2

Acquiring and Manipulating Morphometric Data

The first step of any statistical or morphometric analysis is to gather and organize raw data. R offers a graphical interface that allows diverse datasets to be directly captured from digital images, as we will see. Some basic image manipulation and analysis is introduced as well. This chapter explains how to gather morphometric data in an appropriate way and how to assign them to R objects. The quality of data acquisition determines part of the quality of the results: measurement error results both from the user and from the accuracy of the different tools used for measuring data. It may happen that data are incomplete for some objects (e.g., some objects can be broken, and all the landmarks or distances cannot be captured). The last part of the chapter explains how to handle missing data and to estimate measurement error.

2.1 Collecting and Organizing Morphometric Data

2.1.1 Collecting Data

Traditionally, morphometric data are sets of distance, angle, perimeter, surface, or volume measurements. One can obtain them in theory from coordinates of landmarks or pseudolandmarks (see Chapter 1). Manual tools (rulers, calipers), hardware (digitizers, tracing tables), or position of a pointer on digital image allows these coordinates to be recorded in the x, y, and eventually z-dimensions.

Collecting Distances

Although one can directly record distances from digitized pictures, one usually collects these measurements directly from objects using calipers or rulers, or any other manual device. Later, these data are stored in handsheets or directly in a computer file. For microscopic and bigger objects (like geographic data), distances are obligatorily obtained through images acquired through different kinds of lenses and mirrors (microscope, telescope, magnifying lens, etc.) and captured to computer files using a digital camera, or to photographic films using a conventional camera. It is necessary

here to photograph a size standard (a ruler or any other known distance) together with the pictured object for retrieving the size of structures. Nowadays, images can be numerized and their properties can be analyzed using elementary computer image analysis. Distances can be calculated using the coordinates of endpoints for a given measurement.

Collecting Coordinates of Points

Any image-analyzing system can collect 2D cartesian coordinates of points on a picture file. It is important to estimate the size of the image by using a scale such that

$$True\ coordinates = \frac{Image\ coordinates \times Size\ of\ the\ scale\ on\ the\ image}{True\ size\ of\ the\ scale}\ .$$

This relationship holds for matrices of coordinates as well (configuration matrices), meaning that the scalar multiplication is applied to the matrix of coordinates.

The relative positions of coordinates for an object can be estimated from coordinates of points digitized using a cartesian reference. Users are thus invited to define the origin (which can be any landmark of the object or outside the object), and the x-axis, y-axis, and possibly the z-axis directions to fix the orientation of the coordinate system. Defining the orientation of the coordinate system all at once avoids any further problems with reflection between configurations.

For some purposes, we need to define the cartesian system directly from the object. If every landmark you want to digitize on your object is not accessible to your digitizing device without repositioning the object, a protocol is necessary (you may wish to record coordinates on both dorsal and ventral surfaces of an object). For example, you may need to reverse the object to localize landmarks on the ventral side. For keeping the relative position between points digitized on the ventral side and dorsal side invariant to object reposition, you can record coordinates on the whole object by defining your system coordinates with three landmarks shared by the ventral and dorsal side.

One can directly record coordinates on objects using hardware (2D or 3D digitizers) connected to the computer. Some 3D hardware (confocal microscope, Computed Tomography system, etc.) records a series of images that are basically equally spaced slices of the objects. Coordinates of pixels are recorded by the way of 3D image analysis.

Collecting Surfaces and Perimeters

For 2D objects, surfaces and perimeters can be appraised by letting the computer count the pixels of the structure of interest. The surface corresponds to the number of pixels of the object on the image multiplied by a scaling factor. The perimeter corresponds to the number of pixels involved in the outline multiplied by a scaling factor. The pixel is a unit of surface measurement, thus the scaling factor should take into account the width and length of the pixel. For polygonal and known geometrical shapes, classical geometric addition or multiplication of distance measurements

obtained from landmark coordinate data allows the calculation of surfaces or perimeters.

Collecting 3D Surfaces or Volumes

A 3D surface corresponds to the sum of the pixels involved in outlines of each slice belonging to the surface of the object scaled by the inter-image space and the pixel length and width ratio. For volume, the traditional way is to submerge the object in a liquid and measure the volume of liquid that has been displaced, following the Archimedes principle. In addition, volumes can be estimated as the sum of the pixels of the object for each image multiplied by an appropriate scale factor. The surface unit is the pixel so the measure of the volume should take into account the width and length of the pixel and the inter-image space. One can use voxel size to estimate the volume, if one works on voxel formatted files.

Collecting Images

One can obtain most morphometric properties of objects based on pixel (automated) counting or on pixel coordinates with elementary computer image analysis. A large number of inexpensive digital cameras are now available on the market. Hardware used for collecting 3D properties of objects is more expensive and consists of different machines: scanners, stereographic devices, 3D digitizers .... Three dimensional scanners provide a series of images.

Given an appropriate image format, location and color of the pixels of the image can be stored and analyzed. Binary or black and white images are defined by a series of pixels that take two values (0 and 1); gray-scale or monotonic images have pixel values ranging from 1 to 2^n. Here an image file can be organized as a matrix object (column and row indices corresponding to indices of pixel coordinates, and cells to the pixel value). For color images, each pixel location is related to three values, each ranging from 1 to 2^n. In this latter case, the data can be organized in a three-dimensional array where cell values correspond to the level of one of the three basic color channels (red, green, blue). Similarly, each color channel can be stored in a matrix.

Images for morphometric analysis can be stored in files and reworked using your favorite application software for image analysis, but this latter task can be achieved with the help of R as we will see in Section 2.2.6.

2.1.2 Organizing Data

R can navigate in your repertories to read files. R has a working directory returned through the command getwd(). This repository is set for a session using the following command for Windows:

```
> setwd("C:/data")
```

or for Linux:

```
> setwd("/home/juju/data")
```

For files that are not in the working directory of R, you have to specify the path as a string for the function to find their location (e.g., `"/home/juju/myfile.R"`).

It is very important to organize your data files to optimally work through R. If your data are gathered on several files, it is important to keep the same organization throughout these files, to allow repetitive operations to be easily run by your computer. Additionally, it is good to name your file so it can clearly be recognizable. If the names of all files in a directory follow a given logic, R can open a series of related files using loops or logical indexing in the correct order. For this purpose, we can use R to generate sequences and concatenate strings. It is worth using loops if you need to perform the same operation on several files. It will be easier if part of the name of your files follow some regular sequence, without which you will have to create and write a probably long vector containing the name of all your files, or eventually an extra file.

We will first learn how to organize data generated with R before we learn how to read and organize data that are outside of the R environment. As an example, I present the possible ways to store the essential information about a configuration (its names and the coordinates of landmarks) in various R objects. Configurations of landmarks usually correspond to matrices M of p rows for landmarks and of k columns for dimensions. As for distances, the input of landmarks must follow the same order for each configuration to allow comparisons between configurations.

```
>juju<-scan()
  1:  0.92 100.00 0.99 100.25 1.07 99.99 1.26 99.99 1.11
  10: 99.87 1.16 99.70 1.00 99.86 0.87 99.72 0.88 99.89
  19: 0.74 99.98
 Read 20 items
>JUJU<-matrix(juju, 10, 2, byrow=T)
>colnames(JUJU)<-c("x", "y")
>rownames(JUJU)<-paste("Lan", 1:10, sep="")

>JUJU
        x       y
Lan1  0.92 100.00
Lan2  0.99 100.25
Lan3  1.07  99.99
Lan4  1.26  99.99
Lan5  1.11  99.87
Lan6  1.16  99.70
Lan7  1.00  99.86
Lan8  0.87  99.72
Lan9  0.88  99.89
Lan10 0.74  99.98
```

Alternatively, the configuration can be defined to a 1 by $k \times p$ matrix that will store a succession of coordinates x, y, and z, for the p landmarks. The $k \times p$ configuration

matrix (*M*) can be coerced in the corresponding m vector. m is the vectorized form of the *M* matrix.

```
>JOJO<-matrix(c(0.72,100.32,0.75,100.36,0.77,100.32,0.81,
+  100.32,0.77,100.29,0.77,100.24,0.73,100.28,0.7,100.26,
+  0.7,100.3,0.67,100.33), 10, 2, byrow=T)
>colnames(JOJO)<-c("x", "y")
>rownames(JOJO)<-paste("Lan", 1:10, sep="")
>t(JOJO)
     Lan1   Lan2   Lan3   Lan4   Lan5   Lan6   Lan7   Lan8
x    0.72   0.75   0.77   0.81   0.77   0.77   0.73   0.70
y 100.32 100.36 100.32 100.32 100.29 100.24 100.28 100.26
 Lan9  Lan10
x    0.7   0.67
y  100.3 100.33

>as.vector(t(JOJO))
[1]   0.72 100.32   0.75 100.36   0.77 100.32   0.81 100.32
[9]   0.77 100.29   0.77 100.24   0.73 100.28   0.70 100.26
[17]  0.70 100.30   0.67 100.33
>c1name<-expand.grid(colnames(JOJO),rownames(JOJO))
```

The function `expand.grid` *creates a* `data.frame` *object using all combinations of a group of supplied vectors. It has been used here for creating the new row names of the second matrix.*

```
>JOJO1<-matrix(t(JOJO), 1, 20)
>rownames(JOJO1)<-"JOJO"
>colnames(JOJO1)<-paste(c1name[,1], c1name[,2], sep="-")
>JOJO1

      x-Lan1 y-Lan1 x-Lan2 y-Lan2 x-Lan3 y-Lan3 x-Lan4
JOJO   0.72 100.32   0.75 100.36   0.77 100.32   0.81
      y-Lan4 x-Lan5 y-Lan5 x-Lan6 y-Lan6 x-Lan7 y-Lan7
JOJO 100.32   0.77 100.24   0.73 100.28   0.7 100.26
      x-Lan8 y-Lan8 x-Lan9 y-Lan9 x-Lan10 y-Lan10
JOJO   0.7  100.3  0.67 100.33   0.77  100.29
```

A collection of n configurations can be stored in an `array` object of p, k, n dimensions if all configurations $\mathbf{M_{1 \rightarrow n}}$ contain the same numbers of landmarks and dimensions. The full array can be easily transformed in a `data.frame` object with rows corresponding to objects and columns to the succession of x, y (and z for 3D) coordinates for each landmark. Alternatively, the configuration set can be stored as a list if objects contain different numbers of landmarks and dimensions. Here we organize the collection of the configurations through R in three different ways.

