.1 Signed Distance Functions

To date there have been several tools designed to provide interfaces for functional geometry.[9] There is a subtle distinction in functional geometry between an implicit function and signed distance function.[10] An implicit function will have a zero-set (boundary) and sign-change for point membership in/out of the domain. A signed distance function retains these properties, however it adds the additional constraint of a consistent metric. For signed distance functions it is common for this metric to be the minimal distance to the boundary. This property is often non-trivial to retain in composition and iterative root-finding methods are therefore required.[11] A simple example of a signed distance function of a sphere can be seen in Figure 3.

2.2 Overview of Descartes.jl

Descartes is the front-end for several different projects that have been developed for computational geometry in the Julia programming language. The core geometric representation is defined using signed distance functions and is designed to have a syntax similar to the programmatic solid modelling tool, OpenSCAD. The objective of the system is to use func-
functional programming principles coupled with multiple dispatch to enable the compilation of vectorized native code that represents a solid model. This ultimately leads to high performance that fully utilizes all available computational power on the CPU. Moreover, this is accomplished without a domain-specific language, Abstract Syntax Tree modification, or macros. This is only possible due to Julia’s just-in-time compilation model, immutability, parametric types, and type inference.

The system is very easy to write, for example:

```julia
1 using Descartes
2 c = Cuboid([5,5,5])
3 h = translate([2.5,2.5,0])Cylinder(1,5)
4 obj = diff(c,h)
5 m = HomogenousMesh(obj)
6 save("cube_with_hole.stl", m)
```

The mesh output of this script can be seen in Figure 4. In line 1, the Descartes library is imported. In line 2 and 3 we instantiate some basic primitives, in this case a cube and cylinder. Transformations, as seen in line 3, are structure similar to matrix transforms. Rotations, translations, and shears may all be represented this way, following the commutivity rules of matrix operations. In line 4, a difference operation is performed, removing the cylinder, ‘h’, from the cube, ‘c’. A mesh is constructed in line 5, by using the ‘HomogenousMesh’ function, and saved in line 6 is a ‘stl’ file, suitable for visualization or 3D Printing.

One novel aspect of Descartes.jl is that it can generate efficient machine code for a functional geometry. In Figure 5 the process of constructing a specialized function is shown. Given arguments to a set operation, a new type is formed, in this case a CSGUnion. The left and right elements of the operation are stored in the type information to allow the Julia compiler to specialize this function. In this case, an ‘FRep’ function is made which
Figure 4: Output from Descartes.jl script

contains the matrix transforms and actual signed distance function for the geometry. When sampling, we construct a functional-closure to switch the functional representation from a 2-ary function to 1-ary function. This allows use with automatic differentiation tools and isosurface extraction without additional complexity and knowing about our geometry. A full discussion of this can be found in the Descartes.jl documentation on GitHub.

Figure 5: Dispatch graph for a given Constructive Solid Geometry tree in Descartes
3 Surface Meshing Strategies

As mentioned in the previous section, ray tracing and ray marching are the only computationally feasible ways to directly visualize an implicit geometry without discretizing the surface. However discretization using meshing, or isosurface extraction, is one of the more tractable methods for allowing the visualization of implicit geometry via the conversion to a triangulated surface. Modern graphics processors are optimized to display triangular meshes, and ray tracing is still not entirely computationally efficient given hardware constraints. To convert a function to mesh, the space is uniformly or adaptively sampled via cuboids/voxels to find the zero-points (or isosurface) of the underlying geometry.\[12\] There are several different approaches to this problem. The most common mechanism is to use a lookup table to match a triangle orientation case.

The Meshing.jl library was started in August 2015 as a platform for the development of isosurface extraction algorithms in the Julia programming language. At the start of this project several different approaches were implemented, including Marching Cubes, Marching Tetrahedra, and Naive Surface Nets. These are frequently abbreviated as MC, MT, and NS throughout this paper. For this project, several new algorithms and modifications to the existing algorithms were implemented to achieve different outputs and accuracy during isosurface extraction. In Figure 6 three different surface meshes were generated on a sphere using Meshing.jl.

