Intro to R

Dustin Pluta

May 9, 2017

Workshop Objectives

1. Crash course in R programming fundamentals.
2. Learn basic R syntax and style.
3. Perform computations of varying complexity, from simple arithmetic, up through statistical modeling.
4. Practice working with data in R: reading in, viewing, cleaning, manipulating, and analyzing.
5. Learn R base plot utilities.
6. Document results of analysis using R Markdown.

Philosophy of R

- Function as a high-level, interactive "calculator" for statistical modelling
- Facilitate the development of complex data analysis pipelines
- Designed with convenience for data analysis procedures (mostly)
- Supports the development of data analysis reports, visualizations, and interactive apps.
  - R Markdown
  - ggplot, plotly, and many other packages
  - R Shiny

Basic Commands

Variables and Data Types

Here is a list of the most important R data types:

- Numerics: `1, pi, 0.325, -Inf`

```
1 + pi
## [1] 4.141593
2 - Inf
## [1] -Inf
```

Basic Commands

Variables and Data Types

- Characters: ‘a’, “b”, “Hello”, “DSJF(@$U&AJfdaf32”

```
x = 'a'
x
## [1] "a"

y <- "Hello!"
print(y)
## [1] "Hello!"
```

- Booleans: TRUE, FALSE

```
y <- 1
y == 1
## [1] TRUE
```

Basic Commands

Variables and Data Types

- Factors: Categorical strings, used for modeling

```
as.factor(c("female", "male"))
## [1] female male
## Levels: female male
```

Basic Commands

Variables and Data Types

- Missing Values can be represented by `NA` or `NaN`:

```
z <- NA
is.na(z)
## [1] TRUE
y <- NaN
is.na(y)
## [1] TRUE
```

Basic Commands

Variables and Data Types

- Vectors:

```
x = c(1, 2, 3)
x <- 1:3
y <- c(-10, pi, 4, -10:-7)
chars <- c("1", "b", "Blah")
x
## [1] 1 2 3
y
## [1] -10.000000   3.141593   4.000000 -10.000000  -9.000
## [7]  -7.000000
chars
## [1] "1"    "b"    "Blah"
```

Basic Commands

Variables and Data Types

- Lists:

```
dat.list <- list(1, "alpha", c(1, 2))
```

- Matrices: Numeric arrays of numbers

```
X <- matrix(11:19, nrow = 3, ncol = 3)
print(X)
##      [,1] [,2] [,3]
## [1,]   11   14   17
## [2,]   12   15   18
## [3,]   13   16   19
dim(X)
## [1] 3 3
```

Basic Commands

Variables and Data Types

- Perhaps the most important data type in R is the `data.frame`
- Data frames are sort of like matrices, but can contain a mix of numerics, characters, and factors in their columns.
  - Each column must have a consistent type
- Data frames are the primary way to store your data for analysis

```
dat <- data.frame(X)
names(dat) <- c("HW1", "HW2", "Quiz")
print(dat)
##   HW1 HW2 Quiz
## 1  11  14   17
## 2  12  15   18
## 3  13  16   19
```

Basic Commands

Variables and Data Types

- R includes many example data sets to use for convenient testing

```
data(iris)
head(iris)
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Spec
## 1          5.1         3.5          1.4         0.2  set
## 2          4.9         3.0          1.4         0.2  set
## 3          4.7         3.2          1.3         0.2  set
## 4          4.6         3.1          1.5         0.2  set
## 5          5.0         3.6          1.4         0.2  set
## 6          5.4         3.9          1.7         0.4  set
```

Basic Commands

Variables and Data Types

- There are numerous built-in functions that can extract features and information of the data frame

```
names(iris)
## [1] "Sepal.Length" "Sepal.Width"  "Petal.Length" "Petal.
## [5] "Species"
nrow(iris)
## [1] 150
ncol(iris)
## [1] 5
class(iris)
## [1] "data.frame"
```

Basic Commands

Variables and Data Types

- We can access elements of the data frame using index notation (indices start at 1, not 0)

```
iris[1, 1]
## [1] 5.1
iris[3, 4]
## [1] 0.2
iris[1:3, 2:5]
##   Sepal.Width Petal.Length Petal.Width Species
## 1         3.5          1.4         0.2  setosa
## 2         3.0          1.4         0.2  setosa
## 3         3.2          1.3         0.2  setosa
```

Basic Commands

Variables and Data Types

- We can access whole rows or columns by omitting the appropriate index

```
# First Row
iris[1, ]
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Spec
## 1          5.1         3.5          1.4         0.2  set

# 3rd Column
iris[, 3]
##   [1] 1.4 1.4 1.3 1.5 1.4 1.7 1.4 1.5 1.4 1.5 1.5 1.6 1.
##  [18] 1.4 1.7 1.5 1.7 1.5 1.0 1.7 1.9 1.6 1.6 1.5 1.4 1.
##  [35] 1.5 1.2 1.3 1.4 1.3 1.5 1.3 1.3 1.3 1.6 1.9 1.4 1.
##  [52] 4.5 4.9 4.0 4.6 4.5 4.7 3.3 4.6 3.9 3.5 4.2 4.0 4.
##  [69] 4.5 3.9 4.8 4.0 4.9 4.7 4.3 4.4 4.8 5.0 4.5 3.5 3.
##  [86] 4.5 4.7 4.4 4.1 4.0 4.4 4.6 4.0 3.3 4.2 4.2 4.2 4.
```

Basic Commands

Variables and Data Types

- We can also access whole columns using the name of the column

```
iris$Sepal.Length[1:3]
## [1] 5.1 4.9 4.7
iris$Species
##   [1] setosa     setosa     setosa     setosa     setosa
##   [7] setosa     setosa     setosa     setosa     setosa
##  [13] setosa     setosa     setosa     setosa     setosa
##  [19] setosa     setosa     setosa     setosa     setosa
##  [25] setosa     setosa     setosa     setosa     setosa
##  [31] setosa     setosa     setosa     setosa     setosa
##  [37] setosa     setosa     setosa     setosa     setosa
##  [43] setosa     setosa     setosa     setosa     setosa
##  [49] setosa     setosa     versicolor versicolor versic
##  [55] versicolor versicolor versicolor versicolor versic
```

Basic Commands

Arithmetic

```
n <- 100
2*(1 - 1/n)^n
```

```
## [1] 0.7320647
```

```
2*exp(-1)
```

```
## [1] 0.7357589
```

```
log(1)
```

```
## [1] 0
```

Linear Algebra

- Often statistical modeling requires matrix computations.
- Important basic linear algebra operations are:
  - Matrix Multiplication: `X %*% Y`
  - Transpose: `t(X)`
  - Inverse: `solve(X)`
  - Determinant: `det(X)`
  - Eigenvalue Decomposition: `eigen(X)`

Linear Algebra

```
X <- matrix(c(1, 3:10), nrow = 3, byrow = TRUE)
Y <- c(1, 0, -1)
```

```
X + 2 * Y
##      [,1] [,2] [,3]
## [1,]    3    5    6
## [2,]    5    6    7
## [3,]    6    7    8
X %*% Y
##      [,1]
## [1,]   -3
## [2,]   -2
## [3,]   -2
```

