# Computational Methods for Parametrization of Polytopes

Steve Kelly

March 26, 2016

# Contents

**Abstract**

Geometry is fundamental to the development of natural problem statements. Opportunities exist in computational geometry to add rigourous type descriptions and algorithmic relations that do not sacrifice performance.

# Chapter 1

# Introduction

Our objective is to develop a computational environment for the exploration of parametric polytopes. A computational environment is one in which we can apply rigourous defintional constraints on symbolic constructions, and likewise manipulate them to reveal properties that may be of interest. A parametric polytope is the union of two concepts. The first is the idea of parameters, which are our unknown constraints in a system. A polytope is a geometric object that has "flat" sides. The purpose of this paper is to illucidate the prior work in computational representations of polytopes and develop this work further.

## 1.1 Motivation

## 1.2 Prior Work

develop as body progresses

# Chapter 2

# Mathematical Definitions

In this section we will develop a mathematical perspective of our problem such that we can focus clearly on the computational aspects in the following sections. We are interested in parameterizing polytopes for mathematical exploration. A polytope is geometrically realizable graph composed of flat faces. Using the operations of union, intersection, and difference we may say that a polytope of dimensionality N may be constructed using hyperplanes of dimensionality N-1. However the graph and hyperplane constructions are somewhat distinct, and we will begin to see these representations side by side as we move to a type framework. For our purposes we will be concerned with polytopes which form closed solids.

To begin, we will outline how solids may be constructed using hyperplanes.

## 2.1 Closed Spaces

A "closed" space means a subset of something that contains all points in it's boundary, including the bondary. An "open" set conversely will not include the boundary, but everything inside. We sometimes use brackets to aid the representation of sets on a number line for example: (1,2) for an open set and [1,2] for a closed set. The quantity of the objects contained in our sets will vary depending on the numeric domain we choose. For example if we are using integers we have no elements in the open set and 2 in the closed set. If it is in rationals we have 1/2 included, and in the reals we have infinite points! What we have constructed on the number line can also be thought of as open and closed intervals.

In multiple dimensions this is more interesting since we may construct various, rather arbitrary, geometries to make a closed space. One common example is a hyperplane. A hyperplane is simply a generalization of the a plane into arbitrary dimensions, with the property it is of dimensionality N-1. For example if we are in 2D space, our hyperplane is a line since it partitions our space into two parts. Likewise in 3D space this is a plane. If we define a hyperplane functionally using vector notation we can extract some interesting properties. For simplicity in this example let us assume we have a hyperplane which cuts through the origin. If we let $\vec{x}$ be an arbitrary point in $N$ dimensional space, and $\vec{a}$ be the slopes of each axis, then one functional construction is simply the

dot product, $dot(\vec{a}, \vec{x})$. If x is on the hyperplane the function will be equal to zero. If it is not zero, then we may determine which side of the partition the point lies on. We notice that the hyperplane cuts space, but does not create a closed subspace. Likewise the dot product definition gives us a measure of area and we might rather prefer the shortest distance between the point and the hyperplane. What we are after is what is known as a solid, and we would like to do such using functions that return information useful for computation.

## 2.2 Solids and Orientation

Before we define a polyhedra, we must introduce a few notions. These are solids and orientation. Solids have been studied since antiquity and for our purposes we will define them as constructions in three dimensional space with finite volume. For example, a plane which partitions space is not a solid since either partition is unbounded, however the intersection of planes could form a solid.

Orientation is a parallel concept which allows us to specify how geometric objects contain space. As an example, let us go back to our partition of space with a plane. If we are on our way to construct a solid, it is neccessary to choose the one part to keep and the other to discard. This is the purpose of orientation. In the case of the plane, this follows from the definition, $ax + by + cz + d = 0$. More lucidly, lets look at the signed distance of a point from the plane, computed as:

$$D = \frac{ax + by + cz + d}{\sqrt{a^2 + b^2 + c^2}} \tag{2.1}$$

For simplicity, let's look at a plane parallel to the X and Y axes passing through z = 1. Thus $a$ and $b$ are set to 0, $c$ set to 1, and $d = -1$ Simplifying our formulation we have:

$$\text{D} = 0\text{x}+0\text{y}+\text{z}-1 = \text{z}-1 \tag{2.2}$$

At $z = 1$ we see we are on the plane, however at $z = 0$ and $z = 2$ we get -1 and 1 respectively. The sign of the distance is our indicator of orientation. We can choose an arbitrary convention as to which partition we will count, but akin to the "right hand rule" in physics, the normals of the partition must point outwards. In the case of the plane the normal is the vector $(a, b, c)$, which in our realization is the upwards vector $(0, 0, 1)$. Since the convention is such that the normals point outward, the partition we would consider in a solid is all points *in the opposite direction of the normal*. Also to note is the importance of sign in our distance function. We can exploit this behavior to indicate containment when performing logical operations on spatial partitions. We have chosen this convention for this paper due to it's ubiquity in computation frameworks.