Example of configuration set assigned to an `array` *object:*

```
>array(cbind(JUJU, 2, JOJO, 2), dim=c(10, 2, 2))
, , 1

        [,1]    [,2]
 [1,]  0.92 100.00
 [2,]  0.99 100.25
 [3,]  1.07  99.99
 [4,]  1.26  99.99
 [5,]  1.11  99.87
 [6,]  1.16  99.70
 [7,]  1.00  99.86
 [8,]  0.87  99.72
 [9,]  0.88  99.89
[10,]  0.74  99.98

, , 2

        [,1]    [,2]
 [1,]  0.72 100.32
 [2,]  0.75 100.36
 [3,]  0.77 100.32
 [4,]  0.81 100.32
 [5,]  0.77 100.29
 [6,]  0.77 100.24
 [7,]  0.73 100.28
 [8,]  0.70 100.26
 [9,]  0.70 100.30
[10,]  0.67 100.33
```

Example of configuration set assigned to a `data.frame` *object:*

```
>JJ<-data.frame(rbind(as.vector(t(JUJU)),
+               as.vector(t(JOJO))))
>rownames(JJ)<-c("JUJU", "JOJO")
>colnames(JJ)<-paste(c1name[,1], c1name[,2], sep="-")
>JJ

      x-Lan1 y-Lan1 x-Lan2 y-Lan2 x-Lan3 y-Lan3 x-Lan4
JUJU    0.92 100.00   0.99 100.25   1.07  99.99   1.26
JOJO    0.72 100.32   0.75 100.36   0.77 100.32   0.81
      y-Lan4 x-Lan5 y-Lan5 x-Lan6 y-Lan6 x-Lan7 y-Lan7
JUJU   99.99   1.11  99.87   1.16  99.70   1.00  99.86
JOJO  100.32   0.77 100.29   0.77 100.24   0.73 100.28
      x-Lan8 y-Lan8 x-Lan9 y-Lan9 x-Lan10 y-Lan10
JUJU    0.87  99.72   0.88  99.89    0.74   99.98
JOJO    0.70 100.26   0.70 100.30    0.67  100.33
```

Example of a configuration set assigned to a `list` *object:*

```
>list(JUJU=JUJU, JOJO=JOJO)
$JUJU
         x        y
Lan1  0.92 100.00
Lan2  0.99 100.25
Lan3  1.07  99.99
Lan4  1.26  99.99
Lan5  1.11  99.87
Lan6  1.16  99.70
Lan7  1.00  99.86
Lan8  0.87  99.72
Lan9  0.88  99.89
Lan10 0.74  99.98


$JOJO
         x        y
Lan1  0.72 100.32
Lan2  0.75 100.36
Lan3  0.77 100.32
Lan4  0.81 100.32
Lan5  0.77 100.29
Lan6  0.77 100.24
Lan7  0.73 100.28
Lan8  0.70 100.26
Lan9  0.70 100.30
Lan10 0.67 100.33
```

Other ways to organize data are possible. Later in the text, I will usually distinguish sets of configurations (arrays) from single configuration (matrix) with the respective letters *A* and *M*. In the following section, we will see how to import data files in the environment of R.

2.2 Data Acquisition with R

2.2.1 Loading and Reading R Datafiles

The function `data` loads data files followed by certain extensions (.R, .r, .rda, .rdata, .txt, .csv . . .) and is searching by default for sets in every currently loaded package. If the dataset is not assigned to a new object, the name of the data object corresponds to the name of the dataset without any extensions.

```
>data(iris)
>iris[1,]
  Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1          5.1         3.5          1.4         0.2  setosa
```

Several morphometric sets have been stored in R (notably in the packages shapes and ade4) and can be used as tutorials. It is also possible to use the function `data` for opening your own datasets.

2.2.2 Entering Data by Hand

We can enter data by hand using the CLI and/or by filling arguments of some functions. However, it is not very convivial, and the number of entries can become very large and boring. R provides a data editor on some platforms using the function `de` and typing the code `de(NA)`. Once data are typed, the function assigns the value to a new object of the `list` class. Using the data editor depends on the operating system, and users are invited to read the online help. Note that like any other function, arguments can be passed through the prompt. Similarly, an empty data frame can be edited by the command `edit(as.data.frame(NULL))`, and assigned to an object. It is then filled by hand by the user. The `transform` function allows you to transform certain objects in a `data.frame`, and/or to append new columns calculated by manipulation on the original columns.

```
>transform(matrix(rnorm(4),2,2), new=X1*X2)
           X1         X2          new
1   0.54885879 0.3121743   0.171339622
2  -0.01887397 0.4368677  -0.008245428
```

The package Rcmdr ("R commander") provides a graphical interface with several menus that allows direct operations using the menus and mouse buttons. It can do the job of a data editor as well.

```
> library(rcmdr)
> Commander()
```

Working with "R commander" is really straightforward and there is no need for long explanations. My opinion is that you should avoid using it if you are a newcomer to R, because it does not ask you to think too much when producing any kind of beautiful graph. You must be able to think to become a star in shape statistical analysis.

2.2.3 Reading Text Files

Usually raw data are recorded in a data frame where columns represent variables while rows represent individuals. For R to read the file easily, each element of the dataset stored on your computer files must be separated by the same separator character. Blank spaces are among the most commonly used, but R handles any other kinds, like tabulations. It is important that no elements or strings of your data contain the separator character used for separating elements.

The functions `read.table` and `scan` can read ascii files. The first argument of these functions contains the name of the file (with its extension) and the path (if the file is not in the working directory). The following series of arguments can be completed to indicate among others: number of lines to skip, data types, field separator character, and decimal separator.

`read.table` reads tabular data (to avoid error messages, be careful that each line contains the same number of observations and check your field separator; in the case of missing data, fill empty cells with "NA" (`NA` for nonavailable). A number

of related functions can be used (`read.csv`, `read.delim`, and `read.fwf`) that may have some advantages regarding the way that the initial file is stored.

`scan` is much more flexible and can interpret the file as a vector or a list.

Functions of some packages have been developed for importing and reading files in other formats (Excel, SAS, SPSS, matlab, newick, html...), and access SQL-type databases. If you are a Matlab lover, the R.matlab package allows communications between R and a Matlab servers. A similar package exists for using R language and packages on Matlab (RLink).

2.2.4 Reading and Converting Image Files

R does not read just text files; it can also read and display an image file on a graphical device, thanks to the development of several packages and functions (Table 2.1). Accessing pixel values and their coordinates consists of extracting either the slots or the appropriate components of objects returned by these functions.

Table 2.1. Functions for importing image files and related packages

Function	Package	Image Format	Returned Object
read.pnm	pixmap	pbm, pgm, ppm	objects of diverse pixmap classes
read.jpeg	rimage	jpeg	image.matrix object
readTiff	rtiff	tiff	pixmap object

Some code is provided below to demonstrate how R handles various image files with different packages.

Working with the package rimage.

```
>library(rimage)
>x <- read.jpeg(system.file("data", "cat.jpg",
+    package="rimage"))
>dim(x[,,1])
>is.array(x)
[1] TRUE
```

Returning dimensions of the Red level matrix.

```
>dim(x[,,1])
[1] 420 418
```

Working with the Package pixmap.

```
>library(pixmap)
>x <- read.pnm(system.file("pictures/logo.ppm",
+    package="pixmap")[1])
>str(x)
Formal class 'pixmapRGB' [package "pixmap"] with 8 slots
```

```
 ..@ red     : num [1:77, 1:101] 1 1 1 1 1 1 1 1 1 1 ...
 ..@ green   : num [1:77, 1:101] 1 1 1 1 1 1 1 1 1 1 ...
 ..@ blue    : num [1:77, 1:101] 1.000 1.000 0.992 ...
 ..@ channels: chr [1:3] "red" "green" "blue"
 ..@ size    : int [1:2] 77 101
 ..@ cellres : num [1:2] 1 1
 ..@ bbox    : num [1:4] 0 0 101 77
 ..@ bbcent  : logi FALSE
>dim(x@red)
[1] 77 101
>is.matrix(x@red)
[1] TRUE
>x@red[35,8]; x@green[35,8]; x@blue[35,8]
[1] 0.4392157
[1] 0.4392157
[1] 0.4078431
>x@red[35,8]*255
[1] 112
```

Pixels of a gray-scale image take values comprised between zero and one with a step of 1/255, while RGB images have pixels with three values ranging in the same way. Functions in the above packages easily convert color images to gray-scale images.

```
>library(rimage)
>x <- read.jpeg(system.file("data", "cat.jpg",
+     package="rimage"))
>y <- (rgb2grey(x))
>rm(x)
>dim(y)
[1] 420 418
```

The functions that work with image files usually return very long objects that consume the memory. These objects can be removed from the environment using the function rm, if they are no longer useful. As a frequent user, I recommend not using JPEGs exceeding 100 Ko.

Some image file formats are not readable for R, thus it is necessary to convert the format of images. This operation can waste a lot of time if there is a need to convert a large series of images. R offers the possibility to invoke a system command from the CLI with the function shell. One can therefore call the command of an image converter program directly from R. Among software for manipulating image files, Imagemagick[1] is free and available online and can be installed on many operating systems to convert and manipulate image files. Once you have installed Imagemagick, you can directly work from your R environment as follows:

```
>setwd("/usr/lib/R/library/rimage/data")
>shell("convert cat.jpg cat.bmp")
```

---

[1] www.imagemagick.org

2.2.5 Graphical Visualization

Visualizing data is often necessary to check whether you have correctly collected your data. The generic function `plot` plots diverse R objects. Its first arguments can be two vectors that contain the x and y coordinates of points of an object. Alternatively, a two-column matrix can be passed as argument to produce the same result. Many arguments (see the online help) can be entered directly through the function, including arguments concerning parameters of the graphical device. Alternatively, many parameters (font labels, margin widths, frame size, x and y-axis ranges, title positions, background color, and many other) can be set before to open the graphical device through the function `par`. Low-level plotting commands can add objects on the graph using a series of functions (`points`, `abline`, `text`, `segments`, `locator`, `arrows`, `polygon`, `legend`, `rectangle`, `axis`, `lines`, etc.).

Here is an example of script for plotting the configurations called `JUJU` and `JOJO` (see Fig. 2.1).

Draw the two configurations, with two different landmark symbols.
Add the title "Sea stars" to the graph.

```
>plot(rbind(JUJU, JOJO), pch=c(rep(21, 10), rep(20, 10)),
+     asp=1,main="Sea stars")
```

Draw a polygon with vertices corresponding to the configuration coordinates. Use different line types for each configuration.

```
>polygon(JUJU, lty=3)
>polygon(JOJO, lty=1)
```

Add landmark labels for the `JUJU` *configuration.*

```
>text(JUJU, labels=1:10, pos=c(3,2,3,3,3,4,3,2,3,3))
```

Alternatively, we can specify the position of labels relative to the coordinates they design in the plot with the function `identify`, *by left clicking (to the left, right, top, or bottom) near the landmark of interest.*

```
>identify(JUJU, labels=1:10, pos=TRUE)
```

Add eyes to `JUJU`.

```
>points(c(0.95,1.05,0.955,1.055),rep(99.95,4),pch=21,
+     cex=c(1,1,2,2),bg=c(1,1,NA,NA))
```

Add a mouth to `JOJO`.

```
>lines(c(0.76, 0.74,0.72), c(100.3,100.29,100.3))
```

Note that the function `lines` *can draw outlines as well.*

The function `persp` can display 3D plots. The first useful step is often to define the space that is necessary for the display, then points and lines are drawn using low-level plot commands, with the function `trans3d`.