![Figure 6: From left: Marching Cubes, Naive Surface Nets, Marching Tetrahedra](image)

3.1 Marching Cubes

Marching cubes is the most common algorithm for isosurface extraction. It is fairly simple to implement and applies 254 pre-determined cases to determine the orientation of faces within a voxel.\[13\] The algorithm was originally designed for fast extraction of geometry from grid-type data from medical imaging machines such as CT Scanners.\[14\] Naturally, since the prescribed cases do not cover all possible types of orientations of faces within a voxel, the algorithm is ambiguous in some cases and does not provide accurate meshing on sharp corners. In Figure 7 some example cases may be seen.
Several improvements were made to the marching cubes algorithm to reduce the amount of data generated in the reference implementation of the algorithm. In particular, the look-up tables were improved to allow for reduced memory allocation and faster construction of face indices. By convention, and for purposes of fast transforms, vertices are stored in one vector, and the triangular faces are represented as indices into this vector of vertices. In Figure 7 it can be seen that there are voxel cases that produce several triangles. In the improved reduced-vertex method the lookup table is modified such that only unique vertices are stored within a voxel, thereby reducing the vertex count by up to around 30% in tests. These improvements were released in version 0.5.0 of the software, along with extensive improvements to documentation, and future support for multithreading.

3.2 Performance Analysis and Direct Function Sampling

Isosurface extraction is typically a tradeoff between performance and resolution. In order to understand why this is the case, it is important to understand some of the architecture in modern computers. On modern computing devices, cache coherence is one of the keys to good performance. CPUs are typically very fast when using memory close to the arithmetic logic units, in i.e. a register. However, if the data is a large sequence, such as an array of tens of thousands of number, the computer often cannot keep these values localized, so need to store them in cache or RAM. There are various levels of cache in modern computers, denominated by L1, L2, L3, etc. Ideally an algorithm will process, or lookup data, in a linear way. However, this is not the case for isosurface extraction algorithms. As the data represents three dimensions, there are no convenient mechanisms to extract a voxel such that the points are near each other in memory. Since we are in 3D we will frequently have cases where we are NxN elements away. This problem is akin to those found in stenciling and BLAS algorithms.\[15\] It has been shown in some dense matrix operations that small
“chunks” of data can be sized based on system cache sizes and vector instructions such that computational throughput remains high. One performance-focused approach to isosurface extraction on 3D arrays would be to follow the BLAS approach. However, requires detection of CPU features and complicates the algorithm. However, the issue of cache locality and memory management may be eliminated completely by directly sampling a function in the isosurface extraction algorithm.

Since this work is in the Julia programming language, it is possible to just-in-time compile a function representing the surface that can be directly sampled, thereby mitigating much of the concern about cache locality. For this project, a direct sampling method was implemented given a function, bounds, and sample count. In contrast to the original implementation (and comparable implementations such as Matlab and Matplotlib), this does not require construction of a 3D Array to sample a function. This allows the allocation of a large 3D matrix to be avoided completely, by passing concerns about cache and memory handling. This process was tested on a torus and the results are seen Figure 8 and Table 1. This shows that while the isosurface extraction is faster on a 3D Array, the allocation of the array is two orders of magnitude slower than direct function sampling.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Samples</th>
<th>Min (ms)</th>
<th>Median (ms)</th>
<th>Mean (ms)</th>
<th>Max (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>SDF Array Construction</td>
<td>18</td>
<td>267</td>
<td>271</td>
<td>283</td>
<td>378</td>
</tr>
<tr>
<td>Function MC</td>
<td>694</td>
<td>6.073</td>
<td>6.461</td>
<td>7.2</td>
<td>350.119</td>
</tr>
<tr>
<td>Function NS</td>
<td>487</td>
<td>8.126</td>
<td>8.39</td>
<td>10.265</td>
<td>879.713</td>
</tr>
<tr>
<td>SDF MC</td>
<td>1254</td>
<td>2.466</td>
<td>2.761</td>
<td>3.958</td>
<td>1368</td>
</tr>
<tr>
<td>SDF MT</td>
<td>519</td>
<td>7.781</td>
<td>8.175</td>
<td>9.637</td>
<td>545.535</td>
</tr>
<tr>
<td>SDF NS</td>
<td>1439</td>
<td>2.783</td>
<td>2.94</td>
<td>3.454</td>
<td>621.475</td>
</tr>
</tbody>
</table>

