Homology Pt. 1

Christina Lee

July 8, 2017

Category: Grad
Tags: Mathematics

1 Introduction

I've been learning the pure mathematics of topology to understand the physics of topological materials better. Predictably, my sources have a lot of theorems, the examples tend to be esoteric and hard to conceptualize, and I have little idea how it all relates to my pretty crystals. There also aren't that many examples. I almost miss the days back in high school when they forced us to do the same problem 1,000 times over.

So to better understand one of the important topics, simplices, and simplicial complexes, I've decided to take the pure math and put it to programming the best that I can. I don't know if this is the best way of doing things, or if it's useful at all, but I think it helps me understand them. Hopefully, it will help you too.

Topology is the study of deforming things. If we push and pull at something, without tearing or gluing, what properties remain the same? A coffee cup is the same as a donut because they both have one hole.

We are also the same as a donut because of our digestive track... if you ignore all the open cavities we have inside. We can at least deform away our lungs. I'm not an anatomy expert.

Many objects of interest are topologies, which are just collections of sets that obey certain constraints. Every manifold you've ever worked with has been a topology, but it probably was just trivial and didn't need further study. To study more complex topologies, we can find an equivalent simplicial complex. We can then compute a variety of things about the simplicial complex that tells us about the properties of the topology.

Simplicial complexes are made out of simplices.

A simplex is an n-dimensional generalization of a triangle. A 0-dim simplex is a point; a 1-dim simplex is a line; 3-dim a tetrahedron; so on and so forth.

The rules for the simplices in a legitimate simplicial complex are

1. Every point must belong to at least one simplex
2. A point can belong to only a finite number of simplices
3. Two different simplices either have no points in common, or
    1. on is a face (or edge, or vertex) of the other
    2. the set of points in common is the whole of a shared face (or edge, or vertex).

[1]

A simplicial complex of dimension d will contain a finite number of d-dimensional simplices, the faces of those simplices, the faces of the faces, and so on until you get to all the points contained in the original simplices.

I will denote each point by a string of length 1, like `"a"`.

A simplex of dimension d is a string of length d. For example, in the figure above, the two dim 2 simplices are `"abc"` and `"acd"`. For now, the order doesn't matter, but I am also working on a post to discuss p-chains, where order matters.

A simplicial complex of dimension d contains simplices of dimension d, dimension $d-1$, ... , to 0. I create an array of arrays to store these simplices. Each array of simplices corresponds to a different dimension.

```
In [1]: type simplex
            s::String
        end
```

```
In [2]: type SimplexComplex
            d::Int
            s::Array{Array{simplex}}
        end
```

After almost finishing this post and hours of staring at some quite incomprehensible output, I've realized that I can use some nice IO aspects to tailor how Julia displays my abstract types.

I first create a function that outputs some HTML to the IO stream. Then `Base.show` takes that IO stream and renders it as HTML.

I believe that is what is going on. All I know right now is this works.

I particularly used this link https://docs.julialang.org/en/stable/manual/types/#Custom-pretty-printing-1 to influence how I created this function.

```
In [3]: function displayS(io::IO,s::simplex)
            print(io,"<center>Simplex: $(s.s)</center>")
        end
        Base.show(io::IO,::MIME"text/html",s::simplex)=displayS(io,s)
```

```
In [4]: function displaySC(io::IO,sc::SimplexComplex)
            print(io,"<center><h3>Simplicial Complex</h3></center>")
            for kk in 1:sc.d
                print(io,"<center><b>Dimension: $(kk)</b></center>")
                sc_now=sc.s[kk]
                bigstring="<center>"
                for jj in 1:length(sc_now)
                    bigstring=bigstring*"&emsp;"*sc_now[jj].s
                end
                bigstring=bigstring*"</center>"
                print(io,bigstring)
            end
        end
        Base.show(io::IO, ::MIME"text/html", sc::SimplexComplex)=displaySC(io,sc)
```

I could just iterate out every point, edge, and face of a simplicial complex by hand, but why spend my time doing that when I can spend even more time writing code to get my computer to do it for me?

I'll send in the d-dimensional structure, and the computer will figure out everything smaller than that.

My algorithm will go through and compute the face of every block in the last array, then the edges of all the faces from before that, and so forth.

But some edges will belong to more than one face. See Point 3A in what a simplicial complex is, or line `"ac"` in the figure. I could just have `"ac"` in the simplicial complex twice, but then things start getting ugly and clunky.