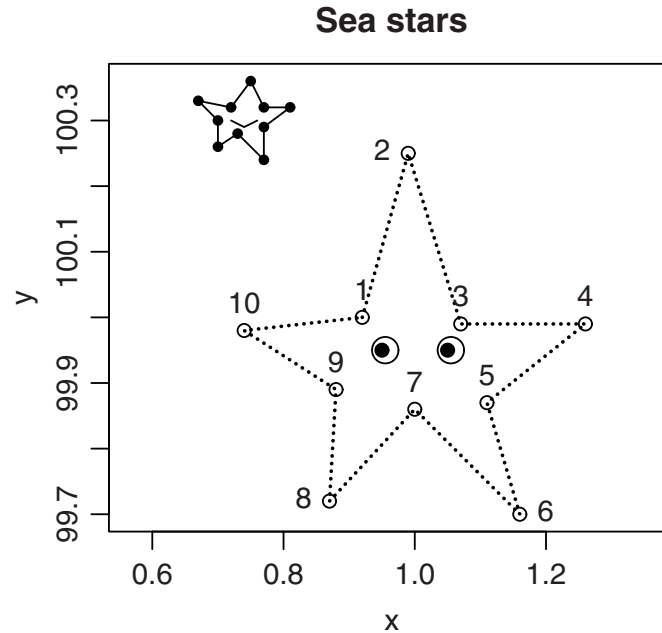*Assign a 3D configuration matrix to the* `tetra` *object.*

Sea stars

Fig. 2.1. Plotting 2D configurations with the function `plot`, and low-level commands

```
>tetra<-matrix(c(0,2,1,1,0,0,1,0.5,0,0,0,1),4,3)
```

Define the range of the space for the visualization and indicate the orientation of the cartesian system to be projected on the screen.

```
>x<-seq(-0.5,2.5, length=10)
>y<-x
>z<-matrix(-0.5, 10, 10)
>res<-persp(x,y,z,zlim=c(-0.5, 1.5),theta=30,phi=15,r=10,
+           scale=F)
```

Plot the landmarks of the configuration.

```
>points(trans3d(tetra[,1],tetra[,2],tetra[,3],pm=res),
+       col=1,pch=16)
```

Add segments ("links") between the landmarks of the configuration.

```
>lines(trans3d(tetra[-3,1],tetra[-3,2],tetra[-3,3],pm=res),
+      col=1,lw=2)
>lines(trans3d(tetra[-c(2,3),1],tetra[-c(2,3),2],
+      tetra[-c(2,3),3],pm =res),lw=2)
>lines(trans3d(tetra[-1,1],tetra[-1,2],tetra[-1,3],pm=res),
+      lty=3,lw=2)
>lines(trans3d(tetra[-c(2,4),1],tetra[-c(2,4),2],
+      tetra[-c(1,4),3],pm=res),lty=3,lw=2)
```
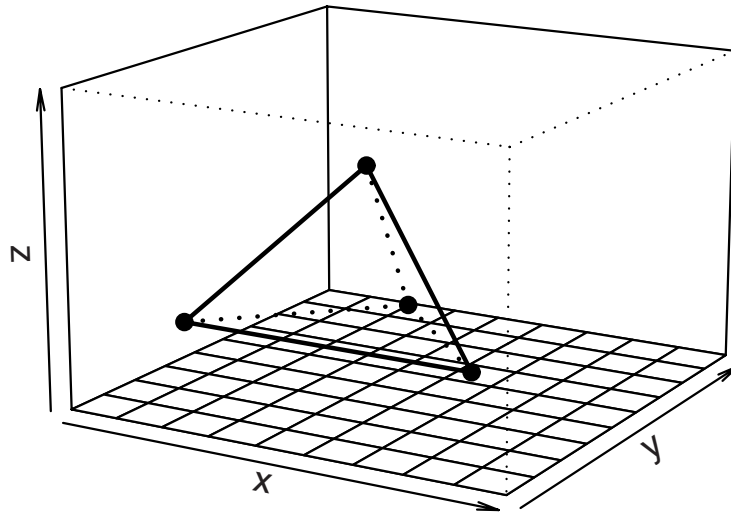
Fig. 2.2. Plotting 3D configurations with the function `persp`

The `Xll()` command opens a supplementary graphical device, and several graphs can be presented on a single device with the `layout` function. The `dev.set (devicename)` and `dev.off(devicename)` commands, respectively, close and activate the device named "devicename." One returns the list of devices by typing the command `dev.list()`. The script below opens three graphs on a single devices and displays x, y, and z-projections of a 3D configuration, the resulting plots are displayed in Fig. 2.3.

```
>layout(matrix(1:4, 2,2))
>res<-persp(x, y, z, zlim=c(-0.5, 2.5),theta=30,phi=30)
>points(trans3d(tetra[,1],tetra[,2],tetra[,3],pm = res),
+       col=1,pch=16)
>lines(trans3d(tetra[-3,1],tetra[-3,2],tetra[-3,3],pm=res),
+    col=1,lw=2)
>lines(trans3d(tetra[-c(2,3),1],tetra[-c(2,3),2],
+    tetra[-c(2,3),3],pm =res),lw=2)
>lines(trans3d(tetra[-1,1],tetra[-1,2],tetra[-1,3],pm=res),
+    lty=3,lw=2)
>lines(trans3d(tetra[-c(2,4),1],tetra[-c(2,4),2],
+    tetra[-c(1,4),3],pm=res),lty=3,lw=2)
>plot(tetra[,2:3],asp=1,xlab="y",ylab="z",
+     main="xprojection")
>polygon(tetra[,2:3])
>plot(tetra[,-2],asp=1,xlab="x",ylab="z",
+     main="yprojection")
>polygon(tetra[,-2])
>lines(tetra[c(2,4),-2])
```

```
>plot(tetra[,1:2],asp=1,xlab="x",ylab="y",
+    main="zprojection")
>polygon(tetra[,1:2])
>lines(tetra[c(2,4),-3])
>lines(tetra[c(1,3),-3])
```
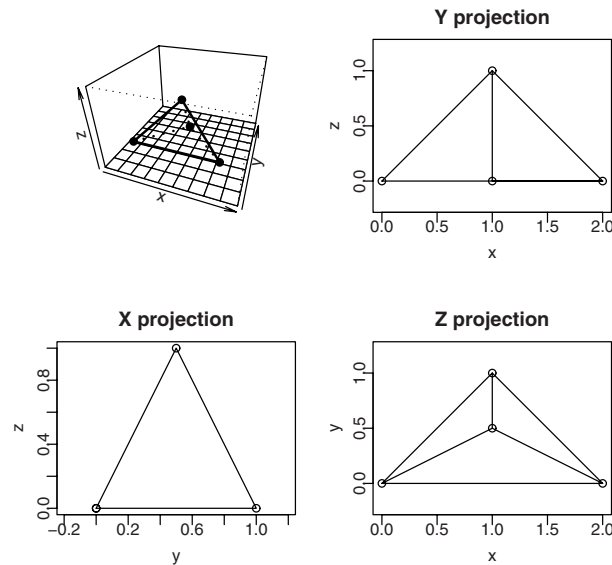

Fig. 2.3. Partitioning the graphical device using the function `layout`

The `scatterplot3d` function of the **scatterplot3d** package produces 3D plots with the coordinates of points. Arguments and scripts are somehow simpler than the `persp` function if the aim of users is to produce a cloud of points (there is no need for specifying the volume of the space on which points will be plotted; coordinates of points can be passed through a vector or a three-column matrix).

The `locator` and `identify` functions directly interact with the graphs. Other possibilities are offered by the **dynamicGraph** and **rgl** packages which build interactive graphs and where users can modify some parameters using the mouse.

The **rgl** package is obviously a useful tool for visualizing and manipulating 3D configurations or surfaces (Fig. 2.4). Let's see some script for visualizing the configuration of the previously defined tetrahedron. In addition, animation of the scene is easily produced.

Load the package **rgl***, and clear the scene of the* `rgl` *graphical device.*

```
>library(rgl)
>rgl.clear()
```

```
>coll<-palette(gray(seq(0.4,.95,length=4)))
```

Set the color of the background.

```
>bg3d("white")
```

Draw triangle surfaces using their coordinates.

```
>for(i in 1:4)
+    {rgl.triangles(tetra[-i,1],tetra[-i,2],tetra[-i,3],
+                  colors=coll[i])}
```

Add spheres at the location of landmarks.

```
>rgl.spheres(tetra[,1],tetra[,2],tetra[,3]
+            ,radius=0.1,col=1)
```

Create an animation.

```
>for (i in 1:360) {rgl.viewpoint(i,45)}
```
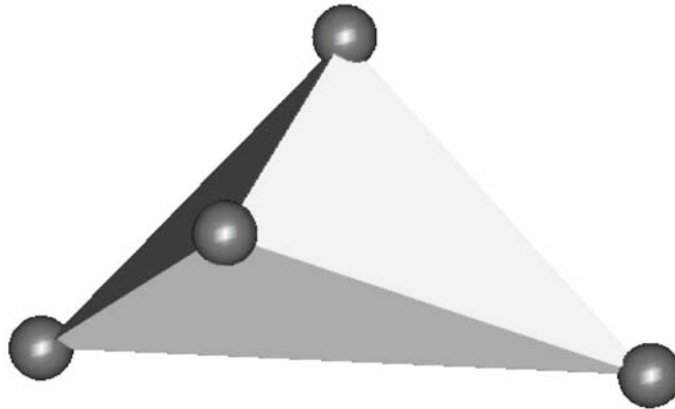

Fig. 2.4. The rgl graphical device

Of course, more complex configurations can be displayed. Fig. 2.5 corresponds to the location of landmarks recorded on the bony shell of the turtle *Dermatemys*. To obtain the graph[2], I adapted the same commands, and I underlined contacts between bones with the rgl.lines function.

In addition to reading image files, the pixmap, rtiff, and rimage packages offer a way to display the image classes as graphs using the generic plot function. The image function of the base package, with care about pixel size, can provide the same results. Indeed, image creates a grid of colored rectangles with colors corresponding to the values of a matrix. Note that x and y-axes have to be inverted and that the plot must be further reflected to display a very similar result (Fig. 2.6). Let's see some applications with the picture of sea shell mytilus.jpg[3] that we have

---

[2] The matrix of landmark coordinates and the code are available in the online supplement.
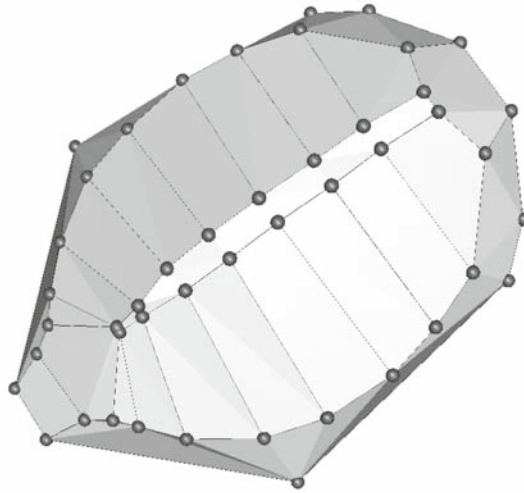
[3] Available in the online supplement

Fig. 2.5. Three dimensional display of a configuration digitized on a turtle shell with the `rgl` graphical device

formerly converted into a *.ppm format. This file is sufficiently small (less that 50 Ko) for being read and rapidly interpreted by R.

```
>setwd("/home/juju/morph")
>shell("convert mytilus.jpg mytilus.ppm")
>library(pixmap)
>M<- read.pnm("mytilus.ppm")
>plot(M)
```

Convert the RGB image to a gray-scale image.

```
>M<- as(M, "pixmapGrey")
>plot(M)
>layout(matrix(1:4, 2,2))
```

Similar operations using the image *function. Note the way to scale the color in the second argument, and the* asp *argument set to 2/3, or 3/4, or 9/16 for avoiding distortion due to pixel width on length ratio (see below). It is possible to play with the depth of the color scale for displaying various images.*

```
>image(t(M@grey[dim(M@grey)[1]:1,]),col=gray(0:255/255),
+   asp=9/16,axes=F,main="Gray-scale: 8-bits")
```

