Using the imagingPC Package

Cameron Miller

2019-06-23

Contents

Introduction 2
Overview 2
Study Overview: Glycosylation Patterns in Metastatic Breast Cancer 3
The Data 3
(1) Merge Region Spots and Spot-Level Data 5
(2) Create AUC Data for Each Spot 6
(3) Create Covariates of Interest 6
(4) Rescale the Data 12
    Data Overview 12
    Rescale Function 13
(5) Estimate the Range Parameter 16
(6) Select Support Structures 20
(7) Create Data Vectors for Fitting Model 23
(8) Write Model 27
(9) Run Model 28
References 32

Introduction

This vignette is designed to teach you how to use the `imagingPC` package and most of its features. It is not meant to be all inclusive, but it should cover enough to get you started.

Before beginning this vignette, make sure that you have NIMBLE (R package name: `nimble`) loaded on your computer, along with an appropriate C++ compiler. In order to speed up computation, NIMBLE writes C++ code, and it needs a C++ compiler that can interact with R. If you're using Windows like me, that means installing Rtools. Check the NIMBLE User Manual to get directions on how to do this for Windows, Linux, and OS systems. Although this may seem like a frustration, it pays off. Compared to other sampling programs, I have found NIMBLE to be ~5x faster. When modeling large datasets like imaging mass spectrometry (IMS) datasets, that makes a big difference.

The `imagingPC` package uses a process convolution (PC) approach to account for the spatial information in imaging data collected on a regular grid. The research that motivated this work was generated using IMS, a technique in which the abundances of ionized molecular fragments are measured over a regular grid. For this vignette, it's not necessary to understand the details of how the data were generated, but what is critical to understand is that we have continuous measures of several molecular fragments over a two-dimensional space.

Overview

The imagingPC package uses the following workflow

1. merge region spots and spot-level data (optional)
2. create area under the curve (AUC) data for each spot (optional)
3. create covariates of interest
4. rescale the data
5. estimate the range parameter
6. select support structure(s)
7. create data vectors for fitting model
8. write model
9. run model

With the exception of Step 3 (create covariates of interest), `imagingPC` contains functions to perform each task (shown below).

Task	Function
merge region spots and spot-level data (optional)	mergeIMSFiles
create area under the curve (AUC) data for each spot (optional)	createAUCData
create covariates of interest	
rescale the data	rScale
estimate the range parameter	estRange
select support structure(s)	chooseStructures
create data vectors for fitting model	createPCData
write model	writePCModel
run model	runPCModel

Note that the first 2 steps are optional. Once you get to the rescaling data, you must stick to the workflow. Each function must be used in order. Now, let's get started. First, make sure the package is loaded.

```
library(imagingPC)
```

Study Overview: Glycosylation Patterns in Metastatic Breast Cancer

Before beginning with the demonstration, it will be helpful to understand the objectives for the study data used here. This study sought to answer 2 questions in metastatic breast cancer using an *in vivo* mouse model.

1. Do glycosylation patterns differ between primary and secondary tumors?
2. Do glycosylation patterns differ between regions containing tumor-associated macrophages (TAMs) and those without TAMs?

Glycosylation is a common protein modification in which chains of sugar groups called glycans are covalently bound to a peptide and modified. This study focused on one type of glycosylation called N-linked glycosylation. To answer the above questions, 4T1 cells were injected in mice and allowed to develop tumors and metastasize, and primary breast tumors and secondary lung tumors were resected. The tumors were sectioned, and N-glycan abundances were measured using IMS. More information will be provided later on the type of IMS used to collect abundance measurements. But for now, let's take a look at the dataset.

The Data

The following two .csv files were exported from SCiLS Lab (Daltonics 2017). The first file is the region spots file, which contains a unique identifier for every spot, or location of data collection, and the corresponding x and y coordinates.

```
regionspots<-read.csv('C:\\Users\\Cameron\\Documents\\region_spots.csv')

head(regionspots, n=15)
#>    X..Exported.with.SCiLS.Lab.2016b.Version.4.01.8781
#> 1                        # Export time: 19-03-04 15:41:02
#> 2   # Generated from file: Z:\\Users\\somefilename.sl
#> 3                             # Object Full Name: Regions
#> 4                                  # Object ID: /Regions
#> 5                                  # Object type: Region
#> 6            # Object creation time: 19-03-04 15:10:26
#> 7                                            Spot index
#> 8                                                     x
#> 9                                                     y
#> 10                                                    0
#> 11                                     5928.18310546875
#> 12                                       4189.064453125
#> 13                                                    1
#> 14                                     6128.18310546875
#> 15                                       4189.064453125
```

The data in this form looks unintelligible. That's because there's a lot of metadata at the top of the file that needs to be removed. When R imports this data, it turns it into a vector. This needs to be reassembled into a matrix. The same thing happens with the second file, which contains all the abundance measures for each fragment.

```
spotdata<-read.csv('C:\\Users\\Cameron\\Documents\\spot_data.csv')
head(spotdata, n=15)
#>    X..Exported.with.SCiLS.Lab.2017a.Version.5.00.9419
#> 1                        # Export time: 19-03-07 15:07:19
#> 2   # Generated from file: Z:\\Users\\somefilename.sl
#> 3                             # Object Full Name: Regions
#> 4                                   # Object ID: Regions
#> 5                                  # Object type: Region
#> 6            # Object creation time: 19-03-04 15:10:26
#> 7                  # MzRange Name: 20190307 TAM tissues
#> 8                      # Normalization: Total Ion Count
#> 9                                                   m/z
#> 10                                     771.255249023438
#> 11                                     771.256469726563
#> 12                                     771.257690429688
#> 13                                     771.258911132813
#> 14                                     771.260131835938
#> 15                                     771.261352539063
```

This file contains the same unique spot identifiers (with a slightly different name) as the region spots file, but it has no x and y coordinates.

(1) Merge Region Spots and Spot-Level Data

To proceed we need to merge these two datasets so that the locations are matched with their corresponding data. At the same time we want to remove the metadata at the top. To do this we use the `mergeIMSFiles` function. The `mergeIMSFiles` function is pretty simple, with only three inputs: the spot coordinates data, the spot-level data, and the `trimTops` argument, specifying whether the tops need to be trimmed to remove metadata.

```
imsdata<-mergeIMSFiles(spotData = spotdata, regionSpots = regionspots,
                       trimTops = TRUE)
imsdata[1:10,1:8]
#>    Spot.index        x        y  771.255249023438  771.256469726563
#> 1           0 5928.183 4189.064                 0                 0
#> 2           1 6128.183 4189.064                 0                 0
#> 3           2 5328.183 3989.064                 0                 0
#> 4           3 5528.183 3989.064                 0                 0
#> 5           4 5728.183 3989.064                 0                 0
#> 6           5 5928.183 3989.064                 0                 0
#> 7           6 6128.183 3989.064                 0                 0
#> 8           7 6328.183 3989.064                 0                 0
#> 9           8 6528.183 3989.064                 0                 0
#> 10          9 6728.183 3989.064                 0                 0
#>     771.257690429688  771.258911132813  771.260131835938
#> 1                0.0                 0                 0
#> 2                0.0                 0                 0
#> 3                0.0                 0                 0
#> 4                0.0                 0                 0
#> 5                0.0                 0                 0
#> 6                0.0                 0                 0
#> 7           258483.2                 0                 0
#> 8                0.0                 0                 0
#> 9                0.0                 0                 0
#> 10               0.0                 0                 0
```

The resulting dataset is now looks the way it should. The first column is the unique spot identifier, the next two are the x and y coordinates, and the remaining are the abundance measures for different m/z values. There are quite a few m/z values (560 to be exact) and lots of zeros.

```
dim(imsdata)
#> [1] 19936   563
```

(2) Create AUC Data for Each Spot

The column names for the abundance measures are given by their m/z. Notice that the increments between m/z values are quite small. The values for an m/z represent the measured abundances at that m/z. Due to reasons beyond the scope of this vignette, the abundance of a molecular fragment is calculated from the abundances across a range of m/z values. We calculate the abundances using the area under the curve (AUC). In doing so we connect abundance measures between the small m/z measurements and calculate the area of the resulting trapezoid (or triangle if one value is a 0). Within the `imagingPC` package, the AUC is calculated using the `createAUCData` function.

```
imsdata<-createAUCData(data = imsdata, mzThreshold = 0.3,
                       usedMergeIMSFiles = TRUE)
imsdata[1:5,1:13]
#>   Spot.index        x        y X771.2656 X933.3204 X1079.3848 X1095.3746
#> 1          0 5928.183 4189.064         0         0     0.0000          0
#> 2          1 6128.183 4189.064         0         0   513.9416          0
#> 3          2 5328.183 3989.064         0         0     0.0000          0
#> 4          3 5528.183 3989.064         0         0     0.0000          0
#> 5          4 5728.183 3989.064         0         0     0.0000          0
#>   X1136.4031 X1257.432 X1282.4658 X1298.4645 X1339.4821 X1419.4883
#> 1          0 3642.2435      0.000  1074.0069          0   8111.131
#> 2          0 7441.2382   1659.731  1633.3009          0   7557.920
#> 3          0 1563.5701      0.000     0.0000          0   3510.445
#> 4          0  813.3568      0.000   635.2662          0   1164.503
#> 5          0 1228.0765      0.000     0.0000          0   1285.796
```