## 2.3 Distance Field Representations of Solids

Implicit functional representation (FRep) in computation centers around a signed, real-value function where the boundary is defined as $f(...) = 0$. In $\mathbb{R}3$ this looks

like $f(x, y, z) = 0$. For modeling purposes we must add the additional constraint that the function evaluates to a negative inside the boundary. Further more the magnitude of the return value must correspond to the minimum distance between the point and the boundary.[1] A sketch of this behavior in one dimension can be seen in Figure 2.1.

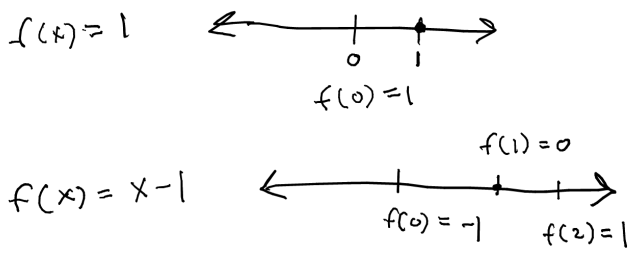


Figure 2.1: Number line illustrating the construction of an implicit signed function

Researchers at MIT have taken these principles to constrain geometric features to ensure a part can be manufactured.[2] They call their system "FabForms" and shows how functional representations can easily accept constraints.

More importantly, functional representations can naturally deal with affine transforms. [3] Given some transform associated with a FRep, one simply applies the inverse transform to check membership.

## 2.4 Logical operations on Distance Fields

One can also compose functional representations with set operations. Below are basic set operations defined for these functions:

$$\cap : \texttt{min}(f_1, f_2)$$
$$\cup : \texttt{max}(f_1, f_2)$$
$$\neg : -\texttt{f}_1$$

It follows that the "difference" of $f_1$ and $f_2$ is the intersection of $f_1$ with the negation of $f_2$, $\texttt{max}(f_1, -f_2)$. The mathematical analyst might have trouble with these formulations because such operations create discontinuities.

### 2.4.1 Rvachev Functions

In the 1960's Vladimir Rvachev produced a method for handling the "inverse problem of analytic geometry". His theory consists of functions which provide a link between logical and set operations in geometric modeling and analytic geometry.[4] While attempting to solve boundary value problems, Rvachev formulated an equation of a square as

$$a^2 + b^2 - x^2 - y^2 + \sqrt{(a^2 - x^2)^2 + (b^2 - y^2)^2} = 0$$

Implicitly, the sides of a square can be defined as $x = +/-a$ and $y = +/-b$. The union of these two is a square. By reducing the formulation of the square we can generalize an expression for the union between two functions.

$$\cup : f_1 + f_2 + \sqrt{f_1^2 + f_2^2} = 0$$

Likewise we can see that intersections and negations can be formed for logical completion.

$$\cap : f_1 + f_2 - \sqrt{f_1^2 + f_2^2} = 0$$

$$\neg : -f_1$$

These formulations can be modified for $C^m$ continuity for any $m$. [5] In addition Pasko, et. al. have shown that Rvachev functions can serve to replace a geometry kernel by creating logical predicates. [6] Their research also establishes the grounds for user interfaces and environment description. For this work a practical implementation will most likely leverage their insights. Rvachev and Shapiro have also shown that using the POLE-PLAST and SAGE systems a user can generate complex semi-analytic geometry as well.[7]

show this construction, it isn't obvious

While a functional representation for geometry is mathematically enticing on its own, the power it gives for numerical analysis might be its greatest virtue. Numerical analysis justified the initial investigation by Rvachev early on. A boundary value problem on a R-Function-predicate domain allows for analysis without construction of a discrete mesh.[7]

One of the most general expositions in the English language of R-Functions applied to BVPs is Vadim Shapiro's"Semi-Analytic Geometry with R-Functions". [5] Unfortunately, no monographs about R-Functions exist in the English literature. Most literature is in Russian, however many articles presenting applied problems using the R-Function Method. [8]

Such a system for analytic geometry can be developed further. In the context of an Eulerian flow field, a distance field over a function that generates partial derivatives could be a fast numerical computation method.

### 2.4.2 Signed Distance Fields

rewrite to be salient

A signed distance field (SDF) is a uniform sampling of an implicit function, or any oriented geometry. Below we can see this in action over the definition of a circle.

didn't introduce orientation, winding order, etc...

```
julia> f(x,y) = sqrt(x^2+y^2) - 1
f (generic function with 1 method)

julia> v = Array{Float64,2}(5,5) # construct a 2D 5x5 array of Float64

julia> for x = 0:4, y = 0:4
           v[x+1,y+1] = f(x,y)
       end

julia> v
5x5 Array{Float64,2}:
 -1.0  0.0       1.0      2.0      3.0
  0.0  0.414214  1.23607  2.16228  3.12311
  1.0  1.23607   1.82843  2.60555  3.47214
  2.0  2.16228   2.60555  3.24264  4.0
  3.0  3.12311   3.47214  4.0      4.65685
```

The results of `v` might be confusing since the matrix is oriented with the origin in the top left corner. At coordinate $(0, 0)$, or entry `v[1,1]`, we see that `f` is equal to `-1`. Likewise we can see $(0, 1)$ and $(1, 0)$ are points on the boundary since the value is `0` and everywhere else is positive.

