Estimating the Area of a Lung Nodule from an X-ray Image

An MAA CCMath Book Chapter

Nadia S. Kennedy* Boyan S. Kostadinov† Ariane M. Masuda‡

February 14, 2022

Abstract

This is the project report, based on R Markdown, which includes the R implementations of the nine tasks from the MAA CCMath Book Chapter with the same title.

Contents

Introduction

This is a computational modeling project, which involves the estimation of a nodule area from its X-ray image. It involves interactively sampling the nodule boundary, estimating the area of the polygon formed by the sampled points, and fitting a circle to the sampled nodule points. This project can be scaffolded into the following components: determining the X-ray image dimensions, adding a coordinate system to the X-ray image, manually sampling the nodule boundary from the X-ray image, recording the x- and y-coordinates of the sampled points, creating a scatterplot of the sampled nodule boundary, computing the area of the polygon formed by the sampled points, fitting a circle to the sampled points using least squares, and transforming the circle to fully contain the lung nodule. The project uses images as matrices of pixel values, the least squares method, discretization of curves and 2D plotting.

In this project report, we implement all computations and visualizations using R and R Markdown in RStudio, as well as a number of R packages, [1, 5, 15]. We suggest that the students use the **RStudio Cloud**, RStudio Cloud [16], rather than installing R, RStudio and additional R packages on their computers.

We recommend the projects to be introduced to the students by scaffolding the mathematical ideas and coding procedures needed for them to work through the different steps that comprise the projects. Prior programming experience in R is not required, but in this case the students are expected to learn the basics of R and R Markdown.

*Mathematics Department, NYC College of Technology, CUNY, nkennedy@citytech.cuny.edu
†Mathematics Department, NYC College of Technology, CUNY, bkostadinov@citytech.cuny.edu
‡Mathematics Department, NYC College of Technology, CUNY, amasuda@citytech.cuny.edu

Technology Framework

We recommend using **R Markdown** notebooks in **RStudio** for all computations and visualizations with R.

RStudio supports R Markdown documents through the **knitr** package. The first time you try to create an R Markdown document in RStudio, you are prompted to install all required packages, including the **knitr** package, and then you can create an R Markdown document and be able to knit it into a pdf or html output, as well as a word document, if you have MS Word installed.

Working with R Markdown documents allows one to unify plain text narrative, mathematical expressions typeset in LaTeX, as well as R and Python code (using the **reticulate** package, Ushey et al. [18]. Another advantage is that it creates fully reproducible, publication quality project reports, presentations and papers.

The following software installers are available for Windows, Mac, and Linux:

- R installers: http://cran.stat.ucla.edu/
- RStudio installers: https://www.rstudio.com/

A cloud-based option (free and paid) for computing with RStudio is provided by the **RStudio Cloud**, RStudio Cloud [16], which also offers an instructor's account for creating virtual classrooms. RStudio Cloud is expected to release in 2022 a new collaboration feature for shared projects in the RStudio Cloud that will be very much like working in Google Docs.

RStudio also offers an R interface to Python via the **reticulate** R package, which provides a comprehensive set of tools for interoperability between Python and R, Ushey et al. [18]. Thus, one can use both R and Python chunks of code in the same R Markdown document, combining the power of both. With R Markdown, one can even create websites, blogs and books, using the **blogdown** and **bookdown** packages. For more details, see the resources tab on the **RStudio** website, Allaire [1].

Another great cloud-based computing environment is **CoCalc**, Stein [17], which offers free and paid accounts for online computing using SageMath, R, Python, Julia, Octave, etc. CoCalc also offers document authoring capabilities using Jupyter Notebooks based on Markdown and LaTeX.

In addition to base R, we use several R packages such as **mosaic**, **mosaicCalc** and **ggformula** [9, 12, 14] developed by **Project Mosaic**, Kaplan et al. [11], which is *"a community of educators working to develop a new way to introduce mathematics, statistics, computation and modeling to students in colleges and universities"*.

For the **mosaic** collection of packages, we recommend installing the developmental version of the **Zcalc** package from its GitHub repository, which installs the entire collection of mosaic-related packages. At the time of writing, we are using the developmental GitHub version 0.5.5 of the **mosaicCalc** package. Installing packages from GitHub is done using the **remotes** package. In particular, the **Zcalc** package can be installed from its GitHub repository with the command:

```
remotes::install_github("dtkaplan/Zcalc")
```

We use the **tidyverse** collection of packages developed by RStudio, and specifically **ggplot2** for visualizations and **dplyr** for data analysis, [20, 21]. We also use the **png** package for reading and manipulating png images, and the **knitr** package for printing tables and including graphics.

We recommend the following resources to learn the basics of R: [4, 6, 19], and R Markdown: [7, 15, 22].

The reader can find additional resources on computational-problem solving with R in the following references [2, 8, 10, 13].

Estimating the Area of a Lung Nodule from an X-ray Image

A lung nodule is a circular-shaped area, which is denser than normal lung tissue. It shows up as a white spot on an X-ray or a CT scan; see Figure 1.

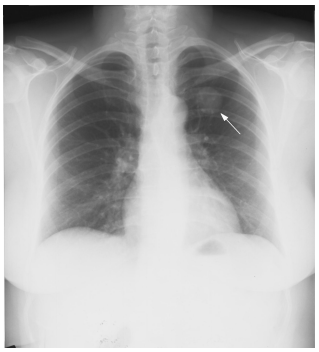

Figure 1: A circular lung nodule on an X-ray image.

