Computational Methods for Parametrization of Polytopes

Steve Kelly

April 28, 2016

Contents

Abstract

The combinatorial and geometric realization of polytopes are outlined in mathematical and computational terminology. With these two representations in hand, various parametric forms may be constructed using vertex locations, edge angles, and symbolic values. We have implemented software which represents polytopes in a way useful for combinatorial inspection and solid modeling using the Julia programming language. These packages have been published to GitHub and are accessible to mathematical researchers around the world through the Julia package manager.

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 rigorous definitional 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. The second is a polytope, which is a some what geometric construction. We will build an intuition of a polytope in this section, and formally define it in the next chapter. Parameters will be applied in the computation realm and are expanded in Section 3.

1.1 A Geometric Intuition of a Polytope

A polytope is a geometrically realizable graph composed of linearly connected vertices[1]. A "graph" is meant in the combinatorial sense of a structure composed of vertices and edges that show connectivity. Thus a geometric realization of a graph with linearly connected vertices is highly analogous to how we would traditionally draw a graph. Figure 1.1 shows a graph in with vertices labeled with numbers and edges between them.

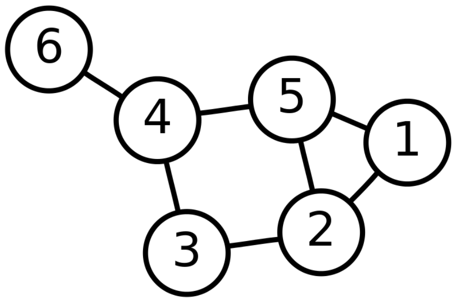


Figure 1.1: An example of a graph.

The informal term "flat" implies the path between two vertices on our polytope are connected by line segments. Thus for something to be considered geometrically realizable, we must be able to assign tuple values to the vertices.

An N-polytope then exists in the corresponding euclidean space dimension. For our purposes we will primarily use the real numbers, $\mathbb{R}^N$, for an N-polytope.

Figure 1.2 shows a two dimensional polytope, more commonly known as a polygon. The polgons we will primarily consider are 1-cycle graphs. 1-cycle implies that given any point there exist one path that exits then returns to the starting point. In three dimensions, polytopes are commonly known as a polyhedra. Figure 1.3 shows a polyhedral representation of a dolphin. Using this picture as reference, we see that given a vertex on a polyhedra there may be multiple cycles, or paths that exit then return to the vertex. Thus many assumptions about the properties of polytopes are contingent upon their dimensionality. Polytopes may have properties of convexity, connectedness, and closure associated with them. In order to study these properties we will elucidate a combinatoric and geometric representation in the coming sections. More importantly we will do so rigorously! These representations are somewhat distinct and we will see the implications as we later develop a computational type framework.

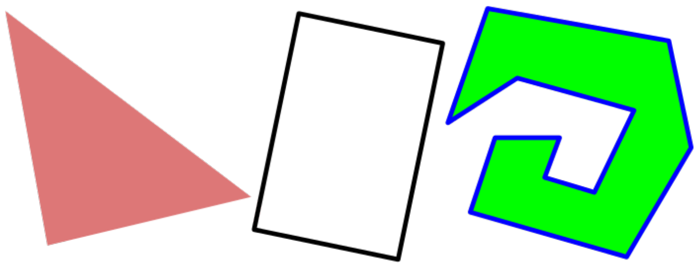


Figure 1.2: Polygons, or two dimensional polytopes

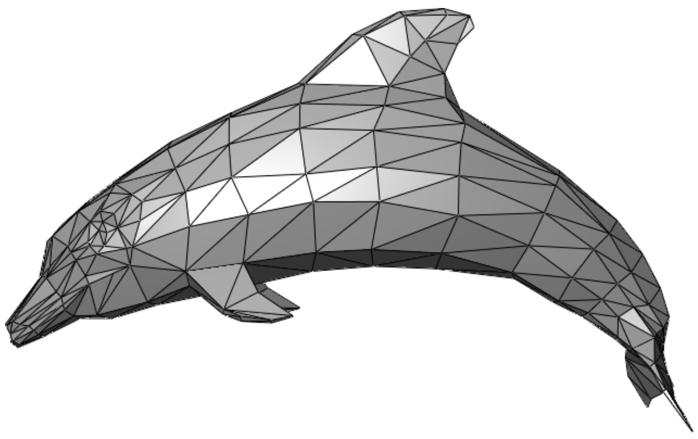


Figure 1.3: A polyhedral representation of a dolphin, or a three dimensional polytope

1.2 Flexibility of Polyhedra

Our initial problem comes via the flexibility of polyhedra. In order to understand flexibility of polyhedra it may be easier first to introduce the concept of rigidity. In this case we will be concerned with 3-dimensional polytopes, polyhedra, and observing the shape of the faces. If we give each edge (connection between vertices) a fixed length and the freedom to pivot around the vertex, a rigid polyhedra will not be able to deform. Cauchy's Ridgidity theorem proves this for any convex polyhedra[2]. If we allow the same degrees of freedom on the edges and the shape of the faces does not change it is called *flexible*[3].

In 2015 Maria Hempel presented an analysis of this problem using a representation of a polyhedra with edges and faces specified via angles and lengths [4]. Such representations may be useful in a variety of disciplines, but our focus will be strictly structural and for enhancing the foundations for discovering flexible polyhedra. We will expand on the significance of this representation in later sections.

1.3 Personal Perspective

This section will outline the personal motivation for this project. My hope is that this section will show where I would like this project to go in the future, and the potential impact I believe it could have.

In 2009 I first became interested with 3D printing. One of the key promises of 3D printing is localized and personalized production. This very quickly lead me into the world of generative geometry and in particular OpenSCAD. OpenSCAD is a solid modeling application which generates solids from scripts. Generative geometry from software allows ua designer to expose parameters for a user to manipulate, thus allowing many designs to be created from one structure and a user to make a solid that fits their unique needs or ecology. Many in this space call such a process "appropriate design".