Table 1: Runtime comparisons for different isosurface extraction algorithms and comparable overhead with 3D Array allocation. (MC - Marching Cubes, MT - Marching Tetrahedra, NS - Naive Surface Nets)

3.3 Multithreaded Marching Cubes

Recent releases of the Julia programming language have stabilized the facilities for multithreading. A multithreaded implementation of Marching Cubes was implemented for this project. For this test, a signed distance function of a torus was created then sampled uniformly into an array (Signed Distance Array). To compare the pre-allocated array and memory-free function sampling, both were passed to the Marching Cubes algorithm and tested with different thread counts to demonstrate the scaling behavior. The timings on a torus with $81^3$ grid samples, median times shown in Table 2.
Figure 8: Histogram of extraction times in Table 1

<table>
<thead>
<tr>
<th>Type</th>
<th>Threads = 1</th>
<th>Threads = 6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Signed Distance Array</td>
<td>2.721 ms</td>
<td>2.212 ms</td>
</tr>
<tr>
<td>Signed Distance Function</td>
<td>6.745 ms</td>
<td>2.699 ms</td>
</tr>
</tbody>
</table>

Table 2: Runtime comparisons for pre-allocated and memory-free Marching Cubes with multithreading

These results for a multithreaded implementation do not scale linearly with the core count, but do achieve good, though non-linear, performance improvements when directly sampling a function. This has been noted by the Visualization Toolkit (VTK) team who developed the Flying Edges algorithm as a modification to the Marching Cubes iteration to scale near-linearly with the amount of threads used. In this approach edges are sampled rather than voxels, and the triangle cases are iteratively assembled. The benefit of edges rather than voxels as a scanning primitive is that only one value lookup or sampling occurs for the whole domain, leading to better efficiency and multithreaded scaling.

4 DistMesh

The DistMesh algorithm was first described in Per-Olof Persson’s Thesis in 2004. The algorithm is designed to produce high-quality meshes on implicit and signed distance functions. The benefit of signed distance functions in meshing is that point membership within the solid may be computed at any point. This is highly useful for refinement strategies where points move within the domain. Persson’s DistMesh algorithm is iterative and models the mesh as a spring system to achieve equilibrium. In Figure 9 the basic process of the iterative algorithm can be seen. The first step is the distribution of points within the space of the signed distance function. This can follow any arbitrary method, such as a uniform spacing,
packed spacing, or random spacing. Once an initial point distribution is established, a delaunay triangulation is computed. These simplices are then decomposed to edges to solve for force equilibrium in an iterative fashion. Between force equilibrium iterations, the original tetrahedra often no longer meet a delaunay criteria (that is, no other point is within the circumsphere of the tetrahedra), so the algorithm re-runs triangulation as necessary. This process leads to force equilibrium on each edge, yielding a high quality mesh.

Figure 9: Visual process of the DistMesh algorithm and quality metrics

The initial thesis on this topic is very broad in nature, and describe a variety of applications such as Level-Set methods, moving boundary methods, and adaptive meshing. The overview and implementation of the algorithm is a roughly 100-line Matlab code. At the outset of the project, the DistMesh algorithm was ported to the Julia programming language, with tetrahedral delaunay triangulation done in TetGen.\cite{19} Matlab by default has many one-line utilities for geometric computation and array processing. In order to replicate the work sufficiently, these had to be un-wrapped into Julia implementations and tested for consistency. The DistMesh.jl repository on GitHub contains several tests and benchmarking tools used throughout this study.

