# Homology From Drawings

Eric Rice
Math 273B

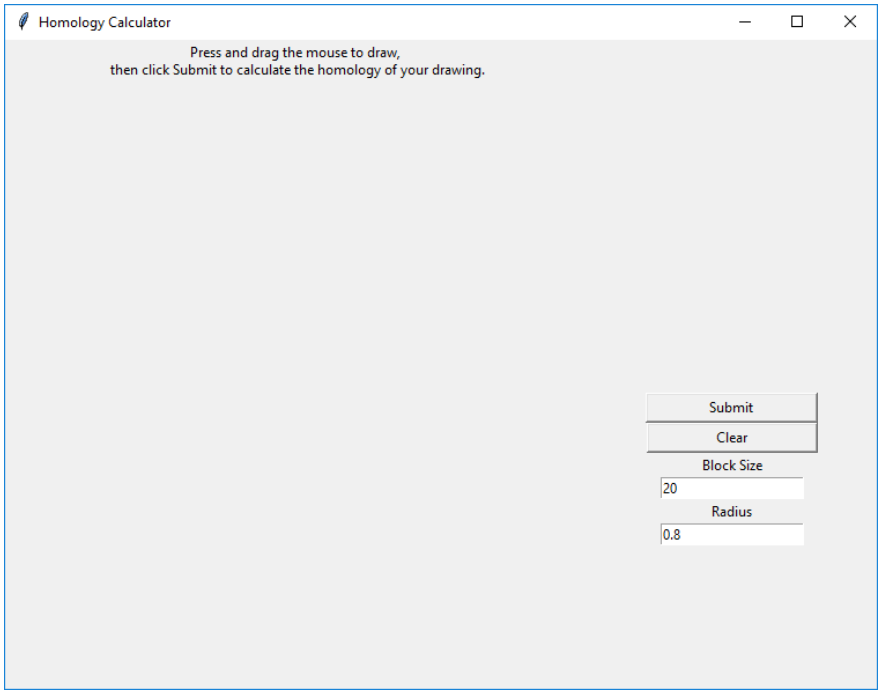
April 29, 2018

## 1 Abstract

In this project we design a program which will calculate the simplicial homology of a black and white drawing generated by user input. Though these drawings will all be two-dimensional, we plan to design our tools in such a way so that they apply to higher dimensional problems. All coding will be done in Python and source code will be given at the end of the written content.

## 2 The GUI

Upon execution of the program, the user is met with the following GUI (Graphical User Interface):




The left side of the GUI is the *canvas* or drawing area. The user can click and drag to create a drawing in this space. On the right, there are two buttons and two entry fields. The "Submit"

button will save the drawing and calculate its simplicial homology, which we will explain in the following sections. The "Clear" button deletes the drawing and resets the application to its initial state. The "Block Size" and "Radius" entry fields are parameters that influence the tolerance levels and accuracy of the calculations. These will be made precise in the construction of the simplicial complex.

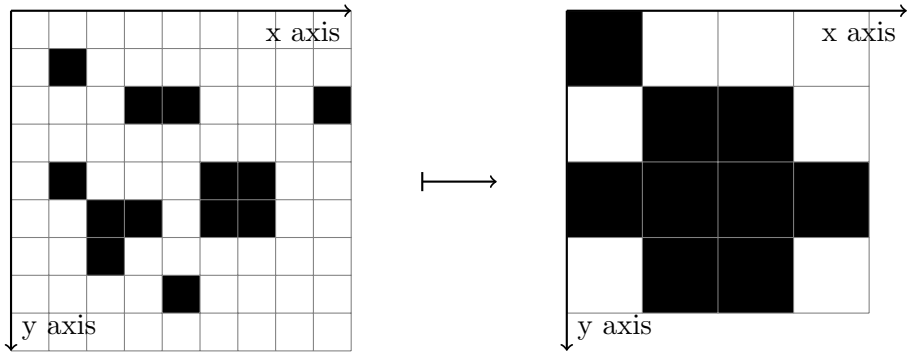
## 3 Constructing a Complex

The main topic of interest is the sequence of actions performed after clicking the "Submit" button. The first of such actions **getValsByCoord** saves the drawing as a PNG file and loads it into memory. With the drawing collected, we extract the pixel data as a two-dimensional array, where the pixel at coordinates $(x, y)$ has value 0 if it is black, and 255 if it is white. Our canvas is by default $380 \times 380$ pixels.