One notable difficulty of OpenSCAD is that it includes its own programming language. In search of deeper understanding I began to work on my own replacements for OpenSCAD. In 2013 this began with the TextCAD project, which allowed one to use the principles of Object Oriented Programming (OOP) via Python [1]. Concurrently, Christopher Olah was working on ImplicitCAD, a pure Haskell implementation of a solid modeler[2]. I had realized that TextCAD had solved, to some extent, the expressibility issue with OpenSCAD but did not solve the complexity issue. Christopher Olah and I were on parallel paths with the expressibility issue, but he had approached the problem from a mathematical perspective of functionally defined solids. In fact, Haskell is a functional language which made this approach intrinsic. After spending some time testing TextCAD, I discovered the principles of OOP to be insufficient for correct representations of solids. One of the key insights made by Christopher was the importance of maintaining functional definitions of solids.

I strongly felt that one of the balances a good generative modeler should possess is readability. Many functional languages, including Haskell and Lisp are difficult for someone to parse. OpenSCAD may read very similar to a markup

[1] `https://github.com/textcad`

[2] `https://github.com/colah/ImplicitCAD`

language. For a year TextCAD sat until I heard about Julia. I began to use it as a replacement for MATLAB, but quickly realized it has all the requirements to be a functional programming language. This lead me to contribute to the geometry ecosystem in hopes of developing a solid modeling framework. Later in 2014 I joined the Lewis Research Group to develop a path planner for 3D printers, and later took a position at their spin-off company Voxel8 to continue my work. This afforded me the opportunity to spend most of my time researching and writing computational geometry software on polygonal meshes and polygons. I found an even deeper passion for the algorithms and theory behind computational geometry.

After leaving Voxel8 I began to pursue my work on solid modeling again. One of the most apparent issue was how to increase performance and propagate parameters in the model. The requirements laid out for computing the flexibility problem were parallel to my challenges. Much of my work in the middle of 2015 was collaborating with people working in the visualization space, so I had a good understanding of polygonal meshes. My hope was that I could make a contribution to this problem in the computational realm. Of course much of this work is nascent and I hope the structure of the data and packages will provide a good base for projects to come. It builds on much of what I have learned in the past 5 years of research in computational geometry. One notable absence from this report is the progress made in the functional representation of solids. Especially towards the convergence of combinatorial and functional polytopes. This work is embedded in the packages we used for this project, and some of the possibilities will be discussed in the conclusion.

Chapter 2

Mathematical Definitions

We are interested in constructing parametric polytopes for mathematical exploration. An informal geometric picture of a polytope has already been developed. In this chapter we will develop a mathematical perspective of our problem such that we can focus clearly on the computational aspects in the following sections.

2.1 Vector Spaces

Given a field (in the algebraic sense) of numbers, a vector of dimensionality N is formed by an N-tuple of numbers in the field. A vector space must be closed under element-wise addition with another vector and element-wise multiplication by a scalar value. Symbolically, given $s \in F$ where F is a field, and $X, Y \in V$ where V is the vector space over F then $X + Y \in V$ and $s * x \in V$ implies our closure property. In this paper we will primarily use the real numbers in euclidean N dimensional space, denoted as $\mathbb{R}^N$. The term "point" will generally be used to describe a vector that may described using numerical values.

2.2 Polytopes

To quote H.S.M. Coxeter, "A polytope is a geometrical figure bounded by portions of lines, planes, or hyperplanes". We have already introduced the notion that a polytope generalizes the notions of polygons and polyhedra, objects that have been studied since antiquity. If we interpret a polytope in one dimension, we may say that it is a line segment. Based on our previous geometric intuition of these objects we may see that a polygon is constructed from line segments and a polyhedron constructed from polygons. It is common a common occurrence in geometrical definitions that a construct of dimensionality N will use objects of dimensionality $N - 1$. We will formally define a polygon and polyhedron now.

2.2.1 Polygons

A polygon is a circuit of line segments formed by points. Let us define our number of points, N, as $V_1, V_2, ...V_N$. We will call these points "vertices", and form the line segments between vertices by $V_1V_2, V_2V_3, ...V_NV_1$ and call these "edges". For our purposes we will only consider vertices existing in the same plane [1]. A polygon forms an interior region and exterior region in the plane of the polygon. The interior region is of finite volume, and is bounded by the edges of the polygon.

2.2.2 Polyhedra

A polyhedron will be defined by a finite set of polygons. Given polygons $F_1, F_2, ..., F_N$ a polyhedron is a finite region bounded by these polygons. These polygons are called faces of the polyhedron.

2.3 Combinatorial Representation of a Polyhedra

Our initial intuition and definitions of polytopes are far from concrete enough to implement in a computational environment. In this section we will define a polytope using simplices, which are more constrained and allow us to more easily infer properties of the polytopes.

2.3.1 Simplices

A simplex (plural simplices) in N dimensional space is the minimal set of points whose convex hull form a closed subset in a space of dimensionality N. In set notation, for M linearly independent points (P) a simplex is the linear combination of these points.

$$\{\sum_{n=1}^{M} s_n \cdot P_n | \sum_{n=1}^{M} s_n = 1\} \tag{2.1}$$

It is often thought of as the generalization of a tetrahedra into N dimensions. In one dimensional space, this is a closed interval or line segment. In two dimensions it is the triangle, and in three it is the tetrahedra. These are called a 1-simplex, 2-simplex, and 3-simplex respectively. Figure 2.1 shows some examples of the connectivity between vertices in simplices.