The lung nodule usually represents a benign tumor such as a granuloma or hamartoma, but in around 20% of cases it represents a malignant cancer. Lung nodules can also occur in immune disorders, such as rheumatoid arthritis. If the patient has a history of smoking or the nodule is growing, the possibility of cancer may need to be excluded through further radiological studies and interventions, possibly including surgical removal.

The goal of this project is to estimate the area of a lung nodule from its X-ray image. Knowing the size of the nodule would be valuable information for the medical team that may be planning a surgical removal of the nodule or a medical treatment.

Representing an Image as a Matrix

Images are represented as 4D arrays. The four dimensions are labelled x, y, z, c. The first two are the usual spatial dimensions, the third one usually corresponds to depth (time), if there are frames at several time points, and the fourth one is for color. In our analysis, we use only the first two dimensions since we have a single frame (no time dependence), and we consider a grayscale image, so we are not interested in color.

In general, an image is just an $m \times n$ matrix $X = (x_{ij})$ of pixels, where the value of the pixel x_{ij}, in position (i, j), is represented by an integer intensity $0 < x_{ij} < W$, using a gray scale from 0 (black) to W (white), where the typical grayscale resolution is given by a value of $W = 255$. However, some image processing packages use the normalized interval $[0, 1]$ for the range of possible pixel values, and we assume this is the case now. Thus, we think of an image X as a matrix $X = (x_{ij})$ with entries in $[0, 1]$. Keep in mind that our images are still 4-dimensional, but the last two dimensions are flat.

Below is an example of a matrix of 16 pixel values and the corresponding grayscale image, shown in Figure 2.

$$X = \begin{bmatrix} 0.0000 & 0.2500 & 0.5000 & 0.7500 \\ 0.0625 & 0.3125 & 0.5625 & 0.8125 \\ 0.1250 & 0.3750 & 0.6250 & 0.8750 \\ 0.1875 & 0.4375 & 0.6875 & 0.9375 \end{bmatrix} \tag{1}$$

```
img <- load.image("nodule.png") # with imager
plot(img, xlim=c(0,ncol(img)), ylim=c(nrow(img),0))
```

In this project, we work with the X-ray image (saved as a png file), shown in Figure 3, which has 2087 rows of pixels and 2293 columns of pixels, for a total of 4,785,491 pixels. Note that we position the image on a coordinate system in such a way that the top-left vertex of the image is placed at the origin of the coordinate system.

We should keep in mind that whatever we do mathematically with matrices can be also done with images since they are represented by matrices. This could be the inspiration for exciting explorations in linear algebra done on images. For example, we can apply mathematical functions elementwise to the matrix representing the lung nodule image, shown in Figure 3. It is interesting to observe how the trigonometric transformation

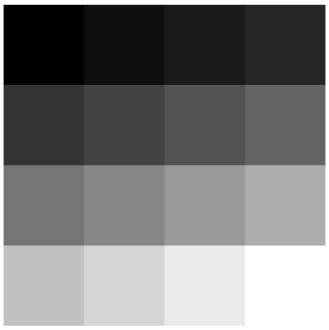

Figure 2: The grayscale image of the matrix X of pixel values.

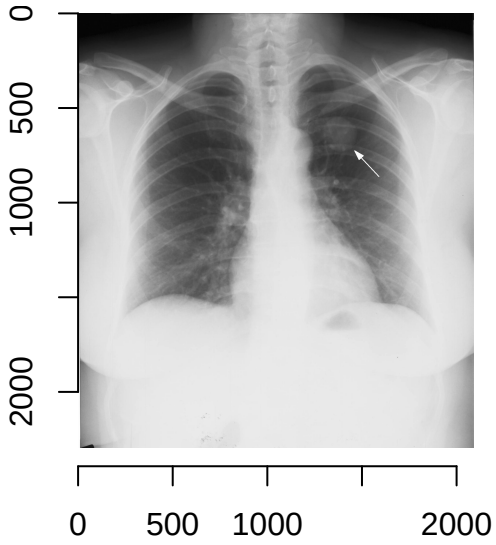

Figure 3: The X-ray image as a matrix of pixels. Source: Google Images.

in (2), switches light to dark regions and vice versa. The result is shown in Figure 4, and we give below the plotting function. This should not be surprising though, given the wave nature of the cosine function.

$$X \to \cos(2X) \tag{2}$$

```
plot(cos(2*img), xlim=c(0,ncol(img)), ylim=c(nrow(img),0))
```

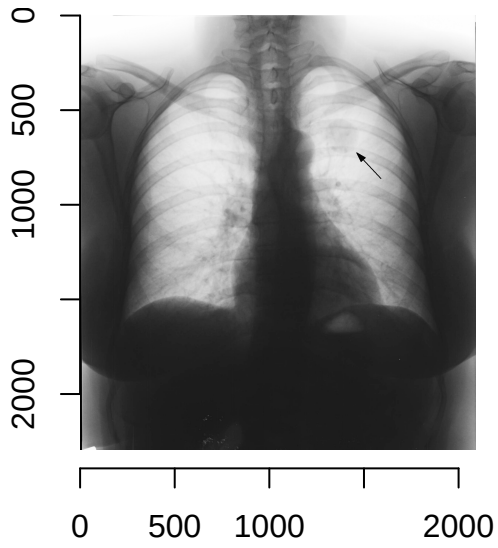

Figure 4: The trigonometric transformation $\cos(2X)$ applied to the X-ray image X.