In the function **getReducedVertexSet** we reduce the original $380 \times 380$ canvas via a block decomposition. Given an integer parameter $b$ which specifies the side lengths of such a block, the code interprets the canvas as a $\frac{380}{b} \times \frac{380}{b}$ array, where the block with block coordinates $(x, y)$ has value 0 if any of its pixels are black, and 255 if all of its pixels are white. The coordinates of blocks with value 0 are saved and returned by the function as a vertex set $S$. Note that if $b$ does not divide 380, then the remainder will be thrown out, leading to a possible loss of data near the edges of the canvas. We give an abbreviated excerpt of this function from the source code below.

```
def getReducedVertexSet(vals, b):
  """Given a rectangular 2D array of pixel values vals
  and an integer b, decomposes vals into b x b
  blocks, ignoring any remainder.  Returns a list S
  of block coordinates of blocks which contain at
  least one black pixel."""

  h = len(vals) // b      # height in block coords
  w = len(vals[0]) // b   # width in block coords

  S = []
  for y in range(0, h):
    for x in range(0, w):
      for s in range(0, b):
        if ((x, y),) in S:
          break
        else:
          for t in range(0, b):
            if vals[y*b + s][x*b + t] == 0:
              S += [((x, y),)]  # tuple of tuple for consistency
              break             # with higher dimensions

  return S
```

This decomposition increases speed of the homology calculations as $b$ increases, but also reduces the accuracy as a result. We now present an example where we take the $9 \times 9$ canvas on the left and reduce it to the $4 \times 4$ canvas on the right with a block size of $b = 2$.




As the name implies, the reduced canvas is exactly what we will use as the vertex set for a simplicial complex. From the block values we construct a vertex set $S$ of tuples, each containing the coordinates of a block with value 0 (i.e., a block which is black). Each vertex is a tuple of ordered pairs for consistency with how we will construct higher order simplices (which will be tuples of multiple ordered pairs).

Given a radius $r$ we construct a Čech complex $K$ from $S$. Recall that as a set, we have

$$K = \check{\text{C}}\text{ech}(S, r) = \left\{ \sigma \subseteq S : \bigcup_{u \in \sigma} \overline{B_r(u)} \neq \emptyset \right\}$$

The elements of $K$ are referred to as *simplices*, and each $\sigma \in K$ is a *simplex*. We say that the simplex $\sigma$ has dimension $p$ if $\sigma \subset S$ contains $p + 1$ vertices. In the application, $K$ is constructed as a class in Python, which essentially creates a new object type which has callable variables and functions (methods). Of interest is the attribute

`K.complex`

Here, `K.complex` is an ordered list of all of the simplices in $K$, subdivided by their dimension. This list is created upon initialization of an instance of the class. In the following, for $p > 0$, we check all combinations of $p$ points from our vertex set $S$ and add the set of these points as a $p$-simplex of $K$ if all of the points are within Euclidean distance $2r$ from each other.

```
class CechComplex():
  """Given a vertex set S of points in R^n (where each point
  is contained in a tuple of length 1) and a radius r,
  returns the Cech complex K of S where:
  - K.complex is a list of lists
  - K.complex[0] = S is the collection of 0-simplices
  - K.complex[1] = all pairs of points in S within Euclidean
  distance 2r from each other; these are
  the 1-simplices
  - K.complex[2] = all triples of points in S within Euclidean
```

```
distance 2r from each other; these are
the 2-simplices, etc."""

complex = None
radius = None

def __init__(self, S, r):
  K = [S, [], []]
  p = 0
  for j in range(0, len(K[p])):
    for k in range(j + 1, len(S)):          # Avoids double checking by not
      if dist(S[j][0], S[k][0]) <= 2*r:     # repeating dist(p, q) via dist(q, p)
        K[1] += [(S[j][0], S[k][0])]

  p = 1
  while len(K[p]) != 0:
    for j in range(0, len(K[p])):
      for k in range(0, len(S)):

        # Checks if the vertex S[k][0] is within 2*r of all of the
        # vertices of the simplex K[p][j]
        close = True
        for b in range(0, len(K[p][j])):
          if dist(S[k][0], K[p][j][b]) > 2*r:
            close = False
        if S[k][0] not in K[p][j] and close == True:

          # Checking for present tuples which are permutations
          check = True
          perms = list(permutations((S[k][0],) + K[p][j]))
          for t in range(0, len(perms)):
            if perms[t] in K[p + 1]:
              check = False

          if check == True:
            K[p + 1] += [(S[k][0],) + K[p][j]]

    p += 1
    K += [[]]

  self.complex = K
  self.radius = r
```