We notice that the 1-simplex is formed by two 0-simplices, a 2-simplex is formed by three 1-simplices, and the 3-simplex by four 2-simplices. The components of these compositions are called “faces”, in a similar manner to the polyhedra. The 0-face is often called a vertex and the 1-face an edge. If we tabled the quantities of each M-face in an N-simplex out, they form Pascal’s triangle and thus follow the binomial coefficient in equation 2.2. This is due to the convex hull and the uniqueness of each point, thus each M-simplex is formed as a constraint on the dimensions.

$$\binom{N+1}{M+1} \tag{2.2}$$

Simplices will be our most basic geometry used for formulating the combinatorial form of a polytope. In fact, the two- and three-simplices are already polytopes in $\mathbb{R}^2$ and $\mathbb{R}^3$ respectively. To generalize independent of a vector, we will define a self-referential abstract form. In order to accomplish this we will introduce a notation of a simplex using a syntax that will be expanded in the computation section. We will let the order or dimensionality, N, of a simplex be

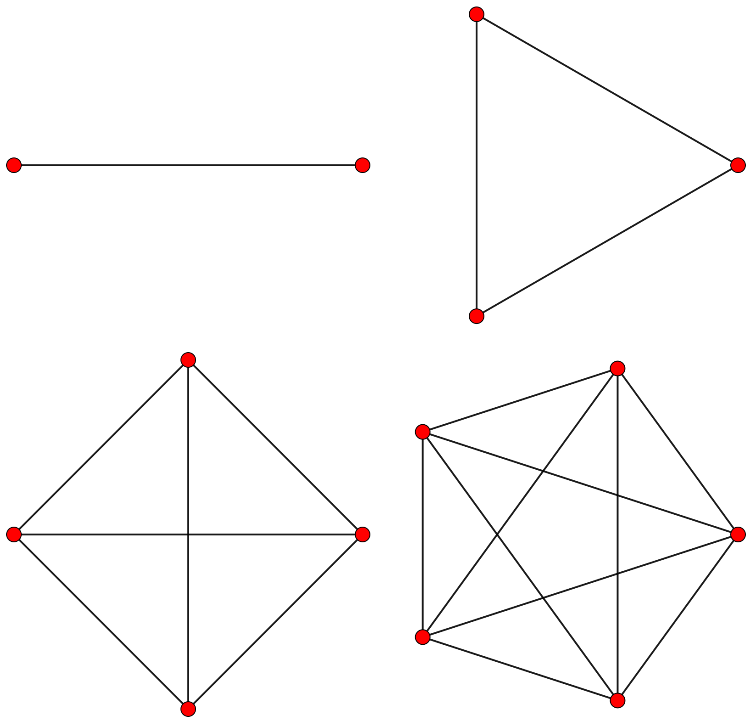


Figure 2.1: Examples of Simplices

denoted by $Simplex\{N\}$, where the brackets are the parameter for the dimensionality. Given a set of points P, let $Simplex\{0\}(P_1) = P_1$. A $Simplex\{N\}$ is constructed by a point with the convex hull of a $Simplex\{N-1\}$ in for a total quantity of points $\binom{N+1}{N}$. Thus given a set of points, P, we may make a recursive construction for a Simplex based on the convex hull outlined in Equation 2.1:

$$Simplex\{N\}(P) = (P_1, Simplex\{N-1\}(P_2, Simplex\{N-2\}(P_3, ...))) \quad (2.3)$$

Such a construction terminates since a $Simplex\{0\}$ is a point. Indeed we also see that this is equivalent to the set of points in the simplex. So simply put in words again this recursive definition for a $Simplex\{N\}$ is the convex hull of a $Simplex\{N-1\}$ plus an additional point, which is ostensibly the convex hull of all points.

2.3.1.1 Orientation on the two-simplex

Before we proceed to construct a polytope with simplices we will discuss orientation on a simplex. The orientation of a simplex is the direction an edge assumes from the ordering. This is simplest to explain in terms of the two-simplex. Given points 1, 2, and 3, we may order then in two ways 123, or 132. Thus our edges have an induced ordering.

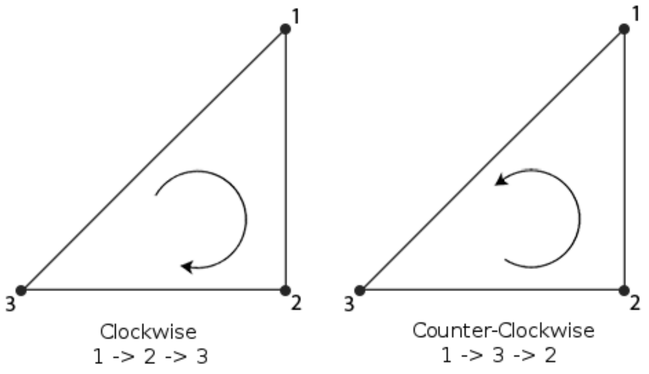


Figure 2.2: The two possible orientations of a triangle given 3 points.

2.3.2 Polygons

In our original definition of a Polygon we said the edges were implicitly formed by the ordering of the points. With the language of Simplices we may explicitly state this as follows. Let P be a set of points defining a polygon. Then a set S of one simplices may be formed from the edges in the polygon, namely $P_1P_2, P_2P_3, ...P_NP_1$, where $S_1 = P_1P_2, S_2 = P_2P_3, ...S_N = P_NP_1$. Thus a polygon may be validly described by the set of one-simplices, S.

2.3.3 Polyhedra

Our original definition for a polyhedra was based on polygons. We previously showed that a polygon may be described by a set of one-simplices. Since a two-simplex is a polygon with three points, this definition is much less tedious. Given a set, S, of two-simplices in $\mathbb{R}^3$, we may also form a polyhedra from S given the two-simplices for a closed space and are consistently oriented. A simple check for consistent orientation in a polyhedra is to check that for each edge (line segment) there exists only one of opposite orientation induced by another face.

Chapter 3

Computational Definitions and Grammar

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[5] [6]. 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, and allow the compiler to infer this information based on the input types. JIT compilation means code is compiled during runtime which allows functions to be recompiled and thus optimized for various data types. 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

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

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.

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. Thus the programmer may write less code and spend less time waiting for results in an interactive environment, which makes Julia a great choice for exploratory programming. Much of this comes down to the innovated type and function system.[7] We will discuss these more in depth later.