After the algorithm was working in Julia, this afforded the opportunity for additional experimentation. One of the initial areas of concern was performance to allow for rapid experimentation. Similarly, we worked to use Signed Distance Functions generated with Descartes with the DistMesh package and perform validations of the signed distance functions. The second area of development was the study of the actual algorithm across a range of parameters. Finally, alternative mechanisms for retriangulation and termination criteria were implemented.
### Parameters

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Matlab (mean runtime)</th>
<th>Julia (mean runtime)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sphere, deltat=0.1, ttol=0.02</td>
<td>18.010 sec (n=7)</td>
<td>0.298 sec (n=17)</td>
</tr>
<tr>
<td>Sphere, deltat=0.1, ttol=0.05</td>
<td>12.330 sec (n=7)</td>
<td>4.003 sec (n=7)</td>
</tr>
<tr>
<td>Sphere, deltat=0.05, ttol=0.02</td>
<td>11.781 sec (n=7)</td>
<td>0.723 sec (n=7)</td>
</tr>
<tr>
<td>Sphere, deltat=0.05, ttol=0.05</td>
<td>11.509 sec (n=7)</td>
<td>138.156 sec (n=1)</td>
</tr>
</tbody>
</table>

Table 3: Runtime comparisons between DistMesh reference implementation in Matlab and this paper’s Julia implementation.

### 4.1 Memory Management and Vectorization

One of the challenges of any type of numerical algorithm is the management of memory. Julia is a garbage collected language, so the extent to which one can manage memory is limited. However, it is possible to use memory more effectively to a great advantage. The first general improvement to the DistMesh algorithm was pre-allocation of memory. In this case, the primary areas of concern are large vectors that contain edges, points, triangles, displacements, and other statistics between each iteration. It is important to note that these gains can be had in the Matlab implementation as well, as it is also a garbage collected language.

Similarly, we worked to vectorize certain operations in the code and store data such that vectorized was generated by the compiler. Vectorization in this case refers to Single Instruction- Multiple Data, or SIMD commands. With the Intel AVX2 instruction set, up to four 64 bit floating point numbers may be operated on at once. For these reasons, points and edges are stored as static vectors and tuples in a heap array which ensures they are contiguous in memory and do not require pointer dereferencing. The Matlab implementation accomplishes something similar by storing each axis in a column of an array. The use of these data structures also facilitates SIMD vectorization of point displacement, sorting, and distance function evaluation.

In order to validate these claims, the reference Matlab and Julia port were compared on a simple sphere of radius 1 and tetrahedral edge length of 0.2 with identical parameters for the algorithm. The results of this trial is shown in Table 3.

In all cases the algorithms generated identical points, however the tetrahedral counts were 2693 for Matlab and 2684 for the Julia implementation. This discrepancy still needs to be investigated further. There are several underlying differences in the delaunay triangulation algorithms that may make this the case. Matlab uses QHULL for delaunay whereas the Julia port uses TetGen. Given equal point distributions in a sphere QHULL and Tetgen give different results. We note that our algorithm generally has better performance and similar results given lower values for ttol.
4.2 Hilbert Sorting

Data ordering is a key aspect to achieve good performance on large data sets. One technique the reference DistMesh algorithm uses is the sorting of edge-tuples to ensure that the point lookup is roughly sequential in memory. However, upon tests it was noted that the points tend to move between each iteration, and initially close point indices will grow farther apart. For example and edge sequence such as: (1,2), (1,4), (2,3) will become: (1,80), (1,30), (2,20) after several retriangulations. Therefore the vector of points needs to be reordered to improve cache locality.