# 4 Homology

Now that we have constructed the simplices in $K$, we turn our attention to the chain groups generated by these simplices (with modulo 2 coefficients). Let $n_p$ be the number of $p$-dimensional simplices in $K$ and order these simplices $\sigma_1, \sigma_2, \ldots, \sigma_{n_p}$. Recall that the chain group of dimension $p \geq 0$ is as a set:

$$C_p := \left\{ \sum_{i=1}^{n_p} \alpha_i \sigma_i : \alpha_i \in \mathbb{Z}_2 \right\}$$

We also define $C_{-1} = \{0\}$. Each $C_p$ is a group under addition modulo 2. We connect these groups via the boundary homomorphism

$$\partial_p : C_p \to C_{p-1}$$

Because this map respects the operations of the chain groups, it suffices to define it under their generators (simplices). We write $\tau \leq \sigma$ if $\tau$ is a *face* of $\sigma$. That is, $\tau \in K$ and $\tau \subset \sigma$.

$$\partial_p \sigma := \sum_{\tau \leq \sigma} \tau, \quad \dim(\sigma) = p,\ \dim(\tau) = p - 1 \tag{1}$$

From here, we define the *cycle* and *boundary subgroups* of $C_p$ denoted $Z_p$ and $B_p$, respectively.

$$Z_p := \{c \in C_p : \partial_p c = 0\} = \ker(\partial_p), \quad B_p := \{c \in C_p : c = \partial_{p+1} k,\ k \in C_{p+1}\} = \mathrm{im}(\partial_{p+1})$$

Applying the Fundamental Lemma of Homology $\partial_p \partial_{p+1} \equiv 0$ [1, IV.1 page 81] shows that $B_p \subset Z_p$. Their quotient gives the *homology group* of dimension $p$, the rank of which is the $p$th *Betti number*:

$$H_p := Z_p / B_p, \quad \beta_p := \mathrm{rank}(H_p)$$

In our application we really are only interested in the calculation of $\beta_p$ for all $p$, as these ranks give an interpretation of the number of $p$-dimensional "holes" of our complex. In particular, the first Betti number $\beta_1$ counts the number of (non-homotopic, non-contractible) "closed loops" in our drawing.

We will see that the Betti numbers of our complex $K$ are encoded in the matrix of its boundary transformation $\partial_p$, where $p \geq 1$. As usual, the columns of such a matrix representation are determined by a choice of ordered basis in both the domain and codomain. We have defined each chain group $C_p$ so that the $p$-dimensional simplices $\sigma_1, \sigma_2, \ldots, \sigma_{n_p}$ are its generators. As such, the $j$th column of $\partial_p$ in this "simplex basis" is the coefficients of $\partial_p \sigma_j$ in the "simplex basis" $\tau_1, \tau_2, \ldots, \tau_{n_{p-1}}$ of $C_{p-1}$. As such, it follows from (1) that we represent $\partial_p$ as the $n_{p-1} \times n_p$ matrix whose $(i, j)$th-entry is

$$\partial_p[i, j] = \begin{cases} 1, & \text{if } \tau_i \leq \sigma_j \\ 0, & \text{otherwise.} \end{cases}$$