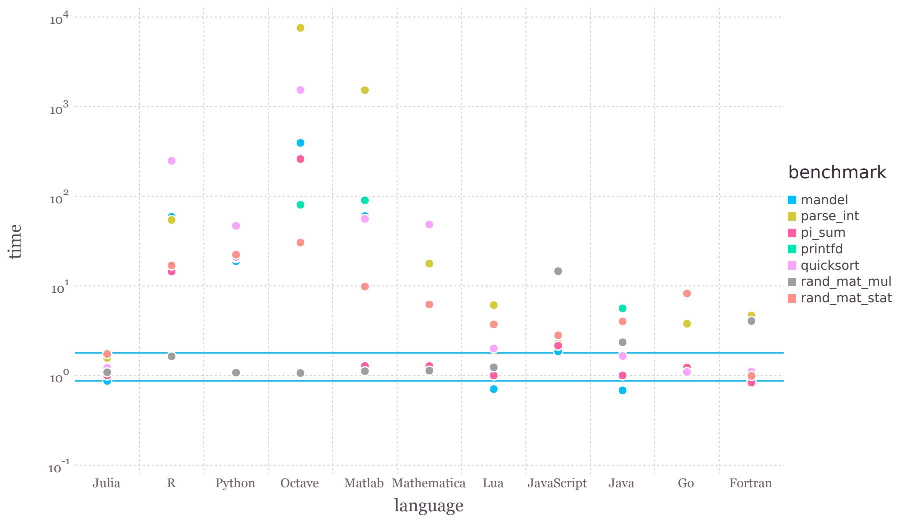


Figure 3.1: A comparison of programming languages and performance.

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):
```

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

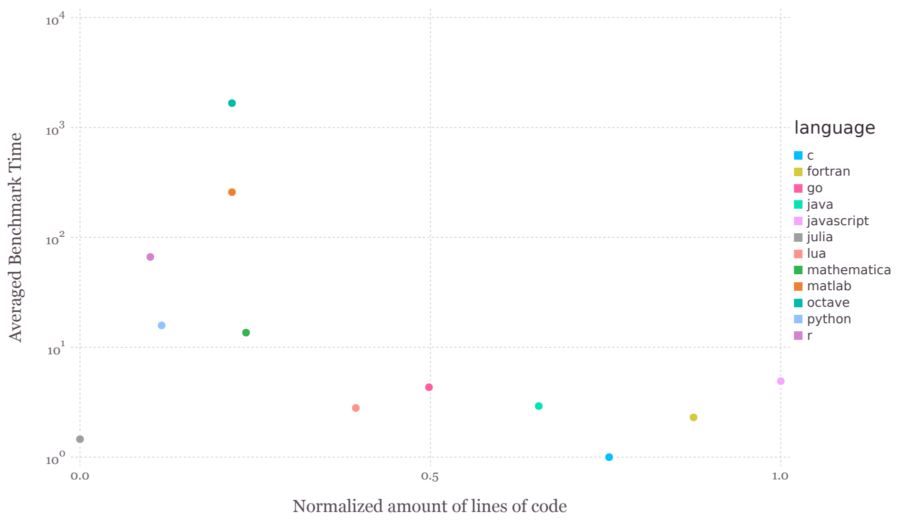


Figure 3.2: The results in Figure 3.1 normalized for code length. (Courtesy of Simon Danish)

```
        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, preferring rather to separate data from functionality. 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, where in a static system the programmer needs to specify to the program compiler how data is transformed in a function. Generic functions allow a single function name, for example `sum`, to have multiple definitions with execution contingent upon the matching of argument types. Many programmers may first encounter generic functions through the term "overloaded" function as well. 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. We will dedicate a section to the explanation of type parameters. 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 or dynamically 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[8]. The compiler analyzes program flow and is able to infer the types of variables and function returns. 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 preferable 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 types to functions, since Julia will default to the *most specific implementation* for the type. 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 sufficiently abstract implementations of functions. If special cases occur,

[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.

multiple functions with different arity or type signatures can be implemented. Explicitly this is called multiple dispatch. In practice by the user this looks like abstracted or generic code if done well so many types can be handled by one function. To the computer, this means choosing or generating the most specific and performant method[5] 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>
}

julia> @code_llvm increment(1.0)

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

Note we 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 expressibility. In this case, multiple dispatch used by the compiler for specialization of functions reveals itself as a way for the user to specialize over many types. In summary, the basic steps in generating native computer code from a function are to:

1. Parse the expression
2. Infer type information
3. Generate native machine code and optimizations

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 mutable, immutables are immutable. What does this mean? Let's break something first via the REPL:

```
julia> type FooIsMutable
           a
       end

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

julia> f.a
1

julia> f.a = 2
2
```

[5]Functions and methods are distinct in Julia. A function may be thought of in the mathematical sense. A method is a function specialized on types with unique machine code.

```
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
```

Our error shows that immutable objects may not have their data (fields) modified. Conversely our mutable object which is an instance of a type (i.e. `f`), can have its fields (i.e. `a`) changed. An immutable object *cannot ever change*. The immutable contract helps develop a notion of functional purity. To the user this means immutables are defined by their values. This can be of great benefit to avoid errors and establish concrete equality between types, such as vectors. Practically this can be of great benefit to the compiler to determine invariants and eliminate pointers in a datatype. 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 without pointers to values"? 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", which means the computer may store the data in registers closer to the CPU. Immutability helps us achieve this by ensuring our data is concretely defined and not a reference to another piece of memory. 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. For example the definition for `Int64` is `bitstype 8 Int64` which means an `Int64` is 8 bytes long. `Array{Int64,1}` is a mutable data type that can vary in size. This means the `Tuple` needs to store the arrays as references to memory, 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 excel when data is linearly packed and pointer-free. The data can be brought into the CPU's memory cache and registers only once and computed without shuffling between cache and RAM. The cost of lookup time between the cache and RAM generally differs by several orders of magnitude.