This next function `AreSame` determines if two strings are permutations of each other so that we can clean up the complex. The even/ odd permutation aspect will play a role next post when I discuss p-chains.

```
In [5]: # Returns 0 is they two strings are not permutations of each other
        # Returns 1 if they are even permutations
        # Returns -1 if they are odd permutations

        function AreSame(a::String,b::String)
            if length(a) != length(b)
                error("Strings not same length in AreSame")
            end

            l=length(a)

            x=repmat(collect(1:l),1,l)
            y=transpose(x)-1
            z=(x+y-1).%l+1
            zp=z[end:-1:1,:]

            for jj in 1:l
                n=""
                m=""
                for ii in 1:l
                    n=string(n,a[z[ii,jj]])
                    m=string(m,a[zp[ii,jj]])
                end

                if b==n
                    return 1
                end
                if b==m
                    return -1
                end

            end

            return 0
        end
```

```
Out[5]: AreSame (generic function with 1 method)
```

The function to create a complex is fairly straight forward, but I want to pull out one part that is simple, but not particularly readable. I use a loop and a modulo to remove one index from a string.
Here's that unreadable chunk:

```
In [6]: d_new_s=3
        d_old_s=4
        for ii in 1:d_old_s
            println(   collect(ii:(ii+d_new_s-1)).%d_old_s  +1 )
        end
```

```
[2, 3, 4]
[3, 4, 1]
[4, 1, 2]
[1, 2, 3]
```

And now the large function to create a simplicial complex.

```
In [7]: # sc = simplecial complex
        # s = simplex

        function CreateComplex(starter::Array{simplex})

            n0=length(starter) # number simplices at a top dimension
            d0=length(starter[1].s) #top dimension

            sc=Array{Array{simplex}}(d0)
            sc[d0]=starter;

            n_new_s=d0*n0 # the number of simplices in the next row
            n_old_s=n0    # the number of simplices in the last row

            d_new_s=d0-1; # the next dimension
            d_old_s=d0;   # the last dimension

            for kk=(d0-1):-1:1

                s_last=sc[kk+1]
                s_next=Array{simplex}(n_new_s)

                for jj in 1:n_old_s
                    for ii in 1:d_old_s

                        s_next[d_old_s*(jj-1)+ii]=
                                simplex(s_last[jj].s[(collect(ii:(ii+d_new_s-1))).%d_old_s+1])

                        end
                    end

                    sc[kk]=s_next;

                    n_old_s=n_new_s
                    n_new_s=n_new_s*d_new_s

                    d_old_s=d_new_s;
                    d_new_s=d_new_s-1;
                end
                return SimplexComplex(d0,sc)
            end
```

```
Out[7]: CreateComplex (generic function with 1 method)
```

I decided to break creating the complex into two functions for the sake of readability. The next function `SimplifyComplex` gets rid of repeated simplices.

We have to be careful when deleting elements of an array if we are still working with it. While removing elements, a `for` loop would be tricky, because my loop changes in size each time. Instead, I use a `while` control loop. Also, I delete matches from the end of the array to the beginning to disturb the fewest elements.

```
In [8]: function SimplifyComplex(sc::SimplexComplex)
            sc=SimplexComplex(sc.d,copy(sc.s))

            for kk in 1:(sc.d-1)
                sc_now=sc.s[kk]

                ii=1
                while ii<length(sc_now)
                    for jj in length(sc_now):-1:(ii+1)
                        if 0 != AreSame(sc_now[jj].s,sc_now[ii].s)
                           deleteat!(sc_now,jj)
                        end
                    end
                    ii+=1
                end
            end

            return sc
        end
```

```
Out[8]: SimplifyComplex (generic function with 1 method)
```

```
In [9]: a=simplex("abc")
```

```
Out[9]: simplex("abc")

In [10]: b=simplex("abd")

Out[10]: simplex("abd")

In [11]: sc=CreateComplex([a,b])

Out[11]: SimplexComplex(3, Array{simplex,N} where N[simplex[simplex("c"), simplex("b"), simplex(

In [12]: sc2=SimplifyComplex(sc)

Out[12]: SimplexComplex(3, Array{simplex,N} where N[simplex[simplex("c"), simplex("b"), simplex(
```

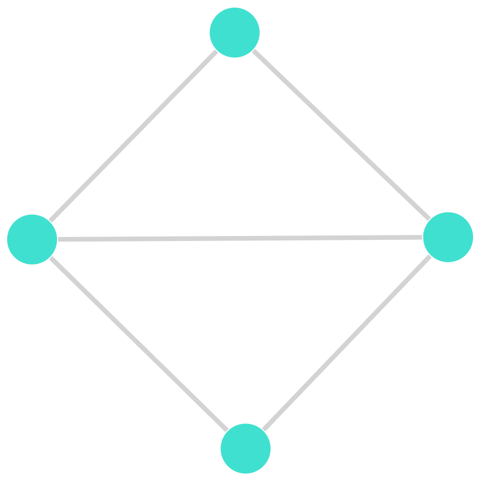
While my HTML output is much easier to read than the original way Julia spit out a simplicial complex, we can still do better.

