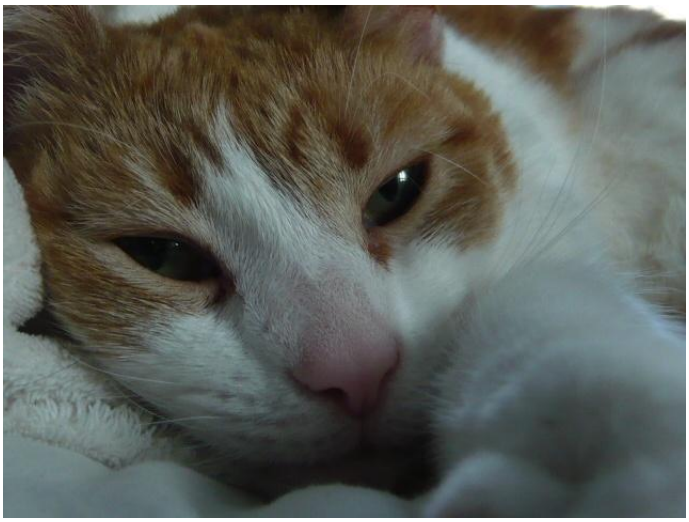
Namir's R 201 Regression Tutorial

by

Namir Shammas

Table of Contents



"Time spent with cats is never wasted."
Sigmund Freud

Introduction

This tutorial complements *Namir's R 101 Tutorial* and *Namir's R 102 Plotting Tutorial*. This tutorial focuses on a few R functions that perform various types of regression and curve fitting calculations. The various packages that are available for R users pack much power into them. The features available often sprint much information with the execution of a single command! This type of capability makes R the choice for statisticians and researchers who study trends in observed data.

My hope is that this tutorial gets your feet wet, so to speak. I encourage you to further experiment with the different regression functions to learn more about how to customize the results they produce. The bottom line is that learning any system must involve working and tinkering with that system. Cursory glances at R documentation or books contribute very little to learning R proficiently.

Regression at the Speed of Sound

Let me first quickly show you what R can do without much explanation and fanfare. So sit tight and buckle up! We'll be moving fast!

We start with the following data stored in file C:\regdata\lr1.txt. Create the host directory C:\regdata and also create the text file lr1.txt that contains the following data:

```
X     Y
1     6.41
```



```
2     9.07
3     10.66
4     13.64
5     15.18
6     18.35
7     23.4
8     24.42
9     28.06
10    31.67
11    33.59
12    36.76
13    42.29
14    45.01
15    46.57
16    49.19
(empty line)
```

Make sure that there is an extra blank line after the last set of values in the file.

Now execute the menu command File | New script in the R environment. R displays a blank script window. Type in the following lines:

```
lr1.data = read.table("C:/regdata/lr1.txt", header = TRUE)
attach(lr1.data, warn.conflicts=FALSE)
lr1 = lm("Y ~ X")
summary(lr1)
confint(lr1)
predict(lr1, interval="confidence")
predict(lr1, interval="prediction")
plot(X, Y, type ="p", main = "Y = a + b X")
abline(lr1)
detach(lr1.data)
```

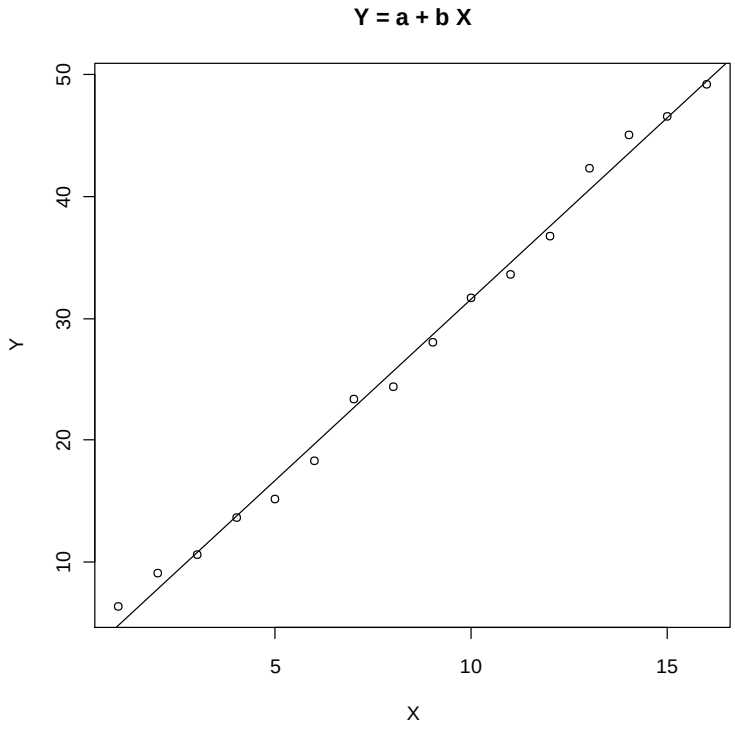
When you are done typing the above lines, select all the lines, by pressing the keys CTRL+A, and then execute the code in these lines by pressing the keys CTRL+R. The above code reads the data in the source file, performs linear regression calculations, and creates the graph you see in Figure 1. This figure shows the points for the input data and the linear regression line that passes through the points. In addition to Figure 1, the R Console window shows the following information which contains a wealth of results related to the linear regression calculations:

```
(output from summary()function)
Call:
lm(formula = "Y ~ X")

Residuals:
   Min     1Q Median     3Q    Max
-1.562 -0.832 -0.134  0.847  1.776

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)   1.8842     0.5910    3.19   0.0066 **
X             2.9715     0.0611   48.61   <2e-16 ***
---
```



```
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 1.13 on 14 degrees of freedom
Multiple R-squared: 0.994,      Adjusted R-squared: 0.994
F-statistic: 2.36e+03 on 1 and 14 DF,  p-value: <2e-16

(output from confint() function)
               2.5 %   97.5 %
(Intercept) 0.6165807 3.151919
X           2.8403861 3.102585

(output from predict(lr1, interval="confidence"))
         fit       lwr       upr
1   4.855735  3.701615  6.009855
2   7.827221  6.782532  8.871910
3  10.798706  9.857892 11.739520
4  13.770191 12.925643 14.614739
5  16.741676 15.982885 17.500468
6  19.713162 19.025672 20.400652
7  22.684647 22.049120 23.320174
8  25.656132 25.048250 26.264015
9  28.627618 28.019735 29.235500
10 31.599103 30.963576 32.234630
11 34.570588 33.883098 35.258078
12 37.542074 36.783282 38.300865
13 40.513559 39.669011 41.358107
14 43.485044 42.544230 44.425858
15 46.456529 45.411840 47.501218
16 49.428015 48.273895 50.582135

(output from predict(lr1, interval="prediction"))
         fit       lwr       upr
1   4.855735  2.177009  7.534462
2   7.827221  5.193790 10.460652
3  10.798706  8.204729 13.392682
4  13.770191 11.209558 16.330825
5  16.741676 14.208033 19.275320
6  19.713162 17.199951 22.226372
7  22.684647 20.185152 25.184143
8  25.656132 23.163522 28.148742
9  28.627618 26.135008 31.120228
10 31.599103 29.099607 34.098598
11 34.570588 32.057378 37.083799
12 37.542074 35.008430 40.075717
13 40.513559 37.952925 43.074192
14 43.485044 40.891068 46.079021
15 46.456529 43.823098 49.089960
16 49.428015 46.749288 52.106741
Warning message:
In predict.lm(lr1, interval = "prediction") :
  Predictions on current data refer to _future_ responses
```


Figure 1. A simple linear fit.



Ok, that was fast! I wanted to give you a taste for what R can do! The next section will perform the above tasks in slow motion and with the *director's comments*, so to speak.

Simple Linear Regression: Take 2

The Hands-On Manual Approach

Before one starts building cool R user-defined functions that perform common sequences of calculations and other tasks, it will be helpful to understand the required steps and how they work.

Consider the case where you want to do the following tasks:

1. You have a data file with X and Y values and want to read that data.
2. You want to perform simple linear regression calculations using the values in the data you read.
3. You want to plot a graph that shows the observed data and the best fitted line.
4. You want to access the regression statistics for further study.

The next subsections focus on the above tasks and see how you can accomplish them in R.

Task 1: Reading the Data

The very first step in the journey of statistical calculations is perhaps the task of data gathering. You have information that resides in a source. You need to access that source, or create the data yourself, if you are seeking test data. In either case, I recommend that you store the data in a spreadsheet like Excel. This approach allows you to perform basic data manipulation, filtering, and massaging before you let an R function loose on it!

Typing in the Data

Once you have the data, residing in Excel, in the form you want, you can then save it to a text file. You can choose to delimit the data (including the names of the variables) using spaces or commas. I chose spaces. The sample data I use appears next, and is stored in file C:\regdata\lr1.txt:

```
X     Y
1     6.41
2     9.07
3     10.66
4     13.64
5     15.18
6     18.35
7     23.4
8     24.42
9     28.06
10    31.67
11    33.59
12    36.76
13    42.29
14    45.01
15    46.57
16    49.19
(empty line)
```

I will be using the directory (or, folder if you prefer) C:\regdata to store data and functions that I present in this tutorial. So to be in sync with the presentation, I suggest you create the C:\regdata in your computer too. After you are done with the tutorial, you can move that directory to another location of your choosing.

Reading the Data File

Once you have the data in place, you can use the R function read.table() to read that data. To perform this step, execute the following command to read the data and store it in the data frame lr1.data:

```
> lr1.data = read.table("C:/regdata/lr1.txt", header = TRUE)
```

The above command reads the data in file C:\regdata\lr1.txt and stores it in the data frame lr1.data. If you type the name of that variable, R generates the following output:

```
> lr1.data
```



```
    X     Y
1   1  6.41
2   2  9.07
3   3 10.66
4   4 13.64
5   5 15.18
6   6 18.35
7   7 23.40
8   8 24.42
9   9 28.06
10 10 31.67
11 11 33.59
12 12 36.76
13 13 42.29
14 14 45.01
15 15 46.57
16 16 49.19
```

Putting the Data Frame Object Under the Microscope

Let's tinker with the variable lr1.data a bit to learn how to extract information from it. Let's try to access its columns of data by using an index. For example, type in the following command:

```
> lr1.data[1]
    X
1   1
2   2
3   3
4   4
5   5
6   6
7   7
8   8
9   9
10 10
11 11
12 12
13 13
14 14
15 15
16 16
```

The expression lr1.data[1] basically displays the first column—both name and values. If you Then execute the command lr1.data[2] you get the name and values of the second column:

```
       Y
1   6.41
2   9.07
3  10.66
4  13.64
5  15.18
6  18.35
7  23.40
8  24.42
9  28.06
```



```
10 31.67
11 33.59
12 36.76
13 42.29
14 45.01
15 46.57
16 49.19
```

Now lets try to access the first column by specifying the name of the variable X as an index:

```
> lr1.data["X"]
    X
1   1
2   2
3   3
4   4
5   5
6   6
7   7
8   8
9   9
10 10
11 11
12 12
13 13
14 14
15 15
16 16
```

Let's try to store the first column in a separate variable and see what we get:

```
> (xv=lr1.data[1])
    X
1   1
2   2
3   3
4   4
5   5
6   6
7   7
8   8
9   9
10 10
11 11
12 12
13 13
14 14
15 15
16 16
```

The variable xv stores the first column of the data frame, including the name of the column. Let's repeat the above task and use function as.vector() to see what we get:

```
> (xv=as.vector(lr1.data[1]))
    X
```



```
1   1
2   2
3   3
4   4
5   5
6   6
7   7
8   8
9   9
10 10
11 11
12 12
13 13
14 14
15 15
16 16
```

Using function as.vector() does not remove the column name!

Let's try another approach that R supports. You can access the data in a column by using the column's name and qualifying it with the name of the data frame. The syntax needed is data.frame$column.name. For example, execute the following command to access the data of column X:

```
> lr1.data$X
 [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16
```

Notice that the result does not include the name of the column—just the values in that column. So using the syntax data.frame$column.name is one approach that shakes off the column name and returns a vector of values.

To get the column names, R offers the function names(data.frame) to return a vector of the column names. Execute the following command to store the column names in variable col.names and to display the column names:

```
> (col.names = names(lr1.data))
[1] "X" "Y"
```

You can access the column names using the varies col.names or by placing an index after the result of function names(). Here is an example that shows how to get the name of the first column using both approaches:

```
> col.names[1]
[1] "X"
> names(lr1.data)[1]
[1] "X"
```

What about accessing specific elements of a range of elements in the data frame? R allows you to access specific elements or a range of rows and columns in a data frame, just like you would

in a matrix. For example, type the following command to get the element at row 2 and column 1 in the data frame lr1.data:

```
> lr1.data[2,1]
[1] 2
```

To access rows 3 to 5 in column 2 of the data frame lr1.data, type the following command:

```
> lr1.data[3:5,2]
[1] 10.66 13.64 15.18
```

Notice that the output does not include the names of the columns. This feature allows us to extract columns (and even rows) from the data frame and store them as regular vectors. To store column 2 in the variable vx, execute the following command:

```
> (xv = lr1.data[,2])
 [1]  6.41  9.07 10.66 13.64 15.18 18.35 23.40 24.42 28.06 31.67 33.59 36.76
[13] 42.29 45.01 46.57 49.19
```

Notice that the variable vx does not have the name of any column included in its information! So the matrix-style indexing provides a second mechanism to extract only the data from a data frame column and store it as a regular vector.

You learned how to read data from a file and into a data frame, then how to access information from the data frame.

Task 2: The Regression Calculations

The next step in performing the linear regression calculations is a practical one. You have seen in the last subsection that you can access the data in the first column using the expression lr1.data$X. This approach is true for all columns in a data frame. Developers of the R language realized that this mandatory qualification was a bit tedious and often error prone—the less you type, the less likely you make a mistake!

Attaching the Data Frame Object

R offers the function attach(data.frame) as a mechanism to free you from qualifying data frame columns . To illustrate the function attach() execute the following command first:

```
> attach(lr1.data, warn.conflicts=FALSE)
```

And then type in the name of the first column:

```
> X
 [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16
```

And now type in the name of the second column:

```
> Y
 [1]  6.41  9.07 10.66 13.64 15.18 18.35 23.40 24.42 28.06 31.67 33.59 36.76
```



```
[13] 42.29 45.01 46.57 49.19
```

You can now access the columns X and Y by simply using their names.

When you are done with a data frame, use the function detach(data.frame) to restore system resources. The action of function attach() makes the R system work harder in trying to figure out that certain variables are really column variables in a data frame.

Performing the Linear Regression Calculations

The next step in this task is to perform the linear regression calculations. R offers the very powerful function lm(). This function requires a minimum of one argument—the expression for the linear model. To perform a simple linear regression between Y and X, execute the following command:

```
> (lr1 = lm("Y ~ X"))
```

```
Call:
lm(formula = "Y ~ X")

Coefficients:
(Intercept)            X
       1.88         2.97
```

The above command does the following:

- Performs the linear regression calculations.
- Displays the fitted model
- Displays the linear regression coefficients.

Notice that the argument for function lm() is the string "Y ~ X". This string tells the function to use variable Y (which we got from the data frame lr1.data) as the dependent variable, and to use the variable X as the independent variable. The tilde character separates the dependent variable and in dependent variable. In the case of a simple linear regression the string "Y ~ X" does the job.

 It is worth pointing out that the formula parameter for function lm(), and other similar regression functions, can be written without the double quotes! This variant works just fine. The advantage of using a formula string lies in the ability to assemble the formula text dynamically and programmatically—by creating a function utilizing statements before the call to function lm() that obtain the names of the variables from the data frame and then assemble the formula string. Such an approach allows such user-defined functions to process variables with names other than X and Y. In fact, you can tell such functions which columns to select using the indices of the column variables in a data frame. These indices select the dependent and independent

variables from the data frame. This approach is even more convenient when reading data with more than two variables. You have the power to select any two variables for linear regression calculations and not worry about the names of these variables.

The Regression Summary

The above command stores the results of the linear regression in the variable lr1. While the output show a few regression statistics, the real power of R shines when you use the summary() function. Type the following command to view a wealth of regression statistics:

```
> summary(lr1)

Call:
lm(formula = "Y ~ X")

Residuals:
   Min     1Q Median     3Q    Max
-1.562 -0.832 -0.134  0.847  1.776

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)   1.8842     0.5910    3.19   0.0066 **
X             2.9715     0.0611   48.61   <2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 1.13 on 14 degrees of freedom
Multiple R-squared: 0.994,      Adjusted R-squared: 0.994
F-statistic: 2.36e+03 on 1 and 14 DF,  p-value: <2e-16
```

The above output shows the regression coefficients along with their standard errors, t-values, probabilities of being zero. the residual standard error, degrees of freedom, multiple coefficient of determination, adjusted coefficient of determination, and the F statistic.

The variable lr1 is an object that contains the mother load!

The Confidence Interval for the Regression Coefficients

R provides the function confint() to calculate and display the confidence interval for the regression coefficients. This function takes as an argument a result of the call to function lm(). Type the following command to view that confidence interval at the default 95% confidence level:

```
> confint(lr1)
                2.5 %    97.5 %
(Intercept) 0.6165807 3.151919
X           2.8403861 3.102585
```

The confidence interval for the intercept is, at 95% confidence level, in the interval of 0.616 and 3.151. Likewise, the confidence interval for the intercept is, at 95% confidence level, in the interval of 2.84 and 3.10.

The Confidence Interval for the Dependent Variable

The function predict() calculates the confidence interval for the dependent variable. The declaration for this function is:

```
predict(object, newdata, se.fit = FALSE, scale = NULL, df = Inf,
        interval = c("none", "confidence", "prediction"),
        level = 0.95, type = c("response", "terms"),
        terms = NULL, na.action = na.pass,
        pred.var = res.var/weights, weights = 1, ...)
```

I will focus on the parameters object, level, and interval. The parameter **object** is a variable that stores the result of calling the function lm(). The parameter **level** specified the confidence level, as a fraction, and has the default value of 0.95. The parameter **interval** specifies what kind of interval to calculate for the observed values of the dependent variable. The arguments for this parameter are either "confidence" or "prediction" and can be abbreviated using "c" and "p", respectively. Both arguments to parameter interval generate three columns: the fitted values for the dependent variable, and its lower and upper intervals. When you specify "confidence" the function returns the confidence interval for the fitted data. When you specify "prediction" the function returns a wider confidence interval for the observed data. The function uses slightly different equations to calculate each kind of confidence interval. One equation regards the data as the ones currently involved in the regression, while the other equations regards the data as "future" values—data that you will use after the regression calculations.

Putting the Regression Summary under the Microscope

To put the variable lr1 under the microscope, execute the following command to get the names of the components in variable lr1:

```
> names(lr1)
 [1] "coefficients"  "residuals"     "effects"       "rank"
 [5] "fitted.values" "assign"        "qr"            "df.residual"
 [9] "xlevels"       "call"          "terms"         "model"
```

The next paragraphs discuss the relevant component of the regression result object.