Sampling the Tumor Boundary

We have an X-ray image of the lung nodule as a png image file, and the first step is to read the image file and plot it on a coordinate system. Depending on the computing environment that you are using, you can work with either pixels or other units, such as centimeters. Usually, on a regular X-ray image, one pixel corresponds to 175 μm (microns) and since 1 $\mu\text{m} = 10^{-4}$ cm, we can approximate the image dimensions in cm using the scaling:

$$\#\text{pixels} \times 175 \times 10^{-4}\,\text{cm}. \tag{3}$$

We can find the dimensions of the X-ray image, first in pixels, and then in cm, by using the conversion formula (3). Keep in mind that the dimensions of the image in pixels are given by the number of rows of the image as a matrix, for the y-dimension, and by the number of columns for the x-dimension. In this way, we find that the dimensions of the image in centimeters are $40.1275\,\text{cm} \times 36.5225\,\text{cm}$.

We can then use the image dimensions to plot the X-ray image in a coordinate system, using the computed dimensions in centimeters, as shown in Figure 5. The unit conversion may be done at a later stage, but it has to be done if we want to estimate the nodule area in cm^2. We provide the code below that generates Figure 5, using the `rasterImage()` function to add an image to an empty plot.

```
# create an empty plot with the image dimensions in cm
plot(NULL, type="n", xlim=c(0, xdim), ylim=c(ydim, 0), xlab="", ylab="", xaxs = "i",
     yaxs = "i", cex.axis=0.7)
# embed the image into the empty plot according to the specified dimensions
rasterImage(img, xleft=par("usr")[1], ybottom=par("usr")[3],
            xright=par("usr")[2], ytop=par("usr")[4])
```

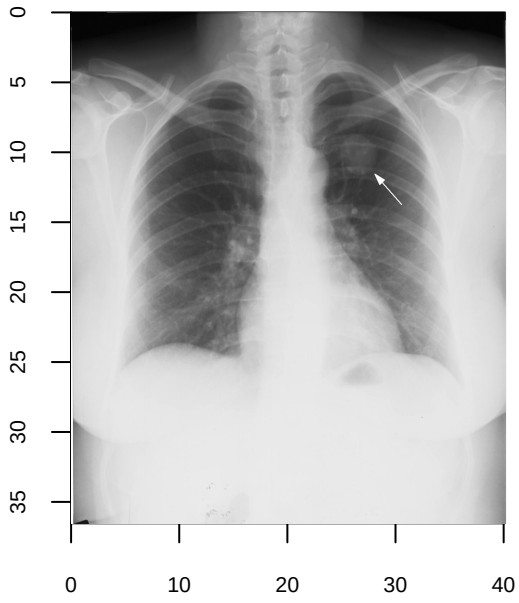

Figure 5: The X-ray image in a coordinate system, where the units are centimeters.

The next step is to interactively sample some points from the nodule boundary. We have done this using RStudio and the `locator()` R function from base graphics.

In general, a manual sampling can be implemented using a built-in function, from your computing environment of choice, to capture a mouse click. Typically, such a function records the coordinates of every point on the plot where you click with the computer mouse. This is how we can get a list with the x- and y-coordinates of the sampled points along the nodule boundary.

We provide below the code for interactively sampling points from the boundary of the nodule, which has to be run from a **plain R script in RStudio** so that the manual sampling can be done in the RStudio Plots window. For this purpose, we use the R function `locator()` to manually sample points from the nodule boundary. The code below allows us to sample 22 points from the X-ray image, once we run the code chunk above to display the X-ray image in the RStudio plotting window, and record their x- and y-coordinates into a csv file.

```
# sample points inside the RStudio Viewer
coords <- locator(n=22, type="o")
# write the x and y coordinates of the sampled points in a csv file
write.csv(coords,file="xy.csv", row.names=FALSE)
```

We sample 22 points from the boundary of the nodule, and since have their x- and y-coordinates, we can plot them on the top of the X-ray image using the same coordinate system, as shown in Figure 6. It is very important to keep in mind that we have to use the same coordinate system for the X-ray image, when we sample the points from the nodule boundary, and when we plot them on the top of the X-ray image, otherwise the sampled points will be shifted relative to the nodule boundary.

Computing the Area of the Polygon with the Shoelace Formula

We can compute the area of the polygon formed by the points sampled from the nodule boundary using the **Shoelace formula**, also known as the **Surveyor's area formula**, given below:

$$\text{Area} = \frac{1}{2}\left|\begin{vmatrix} x_1 & x_2 \\ y_1 & y_2 \end{vmatrix} + \begin{vmatrix} x_2 & x_3 \\ y_2 & y_3 \end{vmatrix} + \cdots + \begin{vmatrix} x_{n-1} & x_n \\ y_{n-1} & y_n \end{vmatrix} + \begin{vmatrix} x_n & x_1 \\ y_n & y_1 \end{vmatrix}\right|, \tag{4}$$

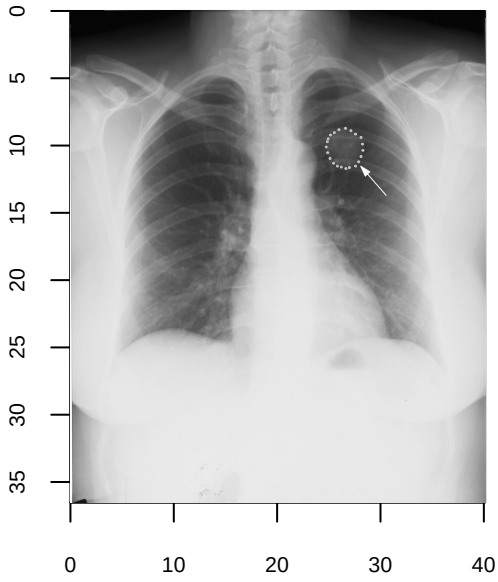

Figure 6: The points sampled from the nodule boundary, superimposed on the X-ray image, in the same coordinate system, in units of centimeters.

where $x_1, x_2, \ldots, x_n$ are the x-coordinates, and $y_1, y_2, \ldots, y_n$ are the y-coordinates of the polygon vertices.