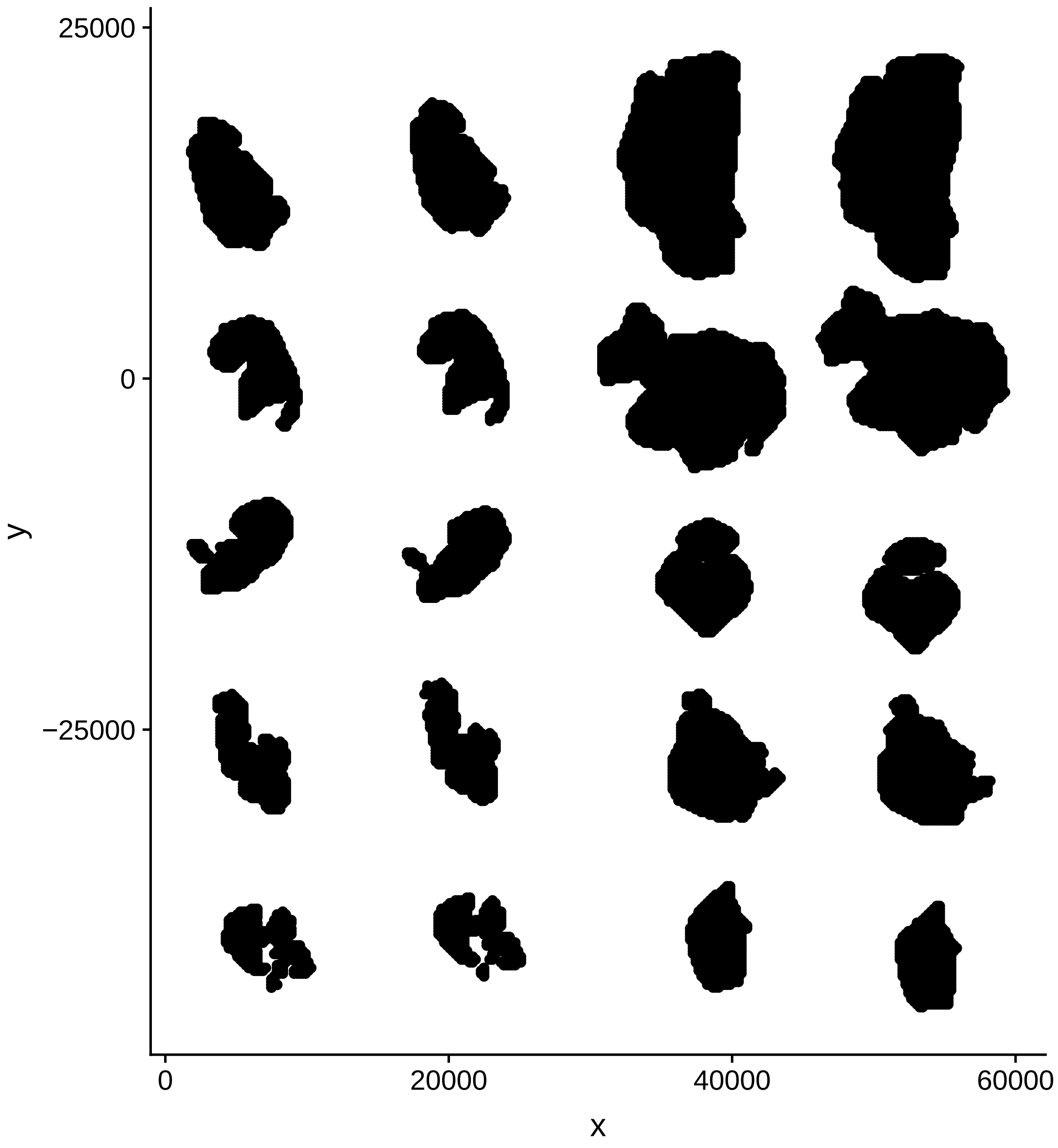
The `data` argument is the only argument that needs to specified. The others are set to their default settings and are shown here only for demonstration. The `mzThreshold` sets up a window of +/- the threshold to determine which m/z values to combine to calculate the AUC. For this reason, the fragments of interest need to be somewhat sparse, and the process of peak-picking needs to be completed before using this function. The `usedMergedIMSFiles` is a TRUE/FALSE variable specifying whether or no that function was used. This argument is important because of the way `R` formats data that's imported.

(3) Create Covariates of Interest

Once the data have been merged and AUC calculated for every spot in every m/z, the covariates of interest need to be defined. Due to the fact that this will vary drastically between studies, this vignette will not cover how to generate the covariates of interest. Instead, I will briefly demostrate how to create a sample variable. Covariates are often defined on the sample level (such as non-tumor vs. tumor samples), and so this is a critical step.

There are at least two ways to create a sample variable. The first is the somewhat manual approach that defines the sample variable based on the x and y coordinates of the data. Let's get a look at the data locations.

```r
ggplot(data=imsdata, aes(x=x, y=y)) +
       geom_point()
```

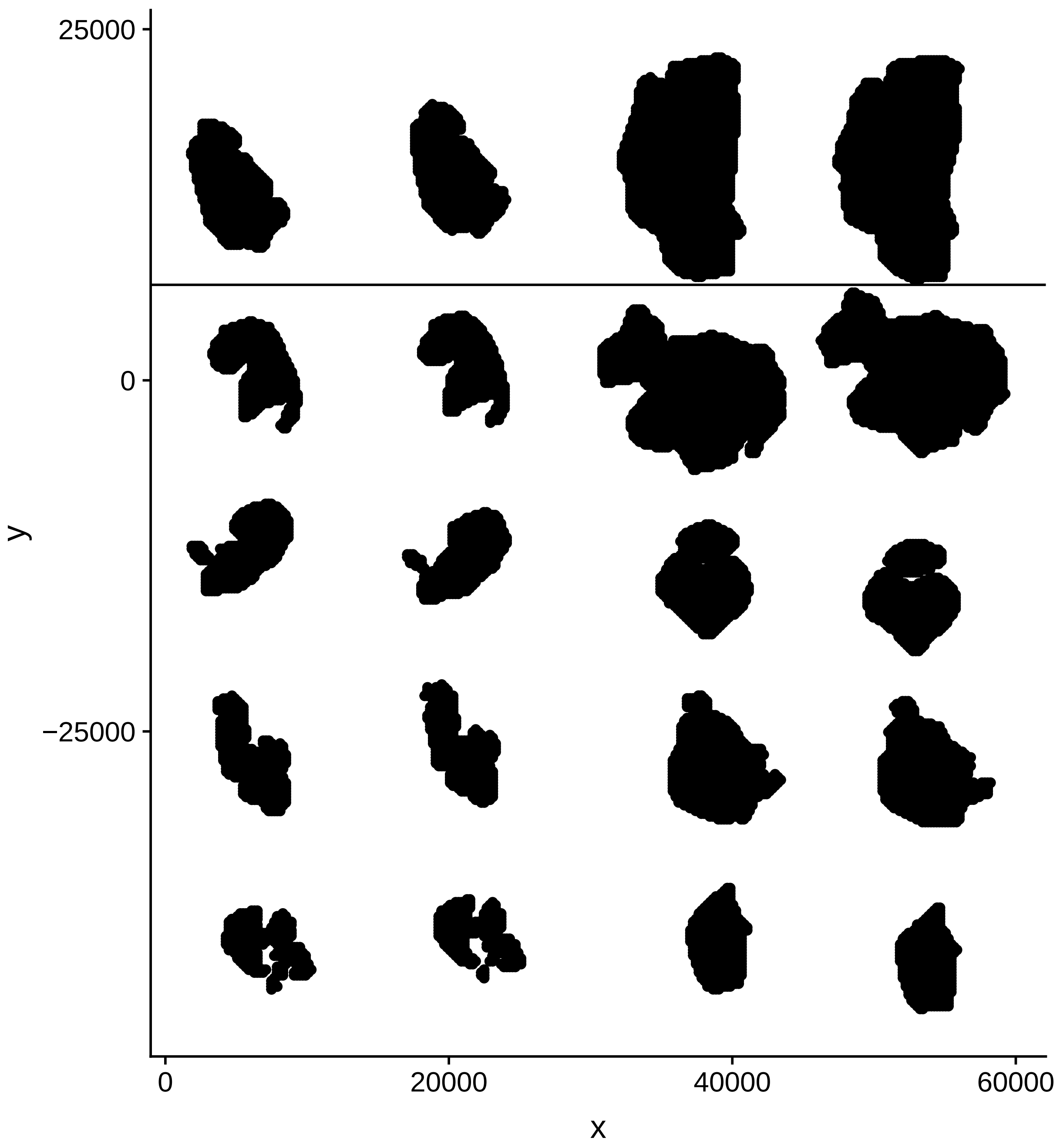


Here, `geom_point` is used instead of `geom_raster` because `geom_raster` has trouble plotting the data. `geom_raster` expects data in which all the observations are the same distance apart. Here, that distance can vary slightly, and that causes problems with `geom_raster`. To generate the sample variable, we can use the following reasoning.

```r
imsdata$sample<-NA
for(i in 1:nrow(imsdata)){
  if(imsdata$x[i]<10000 & imsdata$y[i]>6000){imsdata$sample[i]<-1}
  if(imsdata$x[i]>12000 & imsdata$x[i]<26000 &
     imsdata$y[i]>6000){imsdata$sample[i]<-2}
  if(imsdata$x[i]>30000 & imsdata$x[i]<45000 &
     imsdata$y[i]>6800){imsdata$sample[i]<-3}
  if(imsdata$x[i]>45000 & imsdata$y[i]>6800){imsdata$sample[i]<-4}
  if(imsdata$x[i]<12000 & imsdata$y[i]>-6000 &
     imsdata$y[i]<6800){imsdata$sample[i]<-5}
  if(imsdata$x[i]>12000 & imsdata$x[i]<25000 &
     imsdata$y[i]>-6000 & imsdata$y[i]<6800){imsdata$sample[i]<-6}
}
```

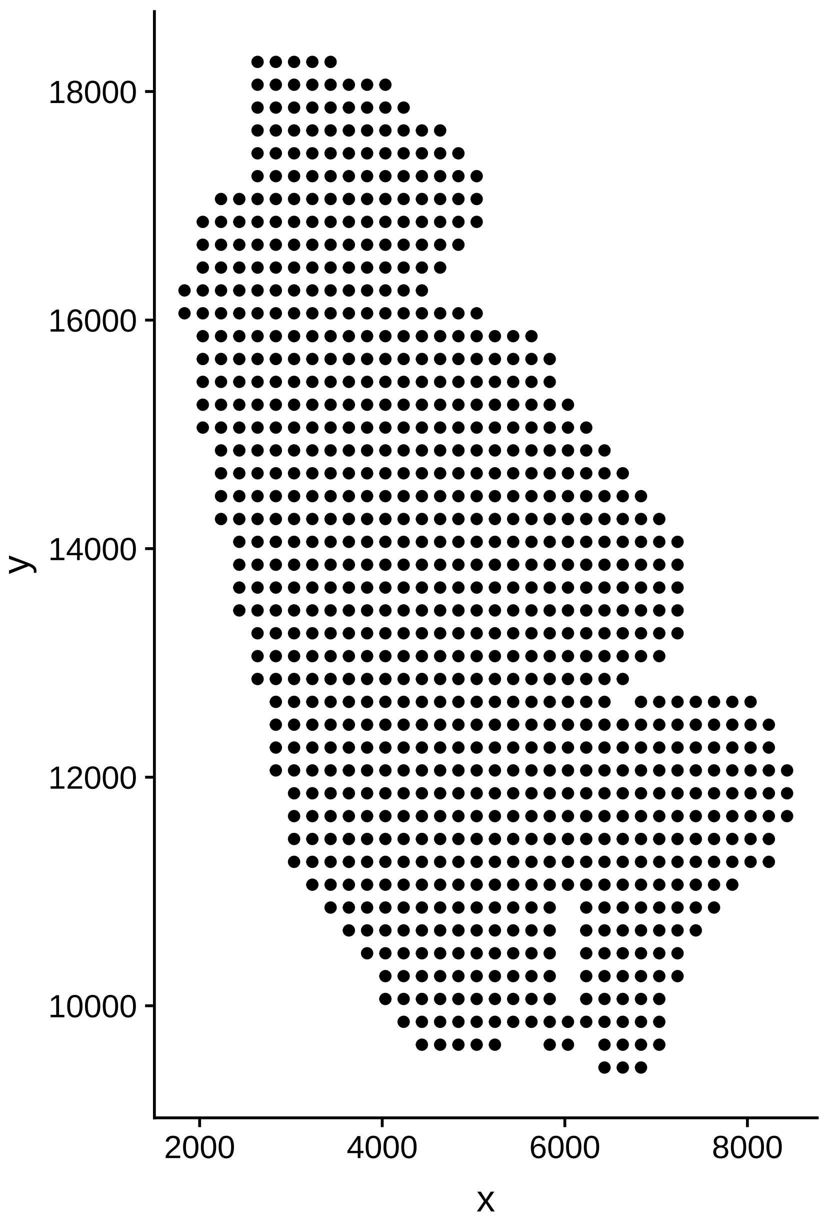
For tissue samples that are close together, you can use horizontal and vertical lines in ggplot to guide you.

```r
ggplot(data=imsdata, aes(x=x, y=y)) +
       geom_point() +
       geom_hline(yintercept = 6800)
```