Plot the same image with a 2-bit gray-scale depth, and with a binary depth.

```
>image(t(M@grey[dim(M@grey)[1]:1,]),col=gray(0:3/3),
+   asp=9/16,axes=F,main="Gray-scale: 2 bits")
>image(t(M@grey[dim(M@grey)[1]:1,]),col=gray(0:1/1),
+   asp=9/16,axes=F,main="Monochrome: 1 bit")
```

Exploring possibilities offered by the contour *function .*

```
>contour(t(M@grey[dim(M@grey)[1]:1,]),asp=9/16,axes=F,
+   levels=0:10/10, main="Contour plot",drawlabels=F)
```

The `image` and `contour` functions displays interesting graphs when their argument is a gray-scale image matrix, or any tone channel matrix.
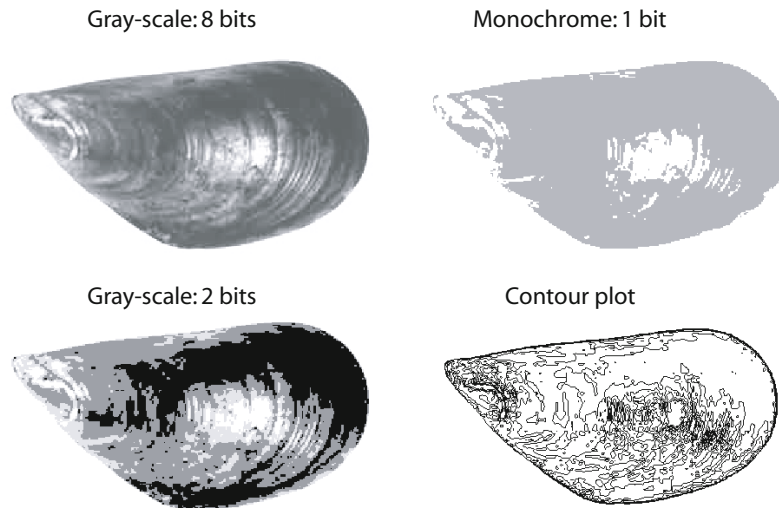
Gray-scale: 8 bits   Monochrome: 1 bit

Gray-scale: 2 bits   Contour plot

Fig. 2.6. Displaying images with various gray-scale depths, or by using contour plot

2.2.6 Image Analysis and Morphometric Data Acquisition with R

The `locator` function reads the position of the graphic cursor when the left mouse button is pressed. It can digitize coordinates of points on any displayed image file by the graphical interface. Specifying arguments in the `locator` function offers further possibilities: printing digitized points, linking them with a line, etc. (see Fig. 2.7). In addition, one can combine `locator` with low-level graphical functions such as `points` or `lines`, or `polygon` to directly interact and draw on the graphical device.

```
>par(mar=c(1,1,1,1))
```

The `mar` *graphical parameter sets the margin of the plot; one can set many other graphical parameters with the* `par` *function. You can find the image file in the online supplement.*

```
>library(rimage)
>x<-read.jpeg("/home/juju/morph/wing.jpg")
>plot(x)
>ji<-locator(5,type="p",pch=3)
```

`locator` *returns an object of the* `list` *class with a vector of* x *and a vector of* y*-coordinates.*

```
>ji
$x
[1] 327.0152 443.4394 662.5909 618.0758 814.9697

$y
[1] 269.3939 207.7576 240.2879 322.4697 320.7576
>text(ji, pos=2, labels=1:5)
>ju<-locator(5,type="l")
>polygon(ju, density=12)
```
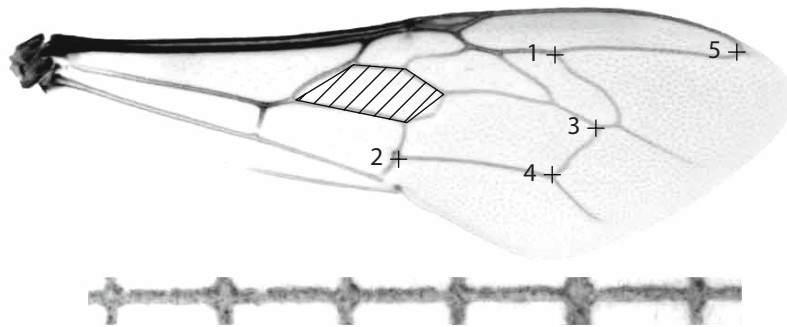

Fig. 2.7. Digitizing landmark locations with R

Calibrating the actual size is possible if one knows any interlandmark distance on the image . I invite users to photograph the object together with a scale (ruler, micrometer . . .). One can then obtain actual interlandmark distances using the cartesian coordinates of the scale (see the script and Fig. 2.8).

We first open an image,[4] and we invert it for an easier digitization process. We produce the inverted image by taking the absolute value of the difference of pixel values minus one.

```
>library(rimage)
>x<-read.jpeg("/home/juju/morph/jawd.jpg")
>x<-rgb2grey(x)
>x<-1-abs(x)
>plot(x)
```

Use `locator` *for localizing landmarks separated by 1 cm on the ruler.*

```
>a<-locator(2,type="o",pch=8,lwd=2,col="grey60",lty="11")
```

Determine the size of a known distance (1 cm) on the graph.

```
>scale1<- sqrt(sum(diff(a$x)^2+diff(a$y)^2))
```

Return the vector of scaled coordinates.

---

[4] The image file of the example is available in the online supplement

```
>b<-unlist(locator(10,type="p",pch=21,bg="white"))/scale1
```

Return the scaled configuration matrix.

```
>matrix(b, 5, 2)
```

An alternative manipulation.

```
>d<- locator (10, type="p"))
>d<- rbind(d$x, d$y)/scale1
```
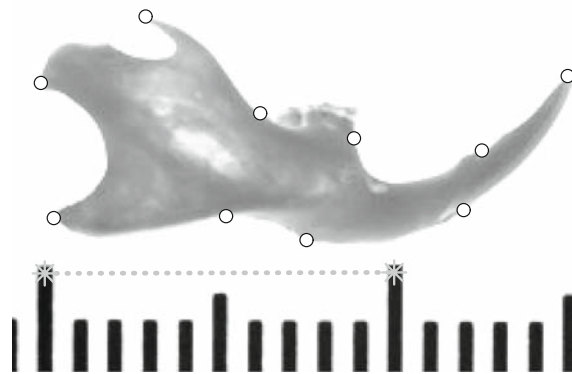

Fig. 2.8. Digitizing landmarks, and measuring the virtual size of the scale with the `locator` function. The digitized scale is indicated by gray stars and dotted segment, while circles indicate digitized landmarks

Depending on the camera or on the screen display, users must be careful with pixel size and non-square pixels that may produce a 3:2, 4:3, or 16:9 aspect ratio (see [68]). Correcting for pixel distortion requires multiplying or dividing matrix indices by corresponding ratios, or resizing one of the two dimensions.

Most of the time, one wants to simplify the image to find pixels of interest more easily. These simplifications involve defining a threshold pixel values and binarizing the images (Fig. 2.9). The following example illustrates these simple manipulations of pixel values. Using thresholds can help in calculating the surface of a specified area, especially when this area is well contrasted from the surrounding background. Logical indexing and the threshold value can be used to specify a desired surface. We can check whether the threshold is reliable for our task by assigning a new value or new color to the pixels of interest, and by plotting the binarized image. I tried three different thresholds on the *Mytilus* shell image.

```
>setwd("/home/juju/morph")
>x<- read.pnm("mytilus.ppm")
>y<- as(x, "pixmapGrey")
>rm(x)
>par(mar=c(1,3,2,1))
```

```
>Y<-y@grey
>layout(matrix(1:4, 2,2))
>plot(y, main="Gray-scale image")
>y@grey[which(Y>=0.1)]<-1
>y@grey[which(Y<0.1)]<-0
>plot(y, main="Bin image, threshold=0.1")
>y@grey[which(Y>=0.3)]<-1
>y@grey[which(Y<0.3)]<-0
>plot(y, main="Bin image, threshold=0.3")
>y@grey[which(Y>=0.9)]<-1
>y@grey[which(Y<0.9)]<-0
>plot(y, main="Bin image, threshold=0.9")
```

Estimate the number of pixels included in the surface of interest (which have values below the threshold in this case).
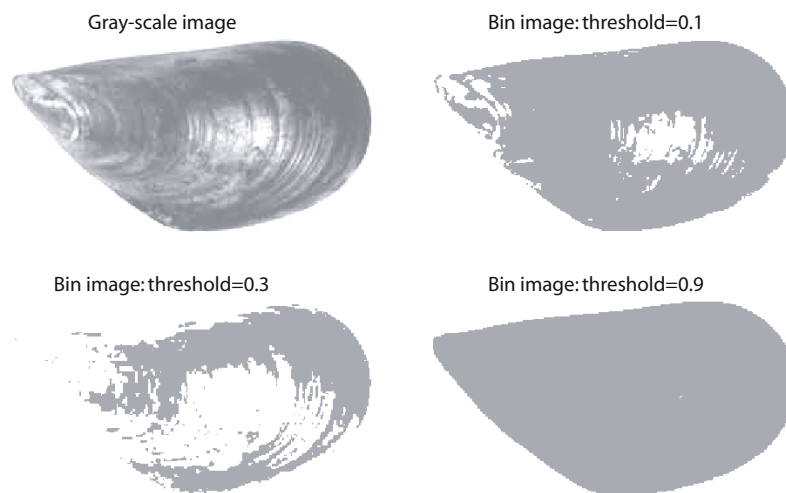
```
>length(y@grey[which(Y<0.9)])
[1] 29102
```


Fig. 2.9. Image binarization using a homemade threshold filter

Applying a threshold to pixel values of an image confers some other advantages for manipulating image files with R. Color image files, indeed, usually require an important memory size; therefore binarizing offers possibilities for compressing the size of the image object. One can extract the coordinates or the indices of the pixel of interest in a two-column matrix to release useless information contained in the background. The `plot` function can easily restitute the part of the image that is interesting for us. These manipulations are especially useful for processing with outline or surface extraction. The image file used for the example is available online.

```
>x <- read.jpeg("/home/juju/morph/jaw3.jpg")
>plot(x)
>dim(x)
[1] 378 891
```

Calculate the matrix indices of pixels of interest, with the integer division and the modulo operators.

```
>xx<-which(x<=0.5)%%378
>yy<-which(x<=0.5)%/%378
>plot(yy, -xx, type="p",pch=24, cex=0.1)
```

Following the same methodology, it is possible to export the slice number (if there is a series of images at different depths for a 3D object) together with the `xx` and `yy` in an object of the `array` or `list` classes for defining 3D surfaces or 3D volumes.