3.4.2 Parameters

We have already seen a rough notion of type parameters in the `Array{Int64,1}` type. The curly brackets, {}, denote the type's parameters, which are separated by commas. For example:

```
julia> [1]
1-element Array{Int64,1}:
 1

julia> [1.0]
1-element Array{Float64,1}:
 1.0

julia> [1 2;3 4]
2x2 Array{Int64,2}:
 1  2
 3  4

julia> [1. 2.;3. 4.]
2x2 Array{Float64,2}:
 1.0  2.0
 3.0  4.0
```

So here we see `Array`s are parameterized on the numeric type and dimensionality. We may construct our own type that will take any type:

```
julia> type Para{T}
           a::T
       end

julia> Para(1)
Para{Int64}(1)

julia> Para(1.0)
Para{Float64}(1.0)

julia> Para([])
Para{Array{Any,1}}(Any[])
```

3.4.2.1 Parametric Functions

The same way a type may be parameterized we my parameterize a function. For example we may only want to handle arrays of different numeric types separately:

```
julia> function whoami{T<:AbstractFloat}(::Array{T})
           print("I am an array of floats!")
       end
whoami (generic function with 1 method)

julia> function whoami{T<:Integer}(::Array{T})
           print("I am an array of integers!")
       end
whoami (generic function with 2 methods)

julia> whoami([1])
I am an array of integers!
julia> whoami([1.])
I am an array of floats!
```

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[9]. The notion of an “S-Expression” was introduced in McCarthy’s seminal work, “Recursive functions of symbolic expressions and their computation by machine”. 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 [10]. Julia also uses ALGOL syntax, but is converted to S-Expressions after parsing [6]. 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 the syntactical 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. This allows the user fine-tuned control of the compilation process, and will allow optimizations to be performed that are not currently available in the compiler. Surveys of Computer Science literature show that such a concept is new in a programming language that uses type inference [7]. However the use of generated functions is generally frowned upon by the Julia community since it makes compilation more difficult since type inference has to be run multiple times before compilation may happen. This often slows down trivial functions by several orders of magnitude, and should be only used if a method is called many times and performance is critical.

We will omit an introduction of macros and generated functions as they are advanced language features. For our purposes a basic understanding of the terminology will be sufficient. However we may expand our initial compilation pipeline to include macros and generated functions:

1. Parse the expression
2. Run macros on the syntax tree
3. Infer type information
4. Allow generated functions to create function bodies
5. Generate native machine code and optimizations

[6]https://www.youtube.com/watch?v=osdeT-tWjzk
[7]http://docs.julialang.org/en/release-0.4/manual/metaprogramming/

3.6 Numerical Robustness

Numerical robustness is a perennial problem in computational geometry[11]. 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[12]. 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[13]. Kettner, et. al. have also developed more examples where numerical robustness is critical[14].

Julia's GeometricalPredicates package [8] uses the approach outlined by Volker Springel, which requires all floating point numbers to be scales between 1 and 2[15]. This has the downside of significantly reducing the available resolution to 50% of the available floating point numbers.

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.

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

Chapter 4

Implementation

In this section we will begin to outline the implementation of various forms of parametric polytopes.

4.1 Survey of Available Packages

In chapter 3 we outlined the rationale for using Julia for mathematical computer programming. An additional impetus was the familiarity of the geometry packages. There will be various references to these and they are outlined below so the reader may become familiar with the utilities available.

4.1.1 GeometryTypes.jl

GeometryTypes.jl provides datatypes and basic operations for computational geometry. This package began as a unification of types located in HyperRectangles.jl, Meshes.jl, and GLAbstraction.jl. The initial types were polygonal meshes and bounding boxes, but now encompasses datatypes for solid modeling, data visualization, and geographic information systems. With the introduction of this package the community made some initial progress on designing types that can be used for computation on the CPU and GPU, however GPU targets are rapidly evolving and the focus has shifted from geometric operations to array operations. Much of our basic combinatorial analysis operations and data types have be contributed to this package.
`https://github.com/JuliaGeometry/GeometryTypes.jl`

4.1.2 FileIO.jl and MeshIO.jl

FileIO.jl is a package that unifies various file loaders that existed in the Julia package ecosystem under one import. The purpose is to allow users to simply call the `save` and `load` functions with file information inferred from file extensions, magic numbers, or data types. MeshIO.jl is one such package that provides file loaders for polygonal mesh data. The file formats supported as of this writing include obj, stl, ply, off, and 2dm. This package may be useful for importing polytope data from other programs such as Blender or AutoCAD, or generating large data sets.

It should be noted that Julia has a `serialize` function, which will save a datatype in full fidelity and in compact binary. Since Julia is yet to reach a 1.0 release, this function is considered unstable. Once `serialize` is stable, it will be the preferred method of saving data sets to the computers storage drive.

`https://github.com/JuliaIO/FileIO.jl`

`https://github.com/JuliaIO/MeshIO.jl`

4.1.3 Meshing.jl

Meshing.jl provides algorithms for converting signed distance field (SDF) data into polytopes. Many algorithms for generating polyhedra from an SDF exist. The most common are Marching Tetrahedra, Marching Cubes, and Dual Contours[16][17][18]. The two algorithms currently provided are the Marching Cubes (MC) and Marching Tetrahedra (MT) algorithms. For this project we added the Marching Cubes algorithm[1] which is twice as fast as the Marching Tetrahedra algorithm. The import difference between the two is performance and manifold mesh generation. The MT algorithm generates manifold meshes, but generates more faces (costing memory) and is slower. It is useful for generating meshes from noisy data or applications where manifold meshes are required such as finite element analysis and 3D printing. The Marching Cubes algorithm is less costly for the computers resources, and is helpful for visualization applications where user experience is important.

`https://github.com/JuliaGeometry/Meshing.jl`

4.1.4 Meshes.jl

Meshes.jl is currently a meta-package[2] that imports elements on GeometryTypes.jl, FileIO.jl, and Meshing.jl. It is one of the older packages in the Julia package ecosystem and was an early center of collaboration before the scopes began to expand. Releases before Meshes.jl became a meta-package are maintained for institutional users[3]. The name space is held to allow for a center for experimentation as stability in the base packages becomes more necessary.