Distance fields are interesting since they provide an intermediate representation between functional space and discrete-geometric space. However they are a very memory hungry data structure. Pixar has published OpenVDB which helps work around these concerns, but such compression can be lossy.[9] With the advent of shader pipelines for GPUs, distance fields have become more popular. Valve has used SDFs with great success for generating smooth text.text renders. [10] Many algorithms for generating polyhedra from an SDF exist. The most common are Marching Tetrahedra, Marching Cubes, and Dual Contours.[11][12][13]

Talk about vert and frag shaders.

Andreas Bærentzen and Henrik Aanæs published methods on the inverse problem of converting a mesh to a signed distance fields.[14] DiFi was introduced in 2004, which demonstrates an algorithm for creating SDFs on multiple types of geometry [15].

Need to re-read this paper

Many necessary algorithms in path planning for digital manufacturing tools fall out of distance fields. For example, offsetting simply becomes an addition or subtraction over the SDF. Computing the medial axis becomes a scan for inflection points. Many path planners need to simplify polygon representations as to not generate move less than the resolution of the machine. Assuming the machine uses a Cartesian system, a SDF can correspond perfectly to the lowest available resolution of the machine. Likewise as Stereolithographic 3D printers begin to use digital mirror devices (commonly known as DLP or DMD) , discrete representations of geometry will become more important in digital manufacturing.

[16]

## 2.5 Simplices

Recently a Simplex type was added to GeometryTypes. A Simplex is defined as the minimum convex set containing the specified points. The initial prototype is very simple yet works well. Below we can see a 0th and 1st order simplex constructed in $\mathbb{R}2$ and $\mathbb{R}3$

```
julia> using GeometryTypes

julia> Simplex(Point(1,2))
GeometryTypes.Simplex{1,FixedSizeArrays.Point{2,Int64}}((FixedSizeArrays.Point{2,Int64}((1,2)),))

julia> Simplex(Point(1,2,3), Point(4,5,6))
GeometryTypes.Simplex{2,FixedSizeArrays.Point{3,Int64}}((FixedSizeArrays.Point{3,Int64}((1,2,3)),FixedSize
```

This representation makes it possible to write code based on the order and dimensionality of a simplex. Few algorithms have been developed around the new Simplex type, and unfortunately it is not integrated as a lower-level construct for the other types yet.

In addition I would like to add a Simplical Complex type.[1]

[1] `https://en.wikipedia.org/wiki/Simplicial_complex`

## 2.6 Polytope

### 2.6.1 Descrete Form

### 2.6.2 Continuous Form

# Chapter 3

# Computational Definitions and Grammar

Needs massive refactor and redux

Programming languages are the grammar and syntax a computer presents to a user. This project is fundamentally exploratory in nature and seeks to generate understanding of geometric relationships using the intersection of mathematical and computational rigor. We have chosen to use the Julia programming language due to comfort of development, and an abundance of supporting libraries for mathematical computation. In this chapter we will give a brief introduction to many computing concepts and illustrate how Julia advances them to meet our needs well.

## 3.1 History

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 code specialization.[17] [18] 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. The syntactical style is similar to MATLAB and Python. The language implementation and many libraries are available under the permissive MIT license.[1]

Benchmarks have shown the language can consistently perform within a factor of two of native C and FORTRAN code.[2] 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". The problem is encountered when abstraction in a high-level language will disproportionately affect performance unless implemented in a low-level language. In the next sections we will compare the expressibility and performance to other languages.

[1] `http://opensource.org/licenses/MIT`
[2] `http://julialang.org/benchmarks`

## 3.2 Comparisons

Many languages are as fast as Julia but sacrifice expressibility. In Figure 3.1 we can see some comparisons to other programming languages. This was developed by the Julia core team, and illustrates that Julia is highly competitive in performance. Again, these results stem from the compiler and language design. In Figure 3.2 we can see these results normalized against code length. The Julia code is quite short, yet consistently achieves good performance. Much of this comes down to the innovated type and function system.[19] We will discuss these more in depth later.