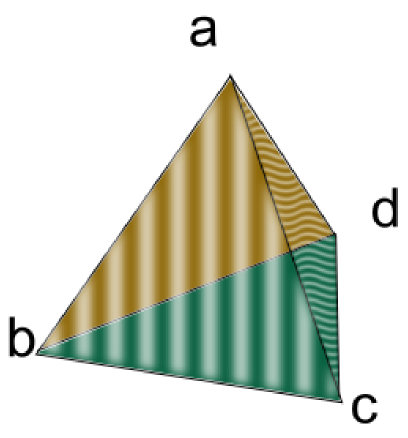
Simplicial complexes are just graphs with additional stuff on top, so we can use some of Julia's graph packages to look at the simplicial complexes.

The nodes are 0-simplices, and edges are 1-simplices.

We can't call the nodes by `"a"`; we have to call them by `1`. Therefore, I create a dictionary from `"a"` to 1.

```
In [13]: using LightGraphs
         using GraphPlot
         using Compose

In [14]: function makegraph(sc::SimplexComplex)
             nodes_num=Dict()
             for ii in 1:length(sc.s[1])
                nodes_num[sc.s[1][ii].s] = ii
             end

             g=Graph(length(sc.s[1]))

             for ii in 1:length(sc.s[2])

                 site1=nodes_num[sc.s[2][ii].s[1:1]]
                 site2=nodes_num[sc.s[2][ii].s[2:2]]

                 add_edge!(g,site1,site2)
             end

             return g
         end

Out[14]: makegraph (generic function with 1 method)

In [ ]: g=makegraph(sc2)
        gplot(g, nodelabel=sc.s[1])
```





a) Graph generated from the simplicial complex 'sc'. b). A sphere S^2 is the four 2-simplices that border a tetrahedron.

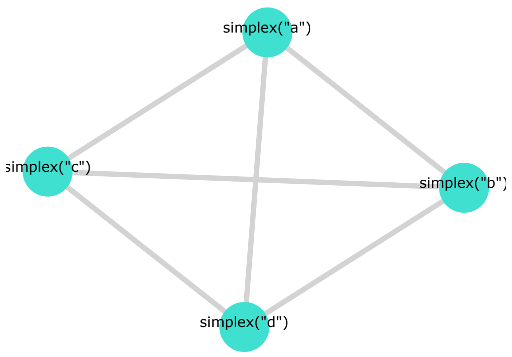
1.1 Sphere S^2

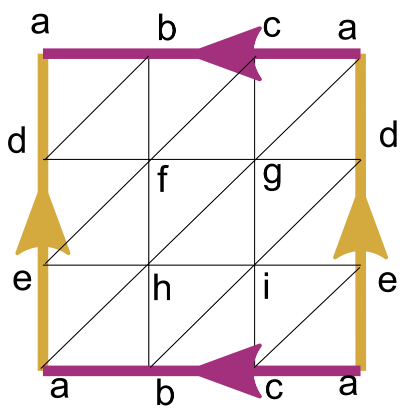
That object is fairly boring, topologically speaking.

Let's crank it up to something to something slightly more complicated, the sphere S^2.

I drew up the simplicial complex of a sphere in Inkscape:

```
In [15]: sphere=["abc","acd","abd","bdc"]

Out[15]: 4-element Array{String,1}:
          "abc"
          "acd"
          "abd"
          "bdc"

In [16]: starter=simplex[]

         for s in sphere
             push!(starter,simplex(s))
         end
         S2=CreateComplex(starter)
         S2=SimplifyComplex(S2)

Out[16]: SimplexComplex(3, Array{simplex,N} where N[simplex[simplex("c"), simplex("b"), simplex(

In [20]: gS2=makegraph(S2)
         gplot(gS2, nodelabel=S2.s[1])

   Out[20]:
```





b) The decomposition of a torus.

1.2 Torus T^2

Similar to a sphere, we also have a torus, T^2.

"Eh?" you might wonder...

Why does a torus have so many triangles?

We can't just write our simplices as "Go from a to a then over to a then back to a". Or aaa.

We have to divide the torus into enough points to uniquely specify each 2-simplex, 1-simplex, and 0-simplex.