`https://github.com/JuliaGeometry/Meshes.jl`

4.1.5 ParametricPolyhedra.jl

ParametricPolyhedra.jl is a package used for solving constraints on triangular faces of a polyhedra. The intention of this package is to allow polyhedra to be specified via angles and edge lengths. It draws heavily from the resources available in GeometryTypes. Since it uses algorithms to define the types and is some what domain specific at this point, we opted to make it a separate package.

`https://github.com/sjkelly/ParametricPolyhedra.jl`

4.1.6 GeometricalPredicates.jl

GeometricalPredicates.jl is a package that provides numerically robust primitives and algorithms for computing incircle, circumcircle, and intriangle calcula-

[1]`https://github.com/JuliaGeometry/Meshing.jl/pull/2`

[2]Meta-package means little or no code is contained in the package besides imported code from other packages. It is often used for version stability or usability purposes.

[3]`https://github.com/JuliaGeometry/Meshes.jl/tree/v0.1.x`

tions. The approach to numerical robustness is used by the Illustric Simulation, and outlines in Volker Springel's paper "Galiliean-invariant cosmological hydrodynamical simulations on a moving mesh"[15]. The essence of the approach is to restrict values in 64 bit floating points between 1 and 2 since the exponent component is constant. This allows 128 bit integers to be used for overflow calculations.

`https://github.com/JuliaGeometry/GeometricalPredicates.jl`

4.2 GeometryTypes.jl Implementations

4.3 Simplex

We began by implementing a Simplex type in GeometryTypes.jl, 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. We will notice that our convention is offset by positive one compared to the mathematical terminology. This is due to Julia not allowing arithmetic in type definitions. There are a few approaches to circumvent this issue, but they either make the datatype larger or sacrifice strong type inference.

The second parameter, `T` is the type of the points. We will see that point may be symbolic in nature, or have their own dimensionality expressed independent of `N`. For example in Julia we may prefix a colon to an identifier and make it a symbolic value which is reflected in the type information:

```
julia> using GeometryTypes

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

In this example we have created a 2-simplex with symbols `:x, :y, :z`. `N` is 3, and `T` has become `Symbol`. Symbolic representation will allow us to create simple combinatorial analysis. 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)),FixedSizeArrays.Point{3,Int64}((1,1,1))))
```

This last example illustrates how `N` and `T` may give us two notions of dimensionality in the Simplex. 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. Unfortunately it may also be possible to construct a Simplex using points of dimension less than that of the Simplex, which would not hold to our contract of linear independence. More so we may also decompose its

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

```
"""
Decompose an N-Simplex into a tuple of Simplex{1}
"""
@generated function decompose{N, T1, T2}(::Type{Simplex{1, T1}},
                                          f::Simplex{N, T2})
    v = Expr(:tuple)
    append!(v.args, [:(Simplex{1,$T1}(f[$i])) for i = 1:N])
    v
end
```

4.4 HomogenousMesh Type

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{VertT, FaceT}
    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 visualization software such as OpenGL. We do not need these in a rigorous 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 analyzing 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, and if it is closed and faces share many vertices this may be up to 3 times faster.

However the most important issue with this type is that it is not parameterized as a Polytope, and simply as a polyhedral mesh.

4.5 Polytope Type

We implemented a Polytope to address some of the issues with the `HomogenousMesh` type[4]. It is defined as follows:

```
"""
A ‘Polytope‘ is an ‘N‘ dimensional object with elements ‘T‘ of the same type.
For example typealias ‘Polygon‘ and ‘Polyhedron‘ exist for dimensions 2 and
3 respectively.
"""
type Polytope{N,T} <: AbstractPolytope{N,T}
    elements::Vector{T}
end
```

The supertype `AbstractPolytope` type is not implied in the mathematical sense, but rather to allow more granular definitions as needed for different computational challenges. The `Polytope` type is parameterized by `N`, the order of the polytope. The following aliases exist for Polytopes with specified values for `N`:

```
"""
A ‘Polygon‘ is a ‘Polytope‘ realizable with only two dimensions.
Generally this will be composed of ‘Points‘ or ‘LineSegment‘s.
"""
typealias Polygon{T} Polytope{2,T}

"""
A ‘Polyhedron‘ is a ‘Polytope‘ realizable with only three dimensions.
Generally this will be composed of ‘Face‘s or two-simplices (‘Simplex{3}‘).
"""
typealias Polyhedron{T} Polytope{3,T}
```

The final parameter, `T`, is the type of the elements. This may simplify many representations, and allow more liberty in Polyhedron representation. For example, constructions of polygons are straight forward and may be a `Vector` of `Symbol` or `Point`. However a Polyhedron may be constructed from `Simplex` or `Polygon`. In this way it behaves as a wrapper of a `Vector` with special type information associated. Of course, nonsensical constructions may be made, but with sufficiently parameterized functions they will not be operable.

4.5.0.1 Functions

Along with defining a `Polytope` we have added calculations for area, volume, centroids, and various decomposition functions.
`https://github.com/JuliaGeometry/GeometryTypes.jl/pull/27`

4.5.1 Signed Distance Fields

A signed distance field (SDF) is a uniform sampling of an implicit function. It was implemented earlier as a Below we can see this in action over the definition of a circle.

```
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
```

[4]`https://github.com/JuliaGeometry/GeometryTypes.jl/pull/27`

```
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. We have created a data type called `SignedDistanceField`, defined below.