By expanding the 2×2 determinants in (4), we obtain an equivalent form:

$$\text{Area} = \frac{1}{2}\Big|\sum_{i=1}^{n-1} x_i y_{i+1} - \sum_{i=1}^{n-1} x_{i+1} y_i + x_n y_1 - x_1 y_n\Big|. \tag{5}$$

It is a good mathematical exercise to derive the Surveyor's area formula. A good reference for its derivation can be found in Braden [3].

A good computational exercise is to implement the Surveyor's formula (4) using a `for loop`, which is illustrated in the code chunk below.

```
xycoords <- read.csv("xy.csv")
x<-xycoords$x # vector of x-coordinates of polygon vertices
y<-xycoords$y # vector of y-coordinates of polygon vertices
n<-length(x) # length of vector
area<-0 # initialize area
for (i in 1:(n-1)){
  area <- area + x[i]*y[i+1] - x[i+1]*y[i]
}
# polygon area
area <- 0.5*abs(area + x[n]*y[1] - x[1]*y[n])
```

Note that the `for loop` implements the difference of the sums $\sum_{i=1}^{n-1} x_i y_{i+1} - \sum_{i=1}^{n-1} x_{i+1} y_i$.

Thus, according to the Shoelace formula, the area of the polygon formed by the sampled points is `area` $= 7.98$ cm^2. To confirm that our computations are correct, we can use the **pathmapping** package, which has the `Shoelace()` function for computing the area.

```
library(pathmapping)
poly <- cbind(x,y) # x and y coordinates of the polygon vertices
polyarea <-shoelace(poly) # polygon area
```

We implement this check above, and it gives `polyarea` $= 7.98\,\mathrm{cm}^2$, which confirms our result.

We can also implement the Shoelace version (5) using a **vectorized programming** approach rather than a `for loop`. Getting used to a functional programming approach based on vectors and matrices and functions applied to them is a very useful coding skill to develop. The key idea here is to represent the two sums in (5) in terms of the vectors $\mathbf{x}$ and $\mathbf{y}$ consisting of the x- and y-coordinates of the polygon vertices, respectively.

Let $\mathbf{x} = (x_1, x_2, \ldots, x_n)$ and $\mathbf{y} = (y_1, y_2, \ldots, y_n)$ be the vectors of x- and y-coordinates of the n polygon vertices. Then, the first sum in (5) can be represented in terms of the vectors $\mathbf{x}$ and $\mathbf{y}$ as follows:

$$\sum_{i=1}^{n-1} x_i y_{i+1} = \texttt{sum(x[1:(n-1)]*y[2:n])}, \tag{6}$$

where the notation `x[1:(n-1)]` means the subvector of $\mathbf{x}$ formed by the first $n-1$ elements of $\mathbf{x}$, that is, all elements of $\mathbf{x}$ except for the last one, and `y[2:n]` means the subvector of $\mathbf{y}$ formed by all elements of the vector $\mathbf{y}$, except for the first one. The multiplication operator `*` is defined as acting elementwise on the two vectors, which have the same length of $n-1$. Thus, the elementwise product `x[1:(n-1)]*y[2:n]` results in a vector of size $n-1$ where the components are obtained by elementwise multiplication. Finally, the expression `sum(x[1:(n-1)]*y[2:n])` simply sums the elements of the vector resulting from the product and it returns the sum on the left-hand side of (6).

Similarly, we can express the second sum in (5) by reversing the roles of the x- and y-coordinates, which represents the "shoe lacing":

$$\sum_{i=1}^{n-1} x_{i+1} y_i = \texttt{sum(x[2:n]*y[1:(n-1)])}. \tag{7}$$

These observations lead us to a vectorized approach for computing the area of the polygon:

```
area<-0.5*abs(sum(x[1:(n-1)]*y[2:n]) - sum(x[2:n]*y[1:(n-1)]) + x[n]*y[1] - x[1]*y[n])
```

The polygon area computed above is `area` $= 7.98\,\mathrm{cm}^2$, just as before.

A good exercise in complex algebra is to rewrite the Surveyor's formula (4) in a complex form. We can represent the kth polygon vertex on the plane as the complex number $z_k = x_k + i\,y_k$, where $i = \sqrt{-1}$ is the imaginary unit.

```
gf_point(y ~ x, col="blue", size=0.6) %>%
  gf_point(col="blue", size=0.3, geom="polygon", fill=NA) %>%
  gf_point(mean(y) ~ mean(x), col="hotpink", size=0.8) %>%
  gf_text(mean(y) ~ mean(x), label="C", col="hotpink", vjust = 1,nudge_y = 0.2) %>%
  gf_labs(x="x", y="y")
```

In Figure 7, we show the sampled points and the polygon they form. In complex coordinates, the Shoelace formula has the following complex form:

$$\text{Area} = \frac{1}{2}\left|\text{Im}\left(\sum_{k=1}^{n} z_k \bar{z}_{k+1}\right)\right|, \tag{8}$$

where the complex vector $\mathbf{z} = (z_k)_{k=1}^{n+1}$ is defined to be periodic modulo n, in the sense that $z_{n+1} = z_1$, and the complex elements $(z_k)_{k=1}^{n}$ represent the n polygon vertices. Here $\bar{z}_{k+1}$ is the complex conjugate, and Im() represents taking the imaginary part. We can implement this complex form of the Shoelace formula by first computing the sum, then taking the imaginary part of the sum followed by its absolute value, and then dividing the result by 2:

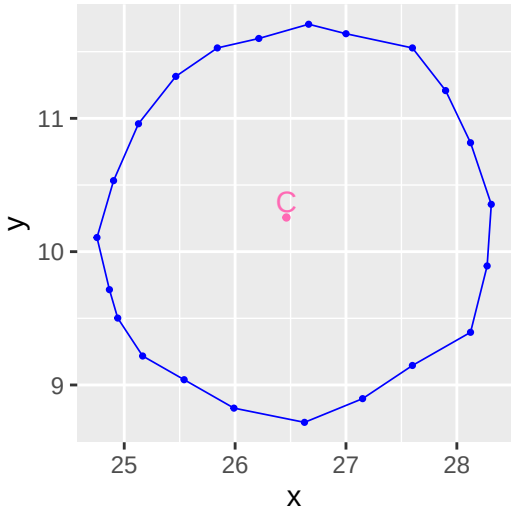

Figure 7: The polygon formed by the sampled points, where the point C is the center of mass of the polygon.

$$\sum_{k=1}^{n} z_k \bar{z}_{k+1} = \texttt{sum(z[1:n]*Conj(z[2:(n+1)]))}. \tag{9}$$

This leads to a vectorized approach for computing the polygon area in a complex form:

$$A = \texttt{0.5*abs(Im(sum(z[1:n]*Conj(z[2:(n+1)]))))}. \tag{10}$$

```
z <- x + 1i*y # vector of complex numbers representing the polygon vertices
n<-length(x) # length of vector x
z <-c(z,z[1]) # periodic extension of z mod n
area <- 0.5*abs(Im(sum(z[1:n]*Conj(z[2:(n+1)])))) # complex form of Shoelace formula
```

The complex form of the Shoelace formula confirms that `area` $= 7.98\,\text{cm}^2$.

The Surveyor's Method for Estimating a Circle from Data

The Surveyor estimates first the location of the center of the circular area. From this point we measure the distances to each one of the polygon vertices that form the boundary of the circular area. Then the distances are averaged to obtain $\bar{r}$ as an average radius of a circle approximating the circular area. The circular area is then approximated by:

$$A = \pi \bar{r}^2. \tag{11}$$

We can find the center of the polygon forming the circular area by computing its **center of mass** $C = (\bar{x}, \bar{y})$, where $\bar{x}$ is the average of the x-coordinates of the polygon vertices, and $\bar{y}$ is the average of the y-coordinates of the n polygon vertices:

$$\bar{x} = \frac{1}{n}\sum_{k=1}^{n} x_k, \quad \bar{y} = \frac{1}{n}\sum_{k=1}^{n} y_k, \tag{12}$$

where $n = 22$ is the number of data points sampled from the nodule boundary. The center of mass C of the polygon is shown in Figure 7.

We can compute the average radius $\bar{r}$ by computing the distance r_k from the center of mass $(\bar{x}, \bar{y})$ to each polygon vertex (x_k, y_k) using the Euclidean distance:

$$r_k = \sqrt{(x_k - \bar{x})^2 + (y_k - \bar{y})^2}. \tag{13}$$

We can compute all these $n = 22$ distances at once using the `mutate()` function from the **dplyr** package.

```
xycoords <- read.csv("xy.csv") # read the x and y coordinates of sampled points
xyrdata <- mutate(xycoords, r = sqrt((x - mean(x))^2 + (y - mean(y))^2))
```

Note that in the `mutate()` function, the formula for `r` is applied to each element of the `x` and `y` columns and the result is a new column labeled `r`, of the same size, but `mean(x)` and `mean(y)` are the averages obtained from applying the `mean()` function to the entire columns `x` and `y`, thus giving us $\bar{x}$ and $\bar{y}$, being the x- and y-coordinates, respectively, of the center of mass. In Table 1, we display the beginning of `xyrdata`.

Table 1: The first 5 rows of the `xyrdata`.

x	y	r
26.62589	8.719669	1.545381
27.14996	8.897418	1.522944
27.59917	9.146267	1.588803
28.12324	9.395116	1.870752
28.27297	9.892814	1.846580

Averaging the vector `r` in the dataframe `xyrdata` gives us the average radius $\bar{r}$. We can visualize the circle $(x - \bar{x})^2 + (y - \bar{y})^2 = \bar{r}^2$ with center $(\bar{x}, \bar{y})$ and radius $\bar{r}$, and superimpose it over the data points. For this purpose, we use the parametric equations of the circle to generate the vectors of x- and y-coordinates for a large number of points on the circle, using a vector of values for the parameter t.

$$x = \bar{x} + \bar{r}\cos(t), \tag{14}$$
$$y = \bar{y} + \bar{r}\sin(t), \tag{15}$$
$$t \in [0, 2\pi].$$

This is implemented in the code below, where the dataframe `circle` contains the vectors of x- and y-coordinates of points on the circle, generated from (14)-(15), for a vector of t values. Note that `with(xyrdata,...)` makes the variables `x`, `y` and `r` in the dataframe `xyrdata` available for computing, and that is why we can compute `mean(x)`, `mean(y)` and `mean(r)`.

```
circle <- with(xyrdata, tibble(t = seq(0, 2*pi, len=100), x = mean(x) + mean(r)*cos(t),
                               y = mean(y) + mean(r)*sin(t)))
```