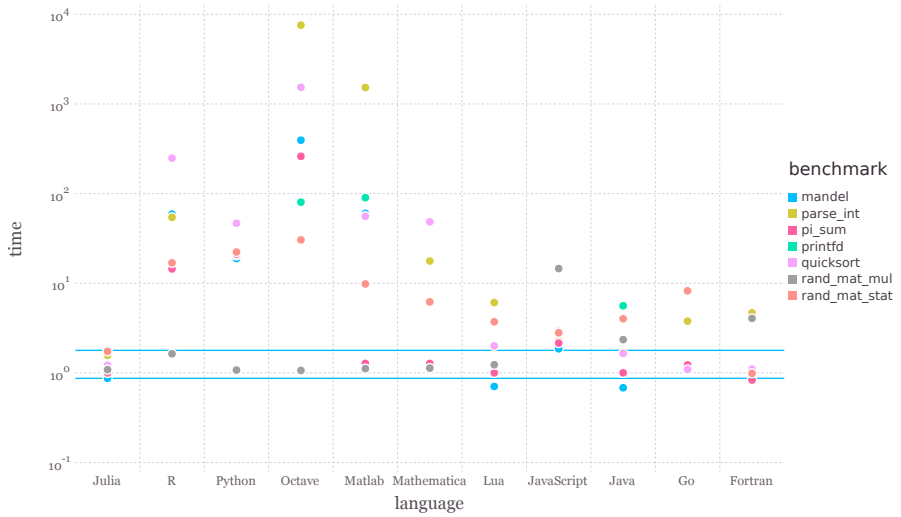


Figure 3.1: A comparison of programming languages and performance.

In 1972 Alan Kay introduced the terms "class" and "object" to describe a coupling of data and functionality.[3] An object is an instance of a class, which contains the definitions of functions and member data. Computer Scientists call this "Object Oriented Programming" (OOP). Languages such as C++, Java, and Python all subscribe to this paradigm. In Python this looks like the following:

```
class Foo:
    foo1
    foo2
    def add_to_foo1(self, x):
        self.foo1 += x
```

This system positively enables specialization of functionality, but due to the coupling of data with functions it becomes a challenge to extend functionality. Languages for scientific computing generally avoid the "traditional" notions of OOP. In Table 3.3 we can see a comparison of type systems used in scientific computing languages. Here "Type system" can be either dynamic or static, which indicates if the programmer needs to specify to the program compiler

[3] http://gagne.homedns.org/~tgagne/contrib/EarlyHistoryST.html

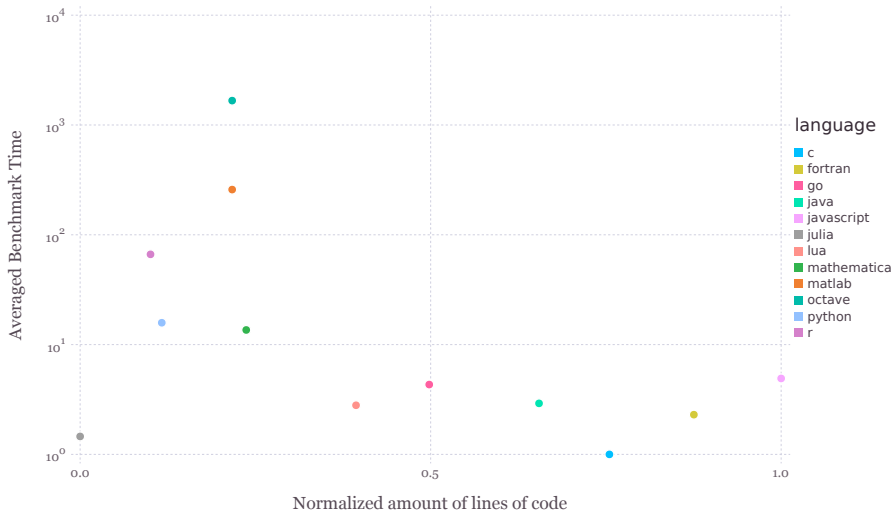


Figure 3.2: The results in Figure 3.1 normalized for code length. (Courtesy of Simon Danish)

how data is transformed in a function. Generic functions allow a single function name, for example "sum", to have multiple defintions with execution contigent upon the matching of argument types. The definition of a parametric type is more nuanced, but generally means that the definition of a type may vary based on the types of it's member data. In the next few sections these ideas will hopefully be clarified and the implications of multiple dispatch and the relation to OOP will be developed further.

Figure 3.3: A comparison of functions, typing, and dispatch.

| Language | Type system | Generic functions | Parametric types |
|---|---|---|---|
| Julia | dynamic | default | yes |
| Common Lisp | dynamic | opt-in | yes (but no dispatch) |
| Dylan | dynamic | default | partial (no dispatch) |
| Fortress | static | default | yes |

## 3.3 Functions

Julia is an experiment in language design. Much of the advancement revolves around the representation of data and the execution of functions. The language is optionally typed, which means function specialization on types is inferred by the compiler without user intervention. This is an idea first utilized in the Hadley Milner's "ML" which was created to develop theorem provers.[20]. A basic example of inference in Julia is shown below:

```
julia> increment(x) = x + 1
increment (generic function with 1 method)
```

```
julia> increment(1)
2

julia> increment(1.0)
2.0
```

[4] The `increment` function was defined for any `x` value. When the `1`, an integer type was passed as an argument, an integer was returned. Likewise when a floating point, `1.0` was passed, the floating point `2.0` was returned.

