Implementation: Fred's Vietoris Rips

Downloading

To download from Github, copy and paste the following into the console

```
if (!require(FredsVietorisRips)) install_github("ftkjr/FredsVietorisRips")
library(FredsVietorisRips)
```

Use

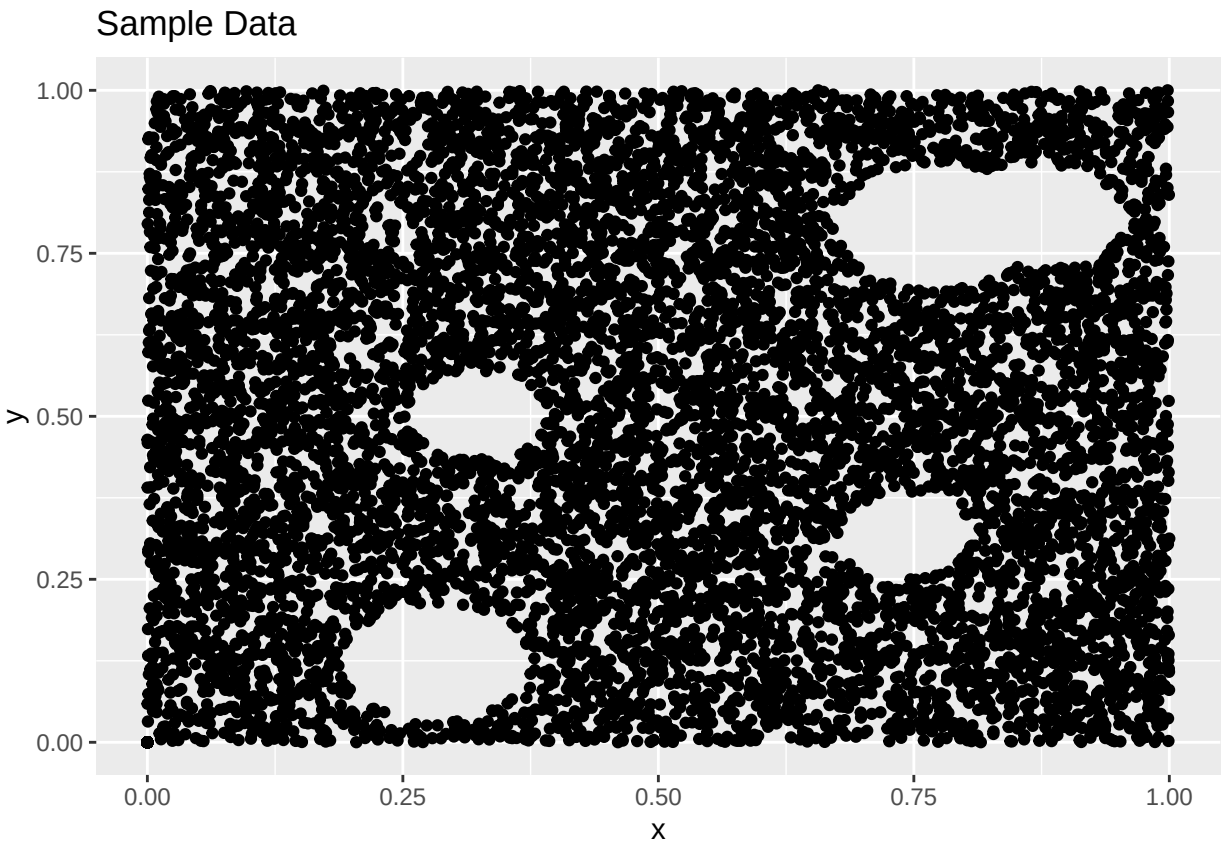
```
##### Packages ####
library(FredsVietorisRips)  # Our Very Own
library(magrittr)
library(dplyr)
library(ggplot2)
library(gridExtra)
library(reticulate)
```

```
##### Source our Python Code for Later Use ####
# All the code
filelist <- list.files("../python/")
for (f in filelist) {
  source_python(paste0("../python/", f))
}
```