Inverting image can reveal features that could have been less visible on the normal image. For this, we have simply to invert the value of pixels for each channel: the biggest becomes the smallest and vice versa. Since the minimal and maximal pixel values are between zero and one, respectively, the inversion is straightforward. We may wish to accentuate the value of a single color channel, or to remove a channel as in the following example.

```
>x<-read.pnm(system.file("pictures/logo.ppm",
+    package="pixmap")[1])
```

Invert the red channel.

```
>x@red<-abs(1-x@red)
```

Invert the green channel.

```
>x@green<-abs(1-x@green)
```

Invert the blue channel.

```
>x@blue<-abs(1-x@blue)
>plot(x)
```

Accentuate the red channel.

```
>x@red<-0.5+x@red/2
```

Remove the green channel.

```
>x@green<-0
```

During image acquisition, it is possible to accentuate the contrast between the object and the background. This makes some manipulation easier, such as contour extraction. If the range of pixel values is less than 1, enlarging the full range to 1 will make details of the object more contrasted to identify specific features (although it does not increase the number of possible pixel values).

```
>x<- read.pnm(system.file("pictures/logo.ppm",
+   package="pixmap")[1])
>x<- as(x, "pixmapGrey")
>x@grey<- x@grey /diff(range (x@grey))
>x@grey <-x@grey - min (xgrey)
>plot(x)
```

We easily modify the brightness of the picture using the power operator.

```
>x<- read.pnm(system.file("pictures/logo.ppm",
+    package="pixmap")[1])
>x<- as(x, "pixmapGrey")
>y<-x
>y@grey<- y@grey^2
>plot(y)
```

Other possibilities are offered for modifying a picture. Displaying the histogram of pixel values (with the `hist` function) can help to understand how to transform pixel values appropriately for a given channel. Histograms can reveal undesirable values of pixels (for example, marginal values) that can be easily eliminated using a few functions, similarly to the way that we have processed for thresholding pixel values). One can also change the distribution of pixels so it could become uniform, or one can modify the symmetry of the distribution using one of the link functions to rescale pixel values between 0 and 1.

Some morphometric techniques are dedicated to the analysis of outline. I present here a small function called `Conte`. It extracts the coordinates of pixels defining an outline from a picture file. The function is the transcription of an outline extraction algorithm. The function starts with a point and looks for the nearest neighbor pixels, rotating and extracting coordinates on the outline in a clockwise way. Notice that we here use a threshold of 0.3 for finding the first point of the outline, and 0.1 for finding the nearest neighbor. One can modify these subjective values to adapt for the outline one wants to extract. You can store the function on an ASCII file somewhere in your computer, and paste it on the computer when you need it.

Function 2.1. `Conte`

Arguments:
>   `x`: *Vector of the x and y-coordinates for the starting point. This starting point must be chosen on the left part and inside the object.*
>   `imagematrix`: `imagematrix` *object (the picture for which one wants to extract the outline).*

Values:
>   `X`: *x-coordinates of the outline.*
>   `Y`: *y-coordinates of the outline.*

```
1  Conte<-function(x, imagematrix)
2  {I<-imagematrix
3  x<-rev(x)
4  x[1]<-dim(I)[1]-x[1]
```

The first step consists of moving a cursor from the selected pixel to the left until you find two pixels that significantly differ in their values for setting the "true starting point" of the outline. The function is not finding the closest pixel to the selected starting points but the pixel of the outline located on the left side of the selected location. This pixel will be the first point of the outline.

```
5  while (abs(I[x[1],x[2]]-I[x[1],(x[2]-1)])<0.1){x[2]<-x[2]-1}
6  a<-1
```

The M *matrix contains the indices (coordinates) of pixels that are located around the current pixel; the current pixel values are momently set to (0,0).*

```
7  M<-matrix(c(0,-1,-1,-1,0,1,1,1,1,1,0,-1,-1,-1,0,1),
8           2,8,byrow=T)
9  M<-cbind(M[,8],M,M[,1])
```

The X, Y, x1, x2, SS, *and* S *values are initialized before the contour extraction to start.*

```
10 X<-0; Y<-0;
11 x1<-x[1]; x2<-x[2]
12 SS<-NA; S<-6
```

The index of the pixel corresponds to a. *It is incremented by 1 every time the next pixel of the contour is found in a clockwise way.* X *and* Y *record the successive coordinates; the algorithm evaluates the following pixel value turning a block of three pixels clockwise around the current pixel, and progresses pixel by pixel until the location of the next pixel belonging to the outline is found.*

```
13 while ((any(c(X[a],Y[a])!=c(x1,x2) ) | length(X)<3))
14   {if (abs(I[x[1]+M[1,S+1],x[2]+M[2,S+1]]-I[x[1],x[2]])<0.1)
15    {a<-a+1;X[a]<-x[1];Y[a]<-x[2];x<-x+M[,S+1]
16     SS[a]<-S+1; S<-(S+7)%%8}
17 else if (abs(I[x[1]+M[1,S+2],x[2]+M[2,S+2]]
18            -I[x[1],x[2]])<0.1)
19    {a<-a+1;X[a]<-x[1];Y[a]<-x[2];x<-x+M[,S+2]
20     SS[a]<-S+2; S<-(S+7)%%8}
21 else if (abs(I[x[1]+M[1,(S+3)],x[2]+M[2,(S+3)]]
22            -I[x[1],x[2]])<0.1)
23    {a<-a+1;X[a]<-x[1];Y[a]<-x[2];x<-x+M[,(S+3)]
24     SS[a]<-S+3; S<-(S+7)%%8}
25 else S<-(S+1)%%8}
```

Return the resulting objects of the function under the form of a list *containing the* X *and* Y *vectors of* x *and* y*-coordinates for pixels of the outline.*

```
26 list(X=(Y[-1]), Y=((dim(I)[1]-X))[-1])}
```

We can use this new function to extract the outline coordinates of the shell on the *Mytilus* image.

Binarize the image.

```
>y<- read.pnm("mytilus.ppm")
>y<- as(y, "pixmapGrey")
>y@grey[which(y@grey>=0.9)]<-1
>y@grey[which(y@grey<0.9)]<-0.7
>par(mar=c(1,1,1,1))
>plot(y)
```

Use locator *for defining the starting point.*

```
>start<-locator(1)
>Rc<-Conte(c(round(start$x),round(start$y)),y@grey)
>lines(Rc$X, Rc$Y, lwd=4)
```

Draw an arrow at the starting point.

```
>arrows(0,Rc$Y[1],Rc$X[1],Rc$Y[1],length=0.1)
```
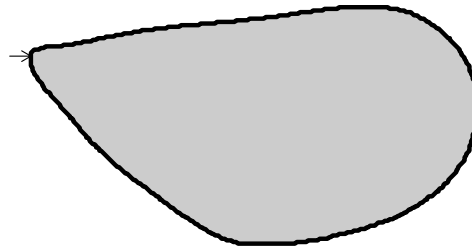

Fig. 2.10. Automated outline extraction. The arrow indicates the starting point

Defining a function for digitizing open curves is rather easy and uses most parts of this algorithm. There is simply a need for inputting the coordinates of the ending point, or rather their approximate position. Notice that you must first have prior knowledge about how coordinates of pixels are handled and how they are plotted on the graphic device; indeed, typing plot(y) here displays something quite different than the image(y@grey) command.

The Conte function can be adapted for a series of scanner images for extracting a 3D surfaces. It presents the advantage that the indices of pixel coordinates correspond to the elementary curvilinear abscissa.

2.3 Manipulating and Creating Data with R

After the acquisition of raw data, it is often necessary to reorganize or slightly transform them to start operationally analyzing the morphometric data. Here are some of

the very usual data manipulations: scaling an image, obtaining angles or distances from coordinates of points. We will also program some more specific tools such as one function for obtaining regularly spaced landmarks on a given outline.

2.3.1 Obtaining Distance from Coordinates of Points

The distance d_{EF} between two landmarks E and F in two or three dimensions is the square root of the sum of the squared differences between each coordinate. It is given by the relationship

$$d_{EF} = \sqrt{\sum_{i=1}^{k} (E_i - F_i)^2} \ .$$

The transcription in R language is simple:

Acquisition of the coordinates of the landmarks E *and* F.

```
>E<-c(1, 4)
>F<-c(6, 8)
```

Computation of the interlandmark distance between E *and* F.

```
>sqrt(sum((E-F)^2))
[1] 6.403124
```

We implement this relationship in the `ild` function. This function computes the distance between any pair of landmarks:

Function 2.2. `ild`

Arguments:
    E: *x and y-coordinates of the first point as a* `vector` *object.*
    F: *x and y-coordinates of the first point as a* `vector` *object.*
Value:
    Distance between the two points.

```
1   ild<-function(E, F){ sqrt(sum((E-F)^2))}
```

```
>ild(E, F)
[1] 6.403124
```

When the function is loaded on the computer, it correctly computes the distances between two landmarks, provided that landmarks have the same number of dimensions and that their coordinates are written in the same order.

2.3.2 Calculating an Angle from Two Interlandmark Vectors

The angle θ between two vectors $\overrightarrow{AB}$ and $\overrightarrow{CD}$ is defined as the difference of their arguments using $\mathbb{C}$ vectors.

```
>CD<-c(2, 4)
>AB<-B-A
>ABc<-complex(real=AB[1], imaginary=AB[2])
>CDc<-complex(real=CD[1], imaginary=CD[2])
>ABc; CDc
[1] 5+4i
[1] 2+4i
```

The `Arg` *function returns the argument of a complex number. The angle between vectors corresponds to the difference of their arguments. The function uses properties of complex numbers for calculating the angle between the* `CD` *and* `AB` *vectors.*

```
>Arg(ABc)
[1] 0.674741
>Arg(CDc)
[1] 1.107149
>Arg(ABc)-Arg(CDc)
[1] -0.4324078
```

Calculate the result in degrees.

```
>Arg(ABc)-Arg(CDc) / pi * 180
[1] -62.76021
```

The `angle2d` function calculates the angle between two 2D `v1` and `v2` vectors.

Function 2.3. `angle2d`

Arguments:
  `v1`: *2D vector of* `numeric` *mode.*
  `v2`: *2D vector of* `numeric` *mode.*
Value:
  Angle between the two vectors in radians.

```
1  angle2d <- function(v1,v2)
2  {v1<-complex(1,v1[1],v1[2])
3  v2<-complex(1,v2[1],v2[2])
4  (pi+Arg(v1)-Arg(v2))%%(2*pi)-pi}
```

To calculate the angle θ between two vectors of higher dimensions, one must use the relationship between their norm and their dot product such that

$$|\theta| = \frac{\overrightarrow{CD} \cdot \overrightarrow{AB}}{\|\overrightarrow{AB}\|\|\overrightarrow{CD}\|} \; .$$

The `angle` function uses the norm and dot product relationship to calculate the angle between two vectors. However, the orientation of this angle will not be signed.

