Progress report
Project IV: Computational Topology,
Michaelmas Term

Ben Napier

In tracking my progress through Michaelmas term, we can divide the work into there distinct sections:

(i) an exploration of the generalisation of graph properties to simplicial complexes;

(ii) an introduction into the computational methods used to calculate simplicial homology; and

(iii) the implementation of such methods in Python.

Invariants of simplicial complexes

In my study of computer science, I was introduced to the concept of graph properties and wondered if there exists generalisation of some common invariants to simplicial complexes.

We first introduce the *chromatic polynomial*.

Definition 1. Let $G = (V, E)$ be a graph. The *chromatic polynomial* of G, $\chi_G \in \mathbb{Z}[\lambda]$, counts the number of its proper λ-colourings.

We take for granted the fact that such a polynomial exists and is unique. Chromatic polynomials do not uniquely identify graphs, but it clearly encodes some information on the number of colourings.

How may we generalise the chromatic polynomial to simplicial complexes? Well, we may define it only on its 1-skeleton, but then we lose all the higher dimensional information and we haven't really generalised anything. May we abstract the notion of a colouring to a higher-dimensional analogue? We now introduce a lemma-generalisation of the chromatic polynomial on graphs.

Definition 2 (Tutte polynomial)**.** Let $G = (V, E)$ be a graph. For $A \subset E$, denote $k(A)$ as the number of connected components on the subgraph (V, A). We define the *Tutte polynomial* of G, $T_G \in \mathbb{Z}[x, y]$ as

$$T_G(x, y) = \sum_{A \subset E} (x-1)^{k(A)-k(E)}(y-1)^{k(A)+|A|-|V|}.$$

This polynomial can shown to specialise to the chromatic polynomial as follows:

$$\chi_G(k) = (-1)^{|V|-k(G)}\lambda^{k(G)}T_G(1-\lambda, 0).$$

The Tutte polynomial encodes much more information than the chromatic polynomial. For example, for a graph $G = (V, E)$, we have the following points:

(i) $T_G(1, 1)$ counts the number of spanning forests;

(ii) $T_G(2, 2) = 2^{|E|}$;

(iii) $T_G(2, 0)$ counts the number of acyclic orientations; and

(iv) $T_G(1,2)$ counts the number of spanning subgraphs.

Now how can we generalise the Tutte polynomial to simplicial complexes? Well, note that in our definition, the Tutte polynomial is a summation over each spanning subgraph, which is generalised with the notion of a *subcomplex* (details to follow). We also claim that the exponents of each term may be reframed using *homology*, something that we may also generalise to simplicial complexes.

Firstly, for a graph $G = (V, E)$, $|H_1(G)| = k(E)+|E|-|V|$. Similarly, $|H_0(G)| = k(E)$. Thus, we may define the Tutte polynomial on a graph as

$$T_G(x,y) = \sum_{G' \subset G} (x-1)^{|H_0(G')|-|H_0(G)|}(y-1)^{|H_1(G')|}$$

where the summation is taken over the *spanning* subgraphs of $G = (V, E)$ (that is, G' is of the form (V, A), $A \subset E$).

It may be clear how we may generalise, but we lack an analogue for spanning subgraph. $L \subset K^{(n)}$ is a spanning n-dimensional subcomplex of a simplicial complex K of dimension $\geq n$ if the $(n-1)$-skeleton of L and K coincide.

For a simplicial complex K, we define the n-Tutte polynomial as

$$T_{K,n}(x,y) = \sum_{L \subset K^{(n)}} (x-1)^{|H_{n-1}(L)|-|H_{n-1}(K)|}(y-1)^{|H_n(L)|}$$

and indeed this generalises as

$$T_{G,1}(x,y) = T_G(x,y).$$

Thus, we may define the n-chromatic polynomial of a simplicial complex K as

$$\chi_{K,n}(k) = (-1)^{|C_{n-1}(K)|-|H_{n-1}(K)|}\lambda^{|H_{n-1}(K)|}T_{K,n}(1-\lambda, 0)$$

where $C_i(K)$ denotes the ith chain group K.