We are now in the regime of having too many simplices to even write out every 2-simplex easily.

To aid me in this task, I wrote out just the strings and put them in an array. I separate all the strings by their location, so I won't get lost as I'm writing them out.

```
In [15]: col1=["abd","bdf","dfe","efh","efa","ahb"]
         col2=["bcf","cfg","fgh","ghi","hib","ibc"]
         col3=["cag","agd","gdi","die","iec","eca"];
```

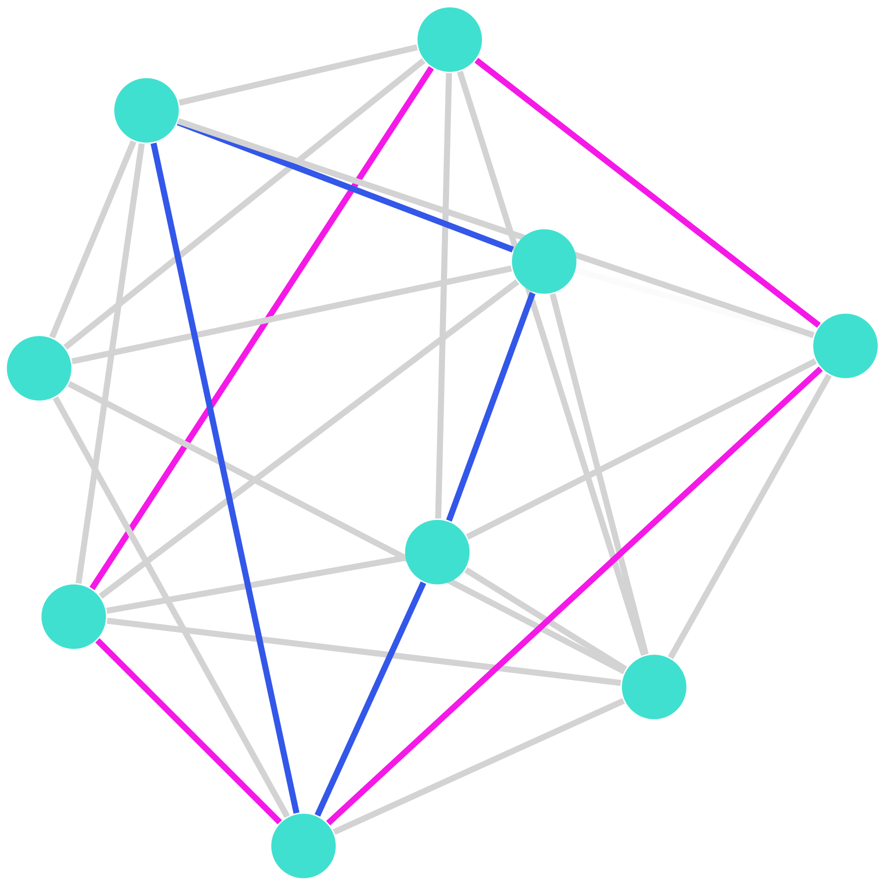
So I've managed to get a list of all the 2-simplices I need, but they are not in the correct form. So I have the next little bit of code to change my three Arrays of strings to one Array of simplices.

```
In [16]: starter=simplex[]

         for s in [col1; col2; col3]
             push!(starter,simplex(s))
         end
         T2=CreateComplex(starter)
         T2=SimplifyComplex(T2)
```

```
Out[16]: SimplexComplex(3, Array{simplex,N} where N[simplex[simplex("d"), simplex("b"), simplex(
```

I think this graph is even worse than the sphere...

```
In [ ]: gT2=makegraph(T2)
        gplot(gT2, nodelabel=T2.s[1])
```



A highlighted version of a torus graph.

I took the plot automatically generated for the torus, and played around with it to try and make sense of the graph. I found two distinct loops that can't be deformed away. This confirms to me that the graph indeed is a torus, even if it looks like a bunch of goble-de-gook.

Try and find a loop that can't be deformed away on S^2. But don't try too long; it's impossible.

Why did I look for "distinct loops that can't be deformed away"? I'll cover that more next time with p-chains, cycles, boundaries, and homology groups.

I highly recommend Allen Hatcher's book on Algebraic Topology for learning this topic. Bonus, it's availible free online at https://www.math.cornell.edu/~hatcher/AT/AT.pdf

```
In [ ]:
```

[1] Stone, Michael, and Paul Goldbart. "Mathematics for Physics." New York: Pimander-Casaubon (2009). [2] Hatcher, Allen. Algebraic topology. , 2002.