Function 2.4. `angle`

 Arguments:
   `v1`*: Vector of* `numeric` *mode.*
   `v2`*: Vector of* `numeric` *mode and of length equal to the length of* `v1`*.*
 Value:
   Angle between the two vectors in radians.

```
1  angle<-function(v1, v2)
2   {temp <- sum(v1*v2)/( sqrt(sum(v1^2))*sqrt(sum(v2^2)) )
3    acos (temp)}
```

For 3D vectors, one must check the sign of the determinant of a 3×3 matrix with the first row being a triple unit vector (**1**), and the next two rows corresponding to the vector coordinates. This operation corresponds to a triple scalar product such that

$$det \begin{vmatrix} \mathbf{1} \\ \overrightarrow{AB} \\ \overrightarrow{CD} \end{vmatrix} = \mathbf{1} \cdot (\overrightarrow{AB} \times \overrightarrow{CD}) \,,$$

where "$\times$" denotes the vector cross-product. The `angle3` function calculates the signed angle between two 3D vectors, it depends on the `angle` function:

Function 2.5. `angle3`

 Arguments:
   `v1`*: 3D vector of* `numeric` *mode.*
   `v2`*: 3D vector of* `numeric` *mode.*
 Value:
   Signed angle between the two vectors in radians.

```
1  angle3<-function(v1, v2)
2   {a<-angle(v1, v2)
3    b<-sign( det(rbind(1, v1, v2)) )
4    if (a == 0 & b == 1){jo<-pi/2}
5      else if (a == 0 & b == -1){jo<- - pi/2}
6      else {jo<- a * b}
7   (pi+jo)%%(2*pi)-pi}
```

2.3.3 Regularly Spaced Pseudolandmarks

In morphometrics, in particular with Fourier analysis of outlines (see Chapter 4), prior operations are usually performed on the collection of coordinates of pixels

defining the outline. One of these operations is to obtain equally spaced pseudoland-marks on the digitized outline (see Fig. 2.12). Our `Conte` function extracts coordi-nates of points on an outline with a one pixel in length curvilinear abscissa. If there are enough pixels, one can extract a given number of equally spaced pixels using the regular sequence-generating function of R.

Obtaining 32 equally spaced pseudolandmarks on the outline of the Mytilus shell (`Rc$X` and `Rc$Y` are coordinates of pixels of the outline).

```
>layout(matrix(c(1,2), 1,2))
>Rc32x<-(Rc$X[seq(1,length(Rc$X),length=33)])[-1]
>Rc32y<-(Rc$Y[seq(1,length(Rc$Y),length=33)])[-1]
>plot(Rc$X, Rc$Y, type="l", lwd=1.5, asp=1, axes=F
+     , main = "curvilinear")
>points(Rc32x, Rc32y)
```

One can also digitize equally spaced pseudolandmarks on any kind of curve that one has approximated by digitizing several points by hand. For acquiring points on the curve, one uses the `locator` function . Indeed, `locator` allows one to collect coordinates that can define lines or segments. Then, one can obtain regularly spaced landmarks on lines or surfaces with the `spsample` function of the **sp** package. We will apply this exercise to the curve depicting the lower part of the rodent jaw.[5] We will sample pseudolandmarks between two well-known landmarks (incisor-bone contact and extremity of the angular apophysis) (see Fig. 2.11).

```
>library(rimage)
>layout(matrix(c(1,2), 1,2))
>par(mar=c(0,1,0,0))
>x<-read.jpeg("/home/juju/morph/jaw2.jpg")
>plot(x)
>dig<-locator(type="o",col="white",lwd=1.5)
```

Draw successive segments with the mouse for digitizing the curve of interest.

```
>DIG<-matrix(unlist(dig),length(dig$x),2)
>library(sp)
>Ldig<-Line(DIG)
```

Transform the object of the `Line` class for sampling pseudolandmarks on successive segments. The matrix of sampled coordinates are inside the `@coords` slot returned by `spsample`.

```
>pseudo<-spsample(Ldig,16,type="regular",
+                 offset=c(0,1))@coords
```

Do not forget to remove the last landmark digitized in the sample (because of the `offset` argument), and plot landmarks and pseudolandmarks on a new graph.

```
>plot(x)
>points(DIG[c(1,dim(DIG)[1]),],cex=1.5,pch=20,
+              frame=F, axes=F,asp=1)
>points(pseudo[-nrow(pseudo),],pch=21,bg="white")
```
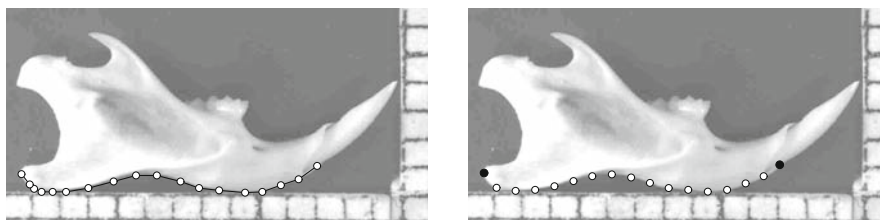
Fig. 2.11. Obtaining equally spaced pseudolandmarks using curve digitizing; pseudoland-marks are white circles, and ending landmarks are black rounds on the right side graph

One can eventually adjust the sampling process by smoothing the original curve (see later in this section).

Rather than selecting equally spaced points according to the curvilinear abscissa, one may prefer to select landmarks that are spaced with a regular sequence of angles taken between the outline coordinates and the centroid. Let say that the origin O is the first digitized point. We must therefore transform every cartesian coordinate into polar coordinates using complex numbers and operations for our task. The `regularradius` function returns n points on equally spaced radii.

Function 2.6. `regularradius`

Arguments:
>   Rx*: Vector containing the x-coordinates of the outline.*
>   Ry*: Vector containing the y-coordinates of the outline.*
>   n*: Number of points to be sampled.*
Values:
>   `pixindices`*: Vector of radius indices.*
>   `radii`*: Vector of sampled radii lengths.*
>   `coord`*: Coordinates of sampled points arranged in a two-column matrix.*

```
1   regularradius<-function(Rx, Ry, n)
2     {le<-length(Rx)
3     M<-matrix(c(Rx, Ry), le, 2)
4     M1<-matrix(c(Rx-mean(Rx), Ry-mean(Ry)), le, 2)
5     V1<-complex(real=M1[,1], imaginary=M1[,2])
6     M2<-matrix(c(Arg(V1), Mod(V1)), le, 2)
7     V2<-NA
```

The following code finds the indices of the nearest pixel on the outline using the angular increment.

```
8     for (i in 0:(n-1))
9         {V2[i+1]<-which.max((cos(M2[,1]-2*i*pi/n)))}
10    V2<-sort(V2)
11    list("pixindices"=V2,"radii"=M2[V2,2],"coord"=M1[V2,])}
```

---

[5] The image file is available in the online supplement

To visualize the outline and the equally spaced radii, we first need to calculate the coordinates of the centroid of the outline. Here we work on the *mytilus* shell image.

The centroid coordinates Xc *and* Yc *of the outline are defined as the mean of x and y-coordinates sampled on the outline and are computed straightforwardly:*

```
>Xc <- mean(Rc$X)
>Yc <- mean(Rc$Y)
>plot(Rc$X,Rc$Y,type="l",lwd=1.5,asp=1,axes=F,main="polar")
>points(Xc, Yc, pch=4)
```

Using a loop, we draw the successive segments linking the centroid to the points sampled on the outline.

```
>ju<-regularradius(Rc$X, Rc$Y, 32)
>points(ju$coord[,1]+Xc, ju$coord[,2]+Yc)
>for (i in 1:32){
+  {segments(0+Xc,0+Yc,ju$coord[,1]+Xc,ju$coord[,2]+Yc)}
```
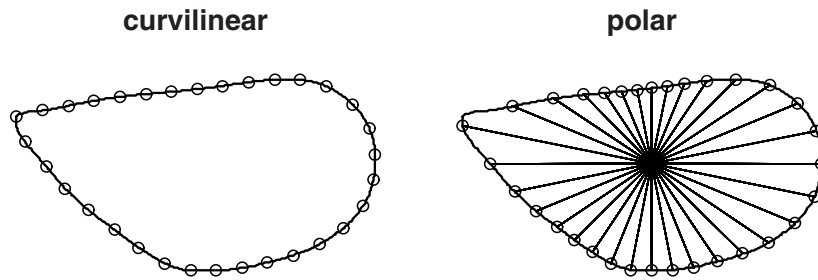
curvilinear   **polar**

Fig. 2.12. Diverse types of pseudolandmarks automated digitizations with the outline of the *Mytilus* shell. On the left, pseudolandmarks are equally spaced following an equal curvilinear abscissa, while on the right, pseudolandmarks are spaced according to equally spaced angles between segments departing from the centroid to the outline

2.3.4 Outline Smoothing

Depending on the resolution and the sensitivity of an automated outline extraction, it is often necessary to smooth the outline for further analyses. When digitized outlines with high resolution produce undesirable irregularities, Haines and Crampton [43] recommend smoothing the outline based on the following formula:

$$(x, y)_i^{new} = \frac{1}{4}(x, y)_{i-1}^{old} + \frac{1}{2}(x, y)_i^{old} + \frac{1}{4}(x, y)_{i+1}^{old} \ .$$

A function that performs this operation requires two arguments; the first is the outline coordinates to be smoothed, and the second is the number of iterations. We want to

apply n times this smoothing function to the raw configuration matrix. We program this under the smoothout function:

Function 2.7. smouthout

Arguments:
   M: *x and y-coordinates of the outline arranged in a two-column matrix.*
   n: *Number of iterations.*
Value:
   Matrix of smoothed coordinates.

```
1  smoothout<-function(M, n)
2    {p<-dim(M)[1]
3    a<-0
4    while (a<=n)
5      {a<-a+1
6      Ms<-rbind(M[p,],M[-p,])
7      Mi<-rbind(M[-1,],M[1,])
8      M<-M/2+Ms/4+Mi/4}
9    M}
```

Low resolution of images can be a source of error during automatic image digitization. Artificially inflating the number of landmarks can provide some approximation of the reality. It can be achieved by interpolating supplementary landmarks that correspond to the mean coordinates of two adjacent landmarks and writing the landmark.addition function. One can later smooth this outline using the smoothout function, if needed.

Function 2.8. landmarkaddition

Arguments:
   M: *x and y-coordinates of the outline arranged in a two-column matrix.*
   n: *Number of iterations.*
Value:
   Matrix of original and interpolated coordinates.

```
1  landmark.addition<-function(M, n)
2    {a<-0
3    while(a<=n)
4      {p<-dim(M)[1]
5      k<-dim(M)[2]
6      N<-matrix (NA,2*p,k)
7      N[((1:p)*2)-1,]<-M
8      N[(1:p)*2,]<-(M+(rbind(M[-1,],M[1,])))/2
9      M<-N}
10    M}
```

2.4 Saving and Converting Data

The easiest way to save objects with R is probably the `save` function which writes a binary R file in a specified folder. The extension can be of any type. For reloading the saved file, the `load` function loads the object on the working environment or follow a specified path entered as argument. The original name of the object is loaded with its value.

The `write.table` function is convenient for storing data frames. Finally, the `cat` function can convert and concatenate objects to character strings, separating them by a specified separator. In specifying `"\n"`, one can write data on different lines.

R can read many formats, most of them being primarily ascii files. Here my goal is to show how one opens, converts, and interprets them through R, and how to perform the reverse operation (for example, digitizing landmarks on R, and then exporting them in the appropriate format).

The *.NTS format is one of the more commonly used in geometric morphometrics. Software performing morphometric analyses (Morpheus,[6] the TPS family[7] etc.) can save or convert data in this format. The data are stored as a matrix with the rows corresponding to the configurations and the columns to the coordinates of each landmark. The first line of the file contains four or five arguments. The first and fourth are fixed, while the second and third respectively correspond to the number of specimens and the number of landmarks multiplied by the number of dimensions ($k \times p$). The fifth and optional argument specifies the number of dimensions, and follows the string "`DIM=`". By correctly filling the arguments of the `scan` function, we open this type of file quite easily. If the string or the character for comments is the double quoting mark along the file, the conversion requires not more than two lines as illustrated below.