Computational methods for simplicial homology

First, we introduce the problem formally.

Problem 1 (BettiNumbers)**.** Let K be a n-dimensional simplicial complex. Calculate the rank of all the non-trivial homology groups of K; that is, the non-zero Betti numbers.

Before approaching this, let us fix a representation of a (abstract) simplicial complex. Let K be a n-dimensional simplicial complex, which we represent as

$$K = (K_0, K_1, \dots, K_n)$$

where K_i is the set of i-simplices in K. K_0 denotes the vertex set of K, and K_i for $i \in \{1, \dots, n\}$ denotes a i-tuple of vertices. For example, we represent a 2-simplex (a triangle) as

$$K = (\{u, v, w\}, \{(u, v), (v, w), (w, u)\}, \{(u, v, w)\}).$$

Note the slight abuse of notation here, for the vertex set we may omit the brackets on the 1-tuple ($u = (u)$). It is clear to see how we may use standard data structures to store K.

Observe that K_i represents the ith chain group, upon which we have a boundary operator defined as

$$\partial_i : K_i \to K_{i-1}, \qquad (u_0, \ldots, u_n) \mapsto \sum_{i=0}^{n} (-1)^i (u_0, \ldots, \hat{u}_i, \ldots, u_n).$$

If we fix the canonical basis for K_i, for each map ∂_i we can construct a matrix representation $D_i \in M_{|K_{i-1}| \times |K_i|}(\{1, -1, 0\})$ in which we can calculate the rank of the image and rank of the kernel by transforming the D_i's to SNF, and thus the ranks of the homology groups.

So the major computation here is to calculate the Smith normal form for a matrix.

Problem 2 (SmithNormalForm)**.** Let M be a $m \times n$ integer matrix. Calculate the Smith normal form of M.

Before introduce our first approach, we need some definitions. We extend the function $\gcd : \mathbb{Z}^2 \to \mathbb{Z}$ to any set of finite size ≥ 2 with $\gcd(\{a_1, \ldots, a_n\}) = \gcd(a_1, \gcd(\{a_2, \ldots, a_n\})) = \ldots = \gcd(a_1, \gcd(a_2, \ldots, \gcd(a_{n-1}, a_n)))$ as one may expect.

Definition 3 (ith determinant divisor)**.** Let M be a $m \times n$ matrix over a commutative ring and denote M_i as the set of order-i minors of M. We define the *ith determinant divisor* of M, denoted $d_i(M)$, as the greatest common divisor of all $i \times i$ minors of M (we trivially define $d_0(M) := 1$). Formally, $d_i(M) = \gcd M_i$.

For brevity, we take for granted the existence and uniqueness (the diagonal entries are unique up to multiplication by a unit, in our case ± 1) of the Smith normal form of any integer matrix M.

Theorem 1. *Let M be a $m \times n$ matrix and $S = (s_{ij})$ be a Smith normal form of M with $\alpha_i = s_{ii}$ for $i \in \{1, \ldots, r\}$ the non-zero diagonal entries of S (so $S_{ij} = 0$ if not $i = j \in \{1, \ldots, r\}$). Then $\alpha_i = \frac{d_i(M)}{d_{i-1}(M)}$ (up to multiplication by a unit).*

Thus, we now need a way to calculate the ith determinant divisor of a matrix. For a $m \times n$ matrix M, there are $\binom{m}{m-i} \times \binom{n}{n-i}$ different $i \times i$ minors, each of which can be computed in $O(i^3)$ time giving a total time for the computation of d_i as $O\left(\binom{m}{i}\binom{n}{i} i^3\right)$ and thus for all d_i,

$$O\left(\sum_{i=1}^{\min\{m,n\}} \binom{m}{i}\binom{n}{i} i^3 \right) = O\left(\sum_{i=1}^{\min\{m,n\}} O\left(i^2 \sqrt{i} \left(\frac{mni}{e} \right)^i \right) \right)$$

using Stirling's approximation, which is indeed not a very useful upper bound. One may reduce this bound by considering a dynamic programming approach along with the following fact. For a $n \times n$ matrix $M = (m_{ij})$, denote M_{ij} denote the minor obtained from deleting the ith row and the jth column. Then

$$\det(M) = \sum_{i=1}^{n} (-1)^{i+1} m_{ij} M_{ij},$$

however, it is unlikely this method would outperform the following method.