Problem Statement

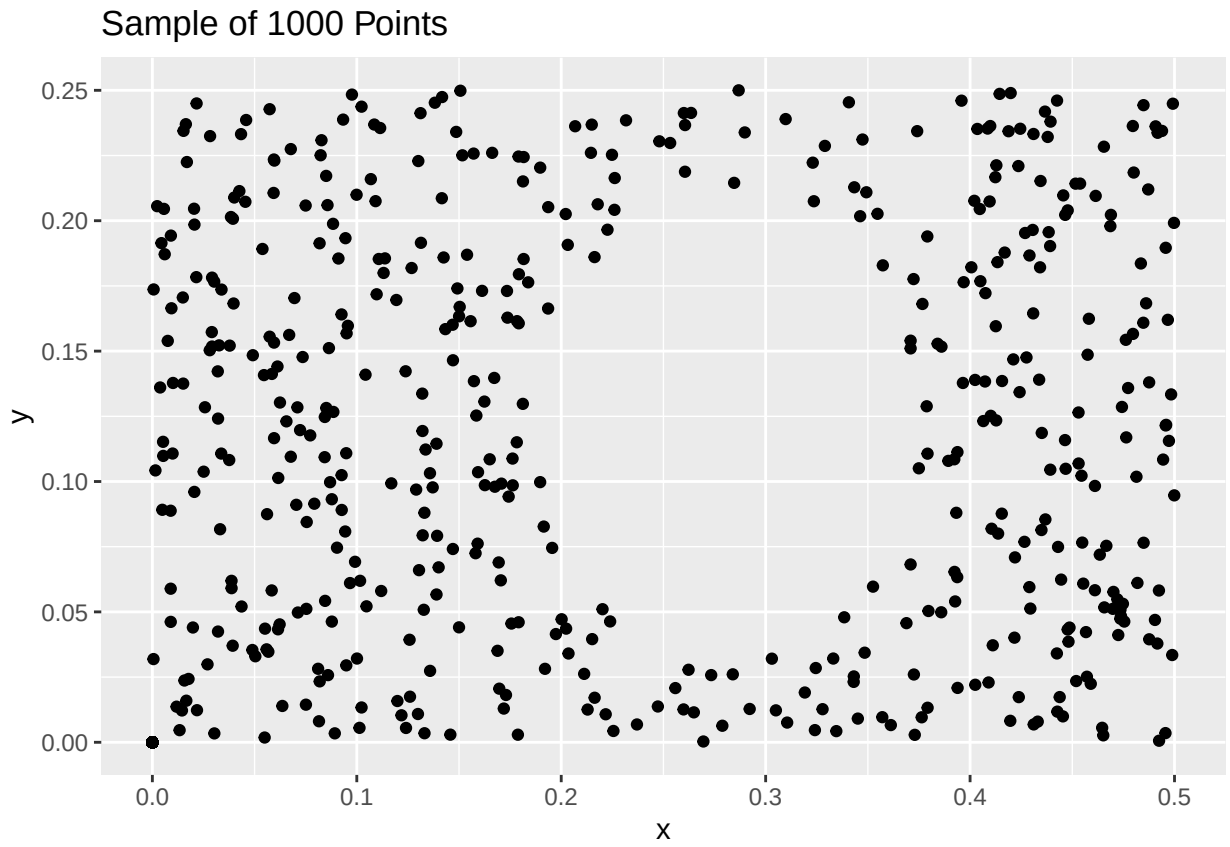
Given some set of data sampled from some unkown object or distribution, we seek to determine the homological features present in the data set.

```
##### Import Data ####
df_original <- read.csv("../data/Clark_Sample_data.csv", col.names = c("x", "y"))

##### Visualize Data ####
df_original %>%
  ggplot(aes(x, y)) +
  geom_point() +
  ggtitle("Sample Data")
```



From the Example Data above, we cut out a section (making sure to have at least one hole present), and pull a sample of 1000 points.

```
##### Take a section of the original data ####
sample_size <- 1000
df <- df_original[df_original$x < 0.5 & df_original$y < 0.25, ]
df <- sample_n(df, sample_size)

# Make it compatible with Pyton, start with zero
rownames(df) <- as.numeric(rownames(df)) - 1

##### Let's seee what we're working with ####
sample_chart <- df %>%
  ggplot(aes(x, y)) +
  geom_point() +
  ggtitle(paste("Sample of", sample_size, "Points"))

sample_chart
```