Let's see what happens when we try a string:

```
julia> increment("a")
ERROR: MethodError: `+` has no method matching +(::ASCIIString, ::Int64)
Closest candidates are:
  +(::Any, ::Any, ::Any, ::Any...)
  +(::Int64, ::Int64)
  +(::Complex{Bool}, ::Real)
  ...
 in increment at none:1
```

The problem is that the `+` function is not implemented between the `ASCIIString` and `Int64` types. We need to either implement a `+` function which might be ambiguous, or specialize the function for `ASCIIString`. A specific implementation is preferrable in this case:

```
julia> function increment(x::ASCIIString)
           ASCIIString([increment(c) for c in x])
       end
increment (generic function with 2 methods)
```

The line `x::ASCIIString` is called a "type annotation" and states that `x` must be a subtype of `ASCIIString`. This allows one to control dispatch of functions, since Julia will default to the *most specific implementation*. Since`ASCIIString` is a series of 8 bit characters, we can iterate over the string and increment each character individually. The `[]` indicates we are constructing an array of characters to pass to be passed to the `ASCIIString` type constructor. Now we see our example works:

```
julia> increment("abc")
"bcd"
```

What was demonstrated here is the concepts of specialization and multiple dispatch, both are highly coupled topics. Each function call in Julia is specialized for types if possible. This means the author only has to write a few sufficently abstract implementations of functions. If special cases occur multiple functions with different arity or type signatures can be implmented. Explicitly this is called multiple dispatch. In practice by the user this looks like abstracted or generic code. To the computer, this means choosing the most specific, and thus performant method. Let's go back to the integer and floating point example. Below is the LLVM assembly generated for each method:

```
julia> @code_llvm increment(1)

define i64 @julia_increment_21458(i64) { // <return type> <function name>(<arg type>)
top:
  %1 = add i64 %0, 1
  ret i64 %1 // return <return type> <return id>
}
```

[4] The REPL (Read-Eval-Print-Loop) allows interactive evaluation of Julia code. It is highly useful for exploration and testing of ideas in the language. Blocks starting with "`julia>`" represent input and the preceding line represents output of the evaluated line.

```
julia> @code_llvm increment(1.0)

define double @julia_increment_21466(double) {
top:
  %1 = fadd double %0, 1.000000e+00
  ret double %1
}
```

Note I have annotated the LLVM code so this is understandable. The only real similarity is the line count. Each one of these functions are generated by the Julia compiler at run time.

Many of the concepts used for performance also serve as methods for expressability. In this case, multiple dispatch used by the compiler for specialization of functions reveals it self as a way for the user to specialize over many types. Revealing the role in which this paradigm allows Julia to achieve high performance is a matter to be developed in further sections.

## 3.4 Types

### 3.4.1 Mutability and Data Packing

Types and immutables are containers of data. The primary difference between the two is the notion of "mutability". Types are mutabile, immutables are immutable. What does this mean? Let's break something first:

```
julia> type FooIsMutable
           a
       end

julia> f = FooIsMutable(1)
FooIsMutable(1)

julia> f.a
1

julia> f.a = 2
2

julia> f.a
2

julia> immutable FooIsImmutable
           a
       end

julia> f = FooIsImmutable(1)
FooIsImmutable(1)

julia> f.a
1

julia> f.a = 2
ERROR: type FooIsImmutable is immutable
```

What just happened demonstrates the contract defined by mutability. Mutable objects, which is an instance of a type (i.e. `f`), can have their fields (i.e. `a`) changed. Immutables cannot. The immutable contract helps develop a notion of functional purity. To the user this means immutables are defined by their values. Practically this can be of great benefit to the compiler. For example:

```
julia> a = (1,2,3)
(1,2,3)

julia> b = typeof(a)
```

```
Tuple{Int64,Int64,Int64}

julia> isbits(b)
true

julia> a = ([1],[2],[3])
([1],[2],[3])

julia> b = typeof(a)
Tuple{Array{Int64,1},Array{Int64,1},Array{Int64,1}}

julia> isbits(b)
false
```

`isbits` ask the question "will this type be tightly packed in memory"? A `Tuple` is a fixed-length set of linear, ordered, data. It has syntax for construction with `()`. In computations we want our data be close together for fast access. In modern times we call such data "cache friendly", or "cache localized". Immutability helps us achieve this. Let's look that the types inside the 3-tuples and see their `isbits` status:

```
julia> isbits(Array{Int64,1})
false

julia> isbits(Int64)
true
```

Why is this the case? We see that `Int64` is bits, because it is literally 64 bits. In Julia a `bitstype` behaves similar to an immutable, and is identified by value. `Array{Int,64}` is a mutable data type that can vary in size. This means the `Tuple` needs to store the arrays as references, in this case a pointer. When iterating over a data set, such a "pointer dereferences" (this is jargon for accessing the data in memory pointed to by a pointer), can be costly. Modern CPUs accell when data is linearly packed and pointer-free. The data can be brought into the CPU's memory cache once and computed without shuffling between cache and RAM.