Now, we can create a scatterplot of the sampled points and superimpose over it a plot of the circle using the data in the dataframe `circle`. Figure 8 shows the sampled points and the circle, where we use the `ggplot()` function from the **ggplot2** package.

```
ggplot(data=xyrdata, aes(x=x,y=y)) +
  geom_point(col="black", size=0.8) +
  geom_polygon(col="gray", size=0.4, fill="gray88", alpha=0.7) +
  geom_point(aes(x=mean(x),y=mean(y)), col="hotpink", size=0.8) +
  geom_path(data=circle, aes(x=x,y=y), col="hotpink") +
  coord_fixed()
```

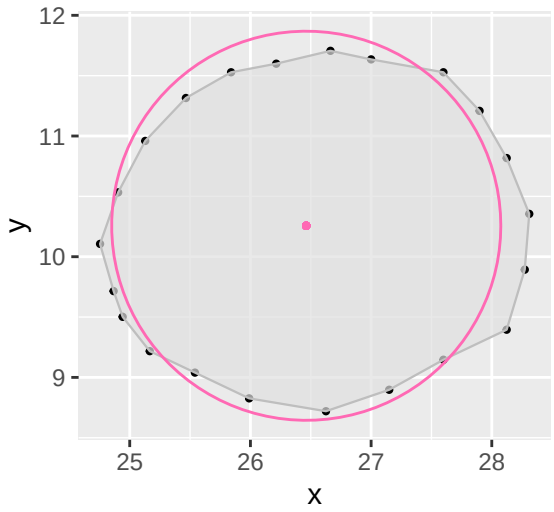

Figure 8: The Surveyor's estimated circle superimposed on the sampled points.

We can compute the parameters of the circle by summarizing the dataframe `xyrdata` and computing the average radius $\bar{r}$ and the center of the circle being the center of mass of the data points $(\bar{x}, \bar{y})$.

```
stats <- xyrdata %>% summarize(rbar=mean(r), xbar=mean(x), ybar=mean(y))
```

The area of the approximating circle is Area $= \pi\bar{r}^2 = 8.17\,\text{cm}^2$.

Figure 8 shows that the Surveyor's estimated circle is not a great fit for the sampled points that form the polygon representing the circular area to be estimated.

Fitting a Circle to the Sampled Points Using Nonlinear Least Squares

Even though the Surveyor's method for estimating a circle does not produce a good fit, we can still use it to initialize a more sophisticated approach to fitting a circle to the nodule boundary by using **non-linear least squares**. The equation of a circle with center at (x_0, y_0) and radius r is given by:

$$(x - x_0)^2 + (y - y_0)^2 = r^2. \tag{16}$$

If all data points (x_j, y_j) for $j = 1, 2, \ldots, n$ $(n = 22)$ were on the circle, we would have n equations:

$$(x_j - x_0)^2 + (y_j - y_0)^2 = r^2,\ j = 1, 2, \ldots, n. \tag{17}$$

However, clearly the data points are not located on a circle so we cannot have all of these equations satisfied. The best thing we can do is to minimize the **sum of squared errors** (SSE), where the jth error is given by $e_j = (x_j - x_0)^2 + (y_j - y_0)^2 - r^2$:

$$\text{SSE}(x_0, y_0, r) = \sum_{j=1}^{n} ((x_j - x_0)^2 + (y_j - y_0)^2 - r^2)^2. \tag{18}$$

This now becomes an optimization problem since we want to minimize the function $\text{SSE}(x_0, y_0, r)$ with respect to the model parameters x_0, y_0 and r. Since the equation of the circle depends quadratically on the circle parameters, this is an example of **nonlinear least squares**. Minimizing a function of three variables is no easy task, but we can use a general-purpose optimizer such as the `optim()` function from base R. It is based on Nelder-Mead, Newton and conjugate-gradient algorithms, but it also has an option for simulated

annealing. The `optim()` function takes two arguments: the first one is a vector with initial guesses for all model parameters, and the second one is the R function to be minimized (the default) with respect to these parameters.

For the initial guesses, we use the Surveyor's estimates $\bar{r}$ and $(\bar{x}, \bar{y})$ for the radius and center, respectively. The objective function to be minimized is the $\text{SSE}(x_0, y_0, r)$, and it has to be defined as an R function with a single, vector argument whose components are the model parameters. We use in the function the previously defined vectors `x` and `y` of the x- and y-coordinates of the sampled points, and we implement the right-hand side of (18) in a vectorized way, as shown in the code chunk below.

```
SSE<-function(params) sum(((x-params[1])^2 + (y-params[2])^2 - params[3]^2)^2)
```

Note that the function argument `params` is a vector whose elements are the circle parameters, with `params[1]` being x_0, `params[2]` being y_0, and `params[3]` being r. Supplying the vector of initial guesses for the model parameters in the correct order and the function `SSE` is all that is needed to minimize (the default optimization) the sum of squared errors.

```
out<-optim(c(stats$xbar,stats$ybar,stats$rbar), SSE) # minimizer
```

The result `out` is a list with several elements, but we use only the first two. The first element `par` is the vector of fitted parameters, the first two parameters being the x- and y-coordinates of the center of the fitted circle, and the third parameter being the radius of the best fit circle. We can extract the first element of the list with `out$par`, which returns the vector of fitted parameters. In particular, we get:

$$x_0 = 26.5547,\ y_0 = 10.1908,\ r = 1.6225,\ \text{SSE}(x_0, y_0, r) = 3.3592. \tag{19}$$

The area of the circle of best fit is then $\text{Area} = \pi r^2 = 8.27\,\text{cm}^2$.