Once we have our sample data, we want to filter our data by a height function where we begin with a resolution r and an overlap, or gain g, and increment through so that for a resolution at some y_j we allocate to cover_j the (x, y) where $y \in [y_j, y_j + r)$, or

$$\text{Cover}_j = \{(x_i, y_i) | y_i \in [y_j, y_j + r)\}$$

our gain, g, determines the overlap in resolutions, so we have each subsequent cover determined by

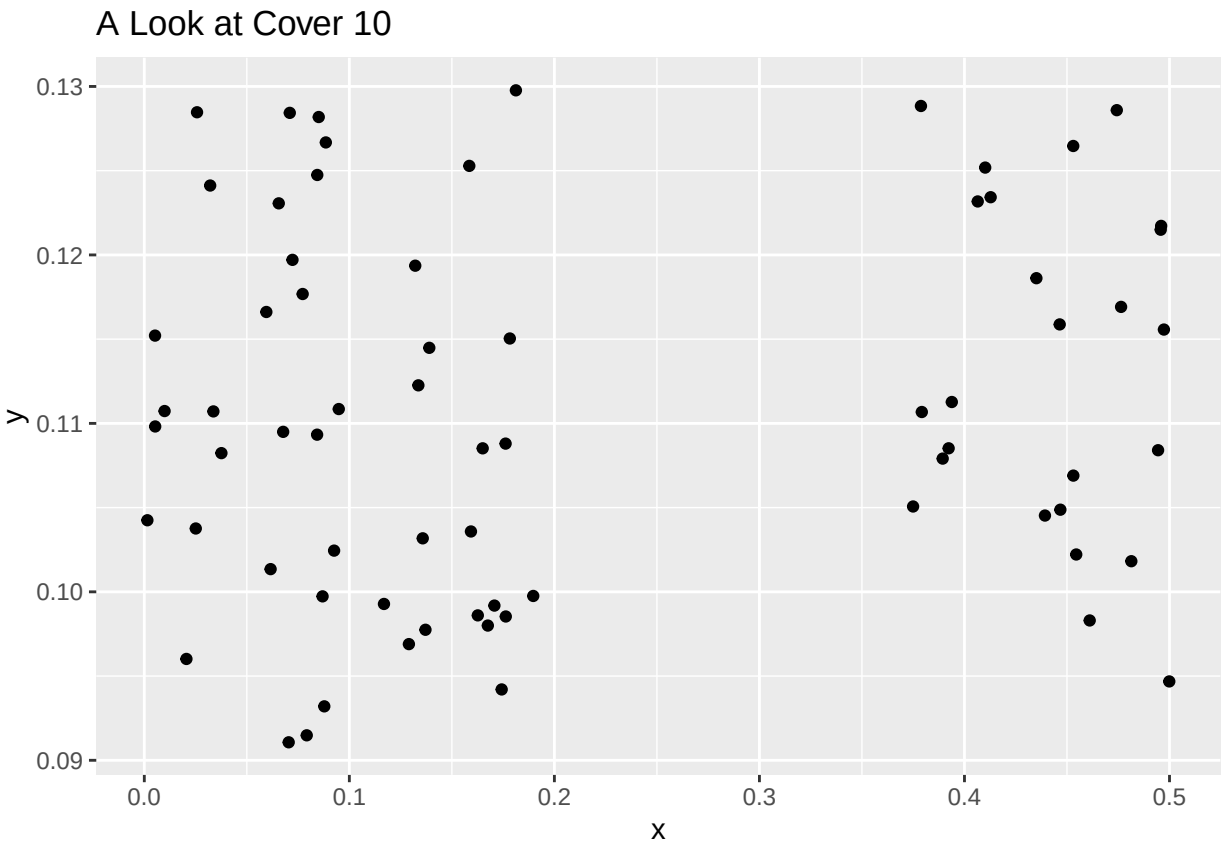
$$\text{Cover}_{j+1} = y_j(1 + (1 - g) * r)$$

Here in our example we have a resolution of $r = 0.04$ and a gain of $g = 0.75$ or an overlap of 75%.

```
##### Filter ####
resolution <- 0.04
gain <- 0.75
list_of_covers <- PullBackCovers(df$x, df$y, resolution, gain)
```

We can pull back the cover and see what's in one of our pullback covers:

```
##### Cover example ####
cover_number <- 10
list_of_covers[[cover_number]] %>%
  ggplot(aes(x, y)) +
  geom_point() +
  ggtitle(paste("A Look at Cover", cover_number))
```