### 3.4.2 Parameters

need to demonstrate why this is HUGELY important for performance and expression

## 3.5 Macros and Generated Functions

Julia is a descendant of the Lisp family of programming languages. Lisp is a portmanteau for "List Processing". The language was designed to address the new notion of "types", specifically in application to Artificial Intelligence (AI) problems.[21] The notion of an "S-Expression" was introduced in McCarthy's seminal work. These statements use parenthesis to denote functions and arguments. Below is an an example of S-Expressions for addition and multiplication.

```
> (+ 1 1)
2

> (* 3 4)
12
```

This syntax is noted for it's mathematical purity. However it can be a syntactic difficulty for many. Most of the current popular programming languages use variants of ALGOL syntax, which is noted for being more readable. [22]

Julia also uses ALGOL syntax, but is lowered to S-Expressions. This enables many of the mathematically pure relations we seek to achieve. In addition S-Expressions are highly conducive to source transforms. This develops a notion of "Homoiconicity", where the representation of program structure is similar to the syntax. In Julia we use this property to make "macros" which enable source code to be transformed based on structure before compilation.

Generated functions perform a similar function as macros, but at the function level. They enable source code to be procedurally generated based on types. Surveys of Computer Science literature show that such a concept is new.

More on generated functions. Not sure if they should even be mentioned since they are usually unnecessary

### 3.5.1 Example

[23]

## 3.6 Numerical Robustness

Numerical robustness is a perennial problem in computational geometry. Multiple approaches exists for various numeric types. Floating points are by far the most difficult to deal with. Tools such as Gappa have been developed so algorithm writers can check their invariants when using floating points. [24] Such tools complicate software development and are not an accessible option for the casual researcher.

One of the most common problems formulated is to determine whether or not a point is collinear with a line segment. Shewchuk has one of the most pragmatic and robust treatments on this topic.[25] Kettner, et. al. have also developed more examples where numerical robustness is critical. [26]

More articulate like Graydon Hoare: `http://graydon2.dreamwidth.org/189377.html`

Julia's GeometricalPredicates package [5] uses the approach outlined by Volker Springel, which requires all floating point numbers to be scales between 1 and 2.[27] This has the downside of significantly reducing the available resolution.

why

A simpler, although less applicable, approach is to work within integer space. Developing a system around this is of interest. For example, it should be possible to specify a minimum unit (e.g. microns) and perform all computations in integer space assuming this does not exceed the needed resolution. More importantly, modern CPUs have integrated 128 bit Integer support. 170141183460469231731687303715884105727 is a lot of microns.

[5] `https://github.com/JuliaGeometry/GeometricalPredicates.jl`

# Chapter 4

# Implementation

## 4.1 Simplex

We began by implementing a Simplex type, defined as follows:

```
"""
A ‘Simplex‘ is a generalization of an N-dimensional tetrahedra and can be thought
of as a minimal convex set containing the specified points.

* A 0-simplex is a point.
* A 1-simplex is a line segment.
* A 2-simplex is a triangle.
* A 3-simplex is a tetrahedron.

Note that this datatype is offset by one compared to the traditional
mathematical terminology. So a one-simplex is represented as ‘Simplex{2,T}‘.
This is for a simpler implementation.

It applies to infinite dimensions. The sturucture of this type is designed
to allow embedding in higher-order spaces by parameterizing on ‘T‘.
"""
immutable Simplex{N,T} <: AbstractSimplex{N,T}
    _::NTuple{N,T}
end
```

With the definition in GeometryTypes, we afford ourselves two notions of dimensionality. Our first parameter ‘N‘ gives us the total dimensionality of the simplex. ‘T‘ is the type of the points. For example in Julia we can prefix a colon to an identifier and make it a symbol which is reflected in the type information:

```
julia> using GeometryTypes

julia> Simplex(:x,:y,:z)
GeometryTypes.Simplex{3,Symbol}((:x,:y,:z))
```

Symbolic representation will allow us to create parametric geometry. Likewise we can construct concrete types:

```
julia> Simplex(Point(0,0,0), Point(1,1,1))
GeometryTypes.Simplex{2,FixedSizeArrays.Point{3,Int64}}((FixedSizeArrays.Point{3,Int64}((0,0,0)),FixedSize
```

This last example illustrates how this type can give us extra generalization. Here we have constructed a line segment in 3D space. The Simplex is of size two but the space it occupies is three dimensional. This way it acts similar to a fixed size vector, but the type implies all points are on the convex hull. (Should update decription to be accurate here).