The **coefficients** component contains the regression coefficients. Execute the following command to view what the vector lr1$coefficients reveals:

```
> lr1$coefficients
(Intercept)           X
    1.88425     2.97149
```

Now, let's attempt to access each coefficient individually. To access the intercept, execute the following command:

```
> lr1$coefficients[1]
(Intercept)
    1.88425
```


To access the slope, execute the following command:

```
> lr1$coefficients[2]
      X
2.97149
```

The component **fitted.values** returns the matrix of fitted Y values calculated based on the linear regression statistics. Execute the following command to view what lr1$fitted.value gives:

```
> lr1$fitted.values
       1        2        3        4        5        6        7        8
 4.85574  7.82722 10.79871 13.77019 16.74168 19.71316 22.68465 25.65613
       9       10       11       12       13       14       15       16
28.62762 31.59910 34.57059 37.54207 40.51356 43.48504 46.45653 49.42801
```

The fitted.values component is an array. You can access any element by its index. Execute the following command to access the values in index 3:

```
> lr1$fitted.values[3]
      3
10.7987
```

The **residuals** component returns values for X and the residuals for Y (calculated as the difference between the observed value and the one calculated using the linear fit). Execute the following command to view the residuals in lr1:

```
> lr1$residuals
        1         2         3         4         5         6         7
8
 1.554265  1.242779 -0.138706 -0.130191 -1.561676 -1.363162  0.715353 -
1.236132
        9        10        11        12        13        14        15
16
-0.567618  0.070897 -0.980588 -0.782074  1.776441  1.524956  0.113471 -
0.238015
```

The **call** component returns the formula used in the regression. Execute the following command to view lr1$call:

```
> lr1$call
lm(formula = "Y ~ X")
```

Well, there is no surprise here!

The **df.residual** component is the residuals degrees of freedom. Execute the following command to obtain the value of lr1$df.residual:

```
> lr1$df.residual
[1] 14
```


Task 3: Graphing the Results

This task concerns itself with plotting the observed data and drawing the best fit line. The first step is to plot the values in variables X and Y, using function plot(), by executing the following command:

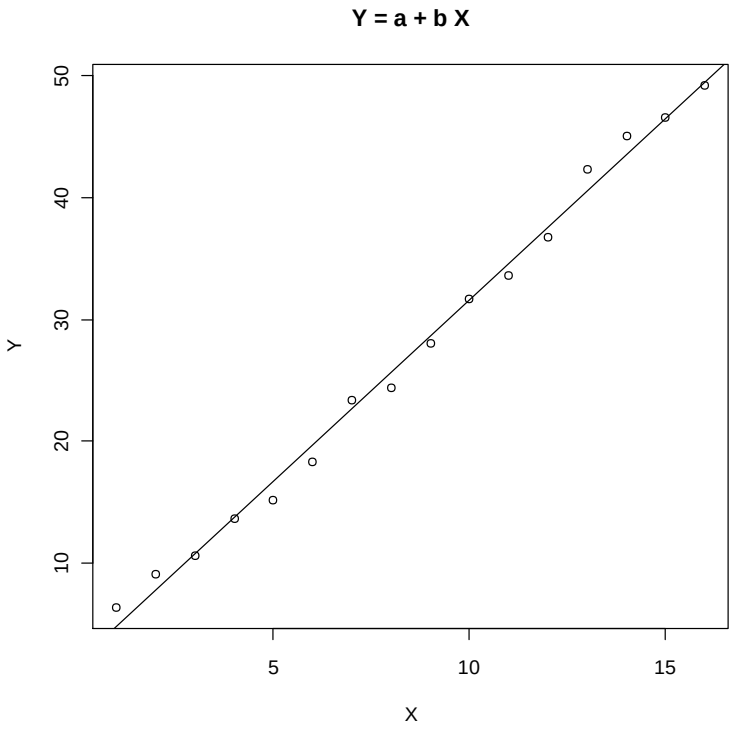
```
> plot(X, Y, type ="p", main = "Y = a + b X")
```

Next draw the line using the function abline(). R allows you to simply specify the linear regression object as an argument and not worry about the specific values in the coefficients components. Execute the following command:

```
> abline(lr1)
```

The above commands generate the graph in Figure 2 which shows the points for the observed data and the regression line. Figures 1 and 2 are identical.

Figure 2. A simple linear regression graph.



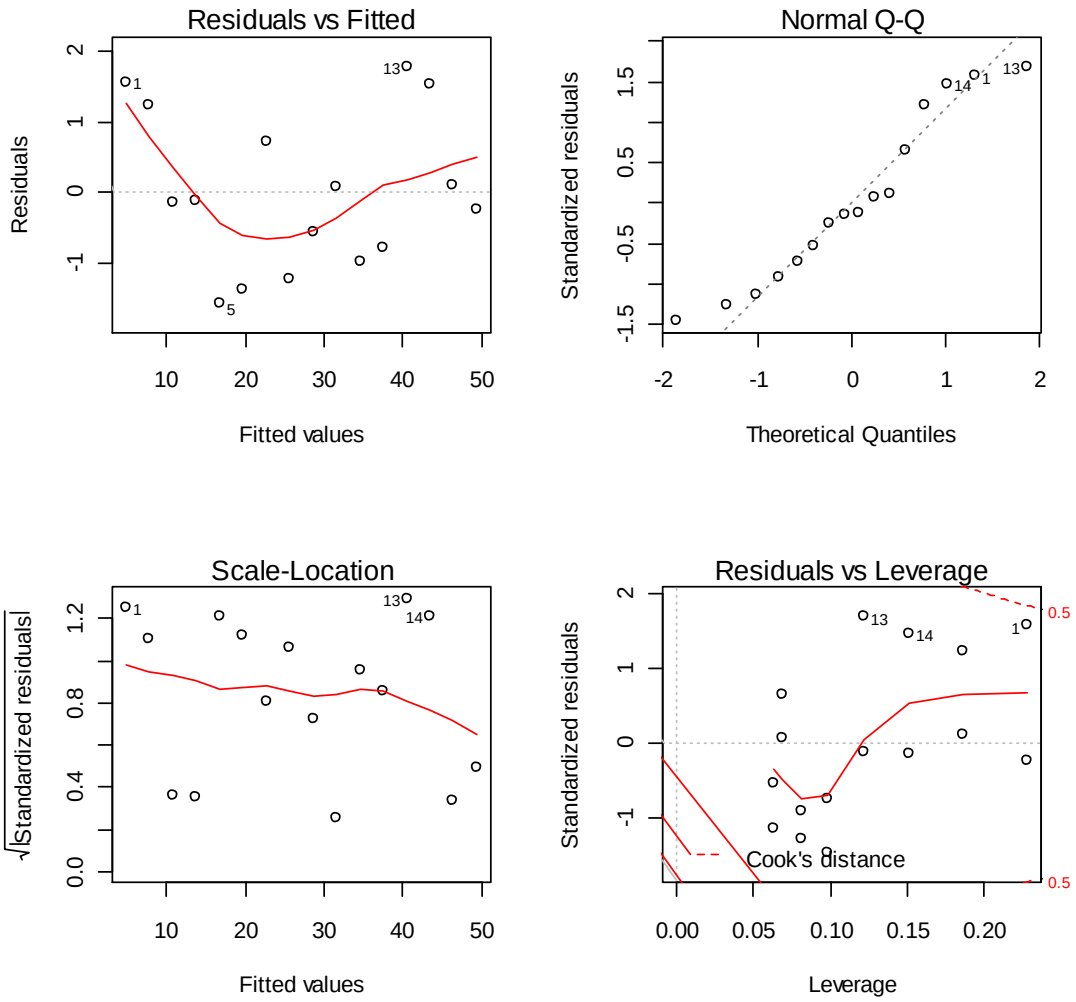
R allows you to use the regression object lr1 with the function plot(). Execute the following commands:

```
> par(mfrow=c(2,2))
> plot(lr1)
> par(mfrow=c(1,1))
```


The call to function plot(lr1) generates Figure 3 with the following four graphs:

1. The Residuals versus fitted data.
2. The Normal Q-Q plot.
3. The Scale Location plot. This plot shows the square root of the standardized residuals versus the fitted data.
4. The standardized residuals versus leverage plot.

Figure 3. A graphical summary for the regression data.



Task 4: Cleanup and Relishing in the Prize

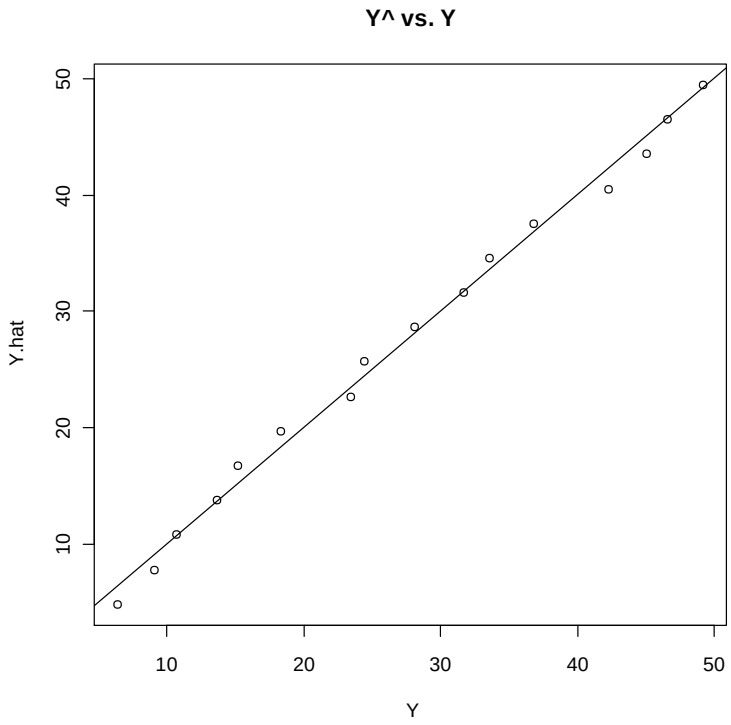
This task is the simplest. Armed with the variable lr1, which is an object that has a lot of information related to the linear regression, you can perform additional calculations if you need to. Essentially the variable that stores the regression result is your prize for going through all these steps.

Before we wind things down, I suggest that you plot the projected Y value versus the observed ones. The vector lr1$fitted.values stores the fitted values of Y. Execute the next two commands to plot the Y.hat vs. Y and then to draw the 45-degree line that goes through the origin (0, 0):

```
> plot(Y, lr1$fitted.values, type = "p", main="Y^ vs. Y")
> abline(0,1)
```

Figure 4 shows the plot that visually compares the calculated values of Y vs. the observed ones.

Figure 4. Y^ vs. Y for the simple linear regression.

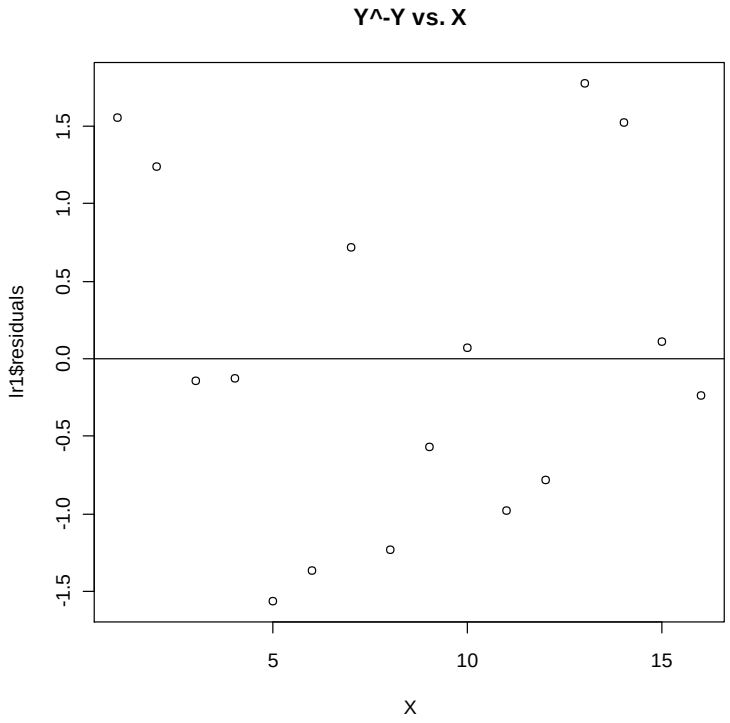


You can plot the residual values vs. the values of Y. Use the vector lr1$residuals to access the residual values of Y. Execute the next two commands to plot the residuals vs. the values of Y and then draw a line passing through Y = 0:

```
> plot(Y, lr1$residuals, type = "p", main="Y^ vs. Y")
> abline(0,1)
```

Figure 5 shows the residuals. The horizontal line through Y=0 separates the positive and the negative residuals. These residuals seem to be evenly scattered.

Figure 5. Residual plot for simple linear regression.



As we wrap up the example, it's a good idea to detach the data frame lr1.data by executing the following command:

```
> detach(lr1.data)
```

Simple Linear Regression: The Function

Armed with the step-by-step knowledge for performing linear regression, one may contemplate taking matters to the next level. By this I mean creating a user-defined function that will automate the tasks presented in the last sections.

The target is to have a function that has the following parameters:

- The data filename
- The Boolean parameter for headers.
- Indices to select the dependent and independent variables. Using such indices will absolve you from knowing the names of the variables.
- A Boolean flag for plotting the observed values and the regression fit.

The function can create the graph for the regression line and observed data, and return the linear regression object from function lm().

Here is the code for the function plot.lr1:

```
plot.lr1 = function(filename, header=TRUE, y.idx=2, x.idx=1, plot.data=TRUE)
{
  # read the data file
  lr1.data = read.table(filename, header = header)
  # attach the data frame
  attach(lr1.data, warn.conflicts=FALSE)
  # perform the linear regression
  sVars = names(lr1.data)
  sVarX = sVars[x.idx]
  sVarY = sVars[y.idx]
  lr1 = lm(paste(sVarY, "~", sVarX))
  if (plot.data) {
    lr.intercept = round(lr1$coefficients[1],2)
    lr.slope = round(lr1$coefficients[2], 2)
    if (lr.slope < 0) {
      sMain = paste(sVarY," = ", as.character(lr.intercept), " + (",
                            as.character(lr.slope), ") ", sVarX, sep="")
    } else {
      sMain = paste(sVarY, " =  ", as.character(lr.intercept), " + ",
                           as.character(lr.slope), " ", sVarX, sep="")
    }
    # plot the observations for x and y
    plot(lr1.data, type ="p", main = sMain)
    # draw the regression lines
    abline(lr1)
    # remove some of the auxiliary variables
    rm(lr.intercept,lr.slope,sMain)
  }
  # detach the data frame lr1.data
  detach(lr1.data)
  # return the regression statistics
  return (lr1)
}
```

The above function has the following parameters:

- The parameter **filename** which represents the source data filename
- The Boolean parameter **header** with a default value of TRUE.
- The parameters **y.idx** and **x.idx** are indices to select the dependent and independent variables, respectively.
- The parameters **show.plot** is a flag for plotting the observed values and the regression fit.

Notice that many statements in the function plot.lr1() are familiar. The function internally handles the names of the variables. You need not be concerned with these names. The variable sVars stores the names of the column variables obtained by the expression names(lr1.data). The variable sVarX stores the name of the independent variable accessed using the expression sVars[x.idx]. Likewise, the variable sVarY contains the name of the dependent variable accessed utilizing the expression sVars[y.idx]. The function then invokes the regression function lm() and supplies a string that specifies the regression model using the text in variables sVarX and sVarY.

As for plotting, the function plot.lr1() adds a title that shows the name of the variables and the regression coefficients (rounded to two decimals). To plot the observed point, the function plot.lr1() calls function plot() and passes the variable lr1.data as the source for the data. The function also draws the regression line using the abline() function. The argument for the latter function is the local variable lr1 which stores the linear regression object.

Save the above function in file C:\regdata\plot.lr1.r. You can load it using the following command:

```
> source("C:/regdata/plot.lr1.r")
```

Once the function is loaded in the workspace, you can use it. Here is an example. Type in the following command:

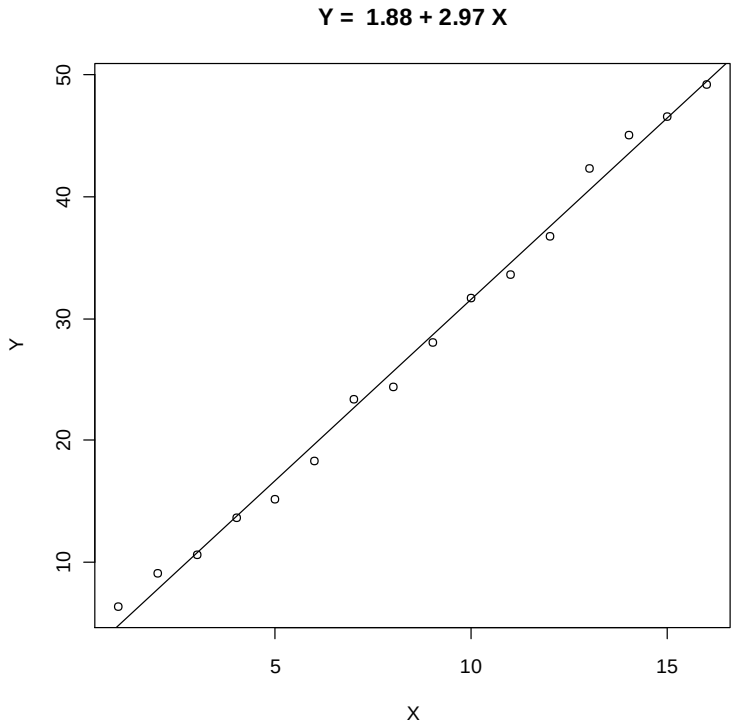
```
> (lr1.res=plot.lr1("C:/regdata/lr1.txt", TRUE, 2, 1))

Call:
lm(formula = paste(sVarY, "~", sVarX))

Coefficients:
(Intercept)            X
      1.884        2.971

> lr1.res

Call:
lm(formula = paste(sVarY, "~", sVarX))

Coefficients:
(Intercept)            X
      1.884        2.971
```

Next type the following command to get the regression summary:

```
> summary(lr1.res)

Call:
lm(formula = paste(sVarY, "~", sVarX))

Residuals:
    Min      1Q  Median      3Q     Max
-1.5617 -0.8317 -0.1344  0.8472  1.7764

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)  1.88425    0.59105   3.188  0.00658 **
X            2.97149    0.06112  48.614  < 2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 1.127 on 14 degrees of freedom
Multiple R-squared: 0.9941,     Adjusted R-squared: 0.9937
F-statistic:  2363 on 1 and 14 DF,  p-value: < 2.2e-16
```


The function also draws the graph shown in Figure 6.

Figure 6. Graph created with plot.lr1().



Simple Multiple Linear Regression

This section focuses on performing multiple linear regression between a dependent variable and several independent variables. These variables are also independent of each other and are not functions of one another. The model we try to fit in this section has the following general form:

$$Y = A_0 + A_1 X_1 + A_2 X_2 + \ldots + A_n X_n$$

The steps involved in performing multiple linear regression are very similar to those of the simple linear regression. The main focus of this section is how to formulate the model that appears in the call to function lm().

As with the linear regression case, we start with the source of data. Create the file C:\regdata\mlr1.txt and enter the following data:

```
Y     X1    X2    X3
10.6  1     45    1.55
16.18 2     55    2.45
14.09 3     67    3.41
```



```
27.26 4     65    7.67
29.57 5     41    6.65
46.03 6     29    9.97
38.94 7     19    5.55
32.22 8     34    3.35
29.57 9     35    1.29
33.25 10    55    4.65
(empty line)
```

Again, include a blank line, in the data file, after the last set of values.