The RATS.NTS dataset can be found in the data sets of the freely available software tpsRegr.[8]

```
>jo<-scan("/home/juju/morph/RATS.NTS",what="char",
+       quote="",sep="\n", comment.char="\"")
```

Obtain the number of dimensions `k`.

```
>jo1<-jo[1]
>l1<-unlist(strsplit(jo1, "="))
>l2<-unlist(strsplit(jo1, " "))
> if(length(l1)==1) {k<-2}
+  else {k<-as.numeric(l1[2])}
>k
[1] 2
```

Extract coordinates.

---

[6] `http://life.bio.sunysb.edu/morph/morpheus/`
[7] `http://life.bio.sunysb.edu/morph/`
[8] `http://life.bio.sunysb.edu/morph/`

```
>jo2<-jo[-1]
>cat(jo2, file="jo2.txt")
>data<-matrix(scan("jo2.txt"), as.numeric(l2[2]),
+    as.numeric(l2[3]), byrow=T)
>unlink ("jo2.txt")
```

Delete the temporary file `jo2.txt`.

Notice the `sep="\n"` argument specified for the `scan` function; it indicates that the separator corresponds to a new line.

A more complex format is the format *.tps that has been developed for a series of programs by James Rohlf.[9] Here is the code for importing configurations, and later for importing names and coordinates that define curves. In the code below, we finally plot configurations, and curves. Similarly one can extract and assign other attributes to a `list` or an `array`.

The file `sneathd.tps` *can be obtained from the datasets of the freely distributed software tpsRelw.*

```
>jo<-scan("/home/juju/morph/sneathd.tps", what="char",
+         quote="", sep="\n", strip.white=T)
>jo<-casefold(jo, upper=F)
```

Find the indices where each configuration starts.

```
>sp<-grep("lm=", jo)
```

Find the n *number of configurations.*

```
>n<-length(sp)
```

Find the p *number of landmarks for each configuration, knowing that it is indicated after each "=".*

```
> p <-as.numeric(unlist(strsplit(unlist(strsplit
+   (jo[sp[1]], "=")))[2], " "))[1])
```

Find the k *number of dimensions.*

```
>k<-length(unlist(strsplit(jo[sp[1]+1],split=" +")))
```

Prepare an empty matrix that you assign to the new `config` *object for storing coordinates of configurations.*

```
>config<-matrix(NA, n, p*k)
>for (i in 1:n)
+    {config[i,]<-as.numeric(unlist(strsplit(
+    jo[(sp[i]+1):(sp[i]+p)], split=" +")))}
```

Read and store the coordinates of the first curve and of the first object.

```
>curve1<-grep("curves", jo)
```

Find the q *number of outlines in the object.*

---

[9] `http://life.bio.sunysb.edu/morph/`

```
>q <-as.numeric(unlist(strsplit(unlist(strsplit(
+   jo[curve1[1]],"="))[2]," "))[1])
>point1<-grep("points", jo[sp[1]:sp[2]])
```

The nb *object is a vector that will contain the number of landmarks stored for each curve.*

```
>nb<-NA
>for (i in 1:length(point1))
+   {nb[i]<- as.numeric(unlist(strsplit(unlist
+   (strsplit(jo[point1[i]], "="))[2], " "))[1])}
```

Store the coordinates of the curve points in a list *of* q *vectors.*

```
>out1<-list()
>for (i in 1:q)
+   {out1[[i]]<- as.numeric(unlist(strsplit(jo[(
+   point1[i]+1):(point1[i]+nb[i])], split=" +")))}
>l<-length(unlist(out1))
```

Prepare the space required for the x *and* y*-axes in the graph.*

```
>m1<- min(unlist(out1)[(1:(l/2))*2])
[1] 90.21
>m2<- min(unlist(out1)[(1:(l/2))*2-1])
[1] 222.15
>M1<- max(unlist(out1)[(1:(l/2))*2])
[1] 1492.27
>M2<- max(unlist(out1)[(1:(l/2))*2-1])
[1] 1860.8
```

Plot the landmarks of the first configuration.

```
>par(mar=c(4,1,1,0))
>plot(matrix(config[1,],p,k,byrow=T),pch=20,
+   cex=1.5, xlim=c(m2-10,M2+10), ylim=c(m1-10,
+   M1+10), asp=1, xlab="Homo", ylab="",axes=F)
```

Plot the curves of the first configuration.

```
>for (i in 1:q)
+    {points(matrix(out1[[i]], nb[i],k, byrow=T),
+    type="l", lw=2, lty="11")}
```

Remark the casefold *function in the script. It translates a character vector in upper or lower case (useful for standardizing a dataset). The result is presented in Fig. 2.13. The* grep *and* sub *functions are used for the conversion of parts of the ascii file because they return indices and replace strings in a vector of characters respectively.*

Exporting R data in one other format corresponds to the reverse operation and is straightforward. The cat function with the separator argument specified as "\n" allows manipulation of data for exporting an ascii file in an appropriate way. Given two configuration matrices in R, M and N, each containing three 2D landmarks, one converts these objects in *.NTS format as follows:

Homo

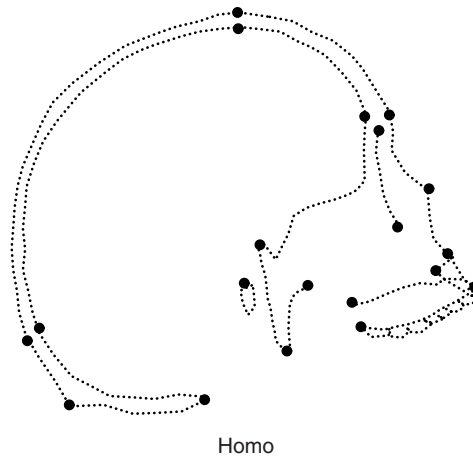**Fig. 2.13.** Opening and printing a *.tps file

```
>M<-matrix(round(rnorm(6)),3, 2)
>M
     [,1] [,2]
[1,]   -2    1
[2,]    0   -1
[3,]    1   -3
>N<-matrix(round(rnorm(6)),3, 2)
>N
     [,1] [,2]
[1,]    2    0
[2,]    0    0
[3,]   -1    0
>config<-c(paste(t(M),collapse=" "),
+                      paste(t(N), collapse=" "))
>commentl<-paste("\"","configurations ","M ","and N")
>firstl<-paste(1, 2, 6, 0, "dim=2")
```

*Save configurations in a *.NTS file.*

```
>cat(firstl,commentl,config,sep="\n",
+         file="/home/juju/morph/juju.NTS")
```

Print the file as it is.

```
>cat(firstl, commentl, config, sep="\n")
1 2 6 0 dim=2
" configurations  M  and N
1 0 -3 0 2 -1
1 2 0 0 1 0
```

Writing a function for exporting data in a desired format involves looping around this theme. Note the backslash for invoking quotation marks or new line in the code.

2.5 Missing Data

Many statistical functions in R contain an argument specifying how to handle missing data. This argument is on the `na.rm`, `na.action`, or `use` forms, and can be specified to change its default value. For example, `na.rm` expects a logical for indicating whether nonavailable data are dropped. Look at the help file of functions such as `cor`, `mean`, and `lm` to understand the different available options. Although one can choose to exclude one measurement or one landmark, one can sometimes estimate missing values from original data.

2.5.1 Estimating Missing Measurements by Multiple Regression

When the studied sample shows some homogeneity in shape variation, measurements are often related. This is often the case with a collection of biological distances. Distances have often a high degree of intercorrelation because they correspond to different body parts that often grow harmoniously during the development. Other causes of the high degree of correlation can be biomechanic, architectural, or genetic. This high correlation can help us infer missing values with regression techniques. It is often better to use all the data to perform these inferences. Multiple regression can be applied as in the following example:

Simulate four distance measurements a, b, c, *and* d *for 30 individuals.*

```
>a<-abs(rnorm(30))
>b<-2*a+rnorm(1, 5)+rnorm(30, 0, 0.5)
>c<-7*a+rnorm(1, 6)+rnorm(30, 0, 4)
>d<-12*a+rnorm(1, 3)+rnorm(30, 0, 3)
>data<-cbind(a, b, c,d)
```

Generate 10 missing values.

```
>data[sample(1:120)[1:10]]<-NA
```

Using the scale *function scales the data with a variance=1 and a mean=0.*

```
>datas<-scale(data)
```

Find the missing values and return the indices of their row with integer division operators and the which *function.*

```
>indic<-which(is.na(data))
>ro<-indic%%30
```

The unique *function removes duplicated values in* ro.

```
>ro<-unique(ro)
```

Change the 0 indices to be 30.

```
>if (any(ro==0)){ro[which(ro==0)]<-30}
>for (i in 1:length(ro))
+     {ind<-which(is.na(data[ro[i],]))
```

The generic `predict` *function is used for appraising fitted values according to a given model (here a multiple linear model). The missing values are estimated with the prediction from the other data. For removing a raw or a column in the matrix, notice the "–" for negative indexing.*

```
>for (j in 1:length(ind))
+     {MOD<-lm(data[, ind[j]] ~ datas[,-ind])
+     data[ro[i],ind[j]]<-predict(MOD,data.frame(datas
+     [,-ind])[ro[i],])[ro[i]]} }
```

2.5.2 Estimating Missing Landmarks on Symmetrical Structures

Several objects exhibit bilateral symmetry. It is often useful to estimate the location of landmarks missing on one side prior to morphometric or statistical analyses. This is not too difficult because symmetric landmarks have a mirror copy. The location of these landmarks is obtained by an appropriate geometric reflection.

The first step for obtaining a mirror image requires estimating the axis or plane of symmetry of the object. Let missing values correspond to NA, and let the *M* configuration be a matrix of p rows and k columns. We will define the indices of landmarks theoretically lying on the symmetry plane or axis in the imp vector, those of left landmarks in the pa1 vector and those of the right landmarks in the pa2 vector. The axis (or plane) that passes through midline landmarks and at the midline of the segments defined by paired landmarks is the axis (or plane) of symmetry. We define the *N* matrix as containing the coordinates of the landmarks expected to be on the symmetry axis, plus the coordinates of points defined at the midline between paired and symmetric landmarks. Because all landmarks of the configuration contribute to the definition of the symmetry axis, coordinates of points defined by left and right sides are weighted twice. To estimate *N*, we have to keep the landmarks of the axis and calculate the midline coordinates of remaining paired landmarks. We obtain *N* using indexing and a few computations.