Linear Algebra

- Careful! Usual multiplication with $*$ is actually elementwise:

```
X * Y
##      [,1] [,2] [,3]
## [1,]    1    3    4
## [2,]    0    0    0
## [3,]   -8   -9  -10
```

Linear Algebra

```
# Transpose
t(X)
##      [,1] [,2] [,3]
## [1,]    1    5    8
## [2,]    3    6    9
## [3,]    4    7   10
```

Linear Algebra

```
# Inverse
solve(X)
##      [,1]      [,2]      [,3]
## [1,]   -1  2.000000 -1.000000
## [2,]    2 -7.333333  4.333333
## [3,]   -1  5.000000 -3.000000
solve(X) %*% X
##      [,1]          [,2]          [,3]
## [1,]    1 -8.881784e-16 -1.776357e-15
## [2,]    0  1.000000e+00  1.776357e-15
## [3,]    0 -8.881784e-16  1.000000e+00
round(solve(X) %*% X, digits = 2)
##      [,1] [,2] [,3]
## [1,]    1    0    0
## [2,]    0    1    0
## [3,]    0    0    1
```

Linear Algebra

- If dimensions don't match, R will give an error:

```
Z <- matrix(c(2, 0, 4, -1), nrow = 2)
Z
##      [,1] [,2]
## [1,]    2    4
## [2,]    0   -1
X %*% Z
## Error in X %*% Z: non-conformable arguments
```

Functions

- Function declaration syntax is:

```
func_name <- function(arg1, arg2) {
  # ... Function Body ... #
  return(result)
}
```

Functions

- For example, let's write a function to find the hypotenuse of a right triangle:

```
pythag <- function(a, b) {
  c <- sqrt(a^2 + b^2)
  return(c)
}
pythag(3, 4)
## [1] 5
```

Functions

Exercise: Write a function that takes in a vector x and returns the mean.

Wrap this code to make it a function that returns the correct value:

```
x.sum <- 0
for (i in 1:length(x)) {
  x.sum <- x.sum + x[i]
}
```

```
my.mean <- function(x) {
  # ...
  return(result)
}
```

Functions

This works:

```
my.mean <- function(x) {
  x.sum <- 0
  for (i in 1:length(x)) {
    x.sum <- x.sum + x[i]
  }
  result <- x.sum/length(x)
  return(result)
}
```

Functions

Test the function:

```
# Our version
x <- runif(100, 0, 1)
my.mean(x)
## [1] 0.472922
```

```
# Built-in Version
mean(x)
## [1] 0.472922
```

Functions

Could also do this:

```
my.mean <- function(x) {
  return(sum(x)/length(x))
}
```

Functions

- **Programming Tips:**
  - Try to write small, tidy functions that do one specific thing well
  - Perform complicated analyses by stringing together many small functions
  - This improves code readability, makes it easier to debug when something goes wrong, and leads to cleaner code that can more easily be reused.

Packages

- There are many useful packages on the R repository CRAN.
- Packages are code modules, usually written by statisticians and academic researchers, or by development teams at companies like R Studio.
- Packages provide implementations of many statistical procedures and saves you the trouble of having to code up complicated algorithms from scratch.
- Relying on the many built-in R functions and the many packages available makes life much easier.

Packages

- Let's install the package for the Time Series Analysis book by Shumway & Stoffer

```
install.packages('astsa')
```

- Load the package with `library('pkg_name')`.

```
library(astsa)
```

- Useful documentation is available online for all packages on CRAN.
- `astsa` documentation

Reading Data

- The easiest file format to work with is '.csv', comma separated values.
- We can read csv files using `read.csv`.
- It is very helpful to read over the documentation carefully

```
?read.csv
```

```
dat <- read.csv("fmri_data.csv")
dim(dat)
## [1] 290 375
head(names(dat))
## [1] "V1" "V2" "V3" "V4" "V5" "V6"
dat[1:4, 1:4]
##          V1       V2       V3       V4
## 1 10468.39 9544.163 9911.728 9951.999
## 2 10447.24 9437.545 9851.216 9895.169
## 3 10418.98 9392.046 9892.233 9909.817
## 4 10422.40 9422.497 9825.961 9897.906
```

Prepping the Data

The data set is for one subject, 375 brain regions, 290 time pts.

```
names(dat) <- paste0("ROI", 1:375)
```

Add a column that indicates the time in seconds (1 time pt $=$ 2 seconds)

```
dat$t <- seq(2, 2*290, 2)
```

Plotting Data

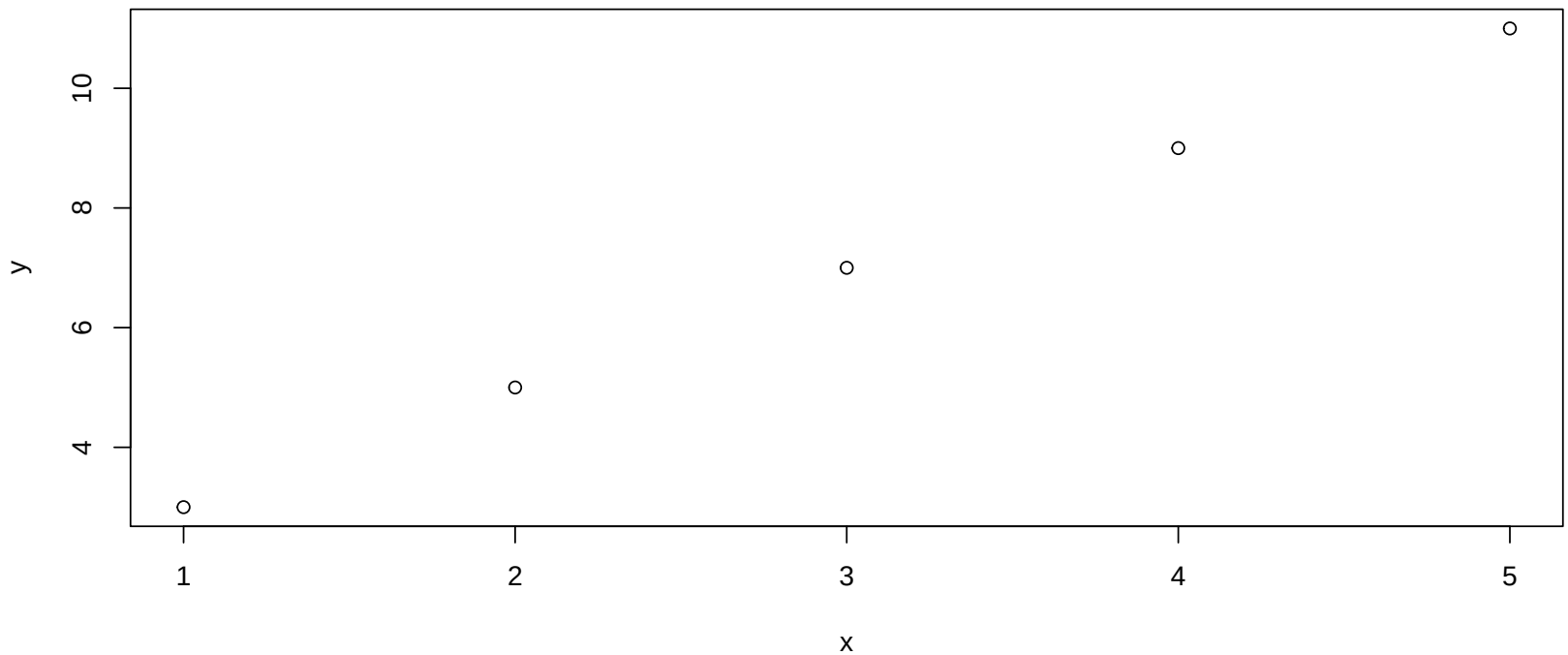
The base `R` function `plot()` works with many data types and models. It is the go-to function for simple plotting.