Now we introduce the standard Smith normal form algorithm. We denote $\operatorname{snf}(M)$ the set of Smith normal forms of integer matrix M. We will take the following for granted.

Theorem 2. *Let M be an $m \times n$ integer matrix, R a $m \times m$ invertible integer matrix, and C a $n \times n$ invertible integer matrix. Then* $\operatorname{snf}(M) = \operatorname{snf}(RM) = \operatorname{snf}(MC)$.

Note that the inverse of an integer matrix is an integer matrix if and only if the determinant of said matrix is ± 1.

This theorem justifies much of the techniques we can use within the standard algorithm. We can apply operations to our matrix M without changing $\operatorname{snf}(M)$, as long as our operations are invertible linear maps. So what kind of operations does this lend us to use? Well, interchanging two rows or columns is clearly an invertible linear map. But we (generally) cannot multiply any row or column by a non-zero element. Indeed, consider the operation of multiplying the ith column by $k \in \mathbb{Z} \setminus \{0\}$. This operation has the following elementary matrix:

$$\begin{pmatrix}
1 & 0 & \dots & 0 & 0 & 0 & \dots & 0 & 0 \\
0 & 1 & \dots & 0 & 0 & 0 & \dots & 0 & 0 \\
\vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\
0 & 0 & \dots & 1 & 0 & 0 & \dots & 0 & 0 \\
0 & 0 & \dots & 0 & k & 0 & \dots & 0 & 0 \\
0 & 0 & \dots & 0 & 0 & 1 & \dots & 0 & 0 \\
\vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\
0 & 0 & \dots & 0 & 0 & 0 & \dots & 1 & 0 \\
0 & 0 & \dots & 0 & 0 & 0 & \dots & 0 & 1
\end{pmatrix}$$

which has determinant k. Thus, unless $k \in \{1, -1\}$, this operation is not invertible. We can however replace two rows with certain linear combinations of each other. First, consider the the operation of replacing row i with a multiplied by row i sum b multiplied by row j and replacing row j with c multiplied by row i

sum d multiplied by row j. This operation may be represented as follows:

$$\begin{pmatrix}
1 & 0 & \dots & 0 & 0 & 0 & \dots & 0 & 0 & 0 & \dots & 0 & 0 \\
0 & 1 & \dots & 0 & 0 & 0 & \dots & 0 & 0 & 0 & \dots & 0 & 0 \\
\vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\
0 & 0 & \dots & 1 & 0 & 0 & \dots & 0 & 0 & 0 & \dots & 0 & 0 \\
0 & 0 & \dots & 0 & a & 0 & \dots & 0 & b & 0 & \dots & 0 & 0 \\
0 & 0 & \dots & 0 & 0 & 1 & \dots & 0 & 0 & 0 & \dots & 0 & 0 \\
\vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\
0 & 0 & \dots & 0 & 0 & 0 & \dots & 1 & 0 & 0 & \dots & 0 & 0 \\
0 & 0 & \dots & 0 & c & 0 & \dots & 0 & d & 0 & \dots & 0 & 0 \\
0 & 0 & \dots & 0 & 0 & 0 & \dots & 0 & 0 & 1 & \dots & 0 & 0 \\
\vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\
0 & 0 & \dots & 0 & 0 & 0 & \dots & 0 & 0 & 0 & \dots & 1 & 0 \\
0 & 0 & \dots & 0 & 0 & 0 & \dots & 0 & 0 & 0 & \dots & 0 & 1
\end{pmatrix}$$