One simple approach is to perform a linear sort. However, in practice this is sub optimal. Given a NxNxN grid, and a tetrahedra spanning a single voxel in this grid, linear sort will lead to some indices being N or NxN units apart. Spatial sorting addresses this by effectively binning the points with their nearest neighbors. A good implementation can have similar performance to a linear sort and improve lookup times. For this project, KD Trees and Hilbert sort were tested. The two methods primarily differ in their structure. KD Trees may be non-uniform box sizes, whereas a hilbert sort has uniform box sizes. The key difference here is that KD Trees will subdivide a space such that each box (or region) hold N items, whereas hilbert sort has no limitations on the amount of points within a region.[20]

Initial experimentation in spatial sorting used KD-Trees to sort these points, however this proved somewhat expensive as these allocate large amounts of memory in the current Julia implementation. However, in certain cases there were performance improvements. Generally, sorting along a Hilbert curve is both efficient and improves performance for large geometries.[21] For smaller geometries, sorting does not have an improvement as the data structures can fit entirely into a CPU cache, and performance regresses due to the sorting overhead. However, for large tetrahedra counts, this may yield a performance improvement. In addition, TetGen was run in non-random, sequential point insertion mode based on the hilbert sort, however performance is greatly variable based on the genus (hole count) and aspect ratio of the geometry. By default, Tetgen uses a random insertion for delaunay construction, which has been shown to have good average performance.[19] Similarly, modified KDTree and Hilbert Sort using biased randomization have been shown to also improve performance, which suggests the Hilbert Sort implementation must be coupled closer to the delaunay triangulation process to see any benefit in this step.[22][23]

Between a single iteration the points tend to not diverge greatly, therefore it is not necessary to run the hilbert sort for each retriangulation. Both a "sort" and "sort_interval" parameter in the DistMeshSetup struct to allow the user to configure if sorting should be enabled, and the frequency at which it occurs. For comparison, a sphere of radius 1 and meshing edge length of 0.15 went from around a 9 second runtime to 1.5 second runtime with hilbert sorting. Similar results have been seen for other large geometries, however the sorting interval must be tuned for each case. This feature is experimental until better heuristics can
be created to determine the optimal sorting interval and further tests run.

4.3 Nonlinear Edge Displacements

One issue in the default DistMesh algorithm is that all edges are treated with equal weight. Persson mentioned in his thesis that this may be solved with a nonlinear equation to apply a greater force when an edge has a greater deviation from the weighted $L_p$ norm of all edge current lengths. Below is the basic equation of the force in each edge.

$$f(l, l_0) = \begin{cases} k(l_0 - l) & l < l_0 \\ 0 & l \geq l_0 \end{cases}$$  \hspace{1cm} (1)

By default, $k = 1$ is chosen for a linear edge displacement. However, a non-linear formulation for the spring constant, $k$, may be formulated as follows:

$$k = \frac{l + l_0}{2l_0}$$  \hspace{1cm} (2)

Figure 10: Meshing on sharp corner. Left: linear force, Right: nonlinear force

The option for using this nonlinear calculation has been added to the DistMesh.jl implementation by setting “nonlinear=true” in the DistMeshSetup struct. This can have a great effect on areas where sliver triangles may be formed, as shown in Figure 10 where a cylinder cuts an edge of a cube. In this example both are missing tetrahedra as the centroids are
near the boundary. Adaptive refinement can greatly help produce a better surface. However, the default linear case leads to a sliver tetrahedra, whereas the nonlinear solution produces well-formed triangles. Figure [11] DistMesh is run on a torus with nonlinear and linear forces respectively, where red vertical lines indicate a retriangulation. We can see that the nonlinear version requires more iterations 25% than the linear version. What is most important to note is that the non-linear version has a smoother convergence on median and average qualities.

![Quality metrics between iterations, Top: nonlinear, Bottom: linear. Vertical red lines indicate retriangulations.](image)

Figure 11: Quality metrics between iterations, Top: nonlinear, Bottom: linear. Vertical red lines indicate retriangulations.
4.4 Distance Caching

Figure 12 shows example profiling results of the DistMesh algorithms in graph form. In these examples (a sphere), roughly 50% of the computation time is spent in evaluating the signed distance function. Unfortunately, a sphere is one of the more simple distance functions that occur in practice. To reduce the number of samples performed, a distance cache was implemented that stores the point distances to minimize the amount of resampling of the signed distance function between iterations. Particularly, this minimizes the amount of check required when determining the point membership in/out of the boundary. The mechanism is roughly as follows:

1. Compute signed distance corresponding to a point and store in array
2. After applying displacement, subtract the euclidean distance of the point movement from the signed distance cache value and store.
3. If this distance is greater or equal to zero, resample the distance function.
4. If point needs to be projected to the boundary, set the point distance to 0.