Again, read the documentation thoroughly!

```
?plot
```

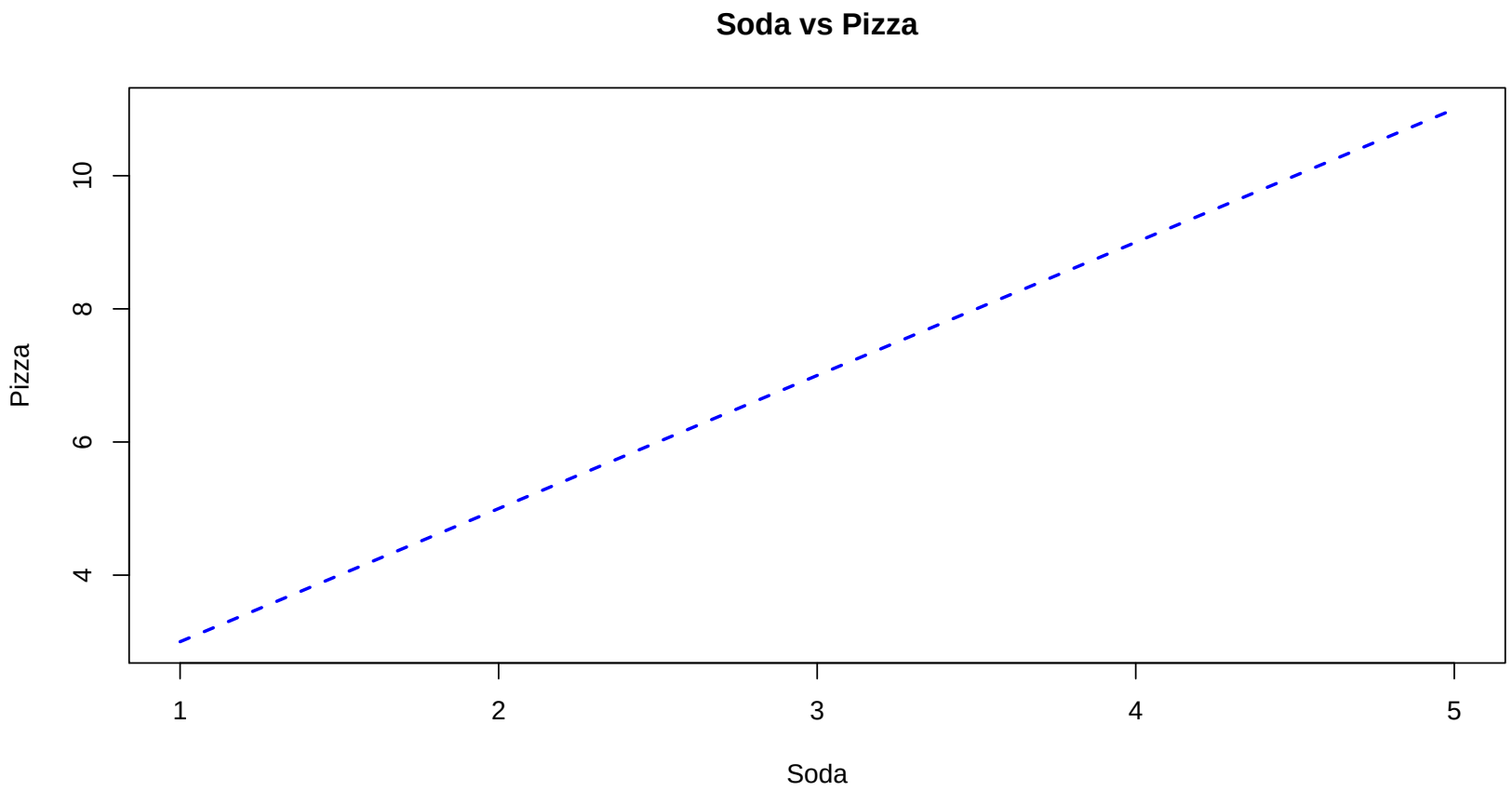
Plotting Data

```
x <- 1:5
y <- 2 * x + 1
plot(x, y)
```



Plotting Data

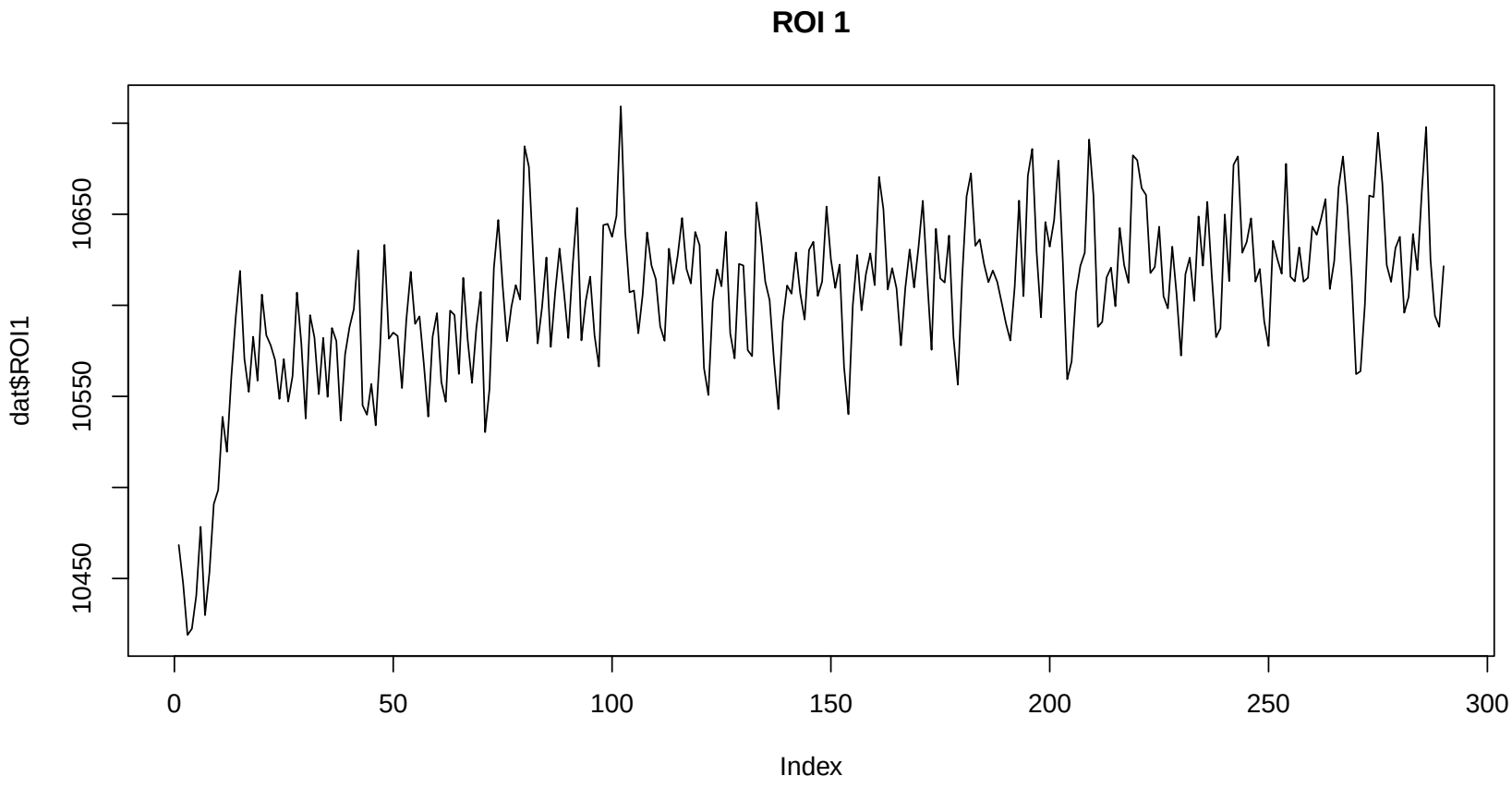
```
plot(x, y, ty = 'l', col = 'blue', lwd = 2, lty = 2,
     xlab = "Soda", ylab = "Pizza", main = "Soda vs Pizza")
```