To check and make sure you've done it correctly, you can plot each sample.

```r
ggplot(data=imsdata[is.na(imsdata$sample)==FALSE & imsdata$sample==1,],
       aes(x=x, y=y)) +
       geom_point()
```



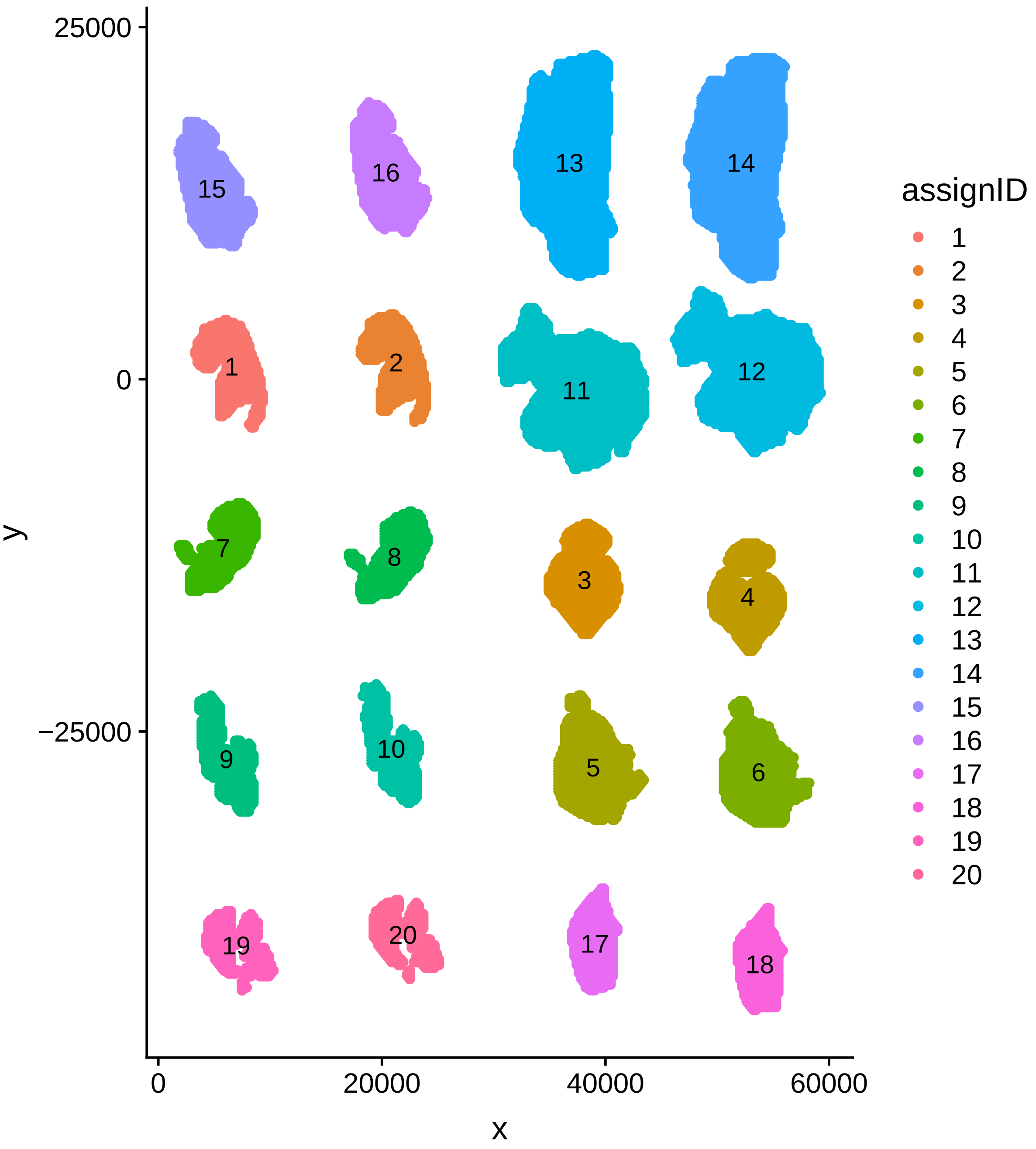
This approach is straightforward and works well for datasets with small numbers of tissue samples. However, if the data come in the form of a tissue microarray (TMA), where hundreds of tissue punches are arranged on a glass slide, this approach quickly becomes tremendously work-intensive.

To make this process a little easier, particularly for datasets with large number of samples, the `imagingPC` package contains a function to assign ID numbers to samples. The only caveat is that the data must be of the same resolution. This means that the distance between observations must be the same for all samples.

The `assignIDs` function is used to assign sample IDs by using the adjacencies of the observations. This function does not require that the data be generated using the `imagingPC` package. It only requires the data as input, though there are a few arguments that can be used to modify the function. This function does not require that the data be generated using the `imagingPC` package.

```
imsdata<-assignIDs(imsdata)
```

```
plotIDs(imsdata)
```



As you can see, the function correctly separated the data into distinct samples. The `assignIDs` function does not reorder the data, so you can try sorting the data if you want to assign IDs in a different order. Otherwise, you can use logical arguments in R to reassign

ID numbers. That is not necessary, though, as subsequent functions will reassign sample ID numbers.

(4) Rescale the Data

Data Overview

For the remainder of the vignette I'll switch to a dataset in which the covariates have been made. This data is included in the R package and is actually the data we were just working with, but it has all the covariates created. I would like to give a special thanks to Dr. Elizabeth Yeh for letting us publish this data. Also, since this data is included in the R package, please feel free to try to improve on our methods.

```
data("TAMdata")
TAMdata[1:10,1:8]
#>    Spot.index        x        y subject ROI slide secondary TAM
#> 1           0 5928.183 4189.064       3   9     5         1   0
#> 2           1 6128.183 4189.064       3   9     5         1   0
#> 3           2 5328.183 3989.064       3   9     5         1   0
#> 4           3 5528.183 3989.064       3   9     5         1   0
#> 5           4 5728.183 3989.064       3   9     5         1   0
#> 6           5 5928.183 3989.064       3   9     5         1   0
#> 7           6 6128.183 3989.064       3   9     5         1   0
#> 8           7 6328.183 3989.064       3   9     5         1   0
#> 9           8 6528.183 3989.064       3   9     5         1   0
#> 10          9 6728.183 3989.064       3   9     5         1   0
```

```
colnames(TAMdata)
#>  [1] "Spot.index" "x"          "y"          "subject"    "ROI"
#>  [6] "slide"      "secondary"  "TAM"        "meanTAM"    "X771.auc"
#> [11] "X933.auc"   "X1079.auc"  "X1095.auc"  "X1136.auc"  "X1257.auc"
#> [16] "X1282.auc"  "X1298.auc"  "X1339.auc"  "X1419.auc"  "X1444.auc"
#> [21] "X1460.auc"  "X1485.auc"  "X1501.auc"  "X1542.auc"  "X1581.auc"
#> [26] "X1606.auc"  "X1645.auc"  "X1647.auc"  "X1663.auc"  "X1688.auc"
#> [31] "X1704.auc"  "X1743.auc"  "X1793.auc"  "X1809.auc"  "X1850.auc"
#> [36] "X1866.auc"  "X1891.auc"  "X1905.auc"  "X1954.auc"  "X1955.auc"
#> [41] "X1970.auc"  "X1995.auc"  "X1996.auc"  "X2012.auc"  "X2017.auc"
#> [46] "X2028.auc"  "X2067.auc"  "X2071.auc"  "X2100.auc"  "X2101.auc"
#> [51] "X2122.auc"  "X2141.auc"  "X2174.auc"  "X2377.auc"  "X2437.auc"
#> [56] "X2465.auc"  "X2466.auc"  "X2539.auc"  "X2540.auc"
```

Notice that this dataset has the same Spot.index, x, and y covariates, as well as several AUC variables for different molecular fragments. The subject and region of interest (ROI) variables are self-explanatory. The slide variable indicates which tissue samples, or ROIs, were run on the same slide. This will become important in downstream analyses and will be explained later. The secondary variable is binary indicator of whether the ROI is from a primary tumor (secondary=0) or a secondary tumor (secondary=1). The TAM is a spot-level, or raster-level, indicator of whether the raster is from a non-TAM (TAM=0) or TAM (TAM=1) region. The word *spot* is often used in IMS to refer to the locations of data collection. We will use this term interchangeably with *raster*. Rasterization is process by which the spots are made to fill a square. This makes visualization of the data much easier.

The last variable that should be explained is the meanTAM variable. To identify TAM regions, the F4/80 stain, which is an immunohistochemical (IHC) stain for macrophage-specific cell surface protein, was used to identify macrophages in the tumors. The stained tissue samples were scanned using a standard photo/document scanner, producing pixelated images of the tissues. For the IHC images, a binary variable was created on the pixel level indicating whether each pixel belonged to a positively stained region. To combine the IHC and IMS data, the IHC coordinates were rescaled so that the outer bounds of the IHC and IMS data matched. Then, for every IMS spot, the binary IHC variable on the pixel level was averaged for all pixels within the bounds of the corresponding IMS raster. This produced a proportion of positively stained pixels correspoding to each raster. This proportion is called meanTAM in the dataset. We'll plot this soon to show what this variable looks like.