The second element is the minimum value of the function, which is 3.36, and it would have been zero if all points lied on a circle. We can now create a dataframe `lscircle` with the x- and y-coordinates of the circle of best fit using the fitted circle parameters. This is done with the code below, where `with(out,...)` makes the variables in the list `out` available for computing without having to extract them from the list. The `tibble()` function creates the dataframe with three variables `t`, `x` and `y`, where `t` is a vector of t values in the interval $(0, 2\pi)$ of length 100, and the vectors of x- and y-coordinates are generated by the parametric equations of the circle of best fit using the optimal values of the fitted parameters.

```
lscircle<-with(out, tibble(t = seq(0, 2*pi, len=100), x = par[1] + par[3]*cos(t),
                           y = par[2] + par[3]*sin(t)))
```

In addition to the circle of best fit, we also want to stretch the radius of the fitted circle, keeping the center fixed, so that it contains all sampled points. We can do that by computing the vector of Euclidean distances between each sampled point and the center of the fitted circle, and then finding the maximum distance in this vector. This is done in the vectorized code below, where the resulting maximum distance is held in `rmax`.

```
rmax <- max(sqrt((xycoords$x - out$par[1])^2 + (xycoords$y - out$par[2])^2))
```

Thus, the area of the smallest circle that contains all sampled points is $\text{Area} = 10.21\,\text{cm}^2$.

Just like before, we can create a dataframe `mincircle` with the x- and y-coordinates of this circle, containing all sampled points, centered at the same point as the fitted circle, but having a radius `rmax`.

```
mincircle <- tibble(t = seq(0, 2*pi, len=100), x = out$par[1] + rmax*cos(t),
                           y = out$par[2] + rmax*sin(t))
```

Finally, we can visualize the sampled data points from the nodule boundary, and the polygon they form, and we can superimpose over them both circles, the circle of best fit and the smallest circle that contains all sampled points. We implement this visualization in Figure 9 using the `ggplot()` function from the **ggplot2** package.

```
ggplot(data=xyrdata, aes(x=x,y=y)) +
  geom_point(col="black", size=0.8) +
  geom_polygon(col="gray", size=0.4, fill="gray88", alpha=0.7) +
  geom_path(data=lscircle, aes(x=x,y=y), col="hotpink") +
  geom_path(data=mincircle, aes(x=x,y=y), col="deeppink") +
  coord_fixed()
```

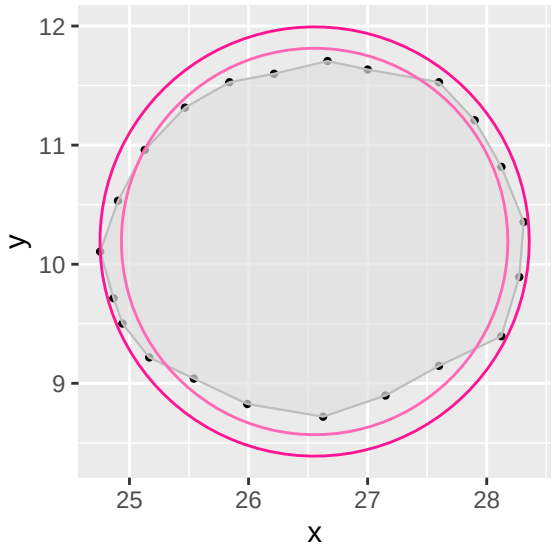

Figure 9: The circle of best fit and the smallest circle containing all points sampled from the nodule boundary, and the polygon they form.

We close this project with the final visualization in Figure 10, where we superimpose on the X-ray image, both the points sampled from the nodule boundary, and the smallest circle that contains them all, stretched from the circle of best fit, keeping the center fixed.

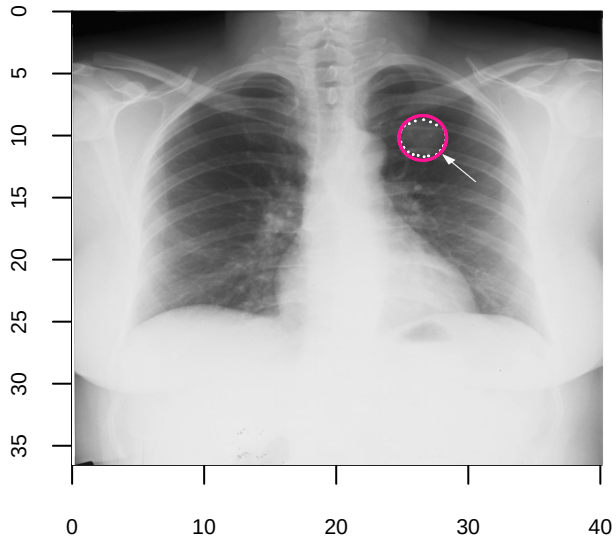

Figure 10: The sampled points from the nodule boundary, superimposed on the X-ray image, along with the optimal circle containing the nodule, obtained from nonlinear least squares.