Plotting Data

Plot the fMRI data for the first region

```
plot(dat$ROI1, ty = 'l', main = "ROI 1")
```



Preprocessing

- Hmmm, looks okay but there seems to be some strange behavior at the beginning of the time series.
- This is likely caused by "scanner drift" in the fMRI machine.
- When working with neurological data, there are typically many preprocessing steps that we need to perform before analyzing the data.

Preprocessing

- **Exercise 2** Remove the first 20 time pts (40 secs) and save the result into a vector named `ROI1`. Then replot the series, including a plot title.
- **Exercise 3** Use the functions `mean` and `sd` to center and scale the time series so that it has mean 0 and variance 1.

Preprocessing

- **Exercise 4** The data looks very noisy with lots of high-frequency oscillations, which is common in fMRI data. Can you think of a reasonable way to smooth the series by reducing these high-frequency oscillations (and hopefully thereby reducing the noise in the data)?
- **Exercise 5** Implement a smoothing procedure in `R` on the centered and scaled series. Plot the smoothed series over the unsmoothed (centered and scaled) series using `lines`.

Preprocessing

- **Exercise 6** Write a function called `preprocess` that takes in a time series x and performs the preprocessing steps in the above exercises:
  1. Remove the first 20 time pts.
  2. Center and scale.
  3. Smooth.
  4. Return the resulting series.

Preprocessing: Trim

```
trim <- function(x) {
  return(x[21:length(x)])
}
```

Preprocessing: Center and Scale

```
my.scale <- function(x) {
  return((x - mean(x))/sd(x))
}
```

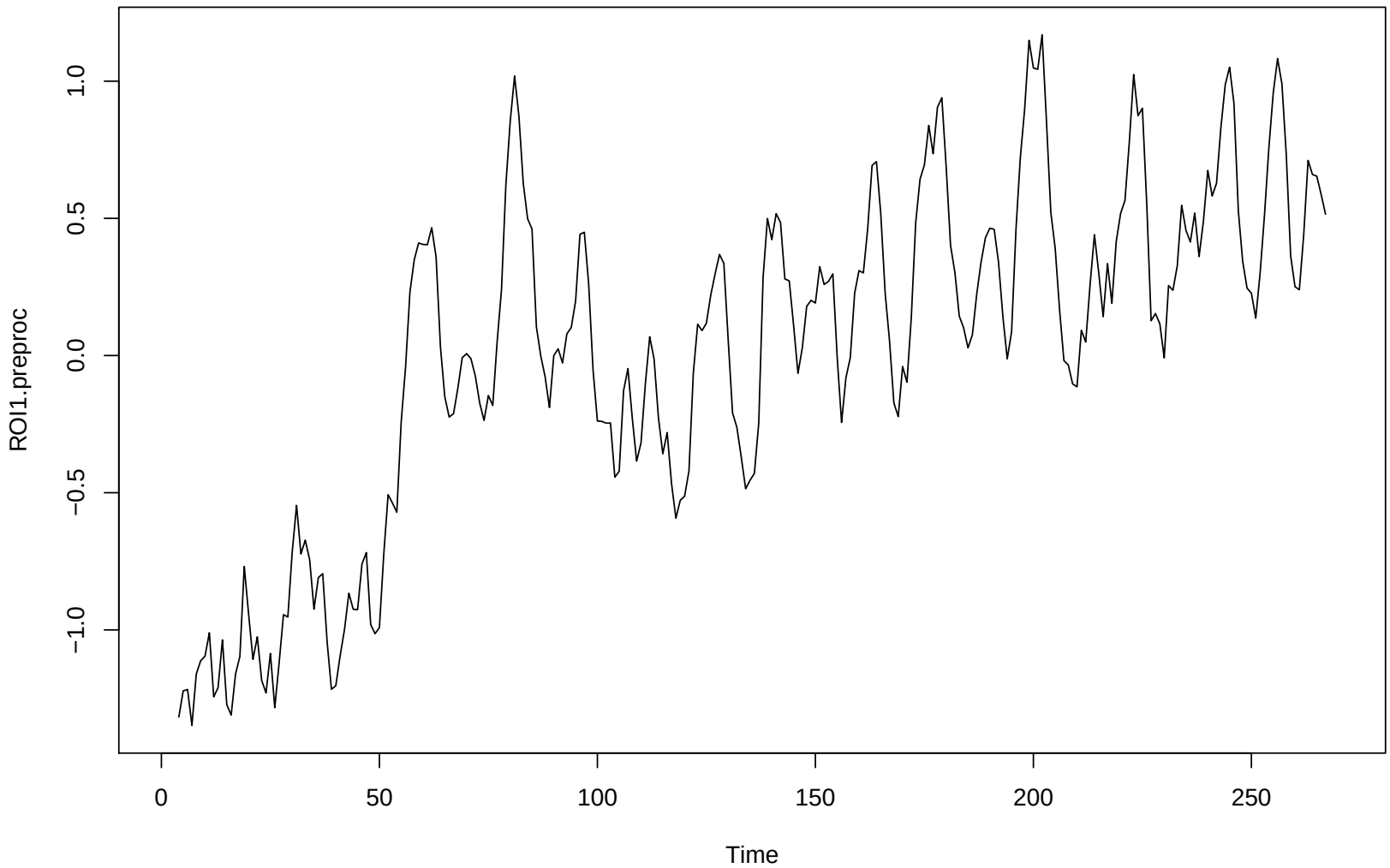
Preprocessing: Smooth

```
my.smooth <- function(x) {
  x <- filter(x, rep(1/7, 7), method = "convolution")
  return(x)
}
```