Given the above data, open a new script editor window in R and type the following lines:

```
mlr1.data = read.table("C:/regdata/mlr1.txt", header = TRUE)
attach(mlr1.data, warn.conflicts=FALSE)
mlr1 = lm("Y ~ X1 + X2 + X3")
summary(mlr1)
confint(mlr1)
predict(mlr1, interval="c")
predict(mlr1, interval="p")
plot(Y, mlr1$residuals, type ="p", main = "Y^ - Y vs. Y")
abline(h=0)
detach(mlr1.data)
```

The above lines perform the following tasks (which by now should be familiar):

1. Read the data from file C:\regdata\mlr1.txt into the data frame mlr1.data.
2. Attach the data frame mlr1.data to allow direct access to the column variables.
3. Perform the multiple linear regression by calling function lm() and storing the result in variable mlr1.
4. Display the summary of the regression that is stored in variable mlr1.
5. Display the confidence interval for the regression coefficients at the default 95% confidence level.
6. Display the confidence intervals for the fitted data, using the default confidence level of 95%.
7. Display the confidence intervals for the observed data, using the default confidence level of 95%.
8. Plot the residuals vs. the observed values of Y.
9. Draw a horizontal line through Y=0.
10. Detach the data frame mlr1.data.

Select the above lines of code (using CTRL+A) and execute them by pressing CTRL+R. The R Console window shows the following summary:

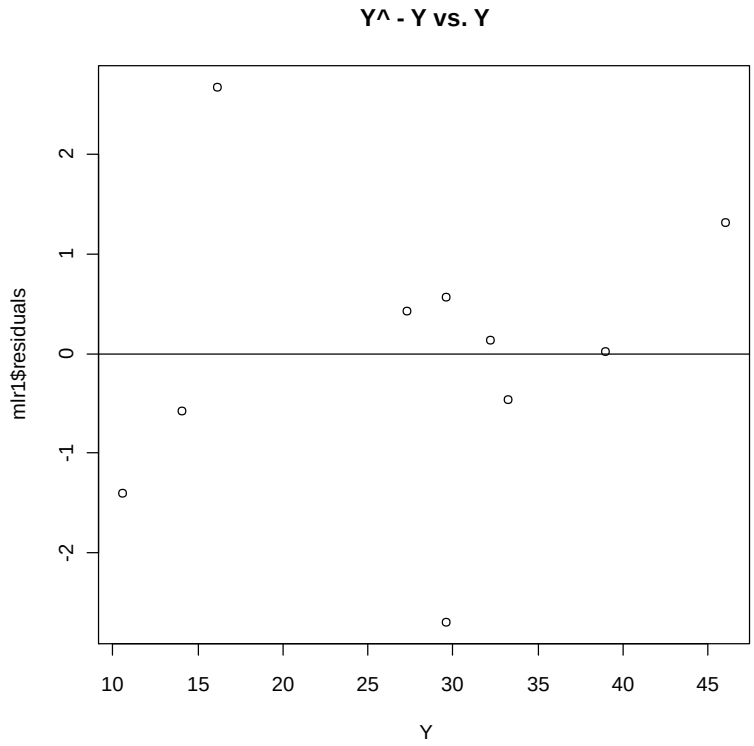
```
(output from function summary())
Call:
lm(formula = "Y ~ X1 + X2 + X3")

Residuals:
```



```
    Min      1Q  Median      3Q     Max
-2.6935 -0.5491  0.0792  0.5304  2.6714

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)  17.5683     2.8855    6.09  0.00089 ***
X1            1.8944     0.2173    8.72  0.00013 ***
X2           -0.2450     0.0418   -5.86  0.00109 **
X3            2.2958     0.2141   10.72  3.9e-05 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 1.79 on 6 degrees of freedom
Multiple R-squared: 0.983,      Adjusted R-squared: 0.975
F-statistic:  116 on 3 and 6 DF,  p-value: 1.05e-05

(output from function confint())
                2.5 %     97.5 %
(Intercept) 10.5077697 24.6289105
X1           1.3626353  2.4261486
X2          -0.3472518 -0.1426816
X3           1.7718171  2.8197262

(output from predict(mlr1, interval="c"))
        fit       lwr      upr
1  11.99768  8.873572 15.12178
2  13.50860 11.119289 15.89791
3  14.66733 12.129049 17.20561
4  26.83164 23.847565 29.81572
5  32.26355 30.466903 34.06019
6  44.71950 41.423671 48.01533
7  38.91625 36.170855 41.66164
8  32.08544 29.995870 34.17502
9  29.00558 26.066733 31.94443
10 33.71443 30.409772 37.01909

(output from predict(mlr1, interval="p"))
        fit       lwr      upr
1  11.99768  6.626270 17.36909
2  13.50860  8.528559 18.48864
3  14.66733  9.614127 19.72053
4  26.83164 21.540451 32.12283
5  32.26355 27.539152 36.98794
6  44.71950 39.246432 50.19257
7  38.91625 33.755903 44.07660
8  32.08544 27.242066 36.92882
9  29.00558 23.739765 34.27139
10 33.71443 28.236041 39.19282
Warning message:
In predict.lm(mlr1, interval = "p") :
  Predictions on current data refer to _future_ responses
```

Figure 7 shows the plot for the residuals vs. the values of variable Y.

Figure 7. The residuals plot.



Let's focus on the call to function lm(). This call, which performs multiple regression, uses the string "Y ~ X1 + X2 + X3" as the regression model. The tilde character separates the dependent variable from the independent ones. Since the function lm() performs a straightforward multiple regression with no transformations, the right hand side of the tilde character consists of "X1 + X2 + X3" which tells the function lm() to use the untransformed data for the variables X1, X2, X3. The lm() function calculates the regression coefficients and other statistics for the following model:

$$Y = A_0 + A_1 X_1 + A_2 X_2 + A_3 X_3$$

Linearized Regression between Two Variables

Let's revert to the case of regression between two variables. The linear model does not hold true in many cases. The reasons may be empirical or based on derived equations that represent the relationship between the dependent variable Y and the dependent variable X.

Let's consider the case where the model used for fitting data is:

$$Y = a X^b$$

We can linearize the above model by taking the logarithm of both sides of the equation:

$$\ln(Y) = \ln(a) + b \ln(X)$$

To illustrate regression with the above model, enter the following data in a text file and save it as file C:\regdata\lr2.txt

```
X     Y
1     2.16
2     10.79
3     29.27
4     49.3
5     76.14
6     110.6
7     148.08
8     192.91
9     241.01
10    301.88
11    365.25
12    431.65
13    506.06
14    588.81
15    675.04
16    765.71
(empty line)
```

Given the above data, open a new script editor window in R and type the following lines:

```
lr2.data = read.table("C:/regdata/lr2.txt", header = TRUE)
attach(lr2.data, warn.conflicts=FALSE)
lr2 = lm("log(Y) ~ log(X)")
summary(lr2)
confint(lr2)
predict(lr2, interval="c")
predict(lr2, interval="p")
plot(Y, lr2$residuals, type ="p", main = "Y^ - Y vs. Y")
abline(h=0)
detach(lr2.data)
```

The above lines perform the following tasks:

1. Read the data from file C:\regdata\lr2.txt into the data frame lr2.data.
2. Attach the data frame lr2.data to allow direct access to the column variables.
3. Perform the linearized regression by calling function lm() and storing the result in variable lr2. The call to function lm() has the argument "log(Y) ~ log(X)". This model tells the function lm() to perform a linear regression between ln(Y) and ln(X).
4. Display the summary of the regression that is stored in variable lr2.
5. Display the confidence interval for the regression coefficients at the default 95% confidence level.

6. Display the confidence intervals for the fitted data, using the default confidence level of 95%.
7. Display the confidence intervals for the transformed values of the observed data, using the default confidence level of 95%.
8. Display the confidence intervals for the fitted data where the data are subjected to an inverse transformation—using the exp() function.
9. Display the confidence intervals for the observed data where the data are subjected to an inverse transformation—using the exp() function.
10. Plot the residuals vs. the observed values of Y. The expression lr2$residuals is the vector containing the residuals for Y.
11. Draw a horizontal line through Y=0.
12. Detach the data frame lr2.data.

Select the above lines of code (using CTRL+A) and execute them by pressing CTRL+R. The R Console window shows the following summary:

```
(output from function summary())
Call:
lm(formula = "log(Y) ~ log(X)")

Residuals:
     Min       1Q   Median       3Q      Max
-0.17316 -0.03177 -0.00769  0.02941  0.15695

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)   0.9433     0.0492    19.2  1.9e-11 ***
log(X)        2.0720     0.0238    87.0  < 2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.0728 on 14 degrees of freedom
Multiple R-squared: 0.998,      Adjusted R-squared: 0.998
F-statistic: 7.56e+03 on 1 and 14 DF,  p-value: <2e-16

(output from function confint())
               2.5 %   97.5 %
(Intercept) 0.8378118 1.048715
log(X)      2.0209262 2.123125

(output from predict(lr2, interval="c"))
         fit       lwr      upr
1  0.9432633 0.8378118 1.048715
2  2.3794818 2.3057564 2.453207
3  3.2196158 3.1624031 3.276828
4  3.8157004 3.7681620 3.863239
5  4.2780595 4.2359704 4.320149
6  4.6558344 4.6162685 4.695400
7  4.9752385 4.9361656 5.014311
8  5.2519189 5.2120013 5.291836
9  5.4959684 5.4543804 5.537556
```



```
10 5.7142780 5.6705432 5.758013
11 5.9117631 5.8656288 5.957897
12 6.0920529 6.0434045 6.140701
13 6.2579034 6.2067098 6.309097
14 6.4114570 6.3577365 6.465178
15 6.5544120 6.4982117 6.610612
16 6.6881375 6.6295203 6.746755

(output from predict(lr2, interval="p"))
         fit       lwr      upr
1  0.9432633 0.7548168 1.131710
2  2.3794818 2.2067754 2.552188
3  3.2196158 3.0532867 3.385945
4  3.8157004 3.6524461 3.978955
5  4.2780595 4.1163079 4.439811
6  4.6558344 4.4947210 4.816948
7  4.9752385 4.8142454 5.136232
8  5.2519189 5.0907188 5.413119
9  5.4959684 5.3343465 5.657590
10 5.7142780 5.5520905 5.876466
11 5.9117631 5.7489121 6.074614
12 6.0920529 5.9284719 6.255634
13 6.2579034 6.0935476 6.422259
14 6.4114570 6.2462966 6.576617
15 6.5544120 6.3884284 6.720396
16 6.6881375 6.5213201 6.854955
Warning message:
In predict.lm(lr2, interval = "p") :
  Predictions on current data refer to _future_ responses

(output from exp(predict(lr2, interval="c")))
          fit        lwr        upr
1    2.568349   2.311304   2.853981
2   10.799305  10.031764  11.625573
3   25.018506  23.627307  26.491621
4   45.408548  43.300404  47.619330
5   72.100391  69.128730  75.199796
6  105.196956 101.116016 109.442597
7  144.783349 139.235344 150.552421
8  190.932300 183.460860 198.708015
9  243.707407 233.779977 254.056405
10 303.165245 290.192127 316.718329
11 369.356806 352.703869 386.796013
12 442.328542 421.325009 464.379124
13 522.123127 496.066423 549.548501
14 608.780048 576.938999 642.378394
15 702.336065 663.953199 742.937830
16 802.825571 757.118902 851.291515

(output from exp(predict(lr2, interval="p")))
          fit        lwr        upr
1    2.568349   2.127222   3.100954
2   10.799305   9.086369  12.835159
3   25.018506  21.184859  29.545897
4   45.408548  38.568892  53.461121
5   72.100391  61.332381  84.758921
```



```
6  105.196956  89.543180 123.587295
7  144.783349 123.253775 170.073640
8  190.932300 162.506627 224.330194
9  243.707407 207.337207 286.457511
10 303.165245 257.775864 356.546824
11 369.356806 313.849050 434.681737
12 442.328542 375.580165 520.939487
13 522.123127 442.990159 615.391907
14 608.780048 516.097976 718.106183
15 702.336065 594.920886 829.145454
16 802.825571 679.474755 948.569308
Warning message:
In predict.lm(lr2, interval = "p") :
  Predictions on current data refer to _future_ responses
```

The regression coefficients are:

Intercept = ln(a) = 0.9433

Slope = b = 2.0720

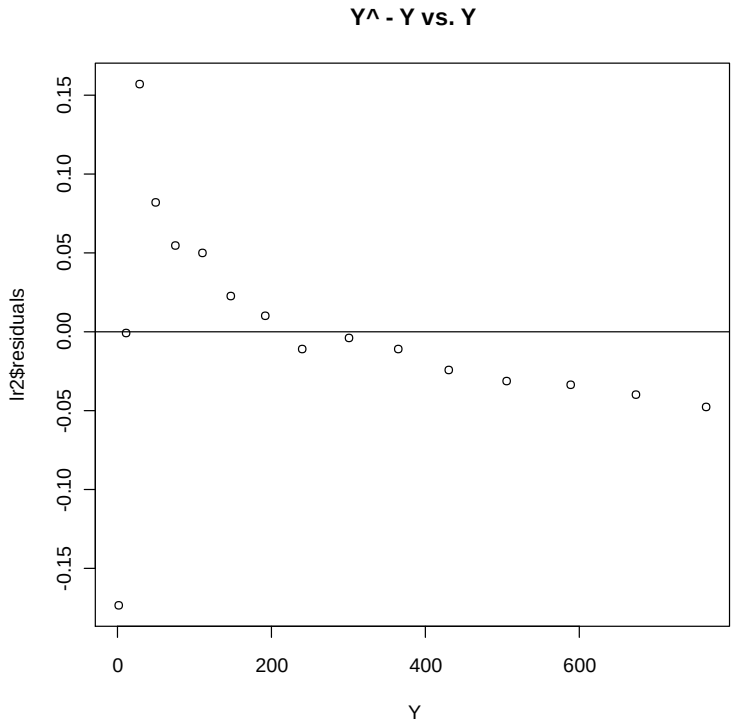
The value of a = exp(0.9433) = 2.56844. The original model used to create the same data is:

Y = 3 X^2

Adding noise to the original model yields the values returned by the function lm().

Figure 8 shows the plot for the residuals vs. the values of variable Y.

Figure 8. The residual plot for a power fit.



Linearized Regression between Multiple Variables

Let's extend on what we learned in the last section. In this section we perform a linearized multiple regression between three independent variables and a dependent variable. The original model used to create the sample data is:

$$Y = a\ X_1^{b_1}\ X_2^{b_2}\ X_3^{b_3}$$

Applying the natural logarithm on both sides of the above equation yields the following linearized model:

$$\ln(Y) = \ln(a) + b_1 \ln(X_1) + b_2 \ln(X_2) + b_3 \ln(X_3)$$

To illustrate regression with the above model, enter the following data in a text file and save it as file C:\regdata\mlr2.txt

```
Y     X1    X2    X3
16.3  1     45    1.55
188.49      2     55    2.45
934.6 3     67    3.41
```



```
12685.28    4     65    7.67
14527.1     5     41    6.65
59624.12    6     29    9.97
19569.11    7     19    5.55
6827.29     8     34    3.35
790.23      9     35    1.29
23076.49    10    55    4.65
(empty line)
```

Given the above data, open a new script editor window in R and type the following lines:

```
mlr2.data = read.table("C:/regdata/mlr2.txt", header = TRUE)
attach(mlr2.data, warn.conflicts=FALSE)
mlr2 = lm("log(Y) ~ log(X1) + log(X2) + log(X3)")
summary(mlr2)
confint(mlr2)
plot(Y, mlr2$residuals, type ="p", main = "Y^ - Y vs. Y")
abline(h=0)
detach(mlr2.data)
```

The above lines perform the following tasks:

1. Read the data from file C:\regdata\mlr2.txt into the data frame mlr2.data.
2. Attach the data frame mlr2.data to allow direct access to the column variables.
3. Perform the linearized regression by calling function lm() and storing the result in variable mlr2. The call to function lm() has the argument "log(Y) ~ log(X1) + log(X2) + log(X3)". This model tells the function lm() to perform a linear regression between ln(Y) as the dependent variable and ln(X1), ln(X2), and ln(X3) as the dependent variables.
4. Display the summary of the regression that is stored in variable mlr2.
5. Display the confidence interval for the regression coefficients at the default 95% confidence level.
6. Display the confidence intervals for the fitted data, using the default confidence level of 95%.
7. Display the confidence intervals for the transformed values of the observed data, using the default confidence level of 95%.
8. Display the confidence intervals for the fitted data where the data are subjected to an inverse transformation—using the exp() function.
9. Display the confidence intervals for the observed data where the data are subjected to an inverse transformation—using the exp() function.
10. Plot the residuals vs. the observed values of Y. The expression lmr2$residuals is the vector containing the residuals for Y.
11. Draw a horizontal line through Y=0.
12. Detach the data frame mlr2.data.

Select the above lines of code (using CTRL+A) and execute them by pressing CTRL+R. The R Console window shows the following summary:


```
(output from function summary())
Call:
lm(formula = "log(Y) ~ log(X1) + log(X2) + log(X3)")

Residuals:
      Min        1Q    Median        3Q       Max
-0.016544 -0.006045  0.000097  0.002423  0.020331

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)  2.12193    0.05885   36.06  3.0e-08 ***
log(X1)      1.97298    0.00801  246.29  3.0e-13 ***
log(X2)     -0.11634    0.01412   -8.24  0.00017 ***
log(X3)      2.49115    0.00801  310.83  7.5e-14 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.0153 on 6 degrees of freedom
Multiple R-squared:    1,       Adjusted R-squared:    1
F-statistic: 8.53e+04 on 3 and 6 DF,  p-value: 2.82e-14

(output from function confint())
                2.5 %     97.5 %
(Intercept)  1.9779269  2.2659249
log(X1)      1.9533794  1.9925824
log(X2)     -0.1508766 -0.0817956
log(X3)      2.4715438  2.5107656

(output predict(mlr2, interval="c")
         fit       lwr       upr
1   2.770835  2.739997  2.801672
2   5.255588  5.236606  5.274571
3   6.856240  6.837133  6.875346
4   9.446697  9.423219  9.470176
5   9.585077  9.569456  9.600699
6  10.993906 10.970260 11.017552
7   9.887961  9.859921  9.916000
8   8.826089  8.809345  8.842832
9   6.677746  6.647131  6.708360
10 10.027259 10.003506 10.051012

(output predict(mlr2, interval="p")
         fit       lwr       upr
1   2.770835  2.722351  2.819318
2   5.255588  5.213636  5.297541
3   6.856240  6.814230  6.898249
4   9.446697  9.402528  9.490867
5   9.585077  9.544534  9.625620
6  10.993906 10.949647 11.038165
7   9.887961  9.841207  9.934714
8   8.826089  8.785100  8.867077
9   6.677746  6.629404  6.726088
10 10.027259  9.982943 10.071575
Warning message:
In predict.lm(mlr2, interval = "p") :
  Predictions on current data refer to _future_ responses
```



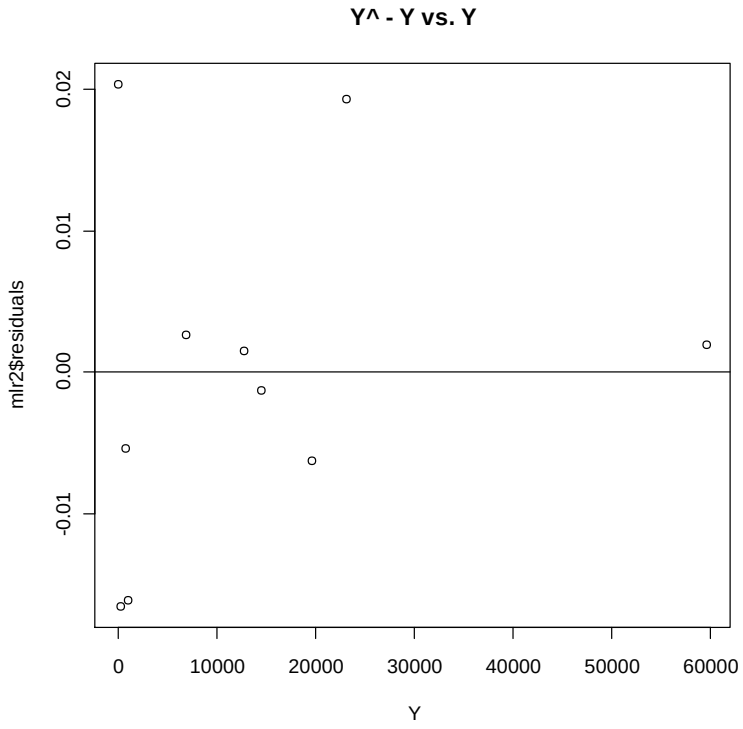
```
(output exp(predict(mlr2, interval="c"))
           fit         lwr         upr
1     15.97196    15.48694    16.47217
2    191.63423   188.03088   195.30662
3    949.78873   931.81355   968.11066
4  12666.26193 12372.34108 12967.16524
5  14546.08853 14320.61756 14775.10943
6  59510.38309 58119.69866 60934.34373
7  19691.86135 19147.38270 20251.82290
8   6809.59944  6696.53430  6924.57359
9    794.52610   770.57054   819.22640
10 22635.13941 22103.82419 23179.22598

(output exp(predict(mlr2, interval="p")))
           fit         lwr         upr
1     15.97196    15.21605    16.76541
2    191.63423   183.76098   199.84480
3    949.78873   910.71542   990.53844
4  12666.26193 12118.97662 13238.26230
5  14546.08853 13968.14125 15147.94902
6  59510.38309 56933.96453 62203.39168
7  19691.86135 18792.38613 20634.38888
8   6809.59944  6536.12938  7094.51142
9    794.52610   757.03071   833.87862
10 22635.13941 21653.94570 23660.79343
Warning message:
In predict.lm(mlr2, interval = "p") :
  Predictions on current data refer to _future_ responses
```

The following table summarizes the regression coefficients and compares them with the values in the original model used, BEFORE noise was added to the sample data:

Coefficient	*From Regression*	*In Original Model*
ln(a)	2.12193	2
b1	2.0720	2
b2	-0.11634	-0.1
b3	2.49115	2.5

Figure 9 shows the plot for the residuals vs. the values of variable Y.