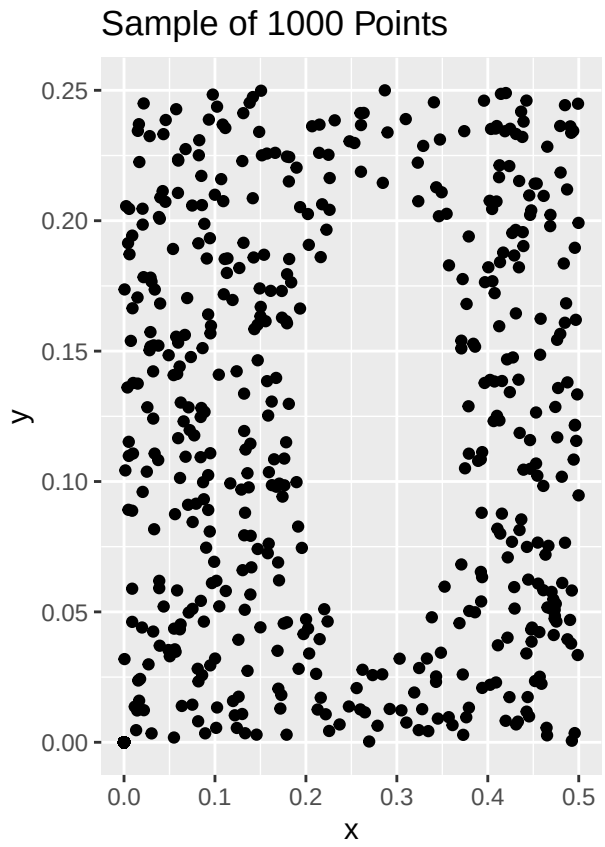
We want to determine the number of clusters present in the cover, so that we can track the homological features as we move through our data.

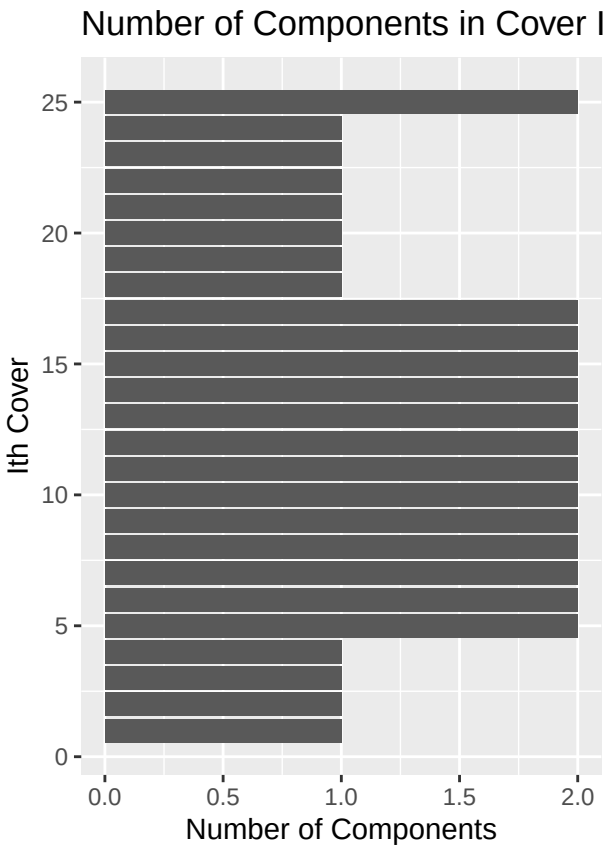
To track these clusters we iterate through our list of covers, creating a distance matrix for each cover then create an adjacency matrix from that distance matrix. We use `FredsDBSCAN` within `FindComponents` to find then count the number of components within in each cover.

```
##### Choose Epsilon and Minimum Connections ####
epsilon <- 0.1
min_connections <- 3

##### How many components do we have? #####
components <- FindComponents(list_of_covers, epsilon, min_connections)
```

After we determining the number of components in each cover, we can plot the data with the number of components present as we have below:

```
##### Component Plot ####
component_chart <- ComponentBarChart(components)


grid.arrange(sample_chart, component_chart, nrow=1)
```