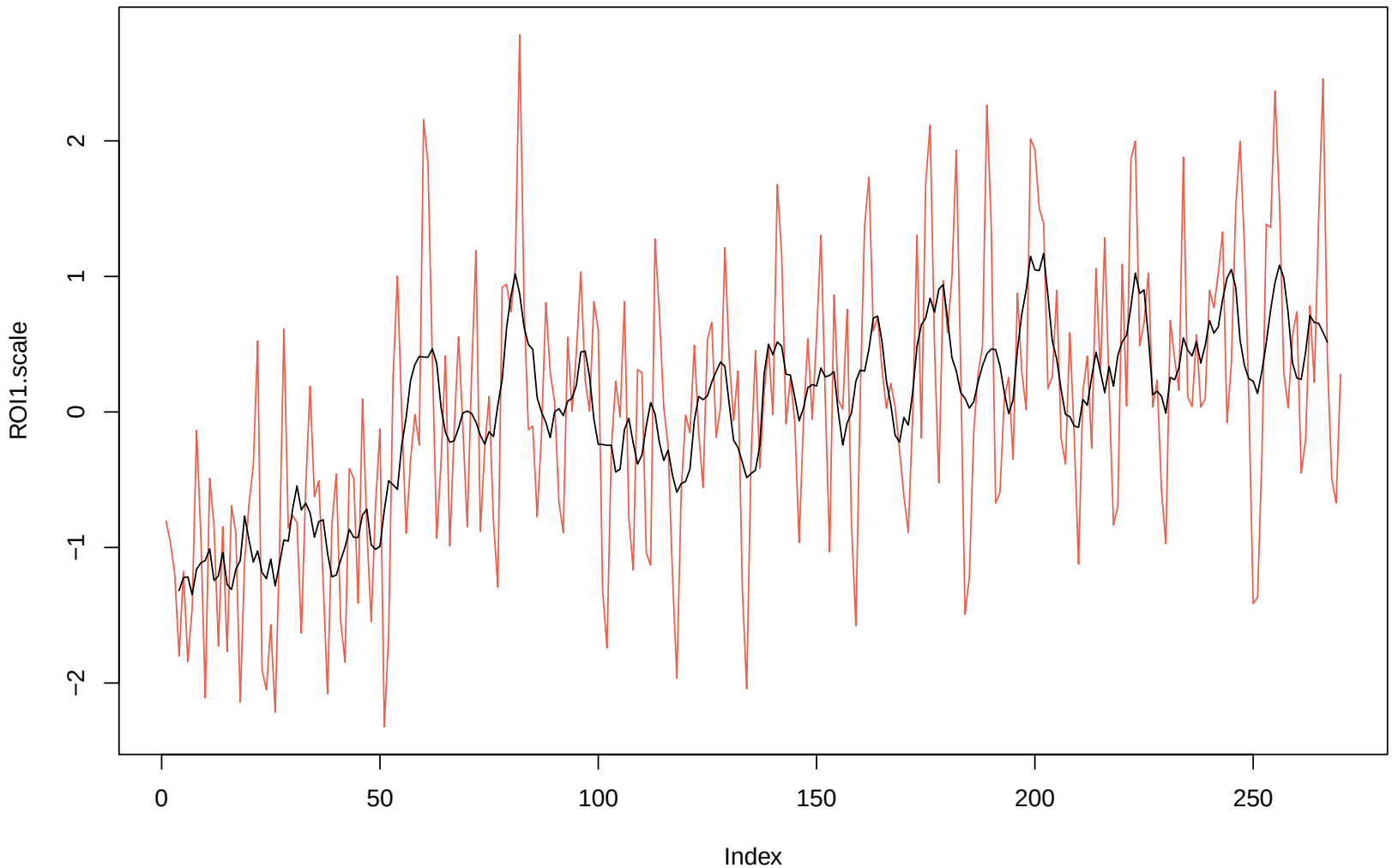
Preprocessing Function

```
preprocess <- function(x) {
  x <- trim(x)
  x <- my.scale(x)
  x <- my.smooth(x)
  return(x)
}
```

```
ROI1.preproc <- preprocess(dat$ROI1)
plot(ROI1.preproc, ty = 'l')
```



```
ROI1.scale <- my.scale(trim(dat$ROI1))
plot(ROI1.scale, ty = 'l', col = rgb(0.9, 0.1, 0, 0.7))
lines(ROI1.preproc, ty = 'l')
```



Preprocessing

Question Looking better, but there still seems to be a linear trend in the data... how can we remove this?

Answer ...

–

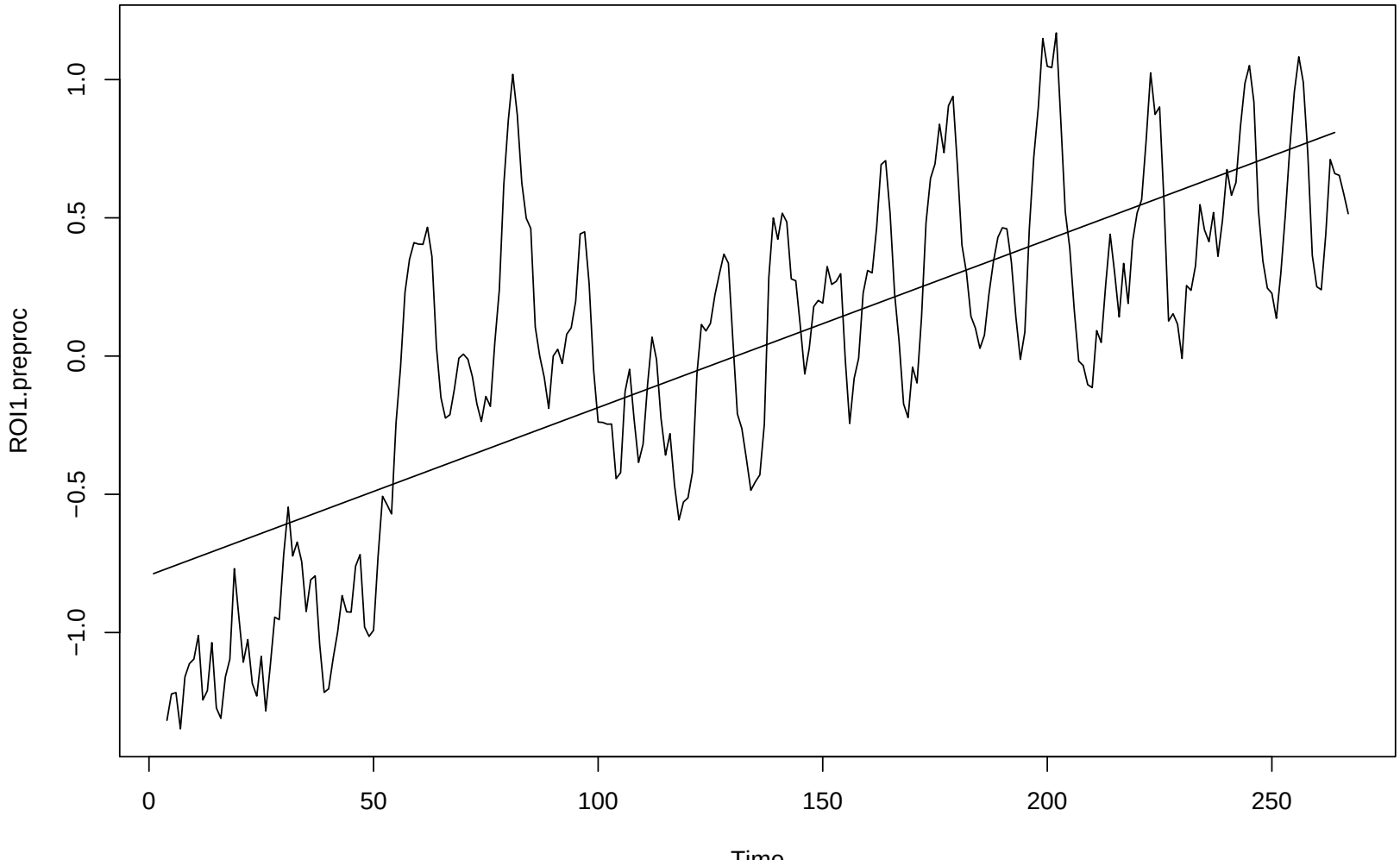
Apply Linear Regression and extract the residuals.