This caching helps performance in almost all cases, and requires little additional storage and computation and, is therefore enabled by default.

Figure 12: Runtime profiling graph of DistMesh. 49% is spent in core routine and 49% in function sampling.
4.5 Analysis of Parameter Selection

There are three key parameters in the original DistMesh algorithm:

- Time Delta
- Displacement Tolerance
- Movement Tolerance

As these parameters are discussed, it is important to remember that each edge in the tetrahedral mesh behaves as a spring. The objective of selecting these parameters is to converge the mesh to a minimal energy state such that all edges are in equilibrium. That is to say, at the next iteration the movement of points is sufficiently small from the forces applied by each edge (spring) on the point.

*Time Delta* (*deltat*) is the amount of force applied by each edge at a given time segment. This ensures that points gradually move to an equilibrium point and such that delaunay properties may be preserved between each iteration.

*Displacement tolerance* (*ptol*) is the maximum amount of displacement induced by a single edge in a given iteration. In the default algorithm if this number is sufficiently small the algorithm will terminate.

*Movement tolerance* (*ttol*) is the maximum allowable movement by a point before retriangulation is required.

These three parameters; deltat, ptol, and ttol allow the DistMesh algorithm to be tuned. Generally a smaller ttol value increases retriangulation and a smaller deltat will create more force iterations (exclusive of triangulations). The ttol parameter is generally a good default at 0.001 as it is scaled by the edge length. Large values will lead to early termination but lower mesh quality, and smaller values will lead to higher mesh quality but longer runtime.

4.6 Output Quality Analysis

The DistMesh algorithm promises high quality mesh output. One of the key techniques for validating and analyzing mesh quality is the dihedral angles of the tetrahedra. Similarly, the minimal dihedral angle in each tetrahedron is also a relevant statistic. The DistMesh implementation in Julia support these metrics and can easily generate histograms. A regular tetrahedron will have dihedral angles of 70.5288° or 0.3918π (1.231) radians. The objective of tetrahedral refinement strategies is to develop a mesh such that the distribution of dihedral angles is as close to a regular tetrahedron as possible for good integration properties.

Figure 13 shows results for dihedral distributions in radians on a sphere comparing linear and nonlinear methods (radius 1, edge length 0.2).
4.7 Quality-Based Retriangulation and Termination

In Per-Olof Persson’s Thesis the idea of quality-based termination was presented. In contrast to the original algorithm, a quality-based termination scheme will continuously analyze the mesh after each iteration to determine if a minimal quality threshold has been achieved. However, it is possible to extend this to retriangulation as well. In this method, the average quality should increase monotonically and termination will be dependent upon the minimal triangle quality. Throughout the project the foundation for this mechanism have been developed with quality metric algorithms use to generate datasets and analyze parameters. The tools to implement this have been used to study the quality of the mesh and algorithm behavior over time. This is an active area of investigation and will be continued into the future. However, it is noted that extracting quality information from the mesh is somewhat costly and requires additional memory and time. One lower-cost and alternate approach tested is “move averaging”.

Figure 13: Minimum and Total Dihedral angle distributions on a sphere with linear and nonlinear forces. Note nonlinear has more regular tetrahedra and a greater minimal dihedral angle.
The core idea of move averaging is to increase the frequency of retriangulation and improve the “smoothness” of quality convergence. In this approach N prior maximum movement values after a retriangulation are stored and compared to the latest movement value. If the current movement is greater than the prior, a retriangulation will be triggered. In this process the Movement Tolerance (ttol) is replaced with a parameter for N iterations to average. We can see in Figure 14, a test of this with N=7 for the retriangulation criteria. In this case, retriangulation occurs more frequently and the average and median quality improvement at each iteration is near monotonic. Similarly, the move averaged version terminates in less than half the iterations. This indicates that the DistMesh algorithm can be more successful if the conditions for retriangulation are better understood.