Figure 9. Residual for linearized multiple regression.



Multiple Regression between Transformed Variables

Many regression models derived from equations can be linearized AND also end up with multiple terms that are based on the same dependent variable. Each term has a different mathematical transformation. This section, considers such as a case. The original model used to create the sample data is:

$$Y = a + bX + c\,X^2 + d\,\sqrt{X}$$

To illustrate regression with the above model, enter the following data in a text file and save it as file C:\regdata\mlr3.txt

```
Y     X
4.18  1
2.16  2
-0.63 3
-1.73 4
-3.82 5
-4.79 6
-5.85 7
```



```
-9.84 8
-7.39 9
-7.72 10
(empty line)
```

Given the above data, open a new script editor window in R and type the following lines:

```
mlr3.data = read.table("C:/regdata/mlr3.txt", header = TRUE)
attach(mlr3.data, warn.conflicts=FALSE)
X1 = X
X2 = X^2
X3 = sqrt(X)
mlr3 = lm("Y ~ X1 + X2 + X3")
rm(X1, X2, X3)
summary(mlr3)
confint(mlr3)
plot(Y, mlr3$residuals, type ="p", main = "Y^ - Y vs. Y")
abline(h=0)
detach(mlr3.data)
```

The above lines perform the following tasks:

1. Read the data from file C:\regdata\mlr3.txt into the data frame mlr3.data.
2. Attach the data frame mlr3.data to allow direct access to the column variables.
3. Create the vectors X1, X2, and X3 by using the statements X1 = X, X2 = X^2, and X3 = sqrt(X) to transform X into regression variables.
4. Perform the linearized regression by calling function lm() and storing the result in variable mlr3. The call to function lm() has the argument "Y ~ X1 + X2 + X3". This model tells the function lm() to perform a linear regression between Y as the dependent variable and X1, X2, and X3 as the independent variables.
5. Remove the temporary vectors X1, X2, and X3.
6. Display the summary of the regression that is stored in variable mlr3.
7. Display the confidence interval for the regression coefficients at the default 95% confidence level.
8. Display the confidence intervals for the fitted data, using the default confidence level of 95%.
9. Display the confidence intervals for the transformed values of the observed data, using the default confidence level of 95%.
10. Plot the residuals vs. the observed values of Y. The expression lmr3$residuals is the vector containing the residuals for Y.
11. Draw a horizontal line through Y=0.
12. Detach the data frame mlr3.data.

Select the above lines of code by pressing the keys CTRL+A, and execute them by pressing the keys CTRL+R. The R Console window shows the following summary:

```
(output from function summary())
```



```
Call:
lm(formula = "Y ~ X1 + X2 + X3")

Residuals:
     Min       1Q   Median       3Q      Max
-2.29882 -0.00174  0.21650  0.52999  0.84884

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)    1.571      8.879    0.18     0.87
X1            -5.998      4.820   -1.24     0.26
X2             0.243      0.173    1.40     0.21
X3             8.235     12.778    0.64     0.54

Residual standard error: 1.12 on 6 degrees of freedom
Multiple R-squared: 0.959,      Adjusted R-squared: 0.938
F-statistic: 46.7 on 3 and 6 DF,  p-value: 0.000149

(output from function confint())
                 2.5 %    97.5 %
(Intercept) -20.155904 23.297849
X1          -17.791172  5.795151
X2           -0.180807  0.667642
X3          -23.031339 39.502128

(output predict(mlr3, interval="c"))
           fit         lwr        upr
1   4.05177375   1.4118129  6.6917346
2   2.19522761   0.4785605  3.9118948
3   0.03181942  -1.6340260  1.6976648
4  -2.05560162  -3.4513065 -0.6598967
5  -3.91874210  -5.2149375 -2.6225466
6  -5.48154812  -6.8853043 -4.0777920
7  -6.69884018  -8.1639286 -5.2337517
8  -7.54118135  -8.9182679 -6.1640948
9  -7.98812465  -9.4412406 -6.5350087
10 -8.02478276 -10.3730077 -5.6765578

(output predict(mlr3, interval="p"))
           fit         lwr        upr
1   4.05177375   0.2492694  7.8542781
2   2.19522761  -1.0353439  5.4257991
3   0.03181942  -3.1720356  3.2356744
4  -2.05560162  -5.1276756  1.0164723
5  -3.91874210  -6.9469047 -0.8905795
6  -5.48154812  -8.5572883 -2.4058080
7  -6.69884018  -9.8030519 -3.5946285
8  -7.54118135 -10.6048415 -4.4775212
9  -7.98812465 -11.0867037 -4.8895456
10 -8.02478276 -11.6308619 -4.4187036
Warning message:
In predict.lm(mlr3, interval = "p") :
  Predictions on current data refer to _future_ responses
```

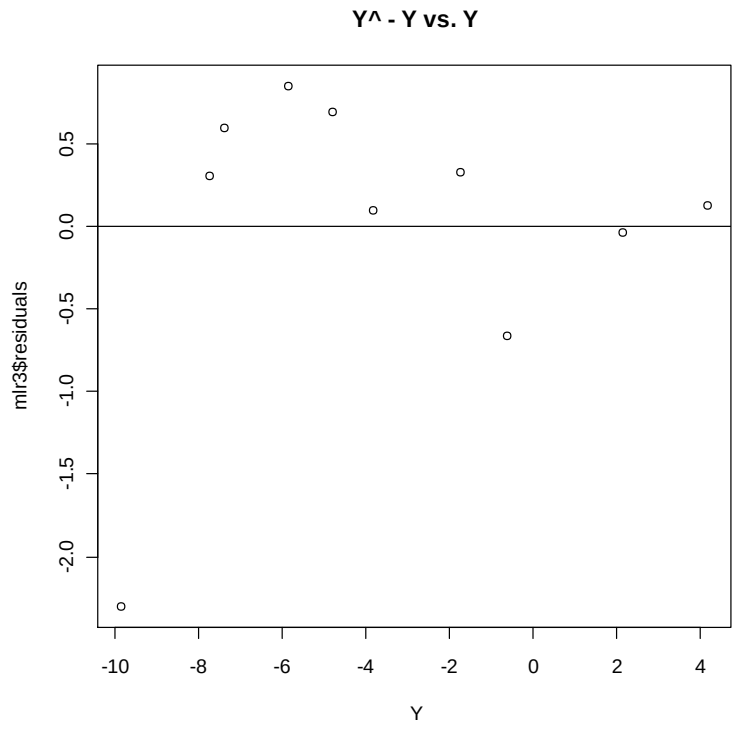

The following table summarizes the regression coefficients and compares them with the values in the original model used, BEFORE noise was added to the sample data:

Coefficient	*From Regression*	*In Original Model*
a	1.571	10
b	-5.998	-2
c	0.243	-0.1
d	8.235	-3

The difference between the model and regression coefficients is the extent of noise added to the model. The above results are also confirmed with Excel's Data Analysis regression routine.

Figure 10 shows the plot for the residuals vs. the values of variable Y.

Figure 10. Residuals for multiple regression between transformed variables.



The last set of code used temporary vectors to create pseudo-variables. The function lm() allows you to use a special syntax to declare transformations without needing to use temporary variables. The function allows you to place ach transformation of a variable in I(). So the formula string "Y ~ X + I(X^2) + I(sqrt(X))" tells the function lm() how to perform a linear regression between Y and with X, X^2, and sqrt(X).

You can replace the above lines that appeared in the last code segment:

```
X1 = X
X2 = X^2
X3 = sqrt(X)
mlr3 = lm("Y ~ X1 + X2 + X3")
rm(X1, X2, X3)
```

With the following line:

```
mlr3 = lm("Y ~ X + I(X^2) + I(sqrt(X))")
```

The output of the summary(ml3) function call is:

```
Call:
lm(formula = "Y ~ X + I(X^2) + I(sqrt(X))")

Residuals:
     Min       1Q   Median       3Q      Max
-2.29882 -0.00174  0.21650  0.52999  0.84884

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)    1.571      8.879    0.18     0.87
X             -5.998      4.820   -1.24     0.26
I(X^2)         0.243      0.173    1.40     0.21
I(sqrt(X))     8.235     12.778    0.64     0.54

Residual standard error: 1.12 on 6 degrees of freedom
Multiple R-squared: 0.959,      Adjusted R-squared: 0.938
F-statistic: 46.7 on 3 and 6 DF,  p-value: 0.000149
```

Regression Transformation Recap

The examples in the last sections illustrated how to transform variables when calling function lm(). The following table summarizes what you have learned and also introduces, what I consider, some quirky syntax for declaring certain kinds of models.

Example for Formula Syntax	*Regression Model*	*Explanation*
Y ~ X	$Y = A + B X$	Simple linear regression.
Y ~ X -1	$Y = B X$	Exclude the intercept.
Y ~ log(X)	$Y = A + B \ln(X)$	Linearized regression.
T ~ X + Y + Z	$T = A + B X + C Y + D Z$	Multiple linear regression.
T ~ X + Y + Z – 1	$T = B X + C Y + D Z$	Multiple linear regression with no intercept.
X1 = f_1(x) X2 = f_2(x) Y ~ X1 + X2	$Y = A + B f_1(X) + C f_2(X)$	Linearized multiple regression
Y ~ X + I(log(X)) + I(1/X)	$Y = A + B \ln(X) + C/X$	Linearized multiple regression
Y ~ X + I(X^2) + I(X^3)	$Y = A + B X + C X^2 + D X^3$	Polynomial regression.

Example for Formula Syntax	*Regression Model*	*Explanation*
Z ~ X:Y	Z = A + B (X*Y)	Linearized regression model with simple product of two variables.
Z ~ X*Y	Z = A + B X + C Y + D (X*Y)	Linearized multiple regression model with linear terms and simple product of two variables.
T ~ (X + Y + Z)^2	T = A + B X + C Y + D Z + E (X*Y) + F (X*Z) + G (Y*Z)	Linearized multiple regression model, with linear terms and multiple cross products of two variables.
T ~ (X + Y + Z)^3	T = A + B X + C Y + D Z + E (X*Y) + F (X*Z) + G (Y*Z) + H (X*Y*Z)	Linearized multiple regression model, with linear terms and multiple cross products for two and three variables.

The last two entries in the above table deserve additional discussion. The syntax for these entries can be misleading. Consider the general form:

$$Y \sim (X_1 + X_2 + \ \ldots + X_m)\text{^}n$$

The above model tells the lm() function to perform a multiple regression such that:

- There are m linear terms--one for each variable X_i.
- The number of cross product terms include those for the combinations of 2, 3, …, and up to n variables (or up to m variables, if m > n).

The above general form works more like if it was written as:

$$Y \sim (X_1 + X_2 + \ \ldots + X_m)\text{^}(\text{if}(m < n, m, n))$$

Here is a short example that illustrates the syntax of the last entry in the above table. Type in the following commands:

```
> x=runif(10,1, 10)
> y=runif(10,11, 100)
> z=runif(10,0.1, 1.0)
> t=runif(10,1, 100)
> tr=lm(t~(x+y+z)^3)
> summary(tr)

Call:
lm(formula = t ~ (x + y + z)^3)

Residuals:
      1       2       3       4       5       6       7       8       9
10
-13.871   4.717  -1.096   5.405  -4.127  -1.893 -10.893  -7.927  28.461
1.223
```



```
Coefficients:
             Estimate Std. Error t value Pr(>|t|)
(Intercept)  -32.6710   312.4774  -0.105    0.926
x             19.6196    41.4649   0.473    0.683
y              2.0938     4.1837   0.500    0.666
z            399.1833   943.2286   0.423    0.713
x:y           -0.4133     0.5932  -0.697    0.558
x:z         -109.0101   166.0086  -0.657    0.579
y:z           -7.9417    13.2759  -0.598    0.610
x:y:z          2.0120     2.4773   0.812    0.502

Residual standard error: 25.09 on 2 degrees of freedom
Multiple R-squared: 0.6571,     Adjusted R-squared: -0.5432
F-statistic: 0.5474 on 7 and 2 DF,  p-value: 0.77
```

The highlight of the above results is not the numerical results, but rather the number of terms and their makeup. The model has simple terms for the variables x, y, and z, cross product terms for paired variables x:y, x:z, and y:z, along with a single term that is for the cross product for all three variables.

If you replace the following commands:

```
> tr=lm(t~(x+y+z)^3)
> summary(tr)
```

With the next ones:

```
> tr=lm(t~(x+y+z)^4)
> summary(tr)
```

You still get the regression for the same number of terms and the same combinations of variables, because the power 4 is greater than 3, the number of independent variables.

Finally, here is a table that summarizes the special characters used in defining the regression model. The model combination directive (the ^ character) might explain why regression terms that raise a variable to a power need to use the I() syntax in order to prevent the R interpreter from distinguishing between a term with a variable raised to a power and combined terms.

Character	***Directive***
~	Described by
+	Additive term
:	Interaction
*	Additive + interaction
-1	Exclude the intercept
^	Combine additive and interaction to a specified level of variable combinations

Nonlinear Regression

Many models that are nonlinear can be mathematically transformed into linear models. You can take reciprocals, apply logarithms, use the square roots, and so on, to turn a nonlinear model into one that can work with linear regression. There are cases where such transformations are not possible. Obtaining the regression coefficients become a process of numerical optimization. You can apply techniques for constrained or unconstrained optimization to reduce the sum of the squared errors—based on the differences between the observed values of the dependent variable and the values calculated using the model. As the optimization algorithm progresses, the regression coefficients are fine tuned, leading to smaller and smaller values for the sum of squared errors. The success of this process depends on the quality of the data, the mathematical equation that represents the model, the initial guesses, and the actual optimization algorithm used. Data with a lot of noise or significant error typically yield poor results. Certain models are harder to optimize than others. Initial guesses that are far from the final answers, or whose values destabilize the calculation, will also cause problems. Finally, the choice of optimization algorithm and its actual implementation can produce results that range from very good to very poor. So the process of performing nonlinear regression is not always an easy one.

The next sections discuss the functions used to perform nonlinear regression.

Simple Nonlinear Regression: Crescent Curve

The function nls() performs nonlinear least squares regression. The function requires the following parameters:

- The model for the nonlinear fit
- The data frame which provides the source for data
- A list of values that specifies the starting guesses for the nonlinear regression coefficients.

The following example illustrates how function nls() woks. Consider the following model for a crescent curve:

$$Y = A_0 (1 - \exp(-k X))$$

Assume that we already have a data frame, call it df, that has data for the variables X and Y. Also assume that you will use the initial guesses for A0 and k as 100 and 01., respectively. A call to function nls() looks as follows:

```
nls("Y ~ A0 * (1 - exp(-k X))", data=df, start=list(A=100, k=0.1))
```


It's that simple! The first argument for function nls() is the model for the nonlinear regression. This model looks very close (except the tilde replaces the equal sign) to the equation for the crescent curve. The regression model lists the variables X and Y, in addition to the regression coefficients A0 and k. How does the function nls() tell which is which? It parses the model and comes up with the identifiers X, Y, A0, and k. The function looks in the names of the column variables in the data frame and determines that X and Y (but not A0 and k) match column names in the data frame. The function concludes that X and Y are regression variables and the source of data. The function has yet to resolve what identifiers A0 and k are. The function looks in the start parameter and sees a list that declares elements with the names A0 and k. These names happen to match the names of the regression coefficients in the regression model. Now that the function knows the role of each of the identifiers X, Y, A0, and k, it goes to work and performs the nonlinear regression. The process is iterative and its success depends on the regression model, the data (and how much noise is in the data), and the initial guesses for the regression coefficients. The rule of thumb is—good data and good guesses yield good results.

To illustrate working with function nls() and with the above model, enter the following data in a text file and save it as file C:\regdata\nlr1.txt:

```
X     Y
1     206.75
2     378.39
3     514.75
4     629.71
5     724.99
6     802.46
7     863.95
8     915.71
9     958.84
10    993.89
11    1026.96
12    1048.76
13    1065.94
14    1083.56
15    1092.68
(empty line)
```

Given the above data, open a new script editor window in R and type the following lines:

```
demo.nlr1 = function(par.A0=1000, par.k=0.1)
{
  # test crecent curve
  # read data
  nlr1.data = read.table("C:/regdata/nlr1.txt", header = TRUE)
  # attach data frame
  attach(nlr1.data, warn.conflicts=FALSE)
  # perform the nonlinear regression
  nlr1 = nls("Y ~ A0*(1-exp(-k * X))", data=nlr1.data,
         start = list(A0=par.A0, k=par.k))
  # store the number of observations
```



```
  ndata = length(T)
  # get the array of regression coefficients
  c=coef(nlr1)
  # store regression coefficient in scalar variables (easier to
  # read in equations below)
  A0=c[1]
  k=c[2]
  # store values for actual model coefficients in scalar variables
  model.A0 = 1150
  model.k = 0.2
  # plot observed points
  plot(X, Y, type ="p", main = "Y vs. X")
  # add two curves: one for the actual model and one for the fitted model
  curve(model.A0*(1-exp(-model.k*x)), from=X[1], to=X[ndata],
        n=501, add=TRUE)
  curve(A0*(1-exp(-k*x)), from=X[1], to=X[ndata], n=501, add=TRUE, col="red")
  # display the regression summary
  show(summary(nlr1))
  # display confidence interval
  show(confint(nlr1))
  # display AIC and Log Likelihood statistics
  cat("\nAIC = ", AIC(nlr1), "\n", "Log Likelihood = ", logLik(nlr1),
      "\n", sep="")
  detach(nlr1.data)
  return ("End of demo")
}
```

The function has the following parameters:

- The parameter par.A0 which passes an argument to the regression coefficient A0. This parameter has a default value of 1000.
- The parameter par.k which passes an argument to the regression coefficient k. This parameter has a default value of 0.1

The function performs the following steps:

1. Reads the data in the file C:\regdata\nlr1.txt. This step stores the resulting data frame in variable nlr1.data.
2. Attaches the data frame nlr1.data to give access to the column variables in that data frame.
3. Performs the nonlinear regression by calling function nls(). The call is nlr1 = nls("Y ~ A0*(1-exp(-k * X))", data=nlr1.data, start = list(A0=par.A0, k=par.k)).
4. Stores the number of observations in variable ndata.
5. Stores the array of regression coefficients in vector c. This step calls function coef(nls.object) to obtain the coefficients.
6. Stores each of the regression coefficients in distinct scalar variables—A0 and k.
7. Stores the model's regression coefficients in distinct scalar variables—model.A0 and model.k.
8. Plots the X and Y observations by calling function plot().