Rescale Function

Before plotting, though, we need to rescale the data. Besides making calculations messy, the way decimals are stored can be problematic for downstream calculations. In the rescaling process, we arbitrarily assign a distance of 1 between successive data collection sites.

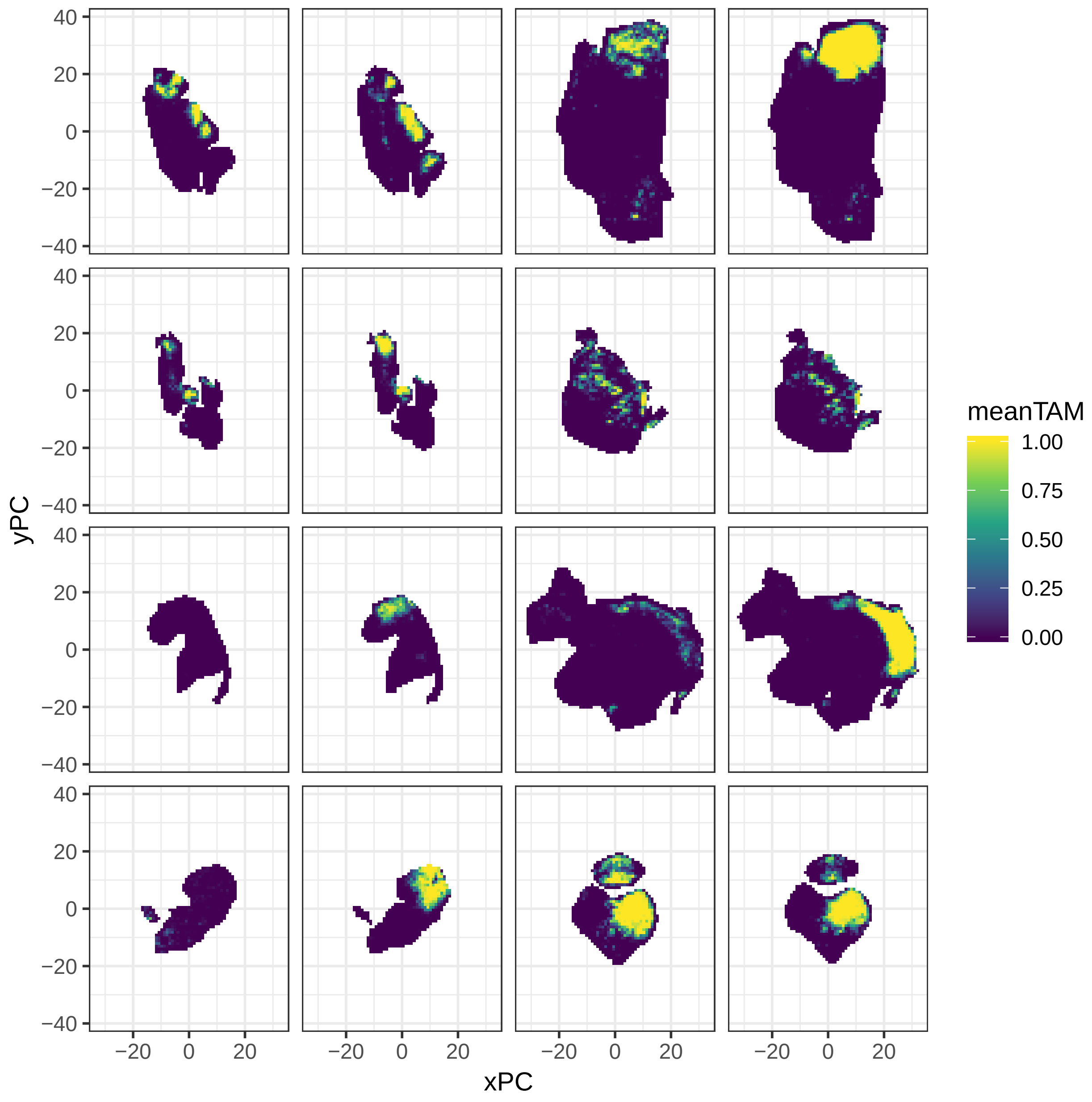
```r
TAMdata <- rScale(TAMdata, subjectVar = 'subject', sampleVar = 'ROI',
                  xCoord = 'x', yCoord = 'y')
class(TAMdata)
#> [1] "rScaleList"
names(TAMdata)
#> [1] "data"       "subjectVar" "sampleVar"
```

The rescaled data is now a list of class "rScaleList". This is a list containing the data with new x- and y- coordinates labeled "xPC" and "yPC", respectively, as well as the subject and sample variables. This information is carried foward to subsequent functions, cutting down on possible errors when copying and pasting code to repeat analyses.

```r
TAMdata$subjectVar
#> [1] "subject"
TAMdata$sampleVar
#> [1] "ROI"
```

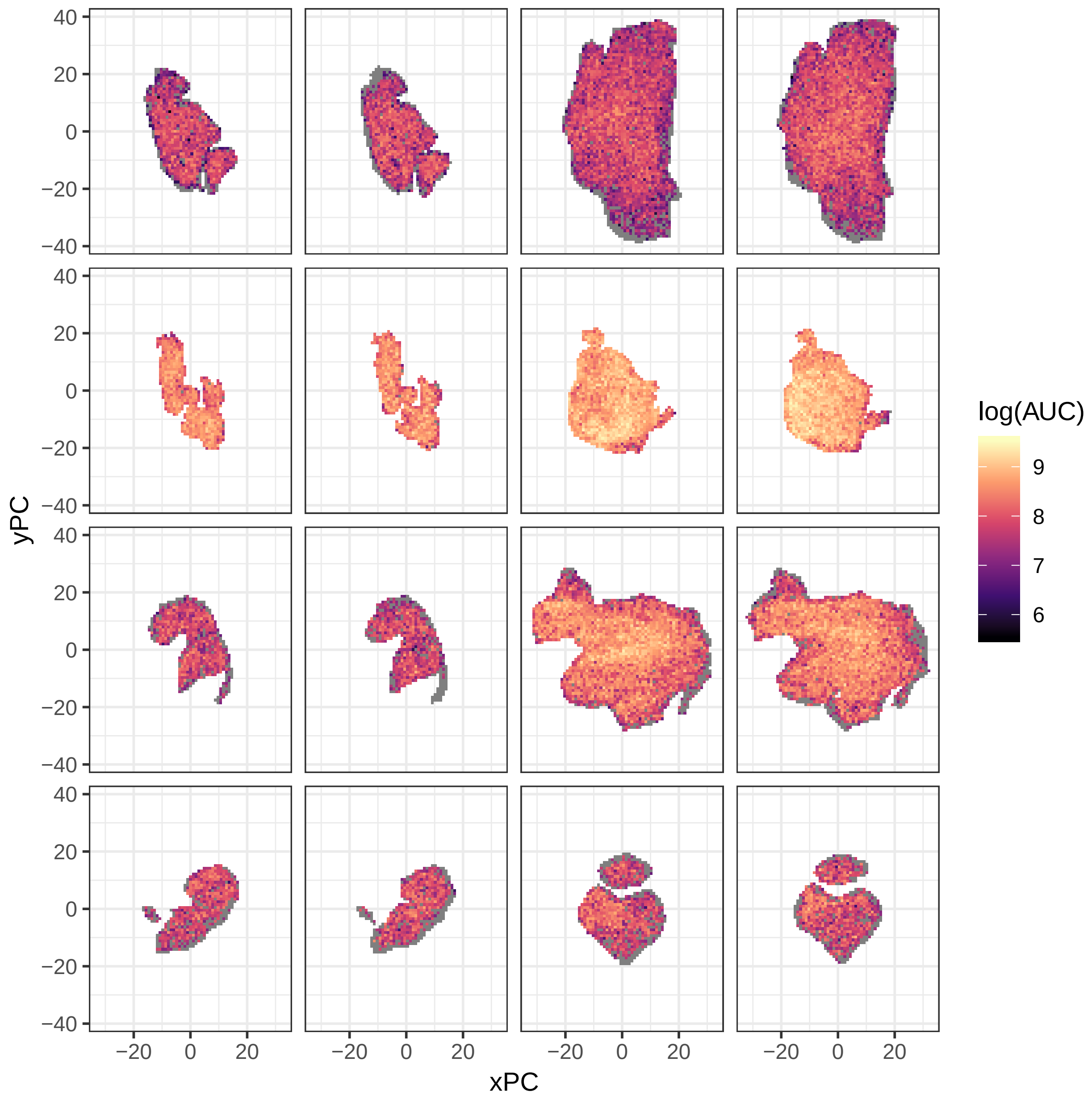
With the data rescaled we can use ggplot to plot the tissue samples. Let's start by plotting the meanTAM variable described earlier. The `rScale` function centers each tissue sample at the origin, so it is necessary to use the facet_wrap or facet_grid option in ggplot.

```
ggplot(data=TAMdata$data, aes(x = xPC, y = yPC)) +
       geom_raster(data = TAMdata$data, aes(fill = meanTAM)) +
       facet_wrap(subject~ROI, nrow = length(unique(TAMdata$data$subject))) +
       scale_fill_viridis_c()  +
       theme_bw() +
       theme(strip.background = element_blank(),
             strip.text.y = element_blank(),
             strip.text.x=element_blank())
```