Random Search Optimization

The sum of squared errors (18) used for the nonlinear least squares can be minimized using a **random search optimization** algorithm, as the one given below.

```
set.seed(1111)
xycoords <- read.csv("xy.csv")
x<-xycoords$x # vector of x-coordinates of polygon vertices
y<-xycoords$y # vector of y-coordinates of polygon vertices
# Sum of squared errors using the vectors of x and y coordinates
S<-function(A,B,R) sum(((x-A)^2 + (y-B)^2 - R^2)^2)
S <- Vectorize(S) # vectorized function
# Initialize the random search using Surveyor's circle
a<-26.45 # lower limit of A
b<-26.65 # upper limit of A
c<-10.10 # lower limit of B
d<-10.30 # upper limit of B
w<-1.50  # lower limit of R
z<-1.70  # upper limit of R
N<-1e5   # number of iterations
# begin random search
Amin<-runif(1,a,b) # random number from U(a,b)
Bmin<-runif(1,c,d) # random number from U(c,d)
Rmin<-runif(1,w,z) # random number from U(w,z)
Smin<-S(Amin,Bmin,Rmin) # objective function to minimize
for (n in 1:N){
  A<-runif(1,a,b) # update A with a random number from U(a,b)
  B<-runif(1,c,d) # update B with a random number from U(c,d)
  R<-runif(1,w,z) # update R with a random number from U(w,z)
  Sval<-S(A,B,R)  # update objective function
  if (Sval < Smin) {
    Amin<-A       # update parameter A
    Bmin<-B       # update parameter B
    Rmin<-R       # update parameter R
    Smin<-Sval    # update function value
  }
}
```

To initialize the random search optimization, we use the parameters of the circle estimated by the Surveyor's method, and we take small intervals around them where we do the random search to find the minimum of the objective function. The random search optimization returns the optimal values for the parameters of the circle of best fit: $x_0 = 26.5436$, $y_0 = 10.101$, $r = 1.614$, $S = 3.3593$.

References

[1] JJ Allaire. RStudio, 2021. https://www.rstudio.com/products/rstudio/.

[2] Nadia Benakli, Boyan Kostadinov, Ashwin Satyanarayana, and Satyanand Singh. Introducing computational thinking through hands-on projects using R with applications to calculus, probability and data analysis. *International Journal of Mathematical Education in Science and Technology, Taylor & Francis*, 48(3):393–427, 2017. URL https://doi.org/10.1080/0020739X.2016.1254296.

[3] Bart Braden. The Surveyor's Area Formula. *The College Mathematics Journal, Taylor & Francis*, 17(4): 326–337, 1986. doi: 10.1080/07468342.1986.11972974.

[4] W. John Braun and Duncan J. Murdoch. *A First Course in Statistical Programming with R*. Cambridge University Press, 2021. doi: 10.1017/9781108993456.

[5] John Chambers. The R Project for Statistical Computing. https://www.r-project.org/.

[6] Jonathan Cornelissen. Introduction to R Online - DataCamp Learn. https://app.datacamp.com/learn/courses/free-introduction-to-r. Accessed: 2022-02-14.

[7] Garrett Grolemund, Xie Yihui, and J. J. Allaire. *R Markdown: The Definitive Guide*. CRC Press, 2021. https://bookdown.org/yihui/rmarkdown/.

[8] Owen Jones, Robert Maillardet, and Andrew Robinson. *Introduction to Scientific Programming and Simulation Using R*. CRC Press, second edition, 2014.

[9] Daniel Kaplan and Randall Pruim. ggformula: Formula Interface to the Grammar of Graphics. https://CRAN.R-project.org/package=ggformula, 2021. Accessed: 2022-02-14.

[10] Daniel Kaplan, Cecylia Bocovich, and Randall Pruim. Modeling-based Calculus with R/mosaic. *The UMAP Journal*, 36(1):29, 2015. URL https://www.comap.com/product/?idx=1470.

[11] Daniel T. Kaplan, Randall Pruim, and Nicholas J. Horton. Project MOSAIC. http://www.mosaic-web.org/.

[12] Daniel T. Kaplan, Randall Pruim, and Nicholas J. Horton. mosaicCalc: Function-Based Numerical and Symbolic Differentiation and Antidifferentiation. https://CRAN.R-project.org/package=mosaicCalc, 2020. Accessed: 2022-02-14.

[13] Boyan Kostadinov, Johann Thiel, and Satyanand Singh. Creating Dynamic Documents with R and Python as a Computational and Visualization Tool for Teaching Differential Equations. *PRIMUS, Taylor & Francis*, 29(6):584–605, 2019. URL https://doi.org/10.1080/10511970.2018.1472161.

[14] Randall Pruim, Daniel T. Kaplan, and Nicholas J. Horton. The mosaic Package: Helping Students to Think with Data Using R. *The R Journal*, 9(1):77–102, 2017. https://journal.r-project.org/archive/2017/RJ-2017-024/index.html.

[15] R Markdown. https://rmarkdown.rstudio.com/. Accessed: 2022-02-14.

[16] RStudio Cloud. https://rstudio.cloud/. Accessed: 2022-02-14.

[17] William Stein. CoCalc – Collaborative Calculation. https://cocalc.com/. Accessed: 2022-02-14.

[18] Kevin Ushey, JJ Allaire, and Yuan Tang. Reticulate: an R Interface to Python. https://rstudio.github.io/reticulate/. Accessed: 2022-02-14.

[19] Hadley Wickham and Garrett Grolemund. *R for Data Science*. O'Reilly Media, 2017. https://r4ds.had.co.nz/.

[20] Hadley Wickham, Winston Chang, Lionel Henry, Thomas Lin Pedersen, Kohske Takahashi, Claus Wilke, Kara Woo, Hiroaki Yutani, and Dewey Dunnington. ggplot2: Create Elegant Data Visualisations Using the Grammar of Graphics. https://CRAN.R-project.org/package=ggplot2, 2021. Accessed: 2022-02-14.

[21] Hadley Wickham, Romain François, Lionel Henry, and Kirill Müller. dplyr: A Grammar of Data Manipulation. https://CRAN.R-project.org/package=dplyr, 2021. Accessed: 2022-02-14.

[22] Yihui Xie, Christophe Dervieux, and Emily Riederer. *R Markdown Cookbook*. CRC Press, 2022. https://bookdown.org/yihui/rmarkdown-cookbook/.