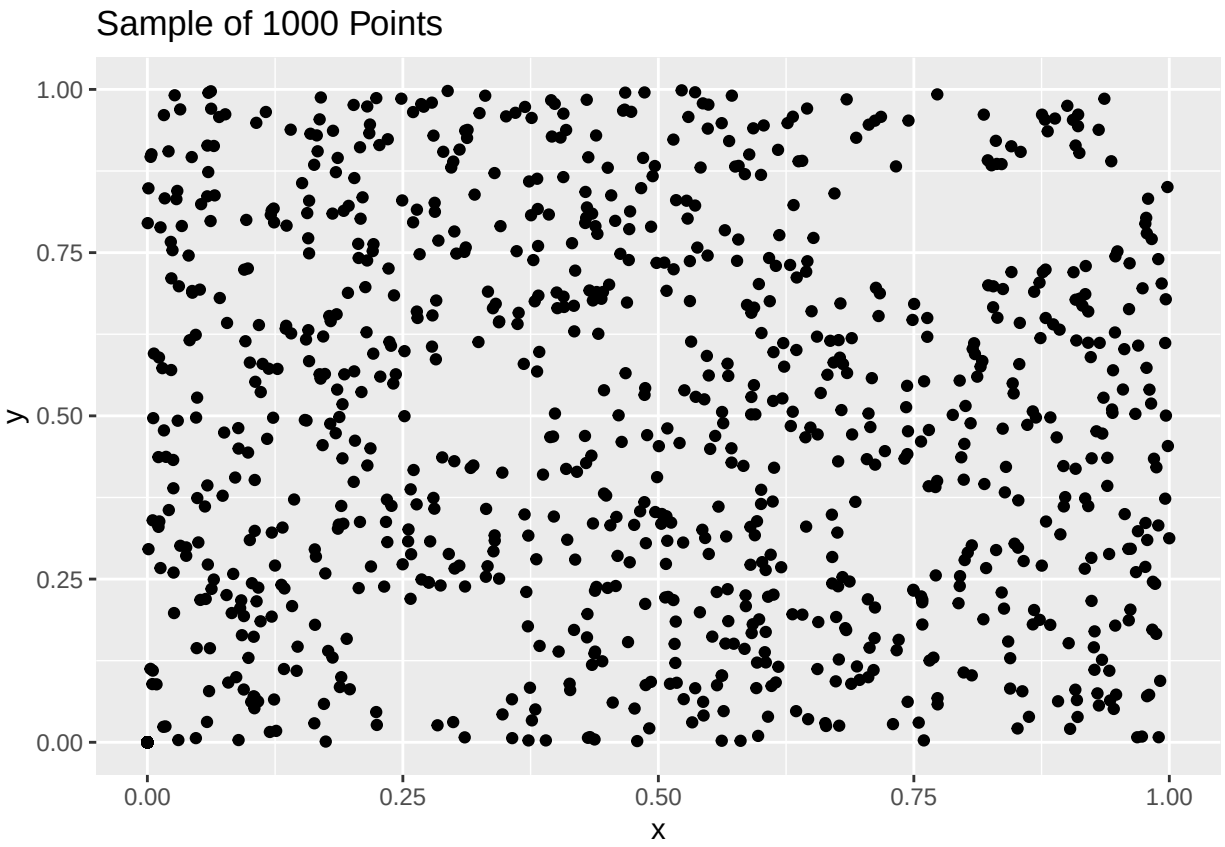




Running a Sample of the Whole

Given the whole set, and not a sample of one specific hole, can we track the various homological features?