which has determinant $ad-bc$ (obtained through repeated expansion along rows that are standard unit vectors). Now suppose we have entries (i,t) and (j,t) such that $m_{it} \nmid m_{jt}$, and set $\beta = \gcd(m_{it}, m_{jt})$. Then there is $a, b \in \mathbb{Z}$ such that $am_{it} + bm_{jt} = \beta$ (Bézout's identity). We then pick (rather purposely) $c = -m_{jt}/\beta$ and $d = m_{it}/\beta$ and note that

$$ad - bc = \frac{am_{it} + bm_{jt}}{\beta} = 1,$$

thus giving an invertible operation. This may seem like a particularly obscure operation, but will come in handy in the algorithm. Finally, if we have entries (i,t) and (j,t) such that $m_{it} \mid m_{jt}$, then we can set $a = d = 1$, $b = 0$, and $c = -m_{jt}/m_{it}$ to get an another invertible operation.

Algorithm 1 (Standard Smith normal form algorithm)**.**
Input: let $M = (m_{ij})$ be a $m \times n$ integer matrix.

(i) Perform the necessary interchange operations to obtain $m_{11} \neq 0$. If this is not possible, M is empty and we go to (vii).

(ii) For each $i \in (2, \dots, m)$, do the following.

  (a) If $m_{11} \mid m_{i1}$, continue to the next i.

  (b) Otherwise, perform the row operation:

$$aR_1 + bR_i \to R_1,$$
$$-\frac{m_{i1}}{\beta}R_1 + \frac{m_{11}}{\beta}R_i \to R_i,$$

  where $\beta = \gcd(m_{11}, m_{i1})$ and a and b are chosen such that $am_{11} + bm_{i1} = \beta$ (existence by Bézout's identity).

(iii) For each $i \in (2, \dots, m)$, do the following.

  (a) If $m_{i1} = 0$, continue to the next i.

(b) Otherwise, perform the row operation:

$$R_{i1} - \frac{m_{i1}}{m_{11}} R_{11} \to R_{i1}.$$

(iv) For each $i \in (2, \ldots, n)$, do the following.

(a) If $m_{11} \mid m_{1i}$, continue to the next i.

(b) Otherwise, perform the column operation:

$$\begin{aligned} aC_1 + bC_i &\to C_1, \\ -\frac{m_{1i}}{\beta} C_1 + \frac{m_{11}}{\beta} C_i &\to C_i, \end{aligned}$$

where $\beta = \gcd(m_{11}, m_{1i})$ and a and b are chosen such that $am_{11} + bm_{1i} = \beta$ (existence by Bézout's identity).

(v) For each $i \in (2, \ldots, n)$, do the following.

(a) If $m_{1i} = 0$, continue to the next i.

(b) Otherwise, perform the row operation:

$$C_{1i} - \frac{m_{1i}}{m_{11}} C_{11} \to C_{1i}.$$

(vi) If all entries of the first row and column except for m_{11} are zero, then we invoke this algorithm on the submatrix obtained by deleting the first column and row (unless this causes an empty matrix, then we continue). Otherwise, we go back to (ii).

(vii) If we are a submatrix invocation, we return. Otherwise, let $r = \arg\max_i \{a_{ii} : a_{ii} \neq 0\}$ and for each $i \in \{r, \ldots, 2\}$ do the following.

(a) Perform the column operation: $C_{i-1} + C_i \to C_{i-1}$.

(b) Mimic the elimination process from (ii) to (v) on the $(i-1)$th row and column.

There is some needed support to this algorithm, which is the succeeding commentary.

An assumption underpinning this algorithm is that the matrix M is stored a single piece of memory, and when we recursively invoke the function we are modifying the same piece of memory every time.

In steps (ii)(b) and (iv)(b), the method of which to choose a and b (called *Bézout coefficients*) is not specified. With the ring of integers (and indeed with any Euclidean domain), we may use the extended Euclidean algorithm, although this implementation was omitted mainly to keep the core ideas at front, as this algorithm may be (slightly) adapted for a general principal ideal domain.