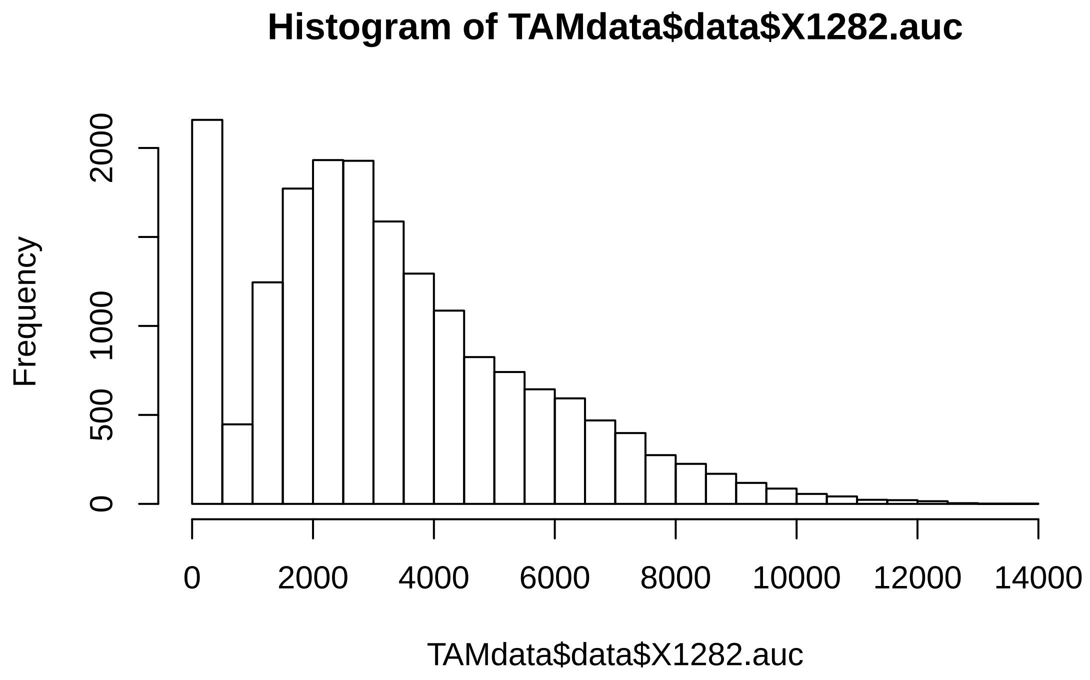
Now we can move on a look at the abundance measures of one of the fragments. We'll start with an N-glycan with a mass of approximately 1282 Daltons. IMS data are typically lognormally distributed, so we use the log transform when plotting and performing analyses.

```
ggplot(data=TAMdata$data, aes(x = xPC, y = yPC)) +
      geom_raster(data = TAMdata$data, aes(fill = log(X1282.auc))) +
      scale_fill_viridis_c(option = 'magma', name = 'log(AUC)') +
      facet_wrap(subject~ROI, nrow = length(unique(TAMdata$data$subject))) +
      theme_bw() +
      theme(strip.background = element_blank(),
            strip.text.y = element_blank(),
            strip.text.x=element_blank())
```



You may notice a lot of grey rasters in the plot. The grey rasters indicate abundance measures of 0, and so when the log is taken, these become -infinity, which can't be plotted. A large proportion of zeros is common in IMS data when using certain mass spectrometry methods. If we generate a histogram, we'll see that a little over 10% of the measurements are zero. For some masses, the percentage of zeros can reach over 90%.

```r
hist(TAMdata$data$X1282.auc, breaks = 20)
```



(5) Estimate the Range Parameter

With the data rescaled, we can finally start employing the techniques for which this package was designed. The first step in utilizing this automated PC procedure is estimating the range. In short, the `estRange` function utilizes the `geoR` package to estimate the semivariance (a measure of variability between observations a distance h apart.) and then fits a covariance model to those semivariance estimates. Currently, this PC method assumes the covariance model is Gaussian.

When estimating the semivariance, data are often binned, meaning the observations across several values of h are often used to estimate a single semivariance. However, IMS data are collected along a regular grid, and distances between observations are repeated, sometimes many thousands of times. In this case, the distances do not need to be binned, and the semivariance is estimated at every distance.

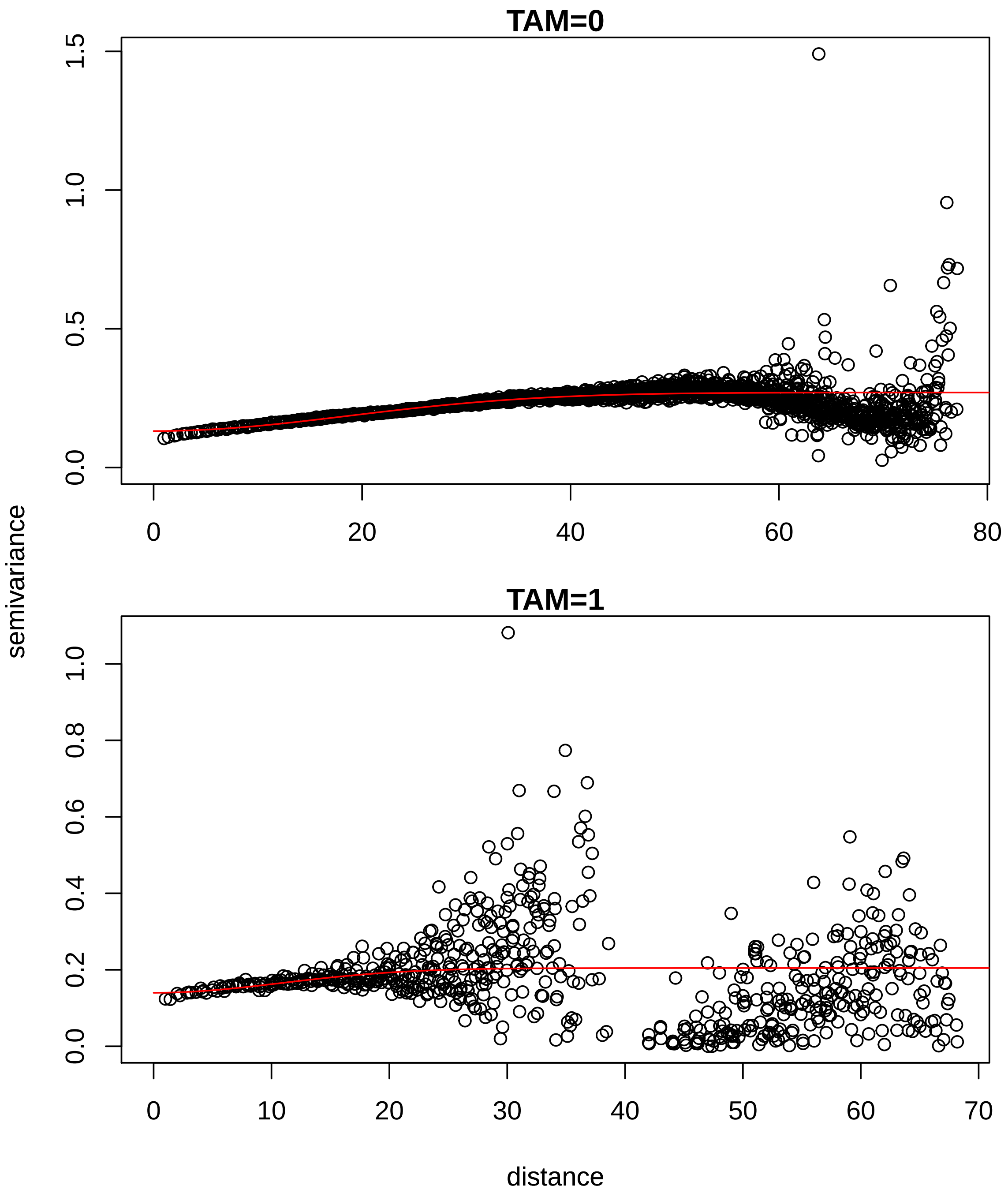
For the study described here, we assume that the underlying spatial processes differ between the non-TAM and TAM regions. To account for different spatial processes, we estimate the range parameter for each level of the TAM variable. This is done by specifying `spatialVar = 'TAM'`.

```r
rangs <- estRange(TAMdata, outcome = 'X1282.auc', spatialVar = 'TAM',
                  semivEst = 'modulus', logTransform = TRUE)
names(rangs)
#> [1] "data"          "subjectVar"   "sampleVar"    "spatialVar"   "outcome"
#> [6] "estRange"      "estSig2"      "semivarFit"   "covModelFit"
```

The `rangs` object is a list with several components. These include those components from the `rScale` function with additional objects. The most important are the last four, which contain the estimated range values, the estimated variance components of the covariance model, the `geoR` object containing semivariance estimates, and the `geoR` object with the fitted covariance model.

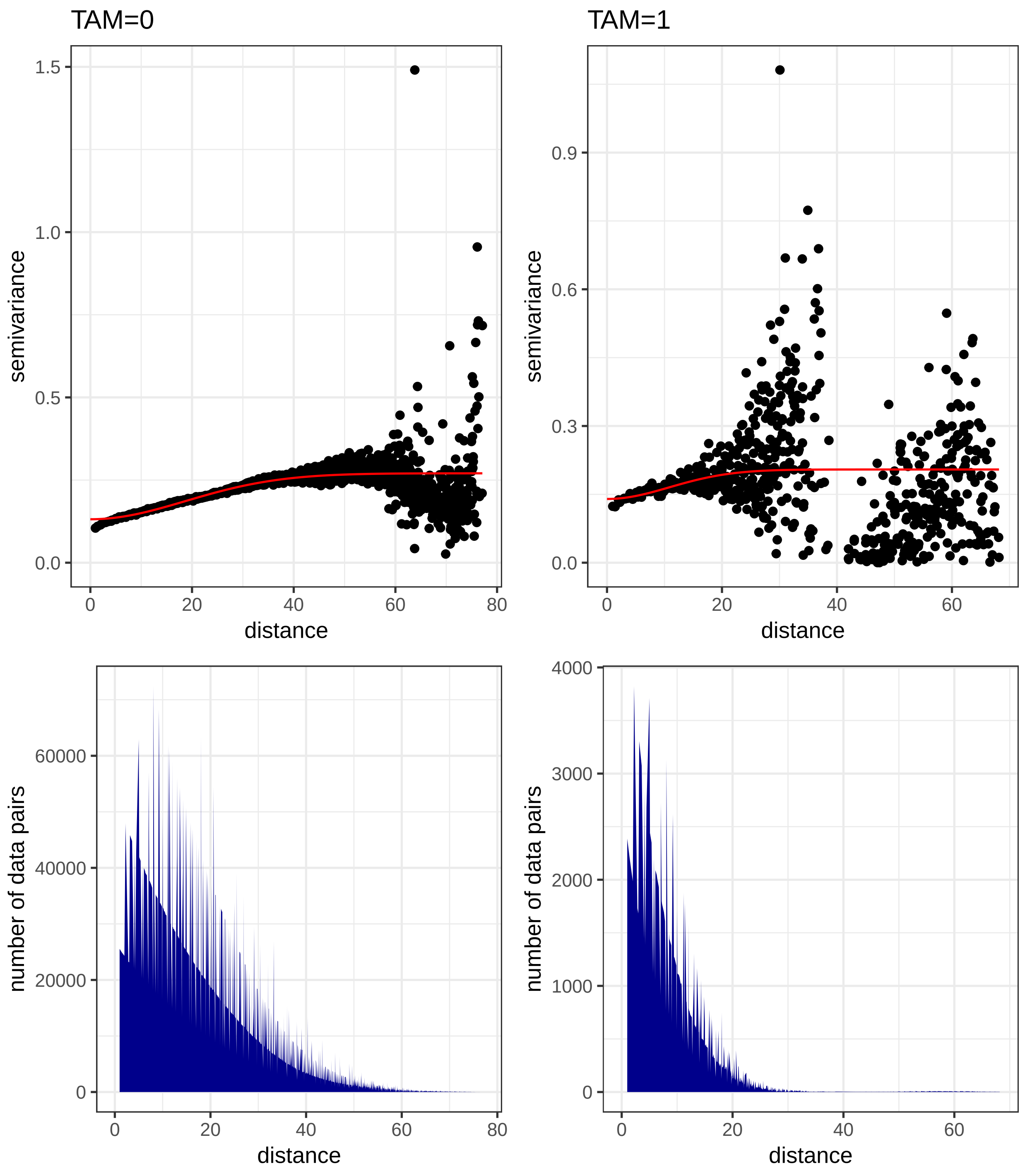
The `imagingPC` package assumes a Gaussian covariance model, and it is important to check that assumption. There are two ways to plot the results of the `estRange` function. The first uses a generic S3 plotting method.

```r
plot(rangs)
```