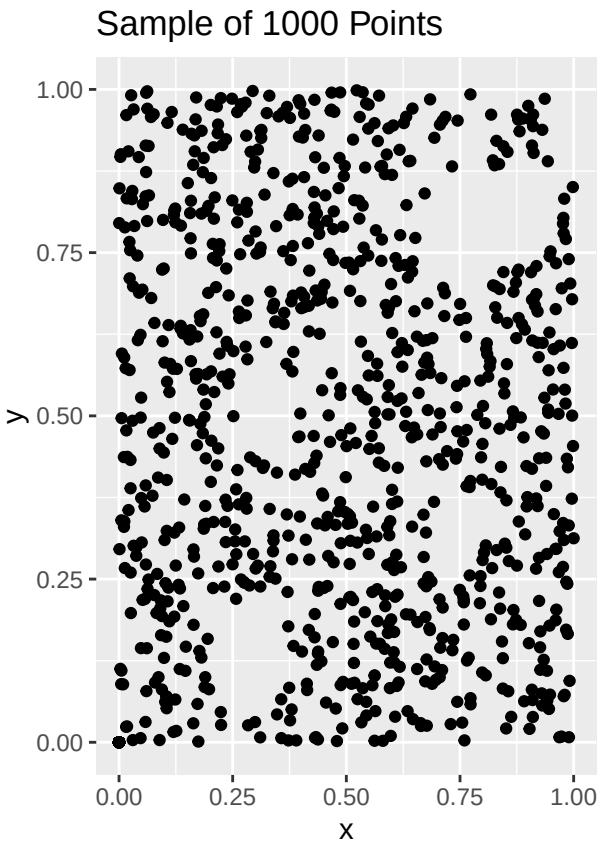
```
##### Sample Entire Set ####
sample_size <- 1000
df <- df_original
df <- sample_n(df, sample_size)

##### What does it look like? ####
sample_chart <- df %>%
  ggplot(aes(x, y)) +
  geom_point() +
  ggtitle(paste("Sample of", sample_size, "Points"))

sample_chart
```

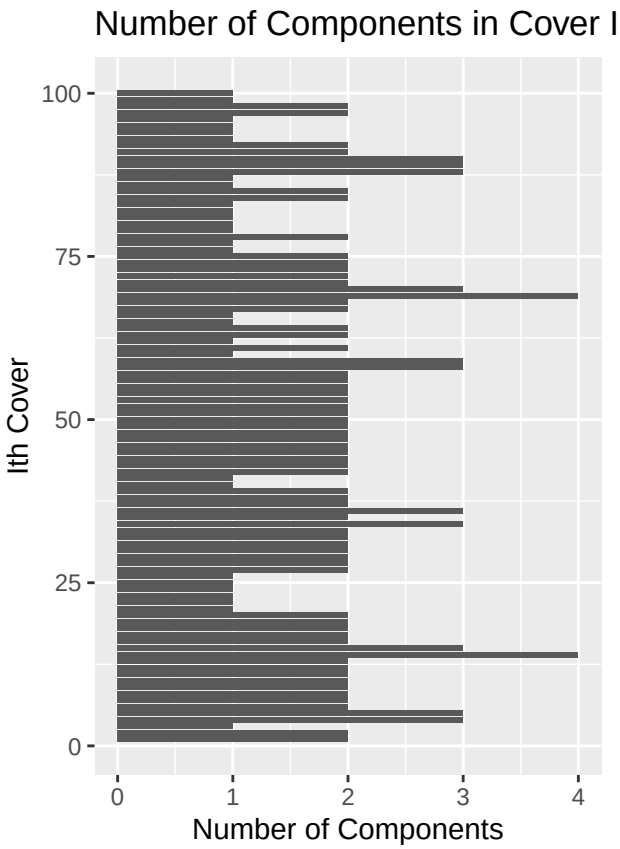


For the purpose of continuity we use the same resolution, gain, epsilon, and minimum connections from above.

```
##### List of Pullback Covers ####
list_of_covers <- PullBackCovers(df$x, df$y, resolution, gain)

##### Number of Components ####
components <- FindComponents(list_of_covers, epsilon, min_connections)
```

```
##### Component Plot ####
component_chart <- ComponentBarChart(components)

grid.arrange(sample_chart, component_chart, nrow=1)
```





Code

PullBackCovers

Construct a list of points within each cover, determined by resolution and gain

```
#' Title PullBackCovers
#'
#' @param x vector of x coordinates
#' @param y vector of y coordinates
#' @param resolution
#' @param gain overlap in terms of percentage
#'
#' @return
#' @export
#'
#' @examples
PullBackCovers <- function(x, y, resolution, gain) {

  ##### Initial Info ####
  y_max <- max(df$y)                                  # When to stop
  increment <- resolution * (1 - gain)                # Lower bound iterations
  n_sets <- ceiling(y_max / increment)                # Number of Sets
```

```
  ##### Cover Boundaries ####
  # Initialize Variables
  lower <- 0 - increment
  upper <- lower + resolution
  covers <- vector("list", n_sets)
  for (cover_number in c(1:n_sets)) {
    covers[[cover_number]] <- df[df$y > lower & df$y < upper, ] # List entry cover
    lower <- (increment * cover_number)                        # Lower Bound
    upper <- lower + resolution                                # Upper Bound

  }
  ##### Meaningful List Entry Names ####
  names(covers) <- paste0("Cover_", c(1:n_sets))

  ##### Return the list ####
  return(covers)
}
```