```
"""
A `SignedDistanceField` is a uniform sampling of an implicit function.
The `bounds` field corresponds to the sampling space intervals on each axis.
The `data` field represents the value at each point whose exact location
can be rationalized from `bounds`.
The type is parameterized by:
* `N` - The dimensionality of the sampling space.
* `SpaceT` - the type of the space where we will uniformly sample.
* `FieldT` - the type resulting from evaluation of the implicit function.
Note that decoupling the space and field types is useful since geometry can
be formulated with integers and distances can be measured with floating points.
"""
type SignedDistanceField{N,SpaceT,FieldT} <: AbstractSignedDistanceField
    bounds::HyperRectangle{N,SpaceT}
    data::Array{FieldT,N}
end
```

4.6 Parametric Polyhedra

The purpose of Parametric Polyhedra is to allow a polytope to be represented with angles and edge lengths.

4.6.1 ParametricTriangle

In order for us to start we must parameterize a triangle. Our first definition is as follows:

```
type ParametricTriangle{T}
    # edge lengths
    a::Nullable{T}
    b::Nullable{T}
    c::Nullable{T}
    # angles (radians)
    alpha::Nullable{T}
    beta::Nullable{T}
    gamma::Nullable{T}
end
```

It uses the `Nullable` type to give values the additional property of being known or unknown. A `Nullable` often checked with the `isnull` function, overloaded as follows:

```
function Base.isnull(p::ParametricTriangle)
    isnull(p.a) || isnull(p.b) || isnull(p.c) ||
    isnull(p.alpha) || isnull(p.beta) || isnull(p.gamma)
end
```

In order to check the configuration space of the `ParametricTriangle` as valid we needed to check all of the values are defined and follow the sine and cosine relations:

```
"""
Test if a ParametricTriangle has a valid configuration.
"""
function Base.isvalid(p::ParametricTriangle)
    # underdetermined case
    isnull(p) && return false
    # otherwise check constraints since all values exist
    a = get(p.a)
    b = get(p.b)
    c = get(p.c)
    alpha = get(p.alpha)
    beta = get(p.beta)
    gamma = get(p.gamma)
    return a*cos(beta) + b*cos(alpha) - c == 0 &&
           b*sin(alpha) - a*sin(beta) == 0 &&
           alpha + beta + gamma - pi == 0
end

# version with isapprox for floats
function Base.isvalid{T<:AbstractFloat}(p::ParametricTriangle{T};
                                        rtol=sqrt(eps(T)),
                                        atol=zero(T))
    # underdetermined case
    isnull(p) && return false
    # otherwise check constraints since all values exist
    a = p.a.value
    b = p.b.value
    c = p.c.value
    alpha = p.alpha.value
    beta = p.beta.value
    gamma = p.gamma.value
    return isapprox(a*cos(beta) + b*cos(alpha) - c,0,
                    rtol=rtol,atol=atol) &&
           isapprox(b*sin(alpha) - a*sin(beta),0,
                    rtol=rtol,atol=atol) &&
           isapprox(alpha + beta + gamma - pi,0,
                    rtol=rtol,atol=atol)
end
```

If some of the edge values in a triangle are unspecified, the following function may complete the `ParametricTriangle`.

```
"""
Given an underdetermined ParametricTriangle, compute the missing values
and return a new ParametricTriangle
"""
function Base.fill(p::ParametricTriangle)
    # all angles must be specified
    if isnull(p.alpha) || isnull(p.alpha) || isnull(p.gamma)
        error("Cannot fill in values for this triangle. All angles must be specified")
    end
    alpha = get(p.alpha)
    beta = get(p.beta)
    gamma = get(p.gamma)
    # no edges given, use circumcircle=1
    if isnull(p.a) && isnull(p.b) && isnull(p.c)
        e = edges(alpha,beta,gamma)
        return ParametricTriangle(e[1],e[2],e[3],p.alpha,p.beta,p.gamma)
    else
        # find the circumcircle
        D = !isnull(p.a) ? get(p.a)/sin(alpha) :
            !isnull(p.b) ? get(p.b)/sin(beta) :
            get(p.c)/sin(gamma) # one must be specified because of prior check
        # we only need to figure one side that is specified
        # so we can (re)compute the other two
        if !isnull(p.a)
            return ParametricTriangle(p.a, _edge(beta,D), _edge(gamma,D),
                                      p.alpha, p.beta, p.gamma)
        elseif !isnull(p.b)
            return ParametricTriangle(_edge(alpha,D), p.b, _edge(gamma,D),
                                      p.alpha, p.beta, p.gamma)
        elseif !isnull(p.c)
            return ParametricTriangle(_edge(alpha,D), _edge(beta,D), p.c,
                                      p.alpha, p.beta, p.gamma)
        end
    end
end
```

4.6.2 ImplicitTriangle

The purpose of the implicit triangle is to use the law of sines to validate a given triangle configuration. The law of sines is given in Equation 4.1, with values correspond to those given in Figure 4.1.

$$\frac{a}{\sin(A)} = \frac{b}{\sin(B)} = \frac{c}{\sin(C)} = d \tag{4.1}$$

The common value, d, is the triangle's circumcircle diameter. Thus if we are given 3 edge lengths (a, b, c) we may compute this value directly with the following:

$$d = \frac{2abc}{\sqrt{(a+b+c)(-a+b+c)(a-b+c)(a+b-c)}} \tag{4.2}$$

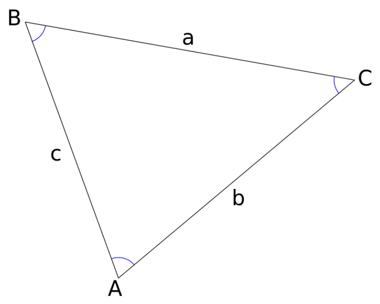


Figure 4.1: A triangle corresponding to the law of sines given in Equation 4.1.

Since we know edge lengths will be strictly positive, and sine is positive in the range of 0 to π. Thus by subtracting the computed circumcircle with Equation 4.2 from the value computed by the law of sines we will have a number strictly less than or equal to zero if $A + B + C = \pi$. The configuration space may be mapped with the following function:

```
function implicit_triangle(a,b,c,alpha,beta,gamma)
    r = a*b*c/sqrt((a+b+c)*(a-b+c)*(a+b-c)*(b+c-a))
    min(sin(alpha)/a - 2r,
        sin(beta)/b - 2r,
        sin(gamma)/c - 2r)
end
```

The purpose of the `min` function is to choose the worst error in the configuration space. For example, this function may be mapped over the range of angle values using a `SignedDistanceField`.