Overall, the Gaussian covariance model provides pretty good fit. For the non-TAM regions (TAM=0) in particular, there is a decrease in the semivariance at larger distances, and this may appear to some as a potential violation of the Gaussian assumption. To get more information on potential assumption violations, we can use the second plot function.

```
plotRangeObj(rangs)
```



This now provides two plots for each level of the TAM variable. The first is the same shown in the generic plotting method. The second plots the number of data pairs used to generate each semivariance estimate. The amount of information used to estimate the semivariance decreases rapidly as the distance increases. We can clearly see that as the number of data pairs decreases, the semivariance estimates become much more spread out. Therefore, in assessing the Gaussian assumption, one should keep in mind that (1) those larger distances occur infrequently, and so some of that behavior may be an artifact of undersampling large distances, and (2) in fitting the semivariance to a covariance model, the `geoR` package minimizes a loss function weighted by the number of observations used to estimate the semivariance at that distance. The smaller distances (such as a distance of 1) occur many more times in the dataset than the larger distances. Therefore, semivariance estimates at shorter distances are much more stable and provide stronger evidence of the form of the covariance model fit.

To get a look at the values of the estimate ranges, we can do the following:

```
rangs$estRange
#>        0        1 
#> 26.38140 15.29135
```

(6) Select Support Structures

After estimating the range, the next step is to select support structures. Without going into too much detail, support structures are collections of *support sites*, points we use to account for the spatial information. The density of support sites required is dependent on the estimated range. The `chooseStructures` function iteratively builds and checks support structures to see if the data are within a certain distance of a support site.

For this study, we assume that the underlying spatial processes differ by levels of the TAM variable, and so we must also select support structures for every level of that variable. The `chooseStructures` function automatically does this if a variable is provided using `spatialVar` in the `estRange` function.

There are several arguments in the `chooseStructures` function, but there are a few that deserve explanation here. When supports structures are built, the largest value in either the x or y direction is used to set square boundaries, and the support structures are built within and just outside of those boundaries. `cDist` is the distance between the corners of those boundaries and the outermost support sites, where the outer support sites are always placed outside those boundaries.

`sdWithin` defines the density of the support structures. It specifies within how many standard deviations the data should be of a support site. The smaller the `sdWithin`, the closer data need to be to a support site, and the more support sites will be used. Our simuations have shown that `sdWithin` should not be greater than 1.

`thresholdNumSup` is a threshold for the number of support sites allowed. It is multiplied by the number of observations in the dataset to set a limit. This limit is set for the sake of computational efficiency. If this number is exceeded, and `defaultStructure="nextHighest"`, then the support structure with the most support sites not exceeding the threshold will be used.

`sdElim` is a threshold for eliminating support sites. For `sdElim=2`, support sites that are not within 2 standard deviations of the smoothing kernel will be removed. Support sites that are removed are not counted towards the total number of support sites when considering the threshold on the number of support sites.

```
structs <- chooseStructures(rangs)
summary(structs)
#> TAM level "0": The 14alternate support structure was chosen.
#>   After removing support sites, there were 211 support sites across
#>   16 samples.
#> TAM level "1": The 33alternate support structure was chosen.
#>   After removing support sites, there were 218 support sites across
#>   15 samples.
```

```
names(structs)
#> [1] "structure_1" "structure_2" "data"        "subjectVar"  "sampleVar"
#> [6] "spatialVar"  "outcome"
```

The `structs` object is a list that includes all the necessary components lay down each set of support sites. We'll focus on the first two components, which contain the support site information for the two levels of the TAM covariate. `structure_1` is the list for the first of the ordered levels (TAM=0), and `structure_2` is the list for the second ordered level (TAM=1). We'll focus on the second list (`structure_2`) for now.

```
names(structs$structure_2)
#> [1] "recommendStruct" "recommendSd"     "nSupportReduced" "coordsU"
#> [5] "coordsData"      "defaultStatus"   "nRowSupport"
```

The first three components of this list are easy to look at.

```
structs$structure_2$recommendStruct
#> [1] 33alternate
#> 9 Levels: 5x 6x 8alternate 9x 10x 12x 14alternate ... 33alternate
structs$structure_2$recommendSd
#> [1] 10.81262
structs$structure_2$nSupportReduced
#> [1] 218
```

`recommendStruct` is the name of the support structure, which includes the number of support sites per tissue sample before removing unecessary support sites, `recommendSd` is the standard deviation used for a Gaussian smoothing kernel, and `nSupportReduced` is the number of support sites across all tissue samples after removing support sites that aren' close to data. It's clear that this number is calculated after removing support sites as 218<(33*16).

The other critical component of each list is `coordsU`. This is a data frame that contains the locations of all support sites, across all tissue samples.

```
dim(structs$structure_2$coordsU)
#> [1] 218   3
head(structs$structure_2$coordsU, n=10)
```

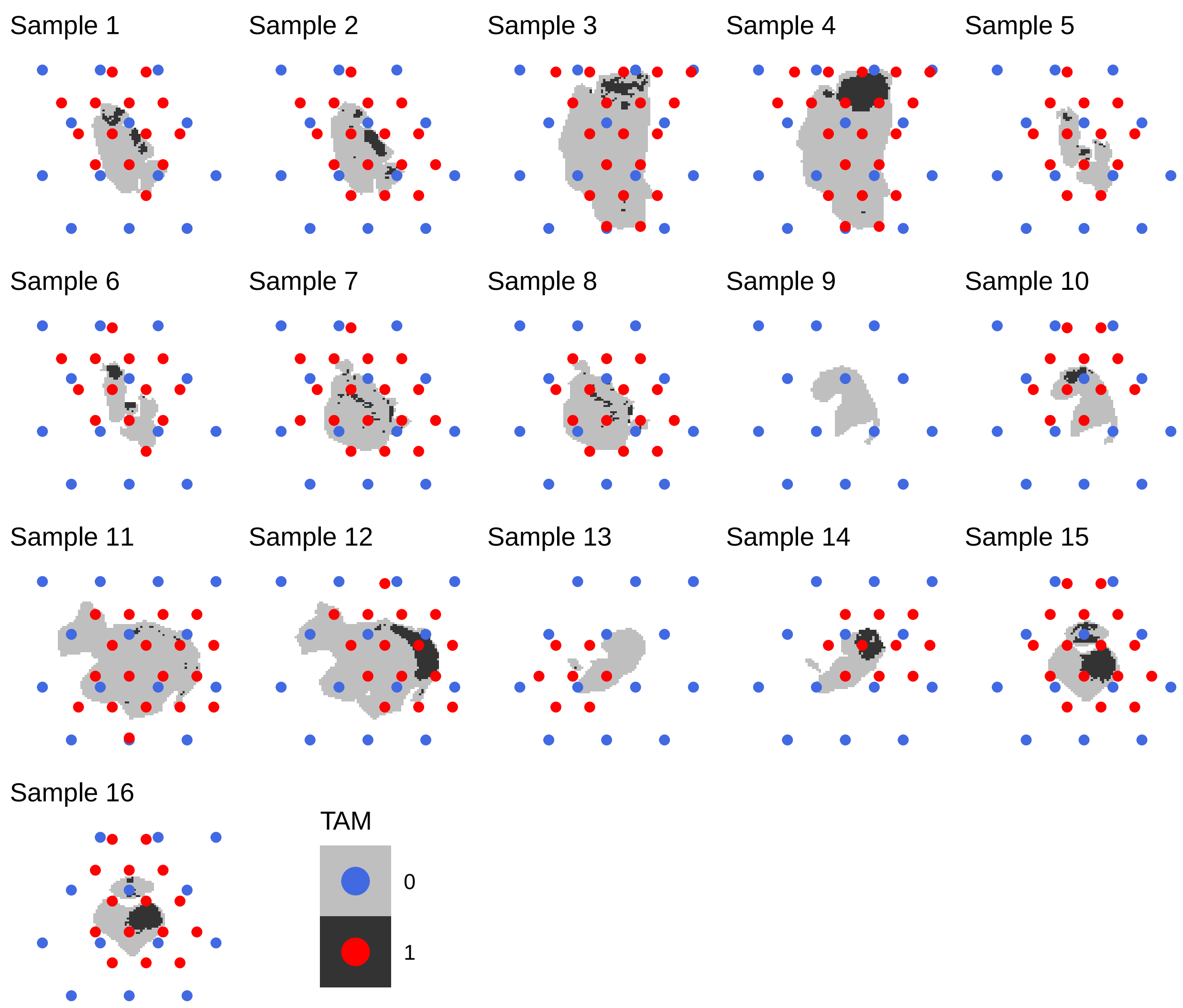
```
#>    sample   x.omega   y.omega
#> 1       1  -7.614142 38.070711
#> 2       1   7.614142 38.070711
#> 3       1 -30.456569 22.842426
#> 4       1 -15.228284 22.842426
#> 5       1   0.000000 22.842426
#> 6       1  15.228284 22.842426
#> 7       1 -22.842426  7.614142
#> 8       1  -7.614142  7.614142
#> 9       1   7.614142  7.614142
#> 10      1  22.842426  7.614142
```

If you feel strongly that additional support sites should be included, this data frame can be modified to include those. Before doing that, though, I would look through the options in the `chooseStructures` function. The density of support sites can be modified using the `sdWithin` argument.

```
structs$structure_1$recommendStruct
#> [1] 14alternate
#> Levels: 5x 6x 8alternate 9x 10x 12x 14alternate
structs$structure_2$recommendStruct
#> [1] 33alternate
#> 9 Levels: 5x 6x 8alternate 9x 10x 12x 14alternate ... 33alternate
```

Going through the `coordsU` data frame can be cumbersome, so the `plotPCStructure` function can be used to plot the support structures. If a `spatialVar` is specified in the `estRange` function, then support structures are plotted for every level of that variable.

```
plotPCStructure(structs, supSiteSize = 1.5, marginProp = 0.25)
```



(7) Create Data Vectors for Fitting Model

Before creating all of the data vectors needed to fit a model, we first need to finish creating all of the covariates. In this study we are interested in whether or not the N-glycan patterns change between primary and secondary tumors, and between non-TAM and TAM regions. It is possible that the difference in an N-glycan abundance between non-TAM and TAM regions could change depending on the tissue sample type (primary or secondary tumor). In other words, there could be an interaction. NIMBLE will not take interactions as part of a linear predictor, or at least I've been unable to get it to work, so an interaction covariate needs to be generated prior to fitting the model. The secondary and TAM covariates are both binary 0-1 variables, so an interaction can be created by just multiplying those to variables. This step would be easiest to perform before starting the steps outlined in this vignette, but we can still access the dataset in the structure object we just created.

```
structs$data$secTAM<-structs$data$secondary*structs$data$TAM
```

This creates a new variable called `secTAM` that takes a value of 1 only if the raster is from a TAM region in a secondary tumor tissue, and a value of 0 otherwise.