Using elementary row and column operations (which do not change the rank of a matrix) we generate and reduce the matrix $\partial_p$ to Smith Normal Form [1, IV.2] in the following method `K.bdryMatrix()`. The Smith Normal Form is always diagonal, which makes it easy to read off the ranks of its kernel ($Z_p$) and image ($B_p$).

```
def bdryMatrix(self):
  """Generates a list of the boundary matrices of the complex
  in Smith normal form, where the index of the list matches
  the dimension."""

  n = [len(self.complex[0])]   # ranks of chain groups, n_p
  bdry = [None]
  for p in range(1, len(self.complex)):
    n += [len(self.complex[p])]
    bdry += [np.zeros((n[p - 1], n[p]), dtype=int)]
    for i in range(0, n[p - 1]):
      for j in range(0, n[p]):
        face = True
        for k in range(0, len(self.complex[p - 1][i])):
          if self.complex[p - 1][i][k] not in self.complex[p][j]:
            face = False
        if face:
          bdry[p][i][j] = 1

  # Reduction to Smith normal form
  k = None
  ell = None
  for p in range(1, len(self.complex)):
    for x in range(0, min(n[p - 1], n[p])):
      for i in range(x, n[p - 1]):
        for j in range(x, n[p]):
          if bdry[p][i][j] == 1:
            k = i
            ell = j

      if k != None and ell != None:
        bdry[p][[x,k]] = bdry[p][[k,x]]           # swaps rows k and x
        bdry[p][:,[x,ell]] = bdry[p][:,[ell,x]]  # swaps columns ell and x

        for i in range(x + 1, n[p - 1]):
          if bdry[p][i][x] == 1:
            for m in range(0, n[p]):
              bdry[p][i][m] = (bdry[p][i][m] + bdry[p][x][m]) % 2
        for j in range(x + 1, n[p]):
          if bdry[p][x][j] == 1:
            for m in range(0, n[p - 1]):
              bdry[p][m][j] = (bdry[p][m][j] + bdry[p][m][x]) % 2

  return bdry
```

Below we define notation for the ranks of the cycle and boundary groups:

$$z_p := \text{rank}(Z_p), \quad b_p := \text{rank}(B_p)$$

Using the Smith Normal Form of $\partial_p$, it is clear that $z_p$ is simply the number of zero columns of the (diagonal) matrix, and $b_{p-1}$ is the number of nonzero columns. Because our matrix has $n_p$ columns, it follows that $z_p = n_p - b_{p-1}$ (a fact which also follows from the first isomorphism theorem of groups). Note that since $C_{-1} = \{0\}$ we have $b_{-1} = 0$ and thus $z_0 = n_0$. Finally, as a quotient, we extract the $p$th Betti number as