Preprocessing

```
ROI1.preproc <- na.omit(ROI1.preproc)
time.pts <- 1:length(ROI1.preproc)
fit <- lm(ROI1.preproc ~ time.pts)
```

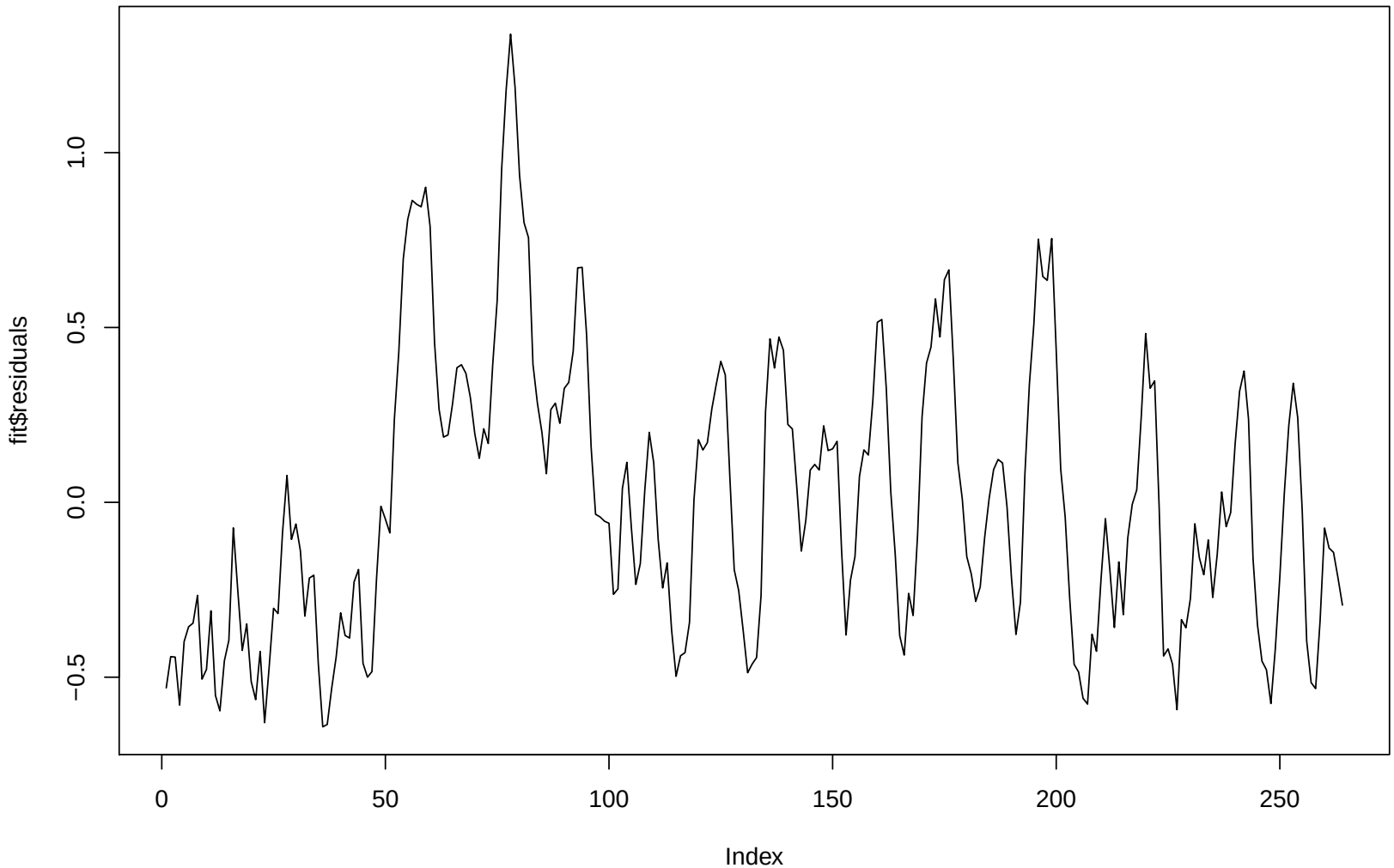
Preprocessing

```
plot(ROI1.preproc)
lines(fit$fitted.values)
```