9. Plots the curve for the original model. This step calls function curve() and uses the values in variables model.A0 and model.k.
10. Plots the curve for the regression model. This step calls function curve() and uses the values in variables A0 and k. The call to function curve() also specifies the color red for the curve drawn.
11. Displays the regression summary. This step calls function show() and supplies the argument summary(nlr1).
12. Displays the confidence interval for the regression coefficients, at the default 95% confidence level. This step calls function show() and supplies the argument confint(nlr1).
13. Displays the AIC and log likelihood statistics. This step calls function AIC(nls.object) and logLik(nls.object) to obtain these statistics.
14. Detaches the data frame nlr1.data.
15. Returns the message *End of demo* to let you know that the function has done its work.

Save the above code in file C:\regdata\demo.nlr1.r, and then load it by using the source function:

```
> source("C:/regdata/demo.nlr1.r")
```

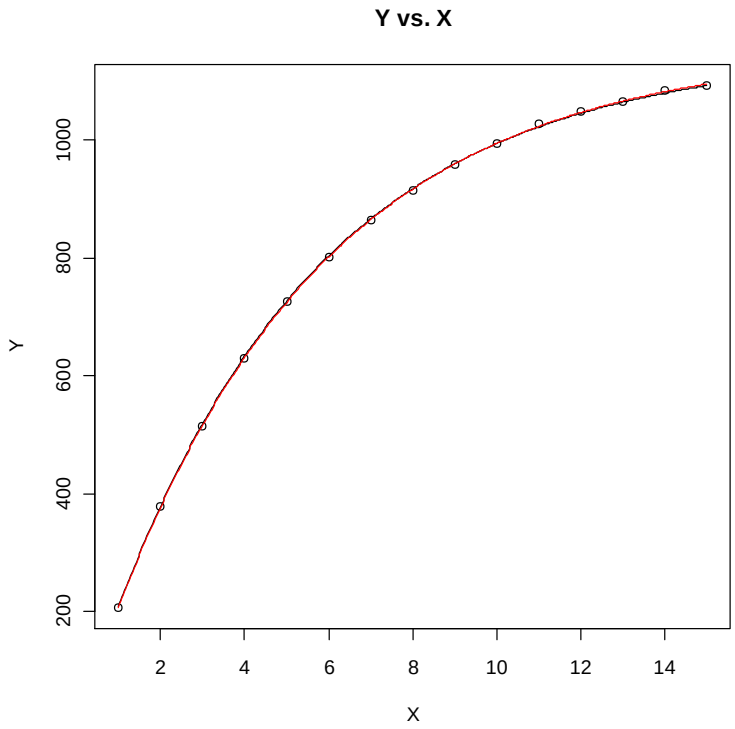
Now execute the following command to use function demo.nlr1() and perform a nonlinear regression for the crescent curve model:

```
> demo.nlr1()

Formula: Y ~ A0 * (1 - exp(-k * X))

Parameters:
    Estimate Std. Error t value Pr(>|t|)
A0 1.155e+03  1.315e+00   878.5   <2e-16 ***
k  1.972e-01  5.938e-04   332.1   <2e-16 ***
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 1.633 on 13 degrees of freedom

Number of iterations to convergence: 5
Achieved convergence tolerance: 2.634e-08

Waiting for profiling to be done...
           2.5%        97.5%
A0 1152.6523208 1158.3346366
k     0.1959321    0.1984974

AIC = 61.13389
Log Likelihood = -27.56695
[1] "End of demo"
```

The above output shows that the default parameters of 1000 and 0.1, for par.A0 and par.k are very good initial values. The result of A0 = 1155 and k = 0.1927 are close to the crescent model's values of A0 = 1155 and k =0.2.

The function demo.nlr1() draws the graph that appears in Figure 11. Notice that the black and red curves are very close to each other. This indicates that the results are very good. You are welcome to experiment with the function demo.nlr1() by supplying various arguments for the initial guesses. This approach lets you see how far you can push the initial guesses before the function nls() fails.

Figure 11. Crescent shape model.



Nonlinear Regression: Chain Reaction Model

The crescent curve model is a simple example for models that are intrinsically nonlinear. A slightly more complicated model is one that is based on a simple chain reaction where chemical A yields chemical B, which in turn yields chemical C. The model we consider is the case when you have only chemical A at the beginning. You start the reaction and observe that chemical A yields chemical B, and then soon chemical C begins to appear. As the reaction progresses, the amount of chemical A decreases continuously. The amount of chemical B typically peaks and then dies out. All the while, the amount of chemical C is increasing continuously.

The above chain reaction is represented by:

$$A \rightarrow B \rightarrow C$$

Assuming that the following first-order rates of reaction apply, we have:

$$dA/dt = -k_1 A$$

$$dB/dt = k_1 A - k_2 B$$

$$dC/dt = k_2 B$$

Where k_1 and k_2 are the reaction rate constants. Assume that initially the concentration of chemical A, B, and C are A_0, 0, and 0, respectively. The above differential equations yield the following equations for the concentrations of chemicals A, B, and C:

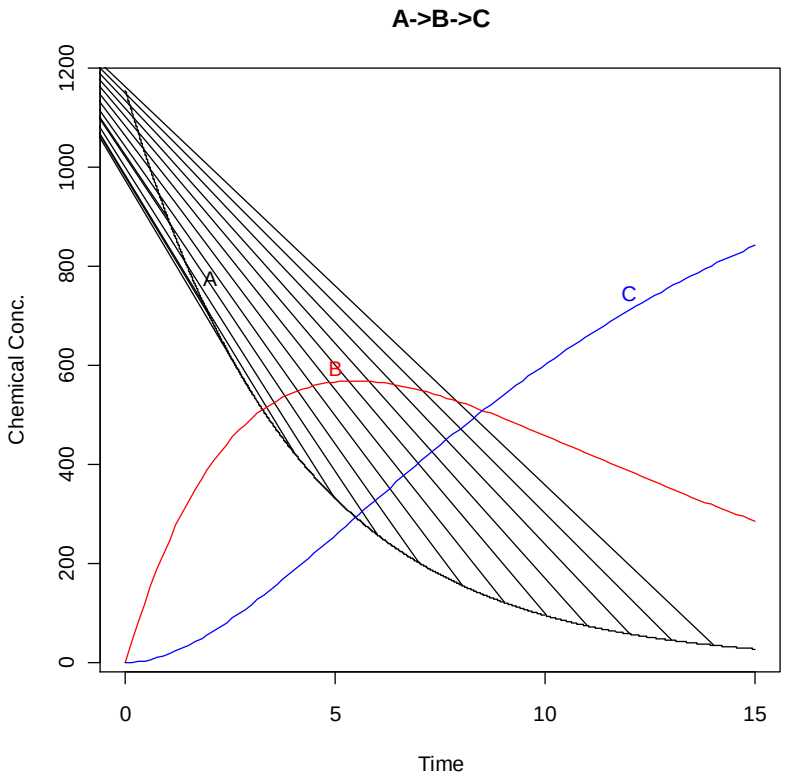
$$A = A_0 \exp(-k_1 t)$$

$$B = A_0 k_1 /(k_2 - k_1)(\exp(-k_1 t) - \exp(-k_2 t))$$

$$C = A_0 - A - B$$

Figure 12 shows the graph for the concentrations of chemicals A, B, and C. The graph is based on the values of A_0= 1155, k_1=0.25, and k_2=0.13.

Figure 12. Graph for chain chemical reaction model.



The equation for chemical B is the one I select for nonlinear regression calculations. Here is the data for the model with A_0=1155, k_1=0.25, and k_2=0.13:

```
T     A     B     C
1     899.5149044 238.9277464 16.55734917
2     700.542912  395.8784784 58.57860963
3     545.5833684 492.5360867 116.8805449
4     424.9007546 545.3551632 184.7440822
5     330.9130404 566.7704829 257.3164767
6     257.715335  566.1325168 331.1521482
7     200.7089047 550.4303627 403.8607327
8     156.3122521 524.8498032 473.8379447
9     121.7361044 493.2035683 540.0603273
10    94.80817341 458.262599  601.9292275
11    73.83667969 422.0106781 659.1526422
12    57.50406396 385.8397881 711.656148
13    44.78421005 350.699667  759.5161229
14    34.87797785 317.2120093 802.9100128
15    27.16299646 285.7574088 842.0795947
(empty line)
```

The above data has no noise added to it. The values represent time t and the concentrations for chemicals A, B, and C. Type in (or cut and paste) the above data and store it in file C:\regdata\nlr2.txt.

Given the above data, open a new script editor window in R and type the following lines:

```
demo.nlr2 = function(par.A0 = 500, par.k1 = 1.0, par.k2 = 0.1)
{
  # Parameters par1.A0, par1.k1, and par1.k2 represent good initial guesses
  # for the regression coefficients

  # read data
  nlr2.data = read.table("C:/regdata/nlr2.txt", header = TRUE)
  # attach data frame
  attach(nlr2.data, warn.conflicts=FALSE)
  # perform the nonlinear regression
  nlr2 = nls("B ~ A0*k1 / (k2-k1) * (exp(-k1 * T) - exp(-k2 * T))",
             data=nlr2.data, start = list(A0=par.A0,k1=par.k1,k2=par.k2))
  # store the number of observations
  ndata = length(T)
  # get the array of regression coefficients
  c=coef(nlr2)
  # store regression coefficient in scalar variables (easier to
  # read in equations below)
  A0=c[1]
  k1=c[2]
  k2=c[3]
  # Store coefficients of actual model in scalar variables
  model.A0 = 1155
  model.k1 = 0.25
  model.k2 = 0.13
  # plot observed points
  plot(T, B, type ="p", main = "B vs. T")
  # add two curves: one for the actual model and one for the fitted model
  curve(model.A0*model.k1/(model.k2-model.k1)*(exp(-model.k1*x) - exp(-
model.k2*x)),
    from=T[1], to=T[ndata], n=501, add=TRUE)
  curve((A0*k1)/(k2-k1)*(exp(-k1*x) - exp(-k2*x)),
    from=T[1], to=T[ndata], n=501, add=TRUE, col="red")
  # display the regression summary
  show(summary(nlr2))
  # display AIC and Log Likelihood statistics
  cat("AIC = ", AIC(nlr2), "\n", "Log Likelihood = ", logLik(nlr2),
      "\n", sep="")
  detach(nlr2.data)
  return ("End of demo")
}
```

The function has the following parameters:

- The parameter par.A0 which passes an argument to the regression coefficient A_0. This parameter has a default value of 500.
- The parameter par.k1 which passes an argument to the regression coefficient k_1. This parameter has a default value of 1.0.
- The parameter par.k2 which passes an argument to the regression coefficient k_2. This parameter has a default value of 0.1.

You are more than welcome to alter the default parameter values and replace them with ones that typify cases of data that you handle.

The function performs the following steps:

1. Reads the data in the file C:\regdata\nlr2.txt. This step stores the resulting data frame in variable nlr2.data.
2. Attaches the data frame nlr2.data to give access to the column variables in that data frame.
3. Performs the nonlinear regression by calling function nls(). The call is nlr1 = nls("B ~ A0*k1 / (k2-k1) * (exp(-k1 * T) - exp(-k2 * T))", data=nlr2.data, start = list(A0=par.A0, k1=par.k1, k2=par.k2)).
4. Stores the number of observations in variable ndata.
5. Stores the array of regression coefficients in vector c. This step calls function coef(nls.object) to obtain the coefficients.
6. Stores each of the regression coefficients in distinct scalar variables—A0, k1 and k2.
7. Stores the model's regression coefficients in distinct scalar variables—model.A0, model.k1, and model.k2.
8. Plots the values for variables T and B observations by calling function plot().
9. Plots the curve for the original model. This step calls function curve() and uses the values in variables model.A0, model.k1, and model.k2.
10. Plots the curve for the regression model. This step calls function curve() and uses the values in variables A0, k1, and k2. The call to function curve() also specifies the color red for the curve draw.
11. Displays the regression summary. This step calls function show() and supplies the argument summary(nlr2).
12. Displays the AIC and log likelihood statistics. This step calls function AIC(nls.object) and logLik(nls.object) to obtain these statistics.
13. Detaches the data frame nlr2.data.
14. Returns the message *End of demo* to let you know that the function has done its work.

Save the above code in file C:\regdata\demo.nlr2.r, and then load it by using the source function:

```
> source("C:/regdata/demo.nlr2.r")
```

Now execute the following command to use function demo.nlr2() and perform a nonlinear regression for the crescent curve model:

```
> demo.nlr2()

Formula: B ~ A0 * k1/(k2 - k1) * (exp(-k1 * T) - exp(-k2 * T))

Parameters:
    Estimate Std. Error   t value Pr(>|t|)
A0 1.155e+03  3.637e-07 3.176e+09   <2e-16 ***
k1 2.500e-01  9.584e-11 2.609e+09   <2e-16 ***
```

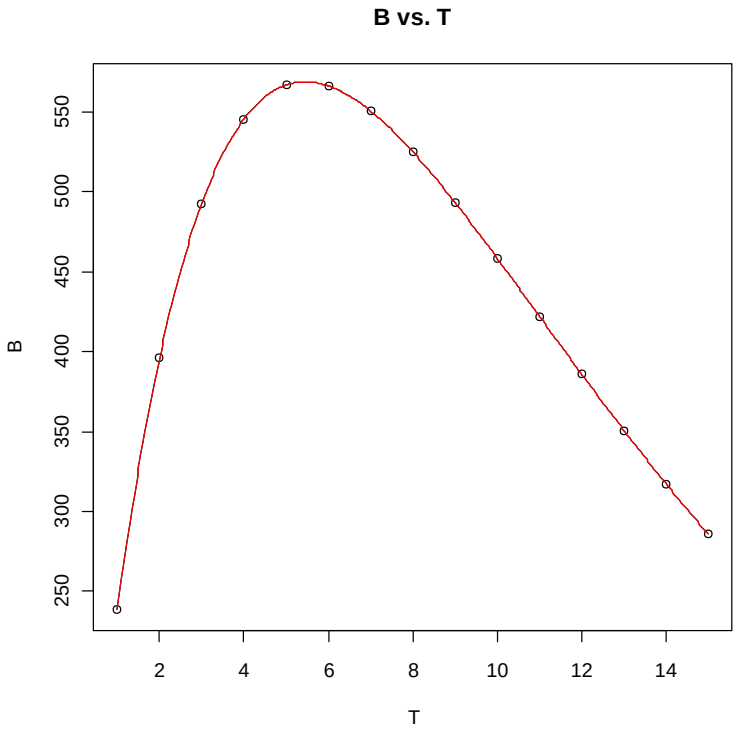


```
k2 1.300e-01  5.135e-11 2.531e+09   <2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 3.113e-08 on 12 degrees of freedom

Number of iterations to convergence: 8
Achieved convergence tolerance: 3.546e-06

AIC = -471.3312
Log Likelihood = 239.6656
[1] "End of demo"
```

The results of the regression coefficients match the model's original values. This ideal type of result is possible because the data has no noise. The function also displays the graph in Figure 13.

You may have noticed that the above example does not use the function confint() to calculate the confidence intervals for the nonlinear regression coefficients. The reason is that using function confint() in the case (at least for the data at hand) generates a runtime error!

Figure 13. Graph for concentration of chemical B.



Let's look at the same data for the chemical reaction. This time the table of data contains values for the time and three versions of the concentrations of chemical B. Each version has a different level of error. The values for variables B10, B20, and B40 are calculated using the following equations:

B10 = B + 10*(UNIF_RAND – 0.5)

B20 = B + 20*(UNIF_RAND(– 0.5)

B40 = B + 40*(UNIF_RAND – 0.5)

Where UNIF_RAND is some uniformly distributed random number generator that yields values in the range of (0, 1). In addition to injecting noise to the variables B10, B20, B40, I rounded their resulting values to one decimal. Here is the table of results:

```
T     B10   B20   B40
1     238.6 248.0 236.7
2     397.4 394.0 410.5
3     497.1 500.6 498.5
4     543.2 550.6 536.9
5     568.1 569.9 554.1
6     568.6 557.4 559.7
7     546.1 546.4 568.6
8     528.4 524.3 534.0
9     489.2 498.4 500.3
10    462.1 466.7 457.3
11    423.9 419.2 410.3
12    382.7 390.5 365.8
13    345.9 358.9 363.7
14    314.5 316.1 312.4
15    282.0 281.6 272.2
(empty line)
```

Type in (or cut and paste) the above data and store it in file C:\regdata\nlr3.txt.

Given the above data, open a new script editor window in R and type in (or cut and paste) the following lines:

```
demo.nlr3 = function(par1.A0 = 1000, par1.k1 = 0.50, par1.k2 = 0.1,
                     par2.A0 = 500,  par2.k1 = 1.0,  par2.k2 = 0.0)
{
  # Parameters par1.A0, par1.k1, and par1.k2 represent good initial guesses
  # for the regression coefficients
  # Parameters par2.A0, par2.k1, and par2.k2 represent poor initial guesses
  # for the regression coefficients
  # test B10, B20 and B40
  nlr3.data = read.table("C:/regdata/nlr3.txt", header = TRUE)
  attach(nlr3.data, warn.conflicts=FALSE)
  # get the array of column names
  sVars = names(nlr3.data)
  # get the total number of variables
  nvars = length(sVars)
```



```
# get the number of data
ndata = length(T)
# set the graph display configuration
par(mfrow=c(2,2))
# Store coefficients of actual model in scalar variables
model.A0 = 1155
model.k1 = 0.25
model.k2 = 0.13
# process B10, B20, and B40 (twice)
for (j in 2:(nvars+1)) {
  # use special index i to access B10, B20, B40, AND B40 again
  # with poor initial guesses for the regression coefficients
  if (j <= nvars) {
    i = j
    # list of good guesses
    coeff.guess.list = list(A0=par1.A0, k1=par1.k1, k2=par1.k2)
  }
  else {
    # when j = n+1, set i = n (again) to access B40
    i = nvars
    # list of poor guesses
    coeff.guess.list = list(A0=par2.A0, k1=par2.k1, k2=par2.k2)
  }
  # get the targeted variable Bxx
  sVarY = sVars[i]
  # build the model
  sExpr = paste(sVarY, " ~ A0*k1/(k2-k1)*(exp(-k1 * T) - exp(-k2 * T))",
              sep="")
  # perform the nonlinear regression
  nlr3 = nls(sExpr, data=nlr3.data, start = coeff.guess.list)
  # get the array of regression coefficients
  c=coef(nlr3)
  # store regression coefficient in scalar variables (easier to
  # read in equations below)
  A0=c[1]
  k1=c[2]
  k2=c[3]
  # plot the points
  if (j <= n) {
    plot(nlr3.data[c(1,i)], type ="p",
         main = paste(sVarY, " vs. T", sep=""))
  }
  else {
    plot(nlr3.data[c(1,i)], type ="p",
         main = paste(sVarY, " vs. T, take 2", sep=""))
  }
  # add two curves: one for the actual model and one for the fitted model
  curve(model.A0*model.k1/(model.k2-model.k1)*(exp(-model.k1*x) - exp(-
      model.k2*x)), from=T[1], to=T[ndata], n=501, add=TRUE)
  curve(A0*k1/(k2-k1)*(exp(-k1*x) - exp(-k2*x)),
      from=T[1], to=T[ndata], n=501, add=TRUE, col="red")
  # display the regression summary
  show(summary(nlr3))
  # display AIC and Log Likelihood statistics
  cat("AIC = ", AIC(nlr3), "\n", "Log Likelihood = ",
      logLik(nlr3), "\n" , sep="")
```

```
  }
  par(mfrow=c(1,1))
  detach(nlr3.data)
  return ("End of demo")
}
```

The function has the following parameters:

- The parameter par1.A0, par1.k1, and par1.k2 represent the regression coefficients A_0, k_1, and k_2. This parameter set supplies *good* initial guesses for the regression coefficients.
- The parameter par2.A0, par2.k1, and par2.k2 represent the regression coefficients A_0, k_1, and k_2. This parameter set supplies *poor* initial guesses for the regression coefficients.