```
using GeometryTypes

res = 0.1

s = SignedDistanceField(HyperRectangle(Vec(0,0.),Vec(pi*1,pi*1)), res) do v
    implicit_triangle(3,3,3,v[1],v[2],pi/3)
end
```

For our purposes a global minima search may be performed fairly quickly and is implemented as follows:

```
"""
Find the value closest to zero and return the linear index of the element.
"""
function find_zeros{T}(mat::Array{T})
    x = typemax(T)
    ind = 0
    # find the value closest to zero
    @inbounds for i = 1:length(mat)
        val = abs(mat[i])
        if val < x
            ind = i
            x = val
        end
    end
    ind
end
```

Other iterative techniques may also be used for solving the configuration space such as gradient descent and the BFGS algorithm. Given more time this is the preferable method, and readily optimized algorithms exist in Optim.jl[5].

4.7 Visualization

One aspect of this project was the visualization of Polyhedra. The requirements are as follows:

1. 3D previews of polyhedra geometry
2. Interface with sliders
3. Code/data input and expandable visualizations

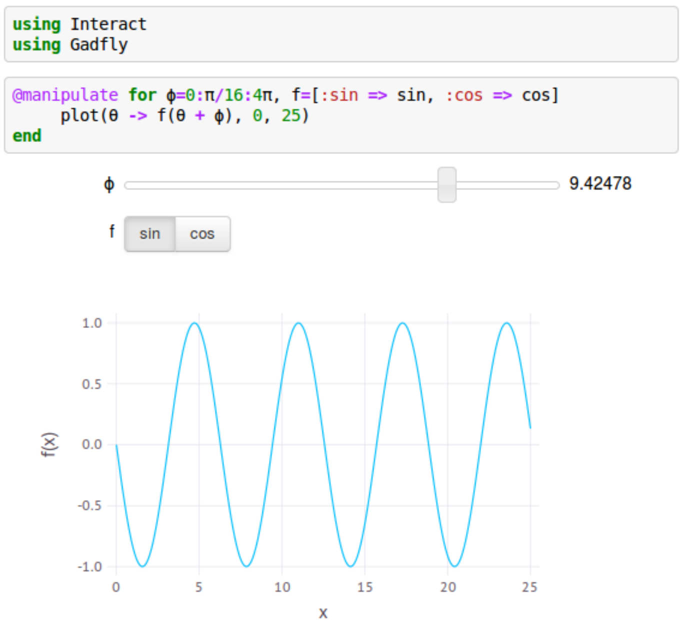
One of our strongest collaborators, Simon Danisch is the lead developer of GLVisualize.jl. GLVisualize.jl meets these goals and has the best performance. However it is still highly experimental and we struggled to achieve stability for this project. This is due to the rapidly changing nature of GPU computing and the OpenGL interface in particular.

Our second option was to leverage the web technologies available. In particular we will be using Interact.jl to provide sliders, ThreeJS.jl to provide 3D previews, and Jupyter notebooks for dynamic evaluation of code. Jupyter, formerly known as IPython, provides a “notebook” for editing code with rich presentation of the code, text, LaTeX, and graphics[19]. Figure 4.2 shows an example of Jupyter notebooks used for interactive Julia development.

At the time ThreeJS.jl was not connected to the datatypes in GeometryTypes.jl. We submitted pull request #12 [6] which added the ability to load and preview a `HomogenousMesh` from GeometryTypes.jl. However the latest update

[5]https://github.com/JuliaOpt/Optim.jl

[6]https://github.com/rohitvarkey/ThreeJS.jl/pull/12



```julia
using Interact
using Gadfly
```

```julia
@manipulate for ϕ=0:π/16:4π, f=[:sin => sin, :cos => cos]
    plot(θ -> f(θ + ϕ), 0, 25)
end
```

Figure 4.2: An example of using Jupyter notebooks with Interact and the 2D plotting package Gadfly to make interactive visuals.

of Jupyter notebooks cannot display plots from ThreeJS.jl. We were able to make specialized interactive webpages that display mesh data with sliders shown in Figure 4.3. As we progress into summer of 2016 it is likely contributions and bug fixes will help accelerate this portion of the project.

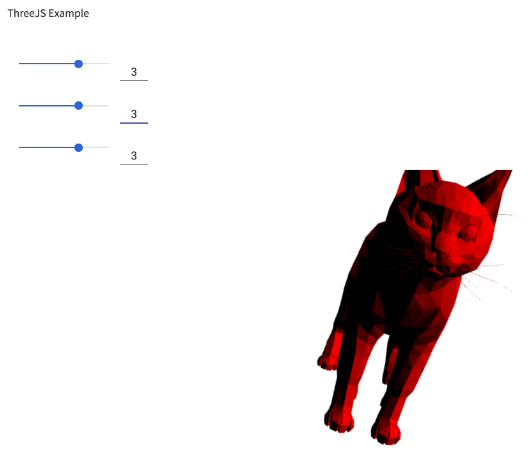


Figure 4.3: An interactive visualization of a mesh with scaling.

Chapter 5

Conclusion

Overall this project has lead to a stronger mathematical perspective of the geometry packages available for Julia. We have identified the weak points in our existing data types and created new ones to achieve better performance and mathematical correctness. Progress was made in the various fronts of combinatorics, numerics, and visualization, and in many cases we brought packages together through this process. Our work is carefully unit tested and available in the Julia package ecosystem. Overtime we hope that this work will progress further with more contribution and continue to progress openly. Computational geometry has been my passion for many years and I am thankful to all who have let me pursue it.

Bibliography

[1] H. S. M. Coxeter, *Regular Polytopes.* Dover Publications, 1973.

[2] H. Gluck, *Almost all simply connected closed surfaces are rigid*, p. 225–239. Springer, 1975.

[3] R. Connelly, "A counterexample to the rigidity conjecture for polyhedra," 1977.

[4] M. Hempel, "An attack on flexibility and stoker's problem," *arXiv preprint arXiv:1512.05230*, 2015.

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

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

[7] 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.

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

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

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

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

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

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

[14] 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.

[15] 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.

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

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

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

[19] F. Pérez and B. E. Granger, "IPython: a system for interactive scientific computing," *Computing in Science and Engineering*, vol. 9, pp. 21–29, May 2007.