Preprocessing

```
plot(fit$residuals, ty = 'l')
```



Preprocessing Function

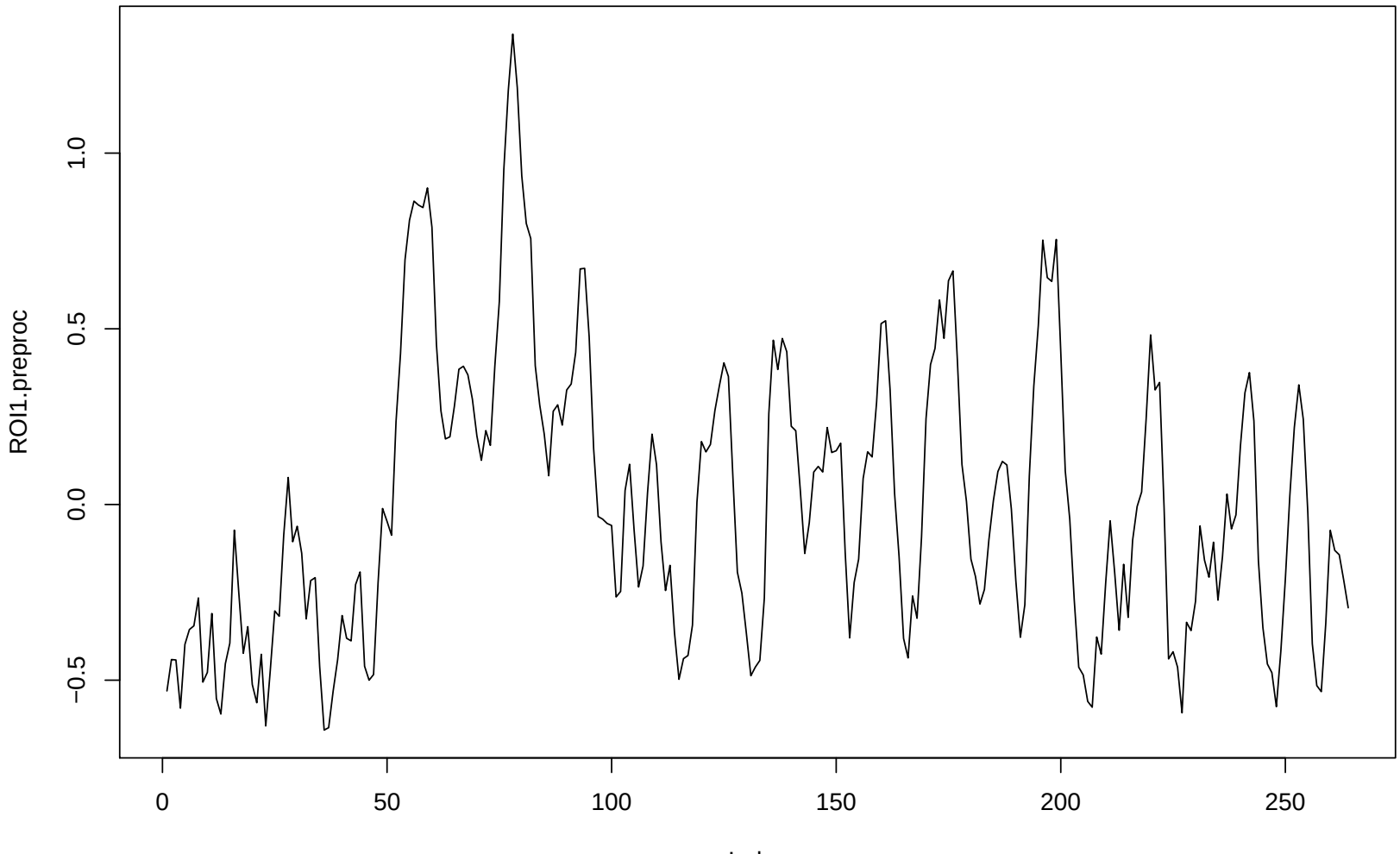
Update the preprocessing function to include this linear detrending.

```
detrend <- function(x) {
  x <- na.omit(x)
  time.pts <- 1:length(x)
  return(lm(x ~ time.pts)$residuals)
}

preprocess <- function(x) {
  x <- trim(x)
  x <- my.scale(x)
  x <- my.smooth(x)
  x <- detrend(x)
  return(x)
}
```

Preprocessing

```
ROI1.preproc <- preprocess(dat$ROI1)
plot(ROI1.preproc, ty = 'l')
```

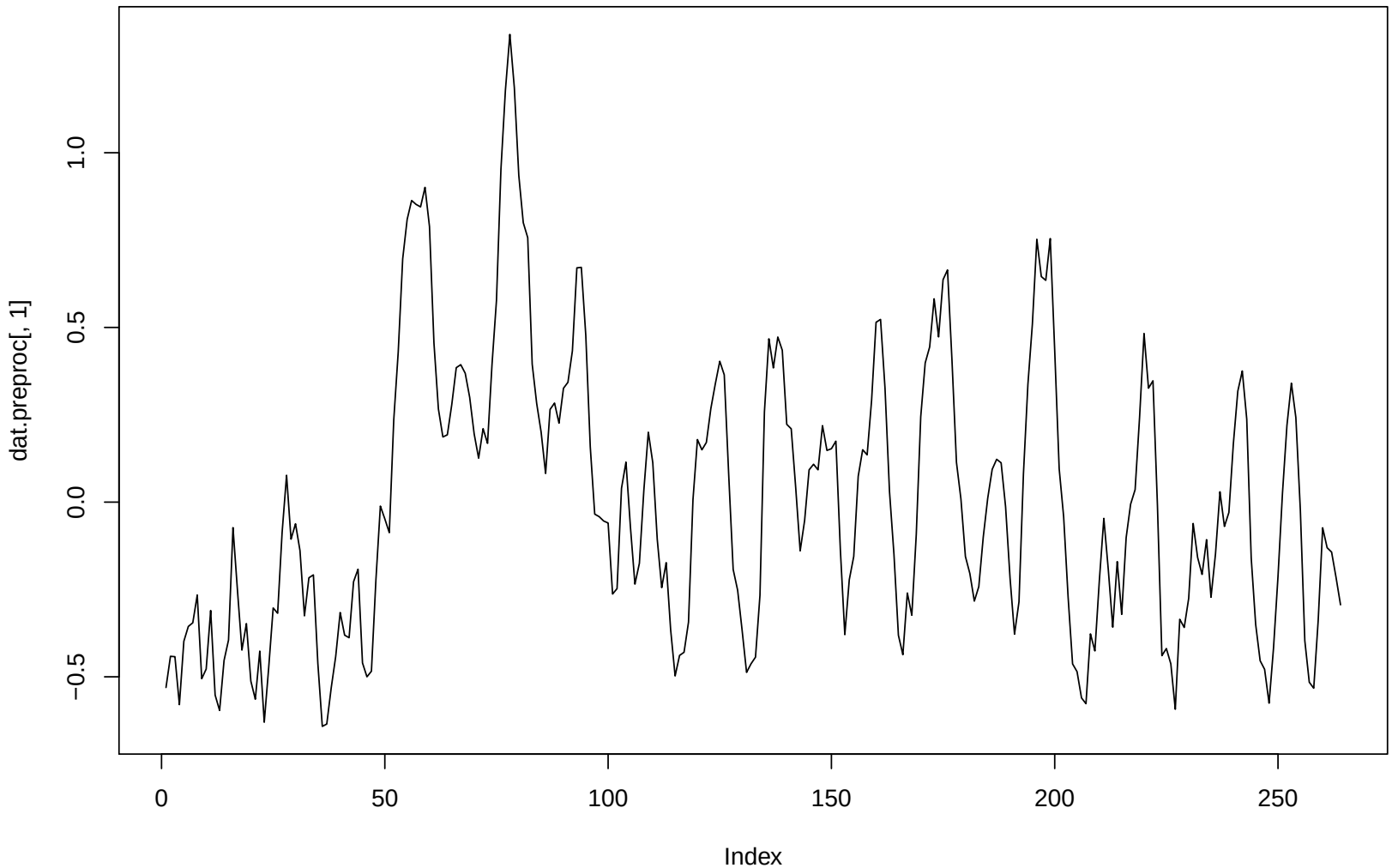


Preprocessing

We can preprocess all of the data simultaneously using our function and `apply`:

```
?apply
dat.preproc <- apply(dat, 2, preprocess)
```

```r
plot(dat.preproc[, 1], ty = 'l')
```



Writing Data

To write out the prepocessed data, we can use the `write.csv` function:

```
write.csv(x, "fmri_ROI1_preproc.csv")
```

Analyzing Data

Analysis steps:

0. Data preparation: collecting, cleaning, organizing, preprocessing.
1. Exploratory data analysis.
2. Choose the model.
3. Fit the model.
4. Diagnostics.
5. Interpret the results.
6. Visualize and report results.

Modelling

Let's fit a simple AR(2) model to the ROI series we just looked at.

The AR(2) model has the form

$$X(t) = \phi_1 X(t-1) + \phi_2 X(t-2) + w_t,$$

where $w_t \sim N(0, \sigma^2)$.