PairwiseDist

Determine distances between all points present in the set

C++ for calculation speed

```
# include <Rcpp.h>
using namespace Rcpp;

// [[Rcpp::export]]

NumericMatrix PairwisedistMatrix(NumericVector x,
                                 NumericVector y){

  int i, j;
  int n = x.size();
  NumericMatrix pwdmat(n);

  for(i = 0; i < n; i++){
    for(j = 0; j < n; j++){
       pwdmat(i, j) = pow(pow((x[i] - x[j]),2) + pow((y[i] - y[j]),2), 0.5);
    }
  }
  return pwdmat;
}
```

```
#' @title Pairwisedist
#'
#' @param x vector of x values
#' @param y vector of y values
#'
#' @return A matrix of pairwise distances
#'
#' @examples Pairwisedist(x_vector, y_vector)
Pairwisedist <- function(x, y){
```

```
  ##### Packages ####
  library(magrittr)

  ##### Errors ####
  if (length(x) != length(y)) {
    stop("Pairwisedist: x and y vectors not of equal length.\n
         Are you sure everything's paired?")
  }

  ##### Vector Length ####
  vec_length <- length(x)

  ##### Populate Distance Matrix ####
  distmat <- PairwisedistMatrix(x, y) %>%
    matrix(nrow = vec_length,
           ncol = vec_length,
           dimnames = list(
             # Rownames
             paste0("P", c(1:vec_length)),
             # Colnames
             paste0("P", c(1:vec_length))
           ))

  ##### Return Pairwise Distance Matrix ####
  return(distmat)
}
```

Adjacency Matrix

Determine which points are adjacent based on some distance epsilon

```
#' @title AdjacencyMatrix
#'
#' @param pairwise_distance_matrix A matrix of pairwise distances
#' @param epsilon A given distance
#'
#' @return A matrix of 1's and 0's where 1's denote adjacency between
#'          points as determined by the given distance, epsilon
#'
AdjacencyMatrix <- function(pairwise_distance_matrix, epsilon,
                            top_right = FALSE){

  ##### Matrix Dimensions ####
  n <- length(pairwise_distance_matrix) ^ (0.5)

  ##### Copy for Replacement ####
  adjacency_mat <- pairwise_distance_matrix

  ##################################################################
  ##### If points (xi, yi) and (xj, yj) are within epsilon    ##
  ##### then assign a 1 to the cell in the adjacency matrix   ##
  ##### else assign a 0                                       ##
```

```
  for (i in c(1:n)) {
    for (j in c(1:n)) {
      if ( i == j) {
        adjacency_mat[i, j] <- 0
      } else if (pairwise_distance_matrix[i, j] < epsilon) {
        adjacency_mat[i, j] <- 1
      } else {
        adjacency_mat[i, j] <- 0
      }

      if (top_right == TRUE) {
        if (i >= j) {
          adjacency_mat[i, j] <- NA
        }

      }
    }
  }
  return(adjacency_mat)
}
```