From there we can use the `createPCData` function to make all the data vectors necessary to fit the model. This is also the first step where we can introduce what covariates we want to put in the model. When we get to writing the model, we can trim the list down and only use a subset, but any covariate that is going to be in the model must be specified at this step.

Along with the covariate names themselves, we need to specify a couple of other arguments, the `covariateTypes` and `covariateLevels` arguments. `covariateTypes` specifies the type of each variable specified in the `covariates` argument. Currently, the only types accepted are binary, categorical, and continuous variables, which are specified using 'binary', 'categorical', and 'continuous', respectively. If a categorical variable is used, then dummy variables are generated, and the differences between all subgroups are automatically assigned as nodes in the MCMC. The `covariateLevels` argument specifies the index for each variable, with the available options being 'subject', 'sample', and 'raster'. Bayesian methods use loop structures to define models, and the `covariateLevels` argument enables downstream functions to assign the correct index to covariates. The secondary tumor variable is a sample-level covariate because its value changes between samples, but not between rasters in the same sample. The TAM variable defines the TAM status for each raster, so it is raster-level covariate. An interaction will take on the lowest-level index of the covariates used to create it, in this case the raster-level TAM covariate.

```
PCdat<-createPCData(structs, trimData = FALSE,
                    covariates = c('secondary', 'TAM', 'secTAM'),
                    covariateTypes = c('binary', 'binary', 'binary'),
                    covariateLevels = c('sample', 'raster', 'raster'))
```

The arguments `covariates`, `covariateTypes`, and `covariateLevels` must be specified using concatenated character strings. Additionally, the order in which each argument is assumed to match. For example, since the 'secondary' covariate is listed first in the `covariates` argument, the first element in the `covariateTypes` and `covariateLevels` are assumed to correspond to the 'secondary' covariate. Lastly, the length of all three arguments must be the same. If `covariateTypes='binary'` was specified, the function would stop and give an error.

Now let's explore the object just made. Like with previous functions, the `createPCData` function generates a list.

```
names(PCdat)
#>  [1] "data"            "nSubjs"          "nSamps"
#>  [4] "cNSampsPerSubj"  "cNRastPerSamp"   "totalRasters"
#>  [7] "covs"            "KMat"            "rastersPerVar"
#> [10] "nSupportSites"   "nObs"            "gTOSupportSites"
#> [13] "nVarLevels"      "subjectVar"      "sampleVar"
#> [16] "spatialVar"      "covariates"      "covariateTypes"
#> [19] "covariateLevels" "outcome"         "recStructures"
```

The list is quite large, with just over 20 elements. Each element contains either covariate information or information that tells the model how to navigate the loop structures and data used to account for spatial information. There are a few list elements that might be useful to understand. The first is the “covs” list. This is a list containing all the covariate information. Here, we are using 3 covariates, so this is a list with length 3.

```
length(PCdat$covs)
#> [1] 3
```

Each list contains is named using the covariate name and contains all the information needed to incorporate that covariate into the model. Some covariates will contain a non-missing `mapping` data frames if covariate values have been reassigned.

```
PCdat$covs$covariate1_secondary
#> $covariate
#> [1] "secondary"
#>
#> $type
#> [1] "binary"
#>
#> $level
#> [1] "sample"
#>
#> $info
#>       sample newvalue
#>  [1,]      1        1
#>  [2,]      2        1
#>  [3,]      3        0
#>  [4,]      4        0
#>  [5,]      5        1
#>  [6,]      6        1
#>  [7,]      7        0
#>  [8,]      8        0
#>  [9,]      9        1
#> [10,]     10        1
#> [11,]     11        0
#> [12,]     12        0
#> [13,]     13        1
#> [14,]     14        1
#> [15,]     15        0
#> [16,]     16        0
```

```
#>
#> $mapping
#>   oldvalue newvalue
#> 1        0        0
#> 2        1        1
```

Another important part of the PCdat object is the `nSupportSites` object. If a spatial variable is provided in the `estRange` function, then this is a data frame showing the number of support sites for each sample (row) and each level of the spatial variable (column). The `.1` and `.2` in the column names indicate that the columns represent the first and second level, resepctively, of the sorted spatial variable. This convention is used throughout the imagingPC package.

```
PCdat$nSupportSites
#>    nsups.1 nsups.2
#> 1       13      14
#> 2       13      16
#> 3       14      19
#> 4       14      20
#> 5       13      13
#> 6       13      13
#> 7       13      17
#> 8       13      14
#> 9       13       0
#> 10      13      11
#> 11      14      18
#> 12      14      15
#> 13      13       7
#> 14      13      10
#> 15      12      16
#> 16      13      15
```

If a spatial variable is provided, another data frame is created that is called `rasterPerVar`. This data frame contains the number of rasters for each sample and each level of the spatial variable. The last two columns represent cumulative raster counts that are used in defining the loop structures within the MCMC.

```
PCdat$rastersPerVar
#>    sample cov.level numraster start  stop
#> 1       1         0       790     1   790
#> 2       1         1        85   791   875
#> 3       2         0       804   876  1679
#> 4       2         1       100  1680  1779
#> 5       3         0      2331  1780  4110
#> 6       3         1       140  4111  4250
#> 7       4       0     2143  4251  6393
#> 8       4       1      351  6394  6744
#> 9       5       0      504  6745  7248
#> 10      5       1       28  7249  7276
#> 11      6       0      478  7277  7754
#> 12      6       1       54  7755  7808
#> 13      7       0      945  7809  8753
#> 14      7       1       77  8754  8830
#> 15      8       0      931  8831  9761
#> 16      8       1       63  9762  9824
#> 17      9       0      613  9825 10437
#> 19     10       0      571 10438 11008
#> 20     10       1       52 11009 11060
#> 21     11       0     2204 11061 13264
#> 22     11       1       25 13265 13289
#> 23     12       0     1924 13290 15213
#> 24     12       1      277 15214 15490
#> 25     13       0      563 15491 16053
#> 26     13       1        1 16054 16054
#> 27     14       0      427 16055 16481
#> 28     14       1      133 16482 16614
#> 29     15       0      530 16615 17144
#> 30     15       1      259 17145 17403
#> 31     16       0      568 17404 17971
#> 32     16       1      183 17972 18154
```

(8) Write Model

With all the necessary information created to run the model, the next step is to write the model. Only two arguments are required to write the model, a `PCDataObj` argument that takes the data object just created, and a `multiSampsPerSubj` argument. This is a TRUE/FALSE argument specifying whether or not there are mutliple samples per subject. If there are, then a subject-specific intercept will be incorporated into the model.

Before writing the model, there are default assumptions that should be understood. By default, all the covariates provided in the `createPCData` function are inserted into the model. If you only wish to include a subset of these, then you must specify that here in the `covariates` argument. The other important default is the assumption about the zero abundance measures in the data.

There are two assumptions that can be made about the zeros in the data. The first assumption is that the zeros are small values that fall below a limit of detection. The second is that the zero are “true” zeros, meaning they represent spots where the molecular fragment is absent. The data here were generated using a technique called matrix-assisted laser desorption/ionization Fourier-transform ion-cyclotron resonance (MALDI FT-ICR). The MALDI part of the acronym specifies the method used to release and ionize fragments, and the FT-ICR part specifies the method used to measure fragment abundance. In FT-ICR, fragment abundances are thresholded as part of data processing to account for electronic noise in the machine. For this reason, when modeling these data we will assume the zeros are left-censored observations. This means the default `typeOfZero="censored"` option is appropriate here.

When `typeOfZero="true"`, the zeros are assumed to be "true" zeros, and a marginalized two-part model is fit. In this case, there are two model parts, a binary and a marginalized part. Different covariates for these two model parts can be specified using the `covariates` and `covariatesForBinary` model arguments.

```
PCmod <- writePCModel(PCdat, multiSampsPerSubj = TRUE, typeOfZero = "censored")
PCmod$model[[2]]
#> for (i in 1:Subj) {
#>     for (j in (Sample[i] + 1):Sample[i + 1]) {
#>         for (k in (Raster[j] + 1):Raster[j + 1]) {
#>             zeros[k] ~ dpois(phi[k])
#>             phi[k] <- -loglik[k] + C
#>             mu[k] <- beta0 + beta1 * secondary[j] + beta2 * TAM[k] +
#>                 beta3 * secTAM[k] + bi[i] + PC[k]
#>             loglik[k] <- llcal.censored(y = y[k], obs = obs[k],
#>                 sigma = sigma[j], mu = mu[k])
#>         }
#>     }
#> }
```

NIMBLE uses a list structure to write a model. You can print the entire model by removing the index. I won't show that now only because it's a long model. Here is just a part of the model, but probably the most important part. In this part of the model the zeros trick is used to specify a left-censored model. μ is the mean of the log-transformed abundance measures, and PC is an assigned node that defines the process convolution approach (specified in another part of the model). Since there were multiple samples per subject (`multiSampsPerSubj=TRUE`), a subject-specific intercept (`bi`) is incorporated into the model.