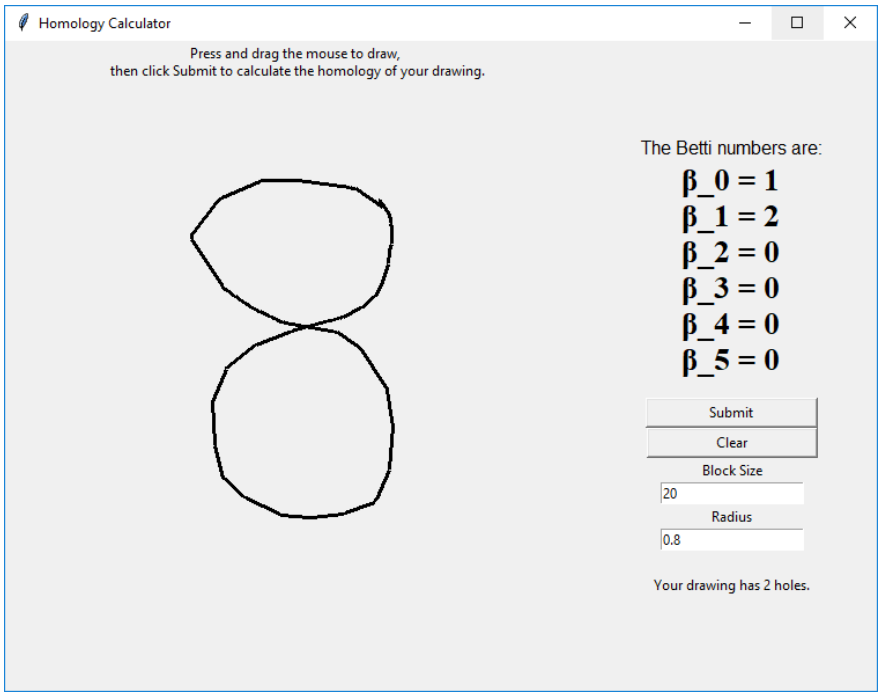
$$\beta_p = z_p - b_p$$

As a remark, above we require $b_p$ instead of $b_{p-1}$. Thus, we must compute the $p$th and $p+1$st boundary matrices in order to find the $p$th Betti number via this method. If $q$ is the highest dimension of any simplex in $K$, then we did not calculate the $q+1$st boundary matrix with the method `K.bdryMatrix()`. But, if we set $C_q = \{0\}$ then $\partial_q$ is the zero homomorphism, hence $b_q = 0$.

The method `K.betti()` is given below.

```
def betti(self):
  """Calculates the Betti numbers of the complex, where
  the output is a list whose index matches the dimension."""

  bdry = self.bdryMatrix()
  n = [] # ranks of chain groups, n_p in text
  for p in range(0, len(self.complex)):
    n += [len(self.complex[p])]

  # Building b_{p - 1} and z_p (the ranks of the boundary and
  # cycle groups) for all p that we have a boundary matrix.
  b = []
  z = [n[0]]
  for p in range(1, len(self.complex)):
    index = -1
    while index + 1 < min(n[p - 1], n[p]):
      if bdry[p][index + 1][index + 1] == 1:
        index += 1
      else:
        break

    b += [index + 1]   # Indices are offset from ranks by 1
    z += [n[p] - b[p - 1]]
  b += [0]   # Highest dimension boundary group is trivial

  beta = []
  for p in range(0, len(self.complex)):
    beta += [z[p] - b[p]]

  return beta
```

This is the function that determines the output when the "Submit" button is pressed from the GUI. An example of the output is shown below.