FredsDBSCAN

The following code is the code behind `FredsDBSCAN`. This iterates through our data set, detecting connected components based on some distance, ϵ and some minimum number of connections.

```
################################################################################
# Find Starting Point:
#    Given an adjacency matrix, a set of untraversed points,
#    and a minimum number of connections for inclusion,
#    we iterate through to find the first point that meets the
#    minimum connection requirements.
#    We then return both the remaining untraversed points and the
#    found starting point
def find_starting_point(adjacency_matrix, untraversed_points, minimum_connections):
    for starting_point in untraversed_points:                                  # Iterate through list of
        if sum(adjacency_matrix[starting_point]) >= minimum_connections:       #     If the sum of the c
            untraversed_points.remove(starting_point)                          #         Remove the poin
#                                                                              #
            return starting_point, untraversed_points                          #     Return the starting
#                                                                              #
    else:                                                                      # If we iterate through e
        return None, untraversed_points                                        #     You Get Nothing! Go
################################################
################################################################################
# Get Cluster
#    Given an adjacency matrix, a minimum connection requirement and a
#    set of points which have yet to be evaluated,
#    iterate through the untraversed points to find the starting point,
#    use this starting point as the first point in our cluster.
#    We then search through the remaining untraversed points
#    to find points adjacent to our starting point.
#    As we iterate through, we remove points we have checked
```

```
#    from the untraversed points array. If a point meets our
#    minimum connection requirement we append it to the array
#    of points in the cluster. As we pass points from our array
#    of untraversed points to our cluster, we repeat our search.
#    We return our array of points in the cluster and the
#    remaining points which were not adjacent to our cluster
#
def get_cluster(adjacency_matrix, minimum_connections, untraversed_points):
    ###############
    # Initialize #
    ###############
    cluster = []                                # List to store clustered points

    ########################
    # Find the first point #
    ########################
    # Start cluster with first point that meets cluster requirements
    starting_point, untraversed_points = find_starting_point(adjacency_matrix,
                                                             untraversed_points,
                                                             minimum_connections)

    # If we can't find a starting point, return None
    if starting_point is None:
        return None, untraversed_points
    else:
        #   otherwise append the starting point and begin search
        cluster.append(starting_point)

    ################################################
    # Begin the search for points in this cluster #
    ################################################
    for point in cluster:                                                    # For points in our clust
        for other_point in untraversed_points:                               #   Search through the po
            if adjacency_matrix[point, other_point] == 1:                    #       if they are adjac
                untraversed_points.remove(other_point)                       #           remove them f
                if sum(adjacency_matrix[other_point]) >= minimum_connections:    #           if they m
                    cluster.append(other_point)                              #               add it to

    ##########
    # Return #
    ##########
    return cluster, untraversed_points
    # Fin
##########################################
#########################################################################################################
# Fred's DBSCAN
#     Given an adjacency matrix and a minimum number of connections
#      we scan the adjacency matrix for disconnected clusters of points
def FredsDBSCAN(adjacency_matrix, minimum_connections):

    ###############
    # Initialize #
    ###############
```

```
    cluster_list = []                                                    # List of Clusters
    cluster = []                                                         # Holds each Cluster as we
    untraversed_points = [p for p in range(len(adjacency_matrix))]       # List of points to hit

    ##################################
    # Traverse our Adjacency Matrix #
    ##################################

    while len(untraversed_points) >= minimum_connections:                # While there's a possibil
        cluster, untraversed_points = get_cluster(adjacency_matrix,
                                                  minimum_connections,
                                                  untraversed_points)    #     find clusters and gi
        if cluster is None:                                              #     if we didn't find an
            break                                                        #         STAHP
        else:                                                            #     assuming we found a
            cluster_list.append(cluster)                                 #         add it to our li

    return cluster_list
##############################
```

FindComponents

```
#' @title FindComponents
#'
#' @param list_of_covers a list of data frames of x and y coordinates
#' @param epsilon distance to determine adjacencies
#' @param minimum_connections minimum number of adjacent points to make a cluster
#'
#' @return a list of the indices of the points in each respective cover
#'
FindComponents <- function(list_of_covers, epsilon, minimum_connections) {
  library(magrittr)

  ##### How many components do we have? #####
  components <- vector("list", length = length(list_of_covers))
  for (cover in c(1:length(list_of_covers))) {
    components[[cover]] <- Pairwisedist(list_of_covers[[cover]]$x, list_of_covers[[cover]]$y) %>%  # Cr
      AdjacencyMatrix(epsilon) %>%                                                                 # Dete
      FredsDBSCAN(min_connections)                                                                 # Scan
  }

  return(components)
}
```

ComponentBarChart

```
#' @title ComponentBarChart
#'
#' @param component_list list of components
```

```r
#'
#' @return bar chart of the number of components in each cover
#'
ComponentBarChart <- function(component_list) {
  library(magrittr)
  library(ggplot2)

  nComponents <- component_list %>%
    lapply(length) %>%
    unlist()

  component_chart <- data.frame(
    index = c(1:length(component_list)),
    ncomp = nComponents
  ) %>%
    ggplot(aes(x = index,
               y = ncomp)) +
    geom_col() +
    xlab("Ith Cover") +
    ylab("Number of Components") +
    coord_flip() +
    ggtitle("Number of Components in Cover I")

  return(component_chart)

}
```