We have a conditional branch at (vi), linking back to (ii). Thus we must assert that this loop will eventually resolve, and not loop forever. For $a \in \mathbb{Z} \setminus \{0\}$, we define $\delta(a)$ to be the number of prime factors of a, which we know to exist and are unique in the ring of integers. Let a be the value of m_{11} before executing a single iteration of (ii) and b be the value of m_{11} after executing, and also assume that through execution a row operation was performed (that is, m_{11} did not divide the entry it was being compared against). We note that b is the greatest common divisor of a and another entry, thus $b \mid a$. We assert that $a \neq b$ as $a \nmid b$ by assumption. Thus we must have that $\delta(b) < \delta(a)$. Notice that each application of (ii) must reduce the value of $\delta(m_{11})$, and so the process must eventually to lead to m_{11} being the only non-zero entry in the first row and column. A similar argument to this can be applied to (vii)(b).

The invertability of the operations has largely been covered, bar (vii)(a). Given a column (or the corresponding row) operation $C_i + C_j \to C_i$, we construct the inverse operation $C_i - C_j \to C_i$.

The complexity of this algorithm is a topic for future work; it is not immediately obvious the number of times (ii) to (iv) must be executed in order to eliminate a row/column.

Implementation

As described in the last section, our input is a list $K = (K_0, \dots, K_n)$ where each K_i itself is a list of simplices (which themselves are lists of vertices), we denote K as `chain_groups` below. Our first step is to generate a representation of the boundary maps. We will do this by choosing the canonical basis and storing the matrix for each map as a two-dimensional array.

```
1  boundary_maps = [[0] * len(chain_groups[1])]
2  for i in range(1, len(chain_groups)):
3      chain_group = chain_groups[i]
4      boundary_map = []
5      for chain in chain_group:
6          boundary_map_col = [0] * len(chain_groups[i - 1])
7          for j in range(0, len(chain)):
8              v = j
9              v_boundary = chain[:j] + chain[j+1:]
10             boundary_map_col[chain_groups[i - 1].index(v_boundary)] = (-1) ** j
11         boundary_map.append(boundary_map_col)
12     boundary_maps.append(boundary_map)
```

We initially cover the trivial case of the boundary map $\partial_0 : K_0 \to 0$. Then at 2, we iterate over the rest of the chain groups in ascending order. At each $i \in (1, \dots, n)$, we construct its boundary matrix (with the canonical basis) and append it to our list of boundary maps, using the formula

$$\partial_i(v_0, \dots, v_i) = \sum_{j=0}^{n} (-1)^j (v_0, \dots, \hat{v}_j, \dots, v_i).$$

Next, we simply calculate the ranks of the images and kernels for each boundary map and thus the ranks of the homology groups.

```python
ranks = []
nullities = []
homology_ranks = []
for i in range(0, len(boundary_maps)):
    ranks.append(rank(boundary_maps[i]))
    nullities.append(len(chain_groups[i]) - ranks[len(ranks) - 1])
ranks.append(0)
for i in range(0, len(boundary_maps)):
    homology_ranks.append(nullities[i] - ranks[i + 1])
```

Here we make use of the `rank` function that we have not defined here, but discussed at depth in the preceding section.

We note that this script only calculates the ranks of the homology group, and the Smith normal form gives us excessive information (that is, the torsion subgroups). Thus there is an opportunity to reduce the complexity by using a different matrix reduction algorithm. In fact, we only need reduce rows and bring the matrix into *Hermite normal form*. Which, for a $m \times n$ matrix, can be done in $O(nm \log N)$ (where N is the number of bits required to store an entry of the matrix). Here we assume that the extended Euclidean algorithm can be executed in $O(\log N)$ time (more specifically, we can calculate the greatest common divisor and Bézout coefficients of $a, b \in \mathbb{Z}$ in time $O(\log \min\{a, b\})$).