(9) Run Model

We have finally reached the last step in the process, and that is running the model. The `runPCModel` function has many arguments that alter the way results are output, and only two are required. Those are the `modelObj` and `PCDataObj` arguments, which are the results of the `writePCModel` and `createPCData` functions, respectively.

For running Bayesian models there are the typical arguments specifying the number of iterations of the burn-in (`nBurnin`), the number of total iterations (burn-in + sample) to take per chain (`nIter`), and the thinning interval (`nThin`).

The `runPCModel` function incorporates a place to specify a slide variable, called `slideVar`. If the zeros are assumed to be censored observations then it is important to specify a slide variable. The slide variable should indicate which tissue samples were run together. The censoring level can change for each run of the mass spectrometer, and incorporating a slide variable will help account for that.

Overall, the `runPCModel` function works by first running the MCMC for the specified number of iterations (`nIter`), with the specified burn-in (`nBurnin`) and thin (`nThin`). The function then checks for convergence. A model is considered converged if the model coefficients have a Brooks-Gelman-Rubin (BGR) statistic of <1.10. The variance components are not checked for convergence, but in our experience, they consistently converge. Furthermore, you can check the convergence yourself to be sure, so long as you use `monitorCoefOnly=FALSE` in the `runPCModel` function. I recommend using the `gelman.diag` function in the `coda` package.

If the model does not converge, then the following happens. The `runPCModel` function will restart the MCMC

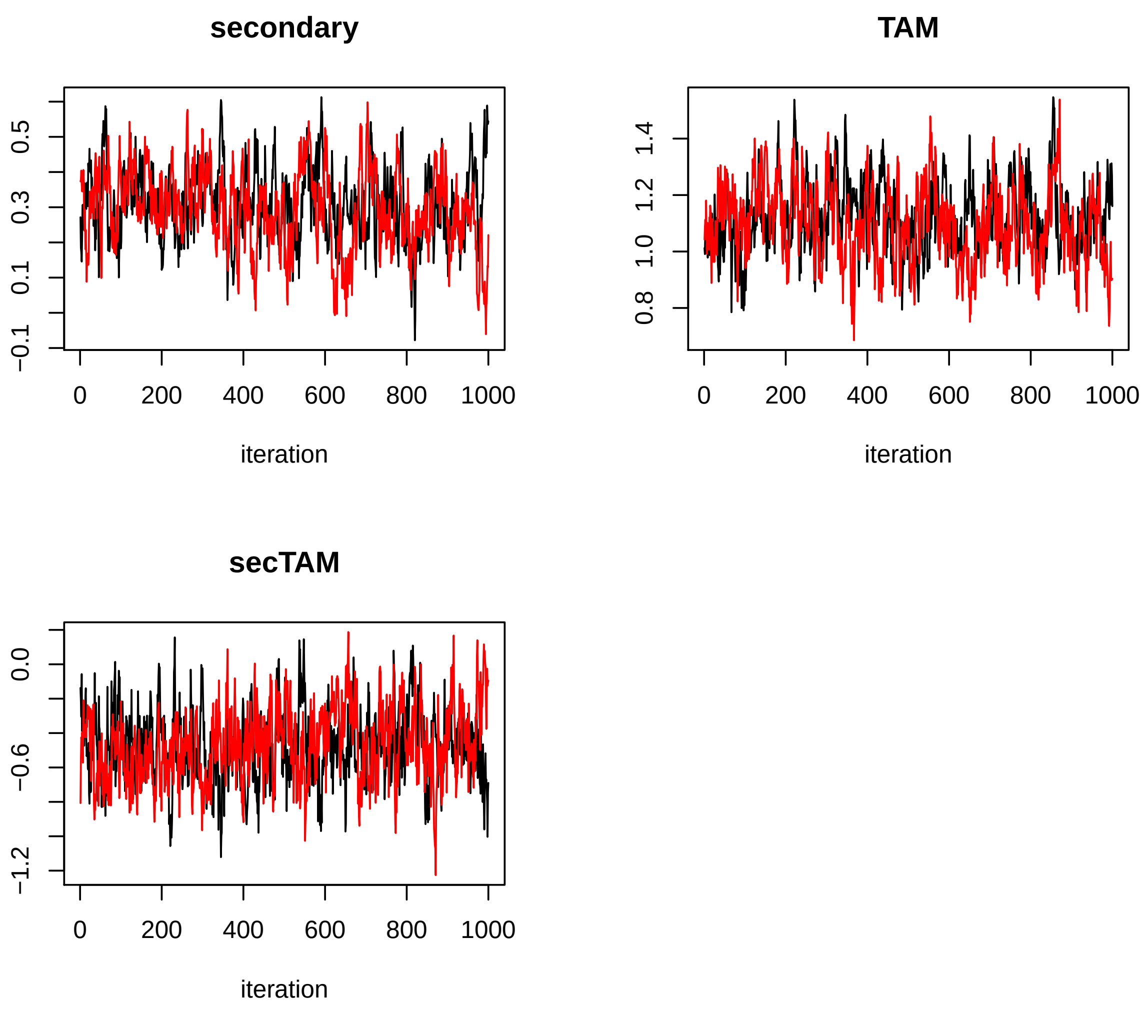
```
set.seed(100780)
PCresults <- runPCModel(modelObj = PCmod, PCDataObj = PCdat, slideVar='slide',
                        monitorCoefOnly = FALSE, sliceSamplers = TRUE,
                        nBurnin = 10000, nIter = 25000, nThin = 15)
#> defining model...
#> building model...
#> setting data and initial values...
#> running calculate on model (any error reports that follow may simply reflect missin
#> checking model sizes and dimensions... This model is not fully initialized. This is
#> model building finished.
#> compiling... this may take a minute. Use 'showCompilerOutput = TRUE' to see C++ com
#> compilation finished.
#> compiling... this may take a minute. Use 'showCompilerOutput = TRUE' to see C++ com
#> compilation finished.
#> runMCMC's handling of nburnin changed in nimble version 0.6-11. Previously, nburnin
#> running chain 1...
#> |-------------|-------------|-------------|-------------|
#> |-------------------------------------------------------|
#> running chain 2...
#> |-------------|-------------|-------------|-------------|
#> |-------------------------------------------------------|
#> The model coefficients converged.
```

We can print the results for the model coefficients using the `summary` function.

```
summary(PCresults)
#>                 Mean        2.5%        25%        50%        75%
#> secondary  0.2994426  0.08739889  0.2272502  0.2996037  0.3709450
#> TAM        1.1088607  0.86252221  1.0206023  1.1052663  1.1938248
#> secTAM    -0.4752669 -0.87419046 -0.6190021 -0.4849626 -0.3370167
#>                97.5%
#> secondary  0.51105757
#> TAM          1.37254496
#> secTAM      -0.02366639
```

From the results table, we see that the interaction, shown by `secTAM`, is significant at the 0.05 level. Therefore, we should not remove that term. The interaction term means that the TAM effect is different within primary tumors compared to secondary tumors.

```
plot(PCresults)
```



Since the interaction is significant, we may want to generate the estimates of the TAM effect in the primary tissues and the TAM effect in the secondary tissues. These are currently not automoatic output in the imagingPC package. I hope to soon provide a feature that allows users to input assigned nodes. For now, such estimates can be produced using the output sample.

Generating the estimates requires an understanding of the modeling coefficients. Refer back to the model to make sure you understand which coefficients go with which covariates, or just print the `coefNameMap` data frame from the fitted model.

```
PCresults$coefNameMap
#>   coefficient variableName
#> 1       beta1    secondary
#> 2       beta2          TAM
#> 3       beta3       secTAM
```

Recall that the secondary covariate is a binary covariate that equals 1 if the tissue sample is a secondary tumor sample and 0 if it is a primary tissue sample. The TAM covariate is a binary covariate that equals 1 if the raster stained positive for TAMs and 0 otherwise. Their interaction, therefore, is binary and equals 1 if the raster is from secondary tumor sample and stained positive for TAMs, and 0 otherwise.

Within primary tissue samples, the TAM effect is equal to β_2. In secondary tissue samples, the TAM effect is equal to $\beta_2 + \beta_3$. The MCMC sample is in a list form that can be found in `sample` in `PCresults`. to get estimates for the TAM effect in primary and secondary tissues, we'll first create one long dataset containing the sampled values for β_1 and β_2.

```
mcmcsample<-rbind(PCresults$sample[[1]],PCresults$sample[[2]])
mcmcsample<-mcmcsample[, c('beta2','beta3')]
head(mcmcsample)
#>          beta2       beta3
#> [1,] 1.0120475 -0.13743753
#> [2,] 0.9907217 -0.21409411
#> [3,] 1.0907758 -0.30556468
#> [4,] 1.0346607 -0.05706372
#> [5,] 1.0715473 -0.38891296
#> [6,] 1.0769893 -0.25338580
```

We then perform the operation $\beta_2 + \beta_3$.

```
mcmcsample<-cbind(mcmcsample,mcmcsample[,'beta2']+mcmcsample[,'beta3'])
colnames(mcmcsample)[3]<-'sec_TAMeff'
head(mcmcsample)
#>          beta2       beta3 sec_TAMeff
#> [1,] 1.0120475 -0.13743753  0.8746100
#> [2,] 0.9907217 -0.21409411  0.7766276
#> [3,] 1.0907758 -0.30556468  0.7852111
#> [4,] 1.0346607 -0.05706372  0.9775970
#> [5,] 1.0715473 -0.38891296  0.6826343
#> [6,] 1.0769893 -0.25338580  0.8236035
```

We can get the mean values using the `colMeans` function and the percentiles using the `quantile` function, making sure to specify the percentiles we want.

```
meanvals<-colMeans(mcmcsample[,c('beta2','sec_TAMeff')])
per_primTAMeff<-quantile(mcmcsample[,'beta2'], probs=c(0.025,0.5,0.975))
per_secTAMeff<-quantile(mcmcsample[,'sec_TAMeff'], probs=c(0.025,0.5,0.975))

itxSummary<-rbind(per_primTAMeff,per_secTAMeff)
itxSummary<-cbind(meanvals,itxSummary)
row.names(itxSummary)<-c('prim_TAMeff','sec_TAMeff')
itxSummary
#>              meanvals      2.5%       50%    97.5%
#> prim_TAMeff 1.1088607 0.8625222 1.1052663 1.372545
#> sec_TAMeff  0.6335938 0.3410565 0.6209102 0.985483
```

Now we have estimates of the TAM effect for both primary and secondary tumor samples. That concludes this vignette. I hope this was helpful in describing how to use the imagingPC package. In the future I hope to expand the utility of this package, such as relaxing some of the Gaussian assumptions.

References

Daltonics, Bruker. 2017. "SCiLS Lab."