Obtain separate matrices for the midline, left-, and right-side coordinates.

```
>N1<-M[imp,]
>NL<-M[pa1,]
>NR<-M[pa2,]
```

Obtain the indices of the missing landmarks. The `unique` *function is used for removing duplicated elements, and* `sort` *allows a vector to be sorted in ascending order.*

```
>no<-sort(unique(c(which(is.na(M[pa1,1])),
+     which(is.na(M[pa2,1])))))
```

Calculate coordinates of the points at the midline of segments defined by corresponding left and right landmarks.

```
>N2<-(NL[-no,]+NR[-no,])/2
>N<-rbind(N1,N2, N2)
```

In practice, coordinates of landmarks of *N* are rarely aligned along a straight line (or on a plane, for 3D data). Indeed, there is usually some variation in the position of landmarks on the axis of symmetry in comparison with their expected position (biological asymmetries due to development, measurement error). For defining our transformation, we could estimate the plane or axis equation using the coordinates contained in *N*, and use this equation to perform our transformation. However, the shortest way is to rotate our configuration and axis so that its coordinates will fit to the line of coordinates $y = 0$ (for 2D data) or for the plane defined by $z = 0$ (for 3D data). This axis or plane is minimizing distances between *N* (the raw rotated axis coordinates) and the symmetry axis we will use for the reflection. The rotation is easily appraised by decomposing the variance and covariance of the 2D or 3D coordinates of *N*. We write the corresponding code in the function `eigenrotation`:

Function 2.9. `eigenrotation`

Arguments:

      M*: k-column matrix of landmark coordinates (missing landmarks excluded) to be rotated.*

      N*: k-column matrix of coordinates that belongs to the symmetry axis or plane (median plane or midline landmarks, plus midline points estimated from paired symmetrical points).*

Value:

    k-column matrix of rotated coordinates.

```
1  eigenrotation<-function(as.matrix(N), as.matrix(M))
```

The `eigen` *function computes the eigenvectors and corresponding eigenvectors for a rectangular matrix.* `eigen` *returns a* `list` *object with the first element being the normalized eigenvector loadings.*

```
2  sN<-eigen(var(N))$vectors
```

Data are projected on the eigenvectors of their own variance-covariance matrix. This performs a rotation of the original data. These are then translated so that axis or the plane of reflection includes the origin (remark that this may introduce undesired reflections).

```
3  k<-dim(N)[2]
4  p<-dim(M)[1]
5  Nn<-N%*%sN
6  Mn<-M%*%sN
```

Compute the vector that translates the landmark of the configuration so their centroid becomes $(0, 0)$.

```
7  uNn<-apply(Nn, 2, mean)
```

Translate the rotated data.

```
8  Mnf<-Mn-rep(1,p)%*%t(uNn)
9  Mnf}
```

When the configuration is rotated, we have first to check whether the orientation between coordinates of landmarks in our data has been preserved. For 2D data, we can check the angle between the first three landmark coordinates. If the sign of the vector has changed, we have to multiply the y- (or z-) coordinates by -1. The sign of the angle can be checked using the `angle2d` function for 2D data, or the `angle3` function for 3D data (see Section 2.3). Once reflection is checked, we appraise coordinates of missing landmarks multiplying the corresponding landmark y (for 2D) or z (for 3D) coordinates by -1, and we duplicate the x. To obtain the reflected 3D missing landmarks, the sign of the third coordinate of the corresponding paired landmark is multiplied by -1.

Other solutions for estimating missing landmarks can involve functions like thin-plate spline (see Section 4.3).

2.6 Measurement Error

Measurement error is defined as "*the variability of repeated measurements of a particular character taken on the same individual, relative to its variability among individuals . . .*" for quoting Bailey and Byrnes [5]. The source of error is multiple.

2.6.1 Sources of Measurement Error

Measurement Error Due to the Measurement Device (Precision)

Measurement error and precision are close concepts. Most digitizing devices and digtizers like observers produce an error that corresponds to their imprecision (precision is the level of similarity among the same, repeated measurement) and to their inaccuracy (inaccuracy is measured as the difference between the measured value and the true value). The cause of this error depends mostly on the reliability and the sensitivity of the measuring device. If you digitize landmarks and measurements on an image, the error depends on the size (number of pixels) and on the resolution of the image (the resolution corresponds to the smaller details that can be seen on a picture).

Measurement Error Due to the Definition of the Measure

Although the definition of the landmark position can be unequivocal (e.g., position of a foramen, intersection between nervations), there can still remain a small variation around the landmark position. If the position of the landmark is as precise as 0.1 mm, this will generate a variation among measurements of the same object and will contribute to the total measurement error. This may have some incidence when the observer wants to compare objects that differ in size, and if recognizable landmarks have the same imprecision between objects. For instance, we could consider a landmark that corresponds to a foramen, and that this landmark would be digitized among small and large skeletons; let the size of the foramen be similar in size for

large and small objects, consequently the percent of error variation of the smaller object will be inflated. Actually, this depends on the range of size variation among objects you are exploring, and you can expect that error to decrease with the size of objects.

Measurement Error Due to the Quality of the Measured Material

Some part of the error is inherent to the data themselves. For example, fossils can show different levels of preservation that may affect the way we measure the object. The imprecision in digitizing a given landmark will depend on the preservation of objects. In this case, one expects a positive relationship between error and level of preservation.

Measurement Error Due to the Measurer

Give a measuring tape to three different people and ask them to measure your own hip circumference with a precision of one mm and you will probably get three different measurements. People necessarily differ in the way that they measure distances, and this depends on their individual condition: their degree of concentration, eye health, stress, or knowledge and interest in the objects they measure may affect the outcome of multiple measurements differently. Among users, the error terms vary not only in intensity, but also in the geometrical way that the error is produced (for example, some people will produce more error in positioning a landmark more laterally than vertically, which will affect the covariation pattern in the error component of the variation).

Measurement Error Due to the Environment of the Measurer

Another source of error comes from the direct environment of the observer. One can expect that a noisy or peaceful environment differently influences the observer. Change in luminosity and lighting may compete with the quality of the measurement made by the observer. This is not only true with direct measurements (using a caliper or a ruler) but also with indirect measurements obtained from pictures. In the latter case, the source and intensity of light should be similar for each capture of image (limiting errors arising from differences in shadows or contrasts). More generally, reducing error requires keeping the same conditions for measurement acquisition throughout the full session.

Measurement Error Due to the Measurement Protocol

The better the measurement protocol is established, and the lower the error is. Consider a photography of a 3D object on which landmarks are later digitized; an important part of error may appear depending on the position of the object under the camera. It is useful to set a reference plane from the object for photographing all

similar objects according to the same orientation. Errors are inherent to most optical devices because lenses usually slightly deform the shape of objects, especially when focal distances are very short. Generating variation by focusing differently when capturing an image inherently inflates error variation. To limit this source of error, the user should set focal distance once and for all before digitizing.

2.6.2 Protocols for Estimating Measurement Error

Estimating measurement error involves taking into account most of its origins. It is often interesting to explore the different possible sources of error variation in categorizing the variation. For some analyses of variation, it is strongly recommended to estimate measurement error (especially when the investigated signal of variation is expected to be low). Yezerinac et al. [124] suggest an ANOVA design to compute a percent measurement error that allows further comparisons between different studies. Using a percent of measurement error allows results to be independent of the range or the units of measured objects.

Repeating measurements on the same objects is always necessary to correctly estimate measurement error. It is best repeat ALL measurements at least twice (measurement error is influenced by objects themselves; we can think that some will be more difficult to accurately measure than others, and the economy of time passed through the digitization of a subsample may change our way of estimating the measurement error). However, estimating measurement error using a smaller representative sample is a possible alternative.

The percentage of measurement error is defined as the ratio of the within-measurement component of variance on the sum of the within- and among-measurement component. Percent of measurement error $\%ME$ can be obtained as follows:

$$\%ME = \frac{s^2_{\text{within}}}{(s^2_{\text{within}} + s^2_{\text{among}})} \times 100 \ .$$

Components of variance [76] are themselves derived from the mean squares of the one-way ANOVA considering the factor individual as a source of variation. The among and within variation are estimated from the mean sum of squares:

$$s^2_{\text{among}} = \frac{\text{MSS}_{\text{among}} - \text{MSS}_{\text{within}}}{m} \ ,$$

and

$$s^2_{\text{within}} = \text{MSS}_{\text{within}} \ ,$$

where m corresponds to the number of repeated measurements.

Doing different sessions under different conditions, with different observers, or using different measurement devices is a way to inspect origins and contributions of the putative candidates that may inflate error. Unfortunately, we can regret that this boring stage is still not systematically present in scientific contributions performing morphometric analyses. One can check whether there are differences in mean measurements between both sessions with the significance of the session effect. I supply

a short simulated example below invoking R commands in the case of a univariate measurement.

Simulation of a set of 20 real distances following a normal distribution, with two measurement sessions and with an error term normally distributed.

```
>truemeasures<-rnorm(20, 20, 3)
>measure1<-truemeasures + rnorm(20, 0, 1)
>measure2<-truemeasures + rnorm(20, 0, 1)
>sessionfactor<-gl(2, 20)
>individualfactor<-as.factor(rep(1:20, 2))
>totalobservation<-c(measure1, measure2)
>summary(aov(totalobservation~sessionfactor))
              Df Sum Sq Mean Sq F value Pr(>F)
sessionfactor  1   1.77    1.77  0.1188 0.7323
Residuals     38 566.47   14.91
```

The one-way ANOVA reveals that there is not a strong influence of session on the measurement and that variation between sessions is lower than within session in this example.

```
> mod<-summary(aov(totalobservation~individualfactor))
> mod
                 Df Sum Sq Mean Sq F value    Pr(>F)
individualfactor 19 544.87   28.68  24.533 6.438e-10 ***
Residuals        20  23.38    1.17
---
Signif. codes:  0 *** 0.001 ** 0.01 * 0.05 . 0.1   1
```

This second one-way ANOVA shows that interindividual variation is much larger than within individual variation, we can consider that measurement error is low enough to interpret interindividual variation.

```
>s2within<-mswithin<-mod[[1]][2,3]
>mod[[1]][2, 3]
[1] 1.168932
>MSamong<-mod[[1]][1, 3]
>s2among<-(MSamong-MSwithin)/2
>s2within/(s2within+s2among)*100
[1] 7.833057
```

The percent measurement error is 7.8% in this example.

If you reiterate this example a large number of times, the averaged measurement should be close to $1/9$. This corresponds to the square of the division of the interindividual variance by the error variance that we introduce in our simulated data ($1/3$).

Problems

2.1. Extracting cartesian coordinates and initializing the cartesian system
Write a function for extracting cartesian coordinates on an image using the `locator` function. This function must set the cartesian system to a 0 origin with x and y-axes

that are initialized by clicking on the picture. The three first clicks correspond to the position of the origin, the direction of the x-axis, and the orientation of the y-axis.

2.2. Calculating the surface of a digitized polygon
Use the splancs package and the `area.pl` function to calculate the surface of a polygon drawn with the `locator` function.

2.3. Changing the class of R objects containing a collection of configurations
Write a function to change the organization of a collection of configuration from an `array` to a `data.frame` object. Write the inverse function (transforming the `data.frame` object into an `array` object).

2.4. Manually acquiring outline coordinates
Write a function that draws and acquires the coordinates of equally spaced points on an outline that is digitized as the succession of small segments digitized by clicking on the image. In this respect, you have to gather not only their coordinates, but also compute the distance of each segments; read the help file of the `locator` and `seq` functions for achieving this aim.

2.5. A magic tool for selecting an area on a picture
Write a function that finds indices of pixels that have values close to the pixel selected by a left click (with the `locator` function).

2.6. Digitizing open curves
Adapt the `Conte` function (defined in Section 2.2.6) for digitizing open curves. The coordinates of the starting and of ending points are entered as arguments with the image file, or by clicking on the picture.

2.7. Writing the **read.nts** function
Use the code of Section 2.4 to write a `read.nts` function that opens morphometric data stored with the *.nts extension.

2.8. Writing the **export.nts** function
Use the code of Section 2.4 to write a `export.nts` function that saves morphometric data acquired with R.

2.9. Writing a comprehensive function for estimating coordinates of missing landmarks of symmetric structures
Use the code of Section 2.5.2 to write a function that estimates the coordinates of missing landmarks on either one or the other side for both 2D or 3D configurations. The number of dimensions must be first estimated to apply adapted functions.

Springer

Springer