## 5 Connectedness and Homology of Each Component

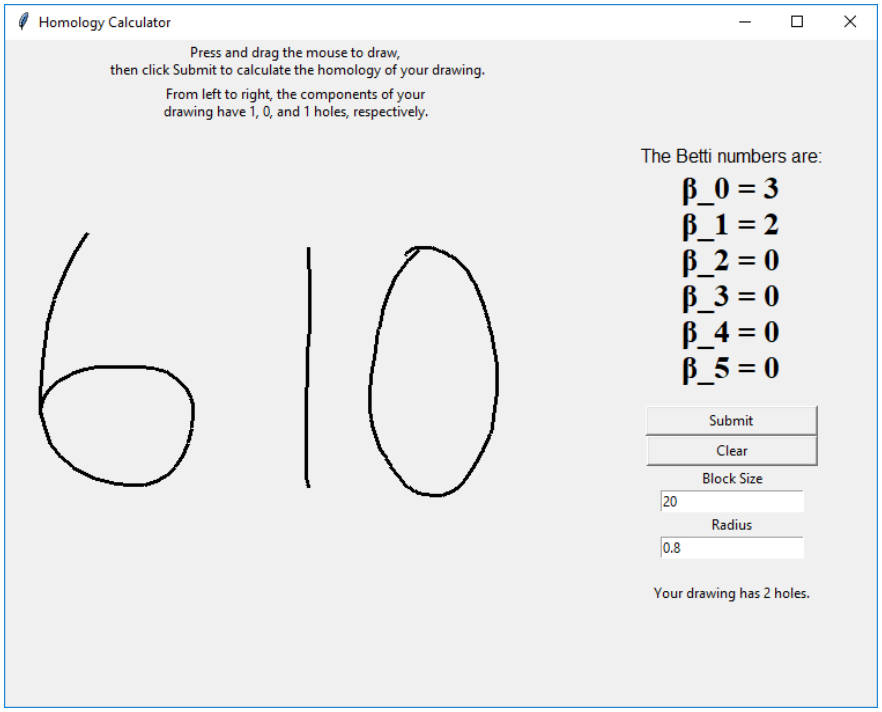
Recall that the zeroth Betti number counts the number of connected components of the complex $K$. When $\beta_0 > 1$, an additional script is run that separates $K$ and creates a connected Čech complex $C_i$ out of each component of $K$, where $i = 0, 1, \ldots, \beta_0 - 1$. The complexes $C_i$ are ordered from left to right as they appear on the canvas. The following code uses the pseudocode given in [1, Chapter I, page 6].

```
def splitByComponent(K):
  """Given a simplicial complex K, returns a list
  of complexes C of the connected components of K."""

  beta = K.betti()
  r = K.radius
  C = []

  # Identifies the connected components of K
  if beta[0] > 1:

    # Initializing a list of sets of vertices, which we will
    # manipulate so that two vertices belong in the same
    # set if and only if they are part of the same component.
```

```
V = []
for i in range(0, len(K.complex[0])):
  V += [set(K.complex[0][i])]

for i in range(0, len(K.complex[0])):
  for j in range(0, len(K.complex[0])):
    find_i = None
    find_j = None
    for k in range(0, len(V)):
      if K.complex[0][i][0] in V[k]:
        find_i = k
      if K.complex[0][j][0] in V[k]:
        find_j = k
    if ((K.complex[0][i][0], K.complex[0][j][0]) in K.complex[1]\
     or (K.complex[0][j][0], K.complex[0][i][0]) in K.complex[1])\
     and find_i != find_j:
      V[find_i] = V[find_i].union(V[find_j])
      del V[find_j]

# Creates a new complex for each component (given as
# a set of vertices)
S = []
for i in range(0, len(V)):
  S += [[]]
  V[i] = list(V[i])
  for j in range(0, len(V[i])):
    S[i] += [(V[i][j],)]   # Putting entries of V[i] in proper form
  C += [CechComplex(S[i], r)]

# Generates a list order of indices so that
# C[order[0]] is the leftmost complex (based on
# the smallest x-coordinate of a vertex) and
# C[order[i]] is further left than C[order[j]]
# if and only if i < j.
mins = []   # list to keep track of smallest x coordinates
for i in range(0, len(C)):
  leftest = C[i].complex[0][0][0][0]
  for k in range(0, len(C[i].complex[0])):
    if C[i].complex[0][k][0][0] < leftest:
      leftest = C[i].complex[0][k][0][0]
  mins += [leftest]

order = []
unordered = C
for i in range(0, len(C)):
```

```
        a = min(mins)
        order += [mins.index(a)]
        mins[mins.index(a)] = max(mins) + 1
        C[i] = unordered[order[i]]

    else:
      C = [K]

    return C
```

If $K$ is split into its components, the GUI is updated to give the number of holes in each connected component, from left to right, at the top of the window.




## 6 Python Code

The complete source code may be downloaded at

https://github.com/EricRiceUCSD/HomologyFromDrawings

## References


[1] Edelsbrunner and Harer, *Computational Topology: An Introduction*, 2010.

[2] Derek Banas, *Learn to Program 20 : TkInter Tutorial*, 2016. (YouTube Tutorial)
https://www.youtube.com/watch?v=-tbWoZSi3LU&index=2&list=PLTnEvfYjec06rD0rjt2bg-AGf_ZmRQ0ES&t=0s

[3] Derek Banas, *Learn to Program 25 Python Paint App*, 2016. (YouTube Tutorial)
https://www.youtube.com/watch?v=cE5kh02CkTY&index=1&list=PLTnEvfYjec06rD0rjt2bg-AGf_ZmRQ0ES&t=0s