Below is an example of a high performance implementation of Simplex decomosition:

```
"""
Decompose an N-Simplex into tuple of Simplex{2}
"""
@generated function decompose{N, T1, T2}(::Type{Simplex{2, T1}},
                                         f::Simplex{N, T2})
    # other wise degenerate
    2 <= N || error("decompose not implented for N <= 2 yet. N: $N")

    v = Expr(:tuple)
    append!(v.args, [:(Simplex{2,$T1}(f[$(i)],
                                      f[$(i+1)])) for i = 1:N-1])
    # connect vertices N and 1
    push!(v.args, :(Simplex{2,$T1}(f[$(N)],
                                   f[$(1)])))
    v
end
```

To some extent, much of the basic defintions have been implemented in libraries such as CGAL, Boost, and ShapeOP. However these libraries are written in C and C++, compiled langauges, that are not amicable to dynamic exploration. Our system should allow us to quickly prototype tests and visualize relationships. This is something that Julia provides extremely well. Primarily it accomplishes this through optional type annotations and multiple dispatch. Secondarily, it has a rich history of integration with interactive computing environments such as a REPL, Jupyter, and Juno.

Prior to this project, GeometryTypes primarily provides for Polygonal Mesh type that is well tuned for operations on the CPU and GPU. It is defined as follows:

```
"""
The ‘HomogenousMesh‘ type describes a polygonal mesh that is useful for
computation on the CPU or on the GPU.
All vectors must have the same length or must be empty, besides the face vector
Type can be void or a value, this way we can create many combinations from this
one mesh type.
This is not perfect, but helps to reduce a type explosion (imagine defining
every attribute combination as a new type).
"""
immutable HomogenousMesh{VertT, FaceT, NormalT, TexCoordT, ColorT, AttribT, AttribIDT} <: AbstractMesh{Ver
    vertices            ::Vector{VertT}
    faces               ::Vector{FaceT}
    normals             ::Vector{NormalT}
    texturecoordinates  ::Vector{TexCoordT}
    color               ::ColorT
    attributes          ::AttribT
    attribute_id        ::Vector{AttribIDT}
end
```

The first thing to note is the provisions for attributes, colors, and textures. These are used for mapping textures and/or colors to polygons via APIs such as OpenGL. We do not need these (at least yet) in a rigourous mathematical definition. Likewise, in a HomogenousMesh we structure the realization as follows: 1. Insert all vertices of the mesh into ‘vertices‘ 2. Construct Faces of at least 3 indices referencing the points in ‘vertices‘.

This gives us certain properties that are nice for computation. Primarily this allows us to observe the combinatorial properties of the mesh by analyizing the faces. In addition, this compacts the data representation of vertices since shared vertices can be represented with a common face index. Affine transforms only need to operate on the vertices. Thus, in the optimistic versus pessimistic case these operations can be 3x faster with this layout assuming all vertices are in 3 faces.

Maybe a good idea to implement on this type too?

## 4.2 Parametric Triangle

## 4.3 Signed Distance Field

## 4.4 Polytope

## 4.5 Combinatorial Operations

# Chapter 5

# Conclusion

## 5.1 Future Work

### 5.1.1 Automatic Differention of Solids

```
julia> using DualNumbers

julia> f(x) = 2x+1
f (generic function with 1 method)

julia> f(Dual(1,1))
3 + 2du
```

### 5.1.2 Ray Tracing and Marching

When we look at the natural world we observe the propogation of light energy. Our eyes recieve this light energy in the form of photons. The study of ray tracing seeks to mimic such behavior for computer visualizations and simulations.

leave this section, would be good to discuss angle-based polyhedra and or deficits for these ops

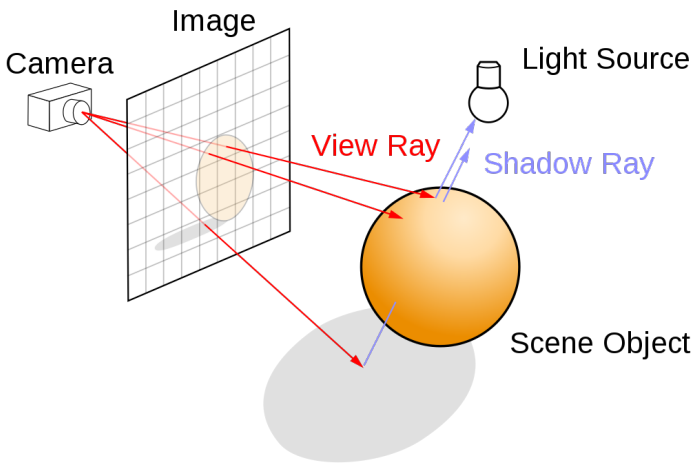


Figure 5.1: An illustration of a Ray tracing.[1]

Íñgo Quílez has done some of the most accessible work on real-time ray tracing. His technique is called ray marching, and leverages the properties of functional geometry.[28]

### 5.1.3 Engineering Solid Analysis

[1]By Henrik (Own work) GFDL or CC BY-SA 4.0-3.0-2.5-2.0-1.0, via Wikimedia Commons

# Bibliography


[1] C. Olah, "Manipulation of implicit functions (with an eye on cad) — christopher olah's blog." `https://christopherolah.wordpress.com/2011/11/06/manipulation-of-implicit-functions-with-an-eye-on-cad/`, Nov 2011.