- We can use the base `R` function `ar` to easily fit the model

Modelling

```
fit <- ar(ROI1.preproc, order.max = 2)
summary(fit)
##                Length Class  Mode
## order             1    -none- numeric
## ar                2    -none- numeric
## var.pred          1    -none- numeric
## x.mean            1    -none- numeric
## aic               3    -none- numeric
## n.used            1    -none- numeric
## order.max         1    -none- numeric
## partialacf        2    -none- numeric
## resid           264    -none- numeric
## method            1    -none- character
## series            1    -none- character
## frequency         1    -none- numeric
## call              3    -none- call
## asy.var.coef      4    -none- numeric
```

Modelling

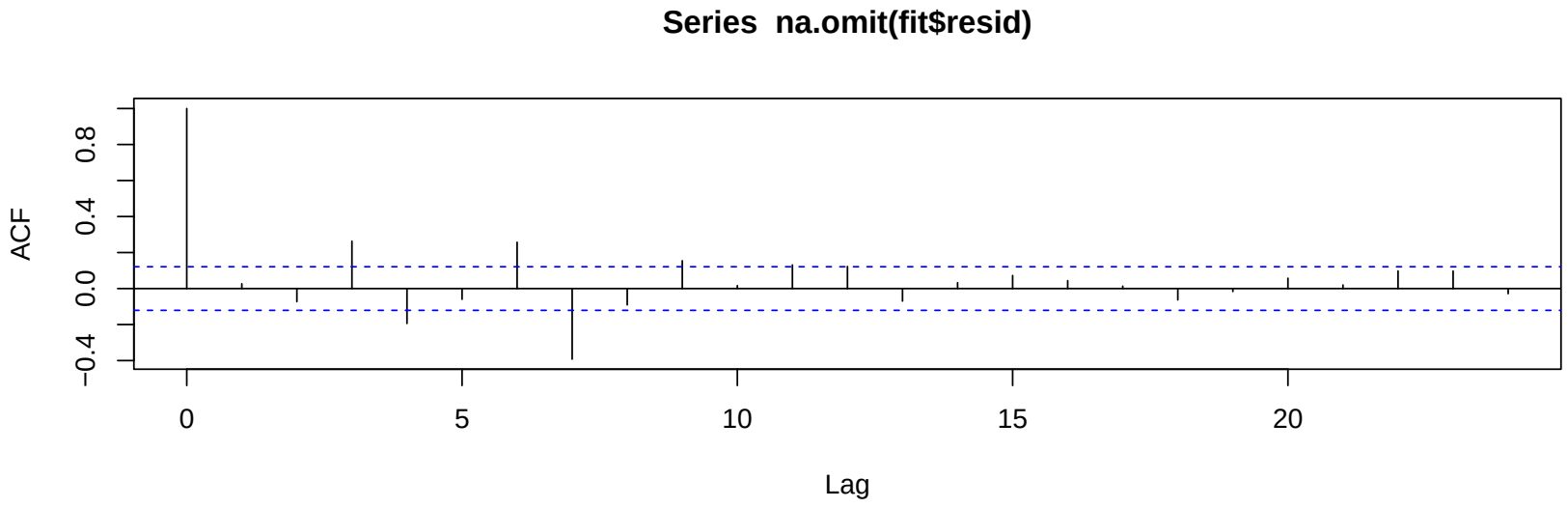
```
fit
##
## Call:
## ar(x = ROI1.preproc, order.max = 2)
##
## Coefficients:
##       1        2
##   1.283   -0.414
##
## Order selected 2  sigma^2 estimated as  0.02313
```

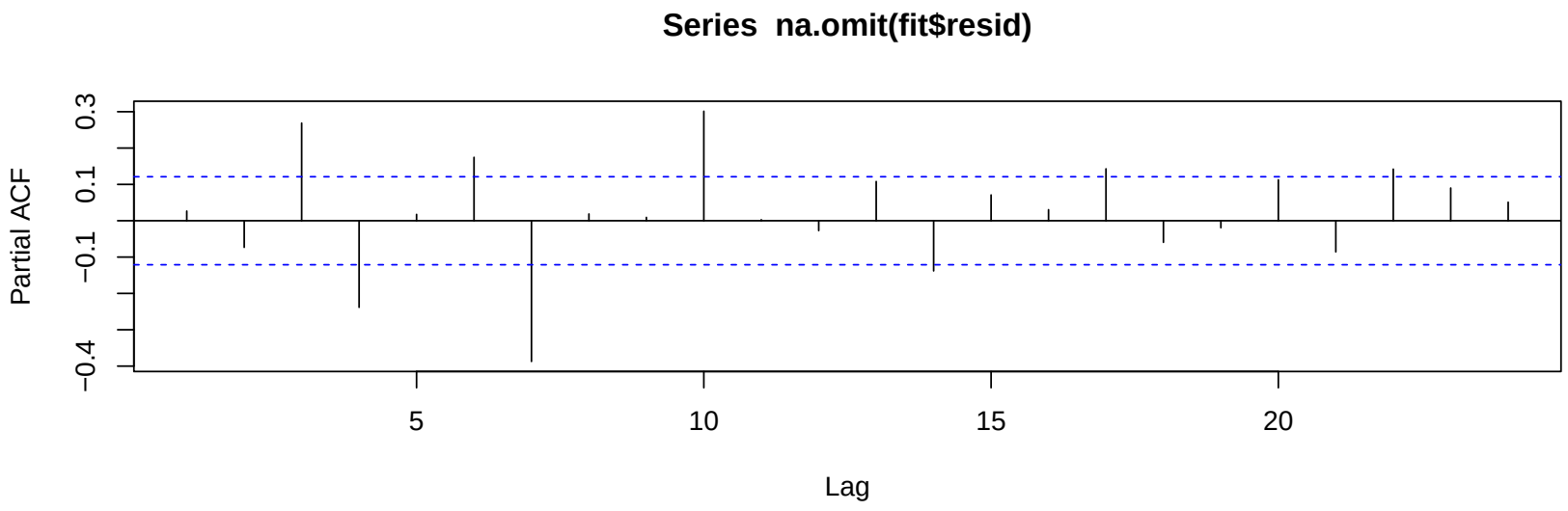
Diagnostics

We can evaluate the fit by looking at the Autocorrelation and Partial Autocorrelation of the residuals.

```
par(mfrow = c(2, 1))
acf(na.omit(fit$resid))
pacf(na.omit(fit$resid))
```

Diagnostics





Diagnostics

If the model fit is good, we expect these plots to be flat and within the blue lines.

It seems the model is a mediocre fit to the data, with some clear correlation left in the residuals for a few lags.

Next Analysis Steps

Likely need to reconsider the model. The plot of the preprocessed data does not seem stationary, so a simple AR model is likely not appropriate, should consider other modelling approaches.

Next Steps

Exercise 7 Use the `ar` function to fit an arbitrary order AR model to the preprocessed ROI1 data. Extract the AIC for this model and the AR(2) and compare.

Exercise 8 Apply a log transformation to the original ROI1 data (before preocessing), then repeat the preprocessing steps and fit an AR(2). Examine the acf and pacf of the residuals. Does the model fit seem to be any better or worse?

Resources

1. R Programming for Data Science
2. Many courses on Coursera, just search for “R”" or “Data Science.”