Figure 14: Top: Default, Bottom: Move Averaged. Red lines indicate retriangulation.
5 Conclusion

In this project the principles of solid modeling, surface meshing, and tetrahedral meshing using signed distance functions were discussed. The primary contribution is the analysis and implementation in Julia of the DistMesh routine first documented by Per-Olof Persson. The three libraries developed and improved for this project are publically available as open source and corresponding documentation has been developed to increase usability.

The DistMesh algorithm has shown promise as a method for improving mesh quality from signed distance functions. In this paper several analyses and alternative approaches have been presented to improve performance and robustness of the algorithm. These include improved memory structures, nonlinear force, quality-based retriangulation, move average retriangulation, and sorting techniques. These developments are available in the mainline branch of the DistMesh repository, however a few still need to be integrated into user-facing API and unit tested in an effective manner. In the future, the features discussed here will be tested further and validated across a broader range of geometries. Overall, the Julia implementation of DistMesh supports a greater range of features that should allow for additional experimentation and support for large node meshes.

Similarly, Descartes, DistMesh, and Meshing form a basis for a design and simulation ecosystem. There is further work on integrations with FEA tools and visualization which must be completed to provide a good user experience, and "close the loop" with simulation studies. However the performance and design of these sub-components has been analyzed to provide a good basis for performance and interaction. This work is freely available on GitHub as open source and has already attracted additional collaborators and users.
References


Appendices

A  Meshing.jl

```julia
using FileIO
using NRRD
using Meshing
using MeshIO
using GeometryTypes

# load the file as an AxisArray
cardio = load("CTA-cardio.nrrd")

# use marching cubes with isolevel at 100
algo = MarchingCubes(iso=100, insidepositive=true)

# generate the mesh using marching cubes
mc = HomogenousMesh(Point(3, Float64), Face(3, Int))(cardio, algo)

# save the file as a PLY-file (change extension to save as STL, OBJ, OFF)
save("cardio_mc.ply", mc)
```
using Meshing
using MAT
using MeshIO
using FileIO
using GeometryTypes

# load microgripper (variable "phi") from mat file
microgripper = matread("microgripper.mat")["phi"]

# extract isosurface
mesh = GtNormals(microgripper, MarchingCubes(insidepositive=true))

# save as STL
save("microgripper.stl", mesh)
B  Descartes.jl

```julia
using Descartes

function beam(beam_size = [50,10,10],
               hole_ct = 5,
               hole_d = 3)

  # computations
  hole_interval = beam_size[1]/(hole_ct + 1)
  c = Square(beam_size)
  for i = 1:hole_ct
    h = translate((hole_interval*i, beam_size[2])/2)Circle(hole_d/2) ←
    c = diff(c, h) ←
  end
  linearExtrude(c, beam_size[3]) ←
end
m = HomogenousMesh(beam())
#save("ld_beam.ply", m)
```

- Define Square boundary (50,10)
- Transforms are associative like matrix
- Difference operation
- Extrude
- Mesh (sample bounds automatically determined)

```julia
using Descartes
c = Cuboid([5,5,5])
h = translate([2.5,2.5,0])Cylinder(1.5)
obj = diff(c,h)
m = HomogenousMesh(obj)
save("cube_with_hole.stl", m)
```

```julia
using Descartes

m = HomogenousMesh( diff( Cuboid([5,5,5]),
translate([2.5,2.5,0])Cylinder(1.5))
save("cube_with_hole.stl", m)
```
using Descartes

# params
beam_size = [50, 10, 10]
hole_ct = 5
hole_d = 3

# computations
hole_interval = beam_size[1] / (hole_ct + 1)

c = Cuboid(beam_size)

for i = 1:hole_ct
    h = translate(hole_interval*i, -1, beam_size[3]/2)
        rotate(-pi/2, [1,0,0])
        Cylinder(hole_d/2, beam_size[2]/2, center=False)
    global c = diff(c, h)
end

save("fea.ply", HomogenousMesh(c))