[2] M. Shugrina, A. Shamir, and W. Matusik, "Fab forms: customizable objects for fabrication with validity and geometry caching," *ACM Transactions on Graphics (TOG)*, vol. 34, no. 4, p. 100, 2015.

[3] P. Henderson, "Functional geometry," *Higher-order and symbolic computation*, vol. 15, no. 4, p. 349–365, 2002.

[4] V. Shapiro, "Theory of r-functions and applications: A primer," tech. rep., Cornell University, 1991.

[5] V. Shapiro, "Semi-analytic geometry with r-functions," *ACTA numerica*, vol. 16, no. 1, pp. 239–303, 2007.

[6] A. Pasko, V. Adzhiev, A. Sourin, and V. Savchenko, "Function representation in geometric modeling: concepts, implementation and applications," *The Visual Computer*, vol. 11, no. 8, pp. 429–446, 1995.

[7] V. L. Rvachev, T. I. Sheiko, V. Shapiro, and I. Tsukanov, "On completeness of rfm solution structures," *Computational Mechanics*, vol. 25, no. 2-3, pp. 305–317, 2000.

[8] M. Voronyanskaya, K. Maksimenko-Sheiko, and T. Sheiko, "Mathematical modeling of heat conduction processes for structural elements of nuclear power plants by the method of r-functions," *Journal of Mathematical Sciences*, vol. 170, no. 6, pp. 776–793, 2010.

[9] `http://www.openvdb.org/`.

[10] C. Green, *Improved alpha-tested magnification for vector textures and special effects*, p. 9–18. ACM, 2007.

[11] H. Müller and M. Wehle, *Visualization of Implicit Surfaces Using Adaptive Tetrahedrizations*, vol. 0, p. 243. IEEE Computer Society, 1997.

[12] T. S. Newman and H. Yi, "A survey of the marching cubes algorithm," *Computers & Graphics*, vol. 30, p. 854–879, Oct 2006.

[13] *SIGGRAPH 95 conference proceedings: August 6 - 11, 1995, [Los Angeles, California]*. Computer graphics Annual conference series, ACM, 1995.

[14] A. Baerentzen and H. Aanaes, "Generating signed distance fields from triangle meshes."

[15] A. Sud, M. A. Otaduy, and D. Manocha, *DiFi: Fast 3D distance field computation using graphics hardware*, vol. 23, p. 557–566. Wiley Online Library, 2004.

[16] A. Pasko, V. Adzhiev, and P. Comninos, *Heterogeneous objects modelling and applications collection of papers on foundations and practice*. Springer, 2008.

[17] J. Bezanson, S. Karpinski, V. B. Shah, and A. Edelman, "Julia: A fast dynamic language for technical computing," *arXiv preprint arXiv:1209.5145*, 2012.

[18] J. Bezanson, A. Edelman, S. Karpinski, and V. B. Shah, "Julia: A fresh approach to numerical computin," *arXiv preprint arXiv:1411.1607*, 2014.

[19] J. Chen and A. Edelman, "Parallel prefix polymorphism permits parallelization, presentation & proof," in *Proceedings of the 1st Workshop on High Performance Technical Computing in Dynamic Languages*, (New York, NY), ACM, 2014.

[20] R. Harper, *Programming in standard ML*. 1997.

[21] J. McCarthy, "4e. recursive functions of symbolic expressions and their computation by machine, part i," *Programming systems and languages*, p. 455, 1966.

[22] C. Hoare, "Hints on programming language design."

[23] M. I. Shamos, *The Early Years of Computational Geometry—a Personal*, vol. 223, p. 313. American Mathematical Soc., 1999.

[24] `http://gappa.gforge.inria.fr/`.

[25] J. Shewchuk, "Fast robust predicates for computational geometry."

[26] L. Kettner, K. Mehlhorn, S. Pion, S. Schirra, and C. Yap, "Classroom examples of robustness problems in geometric computations," *Computational Geometry*, vol. 40, p. 61–78, May 2008.

[27] V. Springel, "E pur si muove: Galiliean-invariant cosmological hydrodynamical simulations on a moving mesh," *Monthly Notices of the Royal Astronomical Society*, vol. 401, p. 791–851, Jan 2010. arXiv: 0901.4107.

[28] I. Quilez, "Rendering worlds with two triangles with raytracing on the gpu in 4096 bytes," Aug 2008.