While function demo.nlr3() resembles demo.nlr2(), it has the following differences that are work mentioning:

1. The function performs four sets of nonlinear regression calculations. The first three sets of calculation use good initial guesses for the regression coefficients with the variables B10, B20, and B40. The fourth set of calculations uses the poor initial guesses for the regression coefficients with the variable B40. To function uses a for loop to process each variable-the variable B40 enjoys two iterations. The loop itself uses the variable j as the loop control variable. However, since function accesses variable B40 twice, the loop uses the variable i as the effective control variable.
2. The function builds a string that represents the regression model. Each loop iteration includes a string with a different *Bnn* variable.
3. The plot() function uses the expression nlr3.data[c(1, i)] to access the variables to plot the data for time T the selected variable B10 or B20 or B40.
4. The function plots four graphs grouped together.

Save the above code in file "C:\regdata\demo.nlr3.r", and then load it by using the source function:

```
> source("C:/regdata/demo.nlr3.r")
```

Now execute the following command to use function demo.nlr3() and perform a nonlinear regression for the crescent curve model:

```
> demo.nlr3()

Formula: B10 ~ A0 * k1/(k2 - k1) * (exp(-k1 * T) - exp(-k2 * T))

Parameters:
    Estimate Std. Error t value Pr(>|t|)
A0 1.190e+03  4.239e+01   28.07 2.59e-12 ***
k1 2.429e-01  1.031e-02   23.55 2.05e-11 ***
k2 1.353e-01  5.891e-03   22.97 2.75e-11 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```



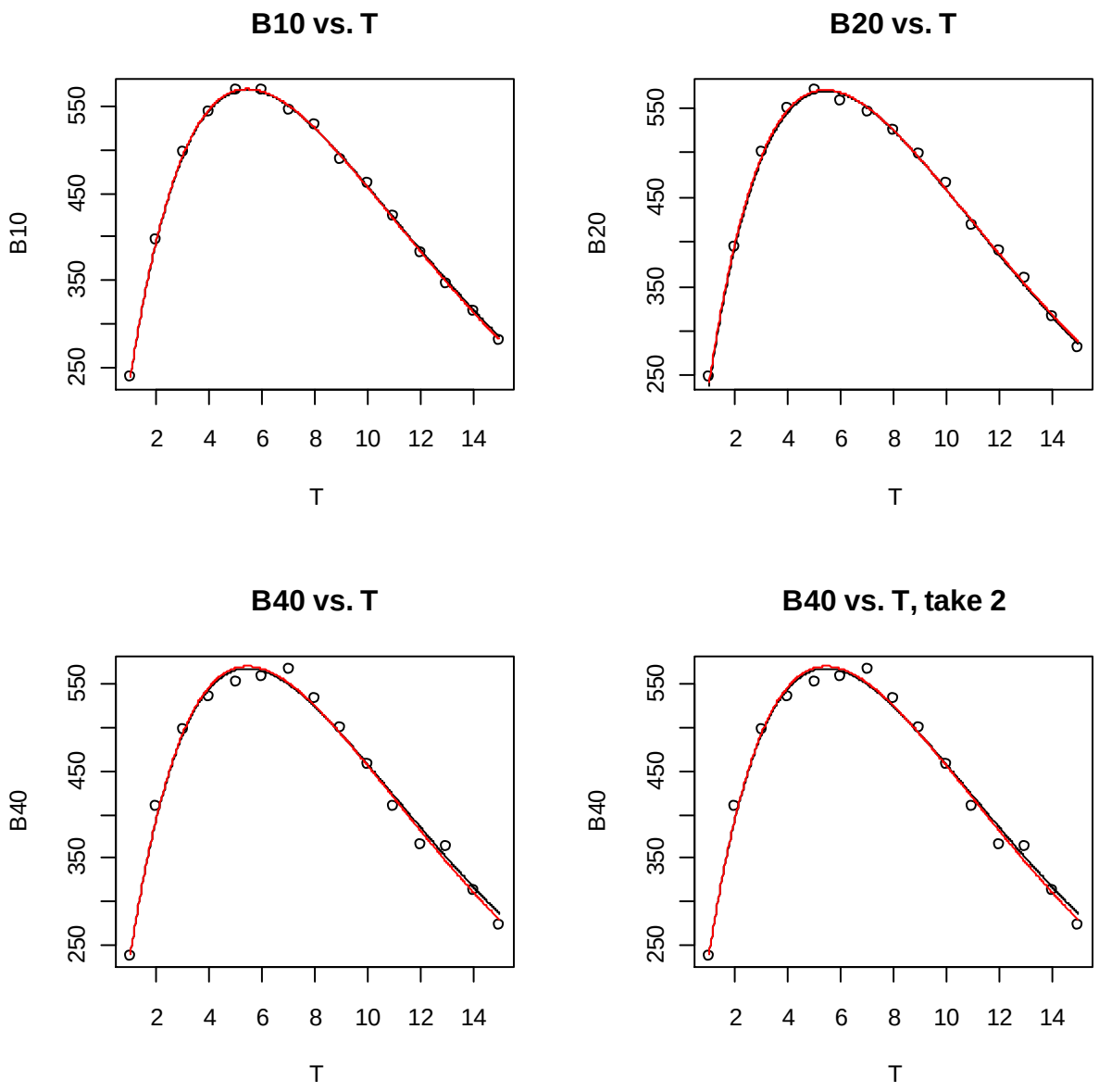
```
Residual standard error: 3.124 on 12 degrees of freedom

Number of iterations to convergence: 5
Achieved convergence tolerance: 1.647e-07

AIC = 81.396
Log Likelihood = -36.698

Formula: B20 ~ A0 * k1/(k2 - k1) * (exp(-k1 * T) - exp(-k2 * T))

Parameters:
    Estimate Std. Error t value Pr(>|t|)
A0 1.107e+03  5.454e+01   20.29 1.18e-10 ***
k1 2.654e-01  1.656e-02   16.02 1.82e-09 ***
k2 1.226e-01  7.921e-03   15.48 2.70e-09 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 5.959 on 12 degrees of freedom

Number of iterations to convergence: 5
Achieved convergence tolerance: 5.967e-07

AIC = 100.7680
Log Likelihood = -46.38402

Formula: B40 ~ A0 * k1/(k2 - k1) * (exp(-k1 * T) - exp(-k2 * T))

Parameters:
    Estimate Std. Error t value Pr(>|t|)
A0 1.258e+03  2.277e+02   5.524 0.000131 ***
k1 2.290e-01  4.744e-02   4.827 0.000414 ***
k2 1.451e-01  3.062e-02   4.739 0.000481 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 12.06 on 12 degrees of freedom

Number of iterations to convergence: 5
Achieved convergence tolerance: 2.484e-06

AIC = 121.9107
Log Likelihood = -56.95535

Formula: B40 ~ A0 * k1/(k2 - k1) * (exp(-k1 * T) - exp(-k2 * T))

Parameters:
    Estimate Std. Error t value Pr(>|t|)
A0 1.258e+03  2.277e+02   5.524 0.000131 ***
k1 2.290e-01  4.744e-02   4.827 0.000414 ***
k2 1.451e-01  3.062e-02   4.739 0.000481 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 12.06 on 12 degrees of freedom
```



```
Number of iterations to convergence: 6
Achieved convergence tolerance: 1.866e-06

AIC = 121.9107
Log Likelihood = -56.95535
[1] "End of demo"
```

Examine the summaries for the variables B10, B20, and B40. Notice that the associated regression coefficients deviate more from the model's original value, as more noise is injected. Figure 14 shows the four graphs generated by the function demo.nlr3(). Experiment with calling the function and supplying different arguments.

Figure 14. Graphs for concentration of chemical B.



The nstools Library

You can download the nstools library from CRAN and install it by typing the following command:

```
> library(nstools)
```

The following table shows a summary of the contents of the nstools library.

Library Function/Data Set	*Description*
albert	Bacterial survival models
baranyi	Bacterial growth models
baranyi_without_lag	Bacterial growth models
baranyi_without_Nmax	Bacterial growth models
bilinear_without_Nres	Bacterial survival models
bilinear_without_Sl	Bacterial survival models
buchanan	Bacterial growth models
buchanan_without_lag	Bacterial growth models
buchanan_without_Nmax	Bacterial growth models
compet_mich	Michaelis-Menten model and derived equations to model competitive and non-competitive inhibition
cpm_aw_2p	Secondary growth models
cpm_aw_3p	Secondary growth models
cpm_pH_3p	Secondary growth models
cpm_pH_4p	Secondary growth models
cpm_T	Secondary growth models
cpm_T_pH_aw	Secondary growth models
geeraerd	Bacterial survival models
geeraerd_without_Nres	Bacterial survival models
geeraerd_without_Sl	Bacterial survival models
gompertzm	Bacterial growth models
growthcurve1	Bacterial kinetics data sets
growthcurve2	Bacterial kinetics data sets
growthcurve3	Bacterial kinetics data sets
growthcurve4	Bacterial kinetics data sets
growthmodels	Bacterial growth models
mafart	Bacterial survival models
michaelis	Michaelis-Menten model and derived equations to model competitive and non-competitive inhibition
michaelismodels	Michaelis-Menten model and derived equations to model competitive and non-competitive inhibition
nlsBoot	Bootstrap resampling
nlsConfRegions	Confidence regions
nlsContourRSS	Surface contour of RSS
nlsJack	Jackknife resampling
nlsResiduals	NLS residuals
nlstools	Nonlinear least squares fit
non_compet_mich	Michaelis-Menten model and derived equations to model competitive and non-competitive inhibition
overview	Nonlinear least squares fit
plot.nlsBoot	Bootstrap resampling
plot.nlsConfRegions	Confidence regions
plot.nlsContourRSS	Surface contour of RSS
plot.nlsJack	Jackknife resampling
plot.nlsResiduals	NLS residuals
plotfit	Nonlinear least squares fit
preview	Nonlinear least squares fit

Library Function/Data Set	*Description*
print.nlsBoot	Bootstrap resampling
print.nlsConfRegions	Confidence regions
print.nlsContourRSS	Surface contour of RSS
print.nlsJack	Jackknife resampling
print.nlsResiduals	NLS residuals
ross	Secondary growth curves
secondary	Secondary growth models
summary.nlsBoot	Bootstrap resampling
summary.nlsJack	Jackknife resampling
survivalcurve1	Bacterial survival data sets
survivalcurve2	Bacterial survival data sets
survivalcurve3	Bacterial survival data sets
survivalmodels	Bacterial survival models
test.nlsResiduals	NLS residuals
trilinear	Bacterial survival models
vmkm	Michaelis Menten data sets
vmkmki	Michaelis Menten data sets

The next subsections discuss a selection of functions in the nstools library that may complement the results of the nls() function.

The overview() Function

The function overview(nls.object) returns an extended summary that includes the confidence interval. While this seems to merely replace the sequence of calling functions summary() and confint(), the function overview() does more! It is able to calculate the confidence interval for the nonlinear regression coefficients in the cases where function confint() fails! If you need the confidence intervals for the nonlinear regression coefficients, then function overview() meets your needs.

To illustrate how the function overview() works, execute the following command to view the extended summary for the variable nlr2 (obtained using ideal data for the chain chemical reaction):

```
> overview(nlr2)

------
Formula: B ~ A0 * k1/(k2 - k1) * (exp(-k1 * T) - exp(-k2 * T))

Parameters:
    Estimate Std. Error   t value Pr(>|t|)
A0 1.155e+03  3.637e-07 3.176e+09   <2e-16 ***
k1 2.500e-01  9.584e-11 2.609e+09   <2e-16 ***
k2 1.300e-01  5.135e-11 2.531e+09   <2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 3.113e-08 on 12 degrees of freedom
```



```
Number of iterations to convergence: 6
Achieved convergence tolerance: 1.406e-06

------
Residual sum of squares: 1.16e-14

------
Asymptotic confidence interval:
      2.5%   97.5%
A0 1155.00 1155.00
k1    0.25    0.25
k2    0.13    0.13

------
Correlation matrix:
           A0         k1         k2
A0  1.0000000 -0.9927989  0.9973905
k1 -0.9927989  1.0000000 -0.9886211
k2  0.9973905 -0.9886211  1.0000000
```

The confidence interval shows that the regression coefficients have no errors! Contrast the above output with the next one obtained from the variable nlr3:

```
> overview(nlr3)

------
Formula: B40 ~ A0 * k1/(k2 - k1) * (exp(-k1 * T) - exp(-k2 * T))

Parameters:
    Estimate Std. Error t value Pr(>|t|)
A0 1.258e+03  2.277e+02   5.524 0.000131 ***
k1 2.290e-01  4.744e-02   4.827 0.000414 ***
k2 1.451e-01  3.062e-02   4.739 0.000481 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 12.06 on 12 degrees of freedom

Number of iterations to convergence: 6
Achieved convergence tolerance: 1.866e-06

------
Residual sum of squares: 1740

------
Asymptotic confidence interval:
           2.5%        97.5%
A0 761.61109324 1753.6944431
k1   0.12561961    0.3323364
k2   0.07840557    0.2118395

------
Correlation matrix:
           A0         k1         k2
A0  1.0000000 -0.9965997  0.9987426
```

```
k1 -0.9965997  1.0000000 -0.9944122
k2  0.9987426 -0.9944122  1.0000000
```

The confidence intervals or the regression coefficients in the case of variable nlr3 show some variation. The coefficients A0, k1, and k2 have ratios of upper value/lower value that are equal to 2.30261,2.64558, and 2.70184, respectively.

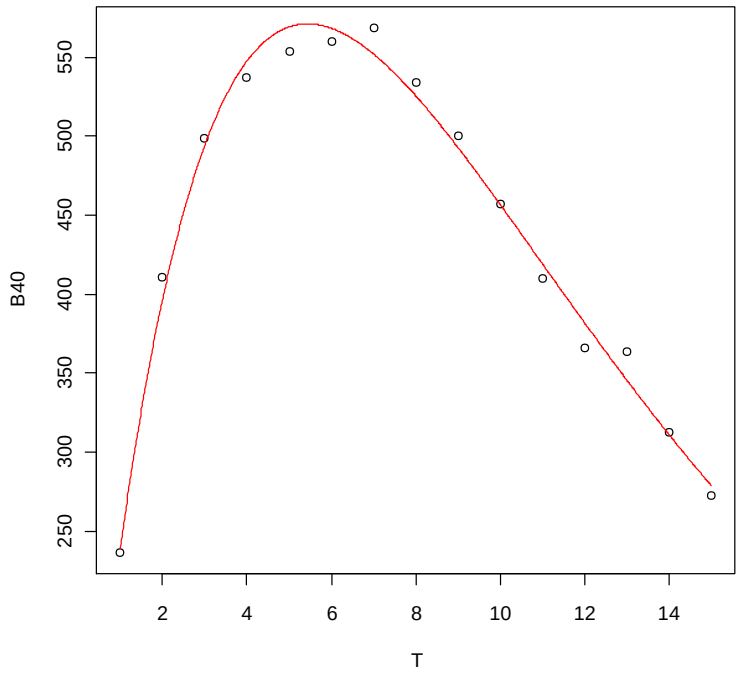
The plotfit() Function

The plotfit(nls.object, smooth=FALSE) draws a graph that contains the observed data and the fitted model curve (when the argument for smooth is TRUE). To illustrate using the plotfit() function, execute the following command:

```
> plotfit(nlr3,smooth=TRUE)
```

The above command generates the graph in Figure 15.

Figure 15. Graph for concentration of chemical B.



The nlsResiduals(nls) Function

The function nlsResiduals(nls.object) returns a collection of matrices that contain values for residuals. Execute the following command to obtain such a matrix:

```
> (res=nlsResiduals(nlr3))
Residuals
```



```
  vector length mode    content
1 $std95 1      numeric Student value for alpha = 0.05

  matrix nrow ncol content
1 $resi1 15   2    fitted values vs. non-transf. resid
2 $resi2 15   2    fitted values vs. standardized resid
3 $resi3 15   2    fitted values vs. sqrt abs std resid
4 $resi4 14   2    resid i vs. resid i+1
```

To view the contents of element res$resi1, execute the following command:

```
> res$resi1
      Fitted values   Residuals
 [1,]      238.9169  -2.2169028
 [2,]      396.6647  13.8352690
 [3,]      494.2147   4.2852796
 [4,]      547.6573 -10.7573407
 [5,]      569.2807 -15.1807172
 [6,]      568.4176  -8.7176024
 [7,]      552.1097  16.4902825
 [8,]      525.6281   8.3719060
 [9,]      492.8800   7.4200275
[10,]      456.7264   0.5735858
[11,]      419.2297  -8.9297096
[12,]      381.8457 -16.0456564
[13,]      345.5726  18.1274054
[14,]      311.0666   1.3334236
[15,]      278.7301  -6.5300544
```

The above matrix represents the fitted values for B40 and their residuals. The next command displays the fitted values and their standardized residuals:

```
> res$resi2
      Fitted values Standardized residuals
 [1,]      238.9169            -0.19525227
 [2,]      396.6647             1.13609012
 [3,]      494.2147             0.34402872
 [4,]      547.6573            -0.90358303
 [5,]      569.2807            -1.27045105
 [6,]      568.4176            -0.73441027
 [7,]      552.1097             1.35629285
 [8,]      525.6281             0.68296721
 [9,]      492.8800             0.60401987
[10,]      456.7264             0.03618655
[11,]      419.2297            -0.75200212
[12,]      381.8457            -1.34218778
[13,]      345.5726             1.49207329
[14,]      311.0666             0.09920632
[15,]      278.7301            -0.55297842
```

The next command displays the fitted values and their square root absolute standardized residuals:

```
> res$resi3
      Fitted values Sqrt abs. standardized residuals
```



```
 [1,]      238.9169                        0.4287966
 [2,]      396.6647                        1.0712030
 [3,]      494.2147                        0.5961665
 [4,]      547.6573                        0.9445619
 [5,]      569.2807                        1.1220808
 [6,]      568.4176                        0.8503085
 [7,]      552.1097                        1.1694779
 [8,]      525.6281                        0.8332785
 [9,]      492.8800                        0.7844779
[10,]      456.7264                        0.2181108
[11,]      419.2297                        0.8605907
[12,]      381.8457                        1.1536039
[13,]      345.5726                        1.2261562
[14,]      311.0666                        0.3325539
[15,]      278.7301                        0.7359298
```

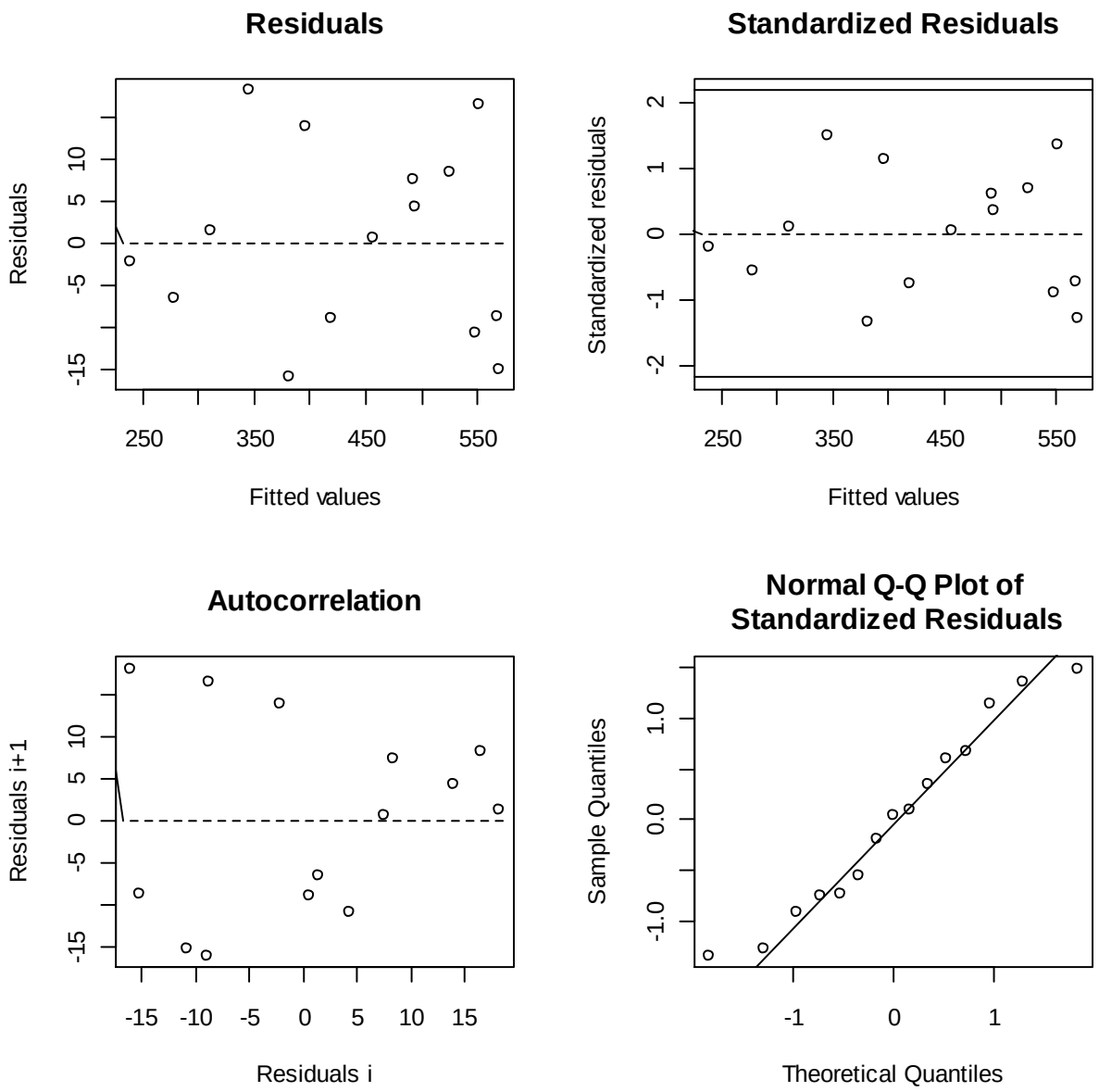
The next command displays the residuals(i) vs. residuals(i+1):

```
> res$resi4
      Residuals i Residuals i+1
 [1,]  -2.2169028    13.8352690
 [2,]  13.8352690     4.2852796
 [3,]   4.2852796   -10.7573407
 [4,] -10.7573407   -15.1807172
 [5,] -15.1807172    -8.7176024
 [6,]  -8.7176024    16.4902825
 [7,]  16.4902825     8.3719060
 [8,]   8.3719060     7.4200275
 [9,]   7.4200275     0.5735858
[10,]   0.5735858    -8.9297096
[11,]  -8.9297096   -16.0456564
[12,] -16.0456564    18.1274054
[13,]  18.1274054     1.3334236
[14,]   1.3334236    -6.5300544
```

If you pass the result of nlsResiduals(nlr3) as an argument to function plot(), you obtain a setoff multiple graphs, shown in Figure 16, that shows the residuals vs. fitted values, standardized residuals vs. fitted values, autocorrelation, and normal Q-Q plot of standardized residuals:

```
> plot(nlsResiduals(nlr3))
```


Figure 16. Residuals plot.



The test.nlsResiduals(nlsResiduals(nls)) Function

The function test.nlsResiduals(nlsResiduals(nls.object)) performs the Shapiro-Wilk normality test. Execute the following command to illustrate this function:

```
> test.nlsResiduals(nlsResiduals(nlr3))

------
        Shapiro-Wilk normality test

data:  stdres
W = 0.9519, p-value = 0.5544
```



```
------
        Runs Test

data:  as.factor(run)
Standard Normal = -0.7898, p-value = 0.4297
alternative hypothesis: two.sided
```

The nlsConfRegions(nls) Function

The nlsConfRegions(nls.object) function calculates the confidence region for an nls object. The function takes a good while to finish calculating and in the process displays percent progress.

To illustrate the nlsConfRegions() execute the following command:

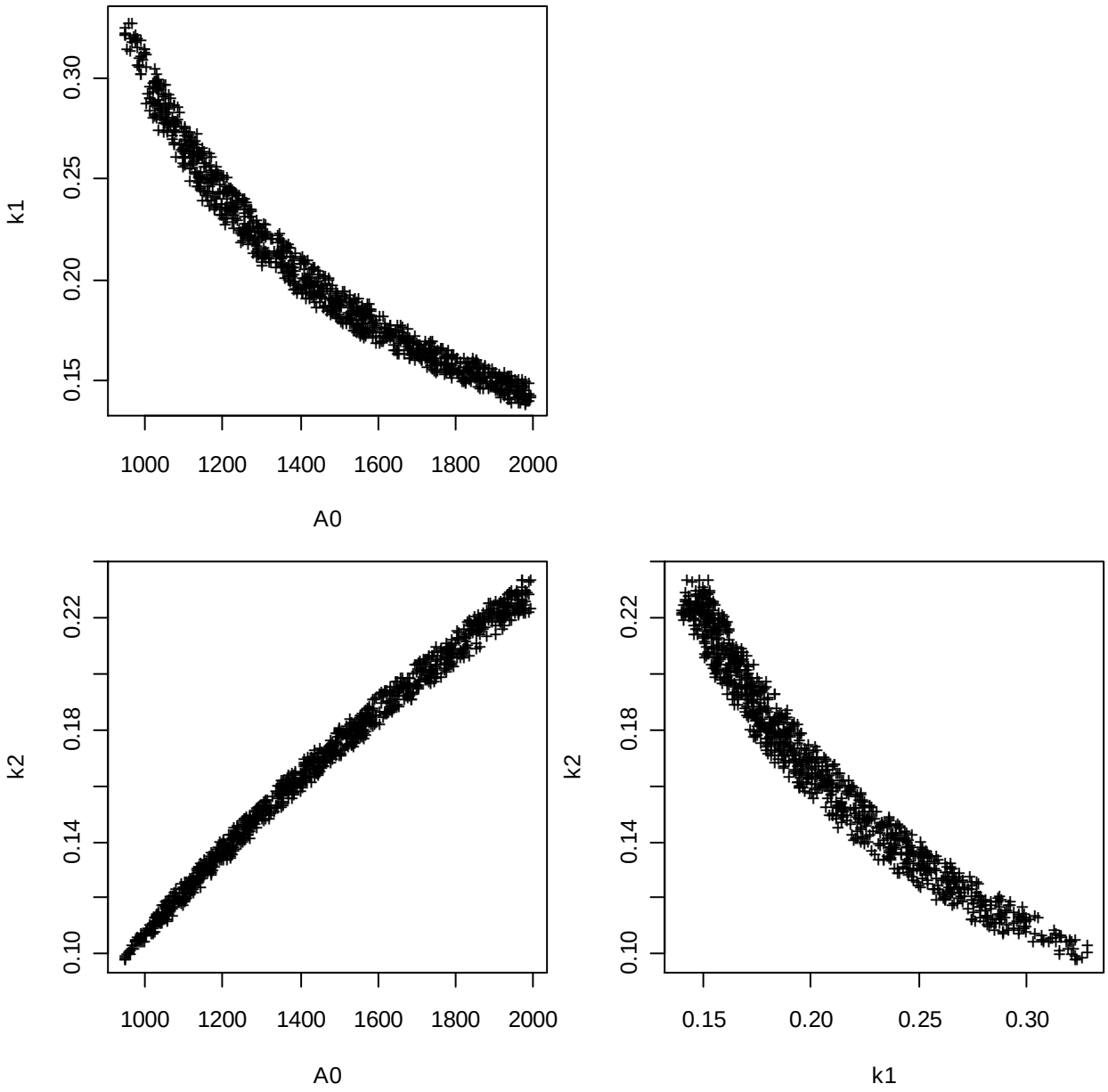
```
> (res=nlsConfRegions(nlr3))

> res
Beale's 95 percent confidence regions

  vector length mode    content
1 $rss   1000   numeric Residual sums of squares
2 $rss95 1      numeric 95 percent RSS threshold

  data.frame nrow ncol content
1 $cr        1000 3    Sample drawn in the confidence region
2 $bounds    3    2    Bounds of the drawing hypercube
```

Plot the information in variable res by executing the following command:

```
> plot(res)
```

Figure 17 shows the diagram created by the above call to function plot().

Figure 17. Confidence regions.



The nlsContourRSS(nls) Function

The mlsContourRSS(nls.object) function returns arrays for the RSS surface contour. To illustrate this function, execute the following command:

```
> (res=(nlsContourRSS(nlr3)))

 RSS contour surface array returned
RSS surface contour

$lrss95 (95 percent RSS threshold):  8.09
  matrix   nrow ncol content
```



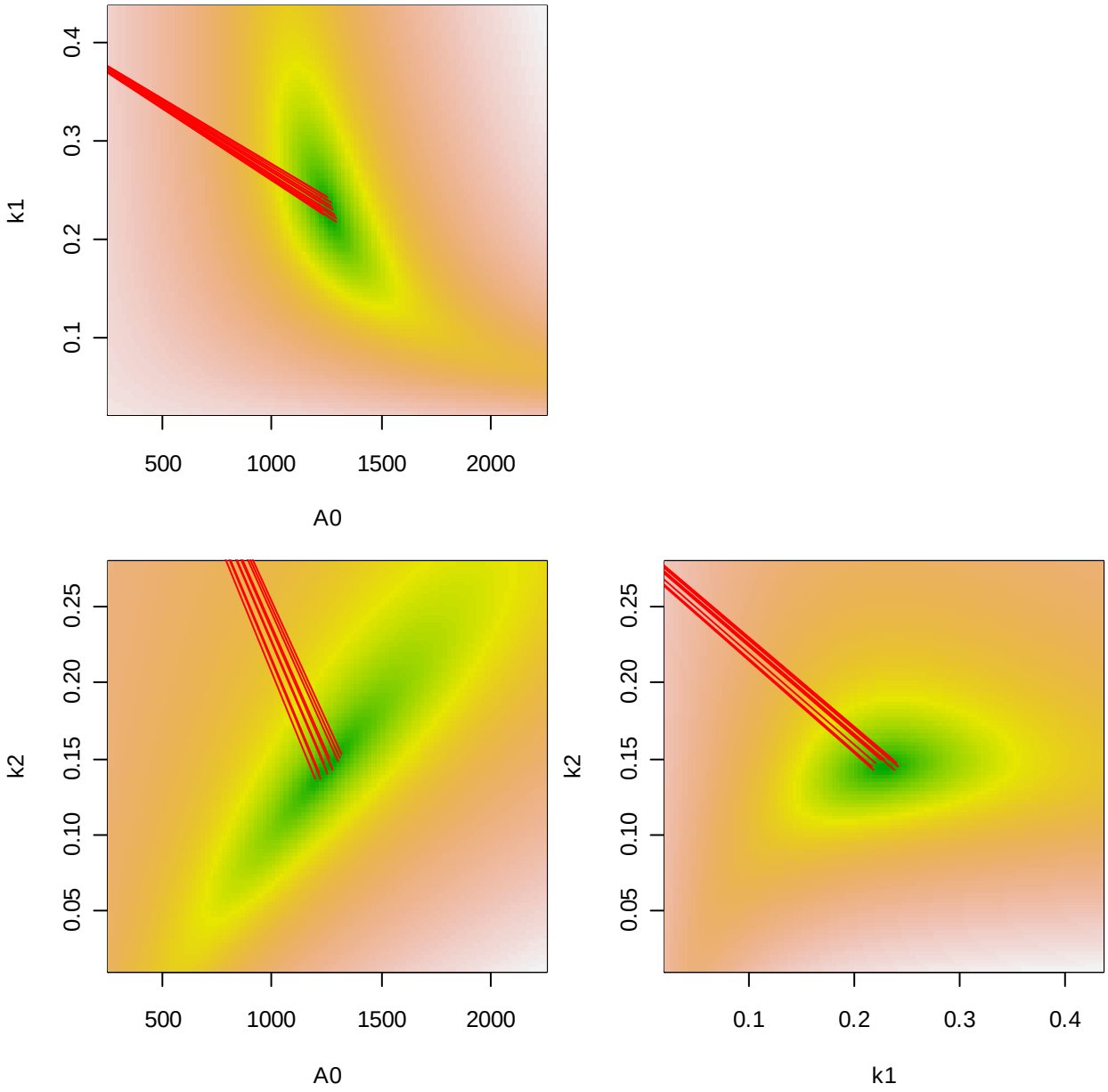
```
1 $seqPara 100  3    Sequence of parameters

  list  length nrow ncol content
1 $lrss 3      100  100  RSS
```

Plot the information in variable res by executing the following command:

```
> plot(res)
```

Figure 18 shows the diagram created by the above call to function plot().

Figure 18. Contours for confidence intervals.



Preparing for the Regression Functions

The next few sections present user-defined functions that perform linear regression, multiple linear regression, and polynomial regression. These functions perform regression calculations, return regression results, and plot graphs.

To use these functions, you should save them in script files and then load them using the source() function. Each function has optional demo statements that run if a global variable TEST.FX is set to TRUE.

So before you start working with the regression functions, execute the following command:

```
> TEST.FX=FALSE
```

If you want to load a regression and automatically run the demo, set the variable TEST.FX to TRUE. If the variable TEST.FX is not defined, it has the same effect as if it were assigned FALSE.

The Linear Regression Function

This section presents the function linreg(). This function supports linear and linearized regression. You need to know the following category of information about this function:

- The function's parameters
- Tasks performed by the function
- Returned data

The Function's Parameters

The function linreg() has the following heading declaration and parameters:

```
linreg = function(x, y, fx, fy, ify, xp=c(), probab=0.95,
                  show.graph=TRUE, col1="red", col2="green",
                  main1="Linearized fit", main2="Original Data")
```

- The parameters **x** and **y** are the vectors for the x and y data.
- The parameter **fx** is the function that transforms the x data.
- The parameter **fy** is the function that transforms the y data.
- The parameter **ify** is the function that reverse transforms the y data.
- The parameter **xp** is a vector of x values to project onto y. The default value for this parameter is an empty vector.
- The parameter **probab** is the confidence level, expressed as a fraction, used to calculated the confidence interval for the fitted and projected values of y.
- The Boolean parameter **show.graph** tells the function whether or not to show any graphs.
- The parameters **col1** and **col2** represent the colors for the observed points and projected points, respectively. The default values for col1 and col2 are red and blue, respectively.
- The parameters **main1** and **main2** specify the titles for the linearized fit and the non-transformed data fit.

The Function's Tasks

The function linreg() performs the following tasks:

1. Transforms the values of vector x and stores the results in variable xt.
2. Transforms the values of vector y and stores the results in variable yt.
3. Stores the number of observations in variable n.
4. Performs the linearized regression. This task calls function lm(yt~xt) and stores the result in variable lr.
5. Stores the coefficient of determination in variable r2.
6. Stores the regression intercept and slope in variables intercept and slope, respectively.
7. Calculates and stores statistics used for later calculating confidence intervals for y.
8. Determines if vector xp has any data. If it has data, the function performs the following subtasks:
   8.1. Creates vector xpr that combines vectors x and xp to store all of the independent variable values.
   8.2. Creates the vector ypr that combines the observed values in vector y and the projected y values obtained from vector xp. These two assignments ensure that the function can graph the data for the observed and the projected data.
   8.3. Resumes in task 10.
9. If the vector xp has no data, simply copy vectors x and y onto vectors xpr and ypr, respectively.
10. Sorts the vectors xpr and ypr using function order() to obtain the sort order from vector xpr. This task ensures that the drawn lines don't zigzag and proceed smoothly like you'd expect them to.
11. Calculates the difference in the projected y using the inverse Student-t distribution value and standard difference in the projected y. This task stores the difference value in variable delta.y.
12. Calculates another difference in the projected y values, assuming that the observations are new. Such a difference is larger in magnitude than the one calculated in task 11. This task stores the difference value in variable delta2.y. Examine the source code and locate the assignment statements for delta.y and delta2.y (the statements appear one right after the other). Compare the equations used to calculate delta.y and delta2.y and see how similar they are. I believe the function predict() uses the same, if not similar, equations to calculate the two flavors for the confidence intervals.
13. Calculates the transformed values for xpr and stores them in vector xt2.
14. Uses the values in vector xt2 to calculated fitted values for y and stores them in vector yt2.
15. Calculates the lower and upper values for the transformed y values, using the variable delta.y, and storing the results in vectors yhat.lo and yhat.hi, respectively.
16. Calculates the lower and upper values for the transformed y values, using the variable delat2.y, storing the results in vectors yhat2.lo and yhat2.hi, respectively.

17. Determines if the parameter show.graph is TRUE. If not, the function skips to task 30.
18. Turns on the ask-user mode for displaying graphs.
19. Plots the points for the transformed values using the values in vectors fx(xpr) and fy(ypr). The plot includes a title and displays the points in the color specified by parameter col1. The graph's title also includes the value of the coefficient of determination, rounded to five decimals.
20. Re-plots the projected points using the color specified by parameter col2. This task calls function points(). The expression fx(xp) supplies the values for the independent variable. The expression intercept + slope * fx(xp) supplies the values for the dependent variable.
21. Draws the regression line by calling function abline(intercept, slope).
22. Draws the confidence interval for the dependent variable by making two calls to function lines()--one for the upper limit and one for the lower limit of the interval. To draw the upper limit, the expressions fx(xpr) and yhat.hi supply the data for the independent and dependent variables, respectively. To draw the lower limit, the expressions fx(xpr) and yhat.lo supply the data for the independent and dependent variables, respectively.
23. Draws the wider confidence interval for the dependent variable by making two calls to function lines()--one for the upper limit and one for the lower limit of the interval. To draw the upper limit, the expressions fx(xpr) and yhat2.hi supply the data for the independent and dependent variables, respectively. To draw the lower limit, the expressions fx(xpr) and yhat2.lo supply the data for the independent and dependent variables, respectively. This task draws the confidence interval as red lines.
24. Calls function plot() to start drawing the graph for the original data. The call uses vectors xpr and ypr to supply data for the independent and dependent variables, respectively. The function plot() draws the observed and projected points using the color specified by parameter col1. The graph's title also includes the value of the coefficient of determination, rounded to five decimals.
25. Re-plots the projected points using the color specified by parameter col2. The expressions fx(xp) and intercept + slope * fx(xp) supply the values for the independent and dependent variables, respectively.
26. Draws the regression curve by calling function lines(). The expressions xpr and ify(intercept + slope * fx(xpr)) provide the values for the independent and dependent variables, respectively.
27. Draws the confidence interval for the dependent variable by making two calls to function lines()--one for the upper limit and one for the lower limit of the interval. To draw the upper limit, the expressions xpr and ify(yhat.hi) supply the data for the independent and dependent variables, respectively. To draw the lower limit, the expressions xpr and ify(yhat.lo) supply the data for the independent and dependent variables, respectively.
28. Draws the wider confidence interval for the dependent variable by making two calls to function lines()--one for the upper limit and one for the lower limit of the interval. To draw the upper limit, the expressions xpr and ify(yhat2.hi) supply the data for the independent and

dependent variables, respectively. To draw the lower limit, the expressions xpr and ify(yhat2.lo) supply the data for the independent and dependent variables, respectively. This task draws the confidence interval as red lines.

29. Turns off the ask-user mode for displaying graphs.
30. Creates a list variable that stores the function's retuned values.
31. Returns the variable res, which is the list of results.

The Returned Data

The function linreg() returns a list with the following elements:

- The lm.obj tag that has the results of calling function lm().
- The tag r2 that contains the coefficient of determination.
- The tag x2 that stores the vector of the combined observed and projection values for the independent variable.
- The tag y2 that stores the vector of the combined observed and projected values for the dependent variable.
- The tag yhat which contains the values for the regression-based values for the dependent variable.
- The tag yhat.hi contains the upper limit for the projected y values.
- The tag yhat.lo contains the lower limit for the projected y values.

Here is the source code for the function linreg() and the demo statements:

```
linreg = function(x, y, fx, fy, ify, xp=c(), probab=0.95,
                  show.graph=TRUE, col1="red", col2="blue",
                  main1="Linearized fit", main2="Original Data")
{
  #----------------- Parameters -------------------
  # x and y are the data for x and y variables
  # fx is the transformation for x
  # fy is the transformation for y
  # ify is the inverse transformation for y
  # xp is the vector of x values for projections
  # probab is the confidence level for confidence interval
  # show.graph is flag to plot graphs
  # col1 is the color for observed data
  # col2 is the color for the projected data
  # main1 is the title for the transformed plot
  # main2 is the title for the untransformed values
  #------------------------------------------------

  #--------------------------- Return ---------------------------------------
  # Function returns the list res that has the following elements:
  #
  # lm.obj is the lm object
  # r.squared is R-squared
  # x2 is the vector for all values of x (observed and projected)
  # y2 is the vector for all values of y (observed and projected)
  # yhat is the vector for projected Y
```



```
# yhat.lo and yhat.hi are the vectors for Y^ CI at specified
#   confidence level
#-------------------------------------------------------------------------

# transform observations to lineaize them
xt = fx(x)
yt = fy(y)
# store the number of observations
n = length(x)
# perform linear regression
lr = lm(yt~xt)
r2 = summary(lr)$r.squared
# store intercept and slope
intercept=lr$coeff[1]
slope=lr$coeff[2]
# calculate mean and other stats
xm = mean(xt)
sxy = sum(xt*yt) - sum(xt)*sum(yt)/n
sxx = sum(xt^2) - sum(xt)^2 / n
syy = sum(yt^2) - sum(yt)^2 / n
MSE = (syy - slope * sxy) / (n - 2)
# any x values for projections?
if (length(xp) > 0) {
  # combine observed values of x with ones used for projections
  xpr = c(x,xp)
  # calculate pseudo-y array so the graph will scale to
  # include projections
  ypr = c(y, ify(intercept + slope * fx(xp)))
}
else {
  # just copy observations
  xpr = x
  ypr = y
}
# get the sort order of the new array
sort.ord = order(xpr)
# sort x and y arrays
xpr = xpr[sort.ord]
ypr = ypr[sort.ord]
# calculate delta y^ for CI
p = 1 - (1 - probab)/2
delta.y = qt(p, n-2) * sqrt(MSE*(1/n+(fx(xpr)-xm)^2/sxx))
delta2.y = qt(p, n-2) * sqrt(MSE*(1+1/n+(fx(xpr)-xm)^2/sxx))
# calculate transformed x including projections
xt2 = fx(xpr)
# calculate Y^
yt2 = intercept + slope * xt2
# calculate Y^ CI
yhat.hi = yt2 + delta.y
yhat.lo = yt2 - delta.y
yhat2.hi = yt2 + delta2.y
yhat2.lo = yt2 - delta2.y
if (show.graph) {
  # turn on ask mode
  par(ask=TRUE)
  # plot transformed values
```



```
    plot(fx(xpr), fy(ypr), type="p", main=paste(main1,
       "with R^2=", as.character(round(r2,5))), col=col1)
    # replot projected points with different color
    points(fx(xp), intercept + slope * fx(xp), col=col2)
    # draw regression line
    abline(intercept, slope)
    # draw confidence interval for Y
    lines(fx(xpr), yhat.hi)
    lines(fx(xpr), yhat.lo)
    lines(fx(xpr), yhat2.hi, col="red")
    lines(fx(xpr), yhat2.lo, col="red")
    # plot untransformed data & projections
    plot(xpr, ypr, type="p", main=paste(main2,
       "with R^2=", as.character(round(r2,5))), col=col1)
    # replot projected points with different color
    points(xp, ify(intercept + slope * fx(xp)), col=col2)
    # draw regression curve
    lines(xpr,  ify(intercept + slope * fx(xpr)))
    # draw confidence interval for Y
    lines(xpr, ify(yhat.hi))
    lines(xpr, ify(yhat.lo))
    lines(xpr, ify(yhat2.hi), col="red")
    lines(xpr, ify(yhat2.lo), col="red")
    # turn off ask mode
    par(ask=FALSE)
  }
  # return linear regression object
  res = list(lm.obj=lr, r2=r2, x2=xpr, y2=ypr, yhat=ify(yt2),
             yhat.hi=ify(yhat.hi), yhat.lo=ify(yhat.lo))
  return (res)
}

if (exists("TEST.FX")) {
if (TEST.FX) {
x=1:10
y=x^2 + x * runif(1,-1,1)
x[1]=2
y[5]=20
y[9] = 111
fx = function(x) { log(x) }
fy = function(x) { log(x) }
ify = function(x) { exp(x) }
lr=linreg(x, y, fx, fy, ify, xp=c(1.5, 5.4, 11,12,15), show.graph=TRUE)
show(summary(lr$lm.obj))
}
}
```

The demo statements in the above code perform the following tasks:

1. Creates the vector x.
2. Creates the vector y using the squared values of x and a random noise.
3. Assigns erroneous values to x[1], y[5], and y[9].
4. Defines the function that transform the x and y values.
5. Define the function for the inverse transformation of y values.

6. Calls function linreg(). The call specifies the projection vector of 11, 12, and 15 and also assigns TRUE to the parameter show.graph.
7. Displays the summary of the lr$lm.obj element.

Save the above code in file linreg.r. Then set the variable TEST.FX to TRUE and load the function linreg() by executing the following command (which shows the following text output):

```
Waiting to confirm page change...
Waiting to confirm page change...

Call:
lm(formula = yt ~ xt)

Residuals:
     Min       1Q   Median       3Q      Max
-2.00201 -0.17683  0.08697  0.44940  1.03121

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)  -2.4822     0.8336  -2.978 0.017659 *
xt            3.2395     0.4970   6.518 0.000185 ***
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 0.8855 on 8 degrees of freedom
Multiple R-squared: 0.8415,     Adjusted R-squared: 0.8217
F-statistic: 42.49 on 1 and 8 DF,  p-value: 0.0001845
```

Figure 19 shows the linearized regression data, regression line, and confidence interval. The observed data appear as red points while the projected ones are in green. Figure 20 shows the untransformed regression data, regression curve, and confidence interval. The observed data appear as red points while the projected ones are in green. Notice that both figures show the values of the coefficient of determination in the graphs' titles.

Figure 19. Linearized regression plot.

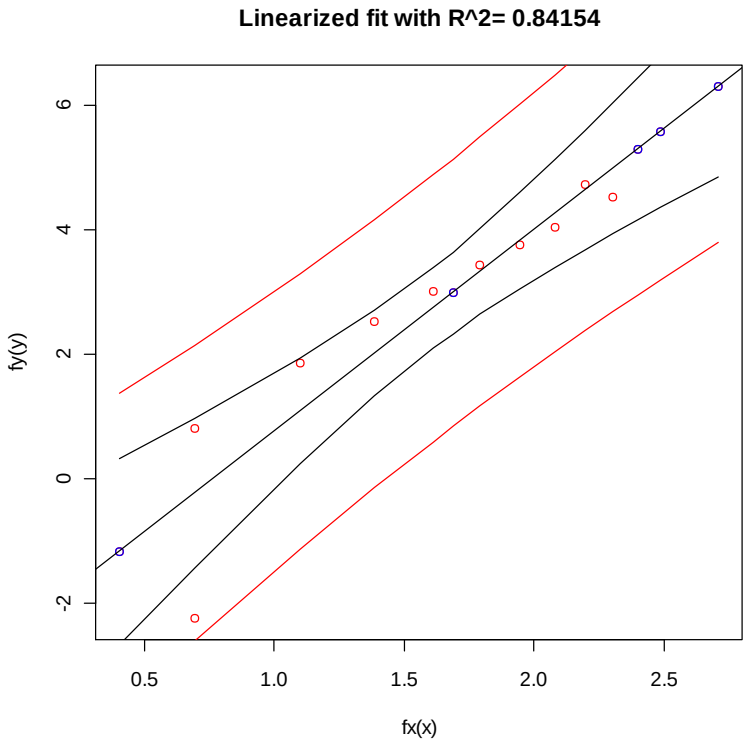
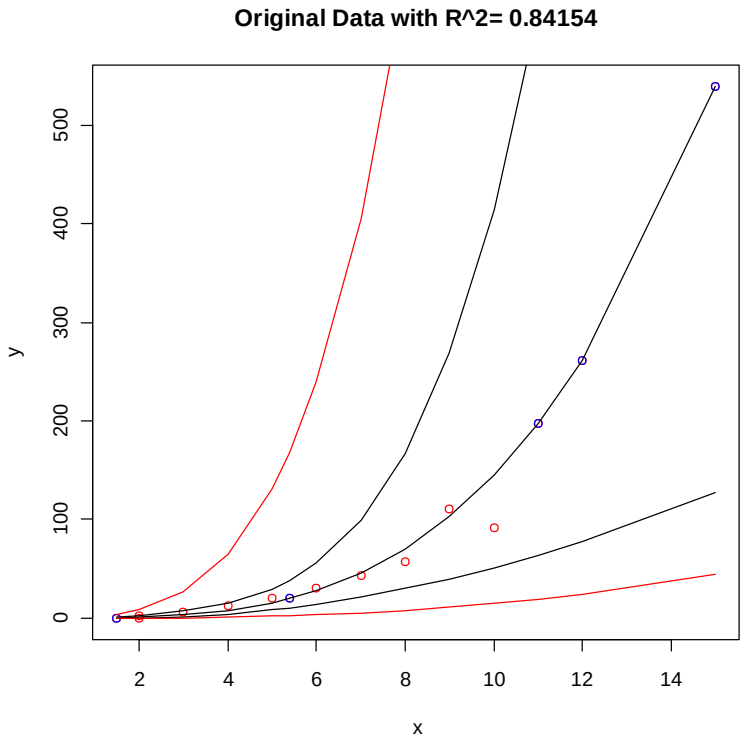


Figure 20. Untransformed data plot.



The Multiple Regression Function

The Function's Parameters

This section presents a function that performs multiple regression between two or more variables. The heading of the function mlreg() is:

```
mlreg = function(x, y, formula = "y~x",
                 fy="function(x) { x }", ify = "function(x) { x }")
```

The function has the following parameters:

- The parameter x is a vector for a single independent variable or a matrix of multiple independent variables.
- The parameter y is a vector for the dependent variable.
- The parameter formula specifies the formula for the regression. The default value for this parameter is for a simple linear regression.
- The parameters fy and ify are the functions that transform and inverse-transform the values the dependent variable. The default values for these parameters are linear functions (i.e. no transformation for the y values).

The Function's Tasks

The function mlreg() performs the following tasks:

1. Copies the values of vector y into vector y0.
2. Transforms the values in vector y.
3. Performs the multiple linear regression using the model in parameter formula.
4. Stores the regression residuals in variable Resid.
5. Calculates the untransformed y^ values.
6. Restores the original values in vector y.
7. Turns on the ask-user mode for graphs.
8. Plots the points for y^ vs. y by calling function plot(). The arguments for the x and y parameters are the vectors y and y.hat, respectively.
9. Draws a 45-degree line by calling the function abline(0,1).
10. Plots the points for residuals vs. y by calling function plot(). The arguments for the x and y parameters are the vectors y and Resid, respectively.
11. Draws the horizontal line through the zero value for the residuals.
12. Turns off the ask-user mode for graphs.
13. Returns the value in variable mlr.

The Source Code

Here is the source code for the function mlreg:

```
mlreg = function(x, y, formula = "y~x",
                 fy="function(x) { x }", ify = "function(x) { x }")
{
  #------------------------- Parameters -------------------
  #
  # x is a vector OR matrix of independent variables
  # y is a vector of the dependent variable
  # formula is the regression formula. An example for formula,
  #   when dealing with a vector x, is:
  #
  #     "y ~ I(x) + I(x^2) + I(1/x)"
  #
  #   An example for formula, when dealing with a matrix x of
  #   variables, is:
  #
  #     "y~I(x[,1])+I(x[,1]^2)+I(1/x[,1])"
  #
  #   The default value for the formula triggers a simple linear regreesion.
  #
  # fy is the function for transforming y values. The default function is
  #   for no transformation.
  # fy is the function for the inverse transformation of y values. The
  # default function is for no transformation.

  #
  # Function plots y^ vs. y and then Residuals vs. y.
  # Function returns the lm() regression object.
```



```
  # make a copy of vector y
  y0 = y
  # transform vector y
  y = fy(y)
  # perform multiple linear regression
  mlr = lm(formula)
  # get the residuals
  Resid = mlr$residuals
  # calculate Y^ for untransformed y values
  y.hat = ify(y - Resid)
  y = y0 # restore vector y to untransformed values
  par(ask=TRUE)
  plot(y, y.hat, type = "p", main = "Y^ vs. Y")
  abline(0,1)
  plot(y, Resid , type = "p", main = "Residual vs. Y")
  abline(h=0)
  par(ask=FALSE)
  return (mlr)
}

if (exists("TEST.FX")) {
if (TEST.FX) {
x1 = runif(10,1,10)
x2 = runif(10,0,2)
x3 = runif(10,2,4)
y = 10 + x1 + 2*x2^2 + 5/x3
for (i in 1:length(y))
 y[i] = y[i] + 10 * runif(1,-1,1)
x = matrix(c(x1,x2,x3),nrow=10,ncol=3,byrow=FALSE)
fy = function(x) { x }
ify = function(x) { x }

mlr = mlreg(x, y, "y~I(x[,1])+I(x[,2]^2)+I(1/x[,3])")
show(mlr)
show(summary(mlr))
}
}
```

The demo statements in the above code perform the following tasks:

1. Creates the random vectors x1, x2, and x3.
2. Creates the vector y based on the values of vectors x1, x2, and x3, plus a random noise.
3. Creates the matrix x by combining vectors x1, x2, and x3.
4. Defines the function that transforms the y values.
5. Define the function for the inverse transformation of y values.
6. Calls function mlreg(). The call specifies the formula of "y~I(x[,1])+I(x[,2]^2)+I(1/x[,3])".
7. Displays the summary of the regression result.

Sample Run

Save the above code in file mlreg.r. Then set the variable TEST.FX to TRUE and load the function mlreg() by executing the following command (which shows the following text output):


```
Waiting to confirm page change...
Waiting to confirm page change...

Call:
lm(formula = formula)

Coefficients:
(Intercept)    I(x[, 1])  I(x[, 2]^2)  I(1/x[, 3])
    12.0952       0.9273       2.0482       0.2579


Call:
lm(formula = formula)

Residuals:
     Min       1Q   Median       3Q      Max
-0.57086 -0.11620 -0.01613  0.19698  0.42975

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept) 12.09520    0.60011  20.155 9.69e-07 ***
I(x[, 1])    0.92732    0.06017  15.413 4.72e-06 ***
I(x[, 2]^2)  2.04819    0.10269  19.946 1.03e-06 ***
I(1/x[, 3])  0.25786    1.68349   0.153    0.883
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.3479 on 6 degrees of freedom
Multiple R-squared: 0.9902,     Adjusted R-squared: 0.9854
F-statistic:   203 on 3 and 6 DF,  p-value: 2.024e-06
```

Figure 21 shows the Y^ vs. Y plot. Figure 22 shows the residuals plot.

Figure 21. Y^ vs Y

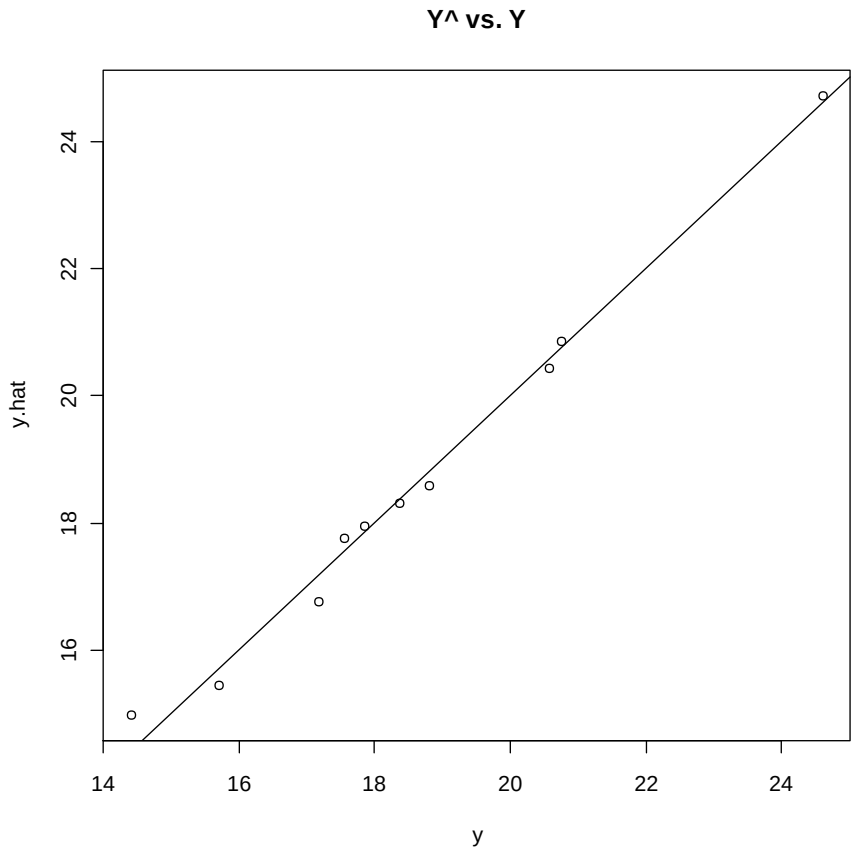
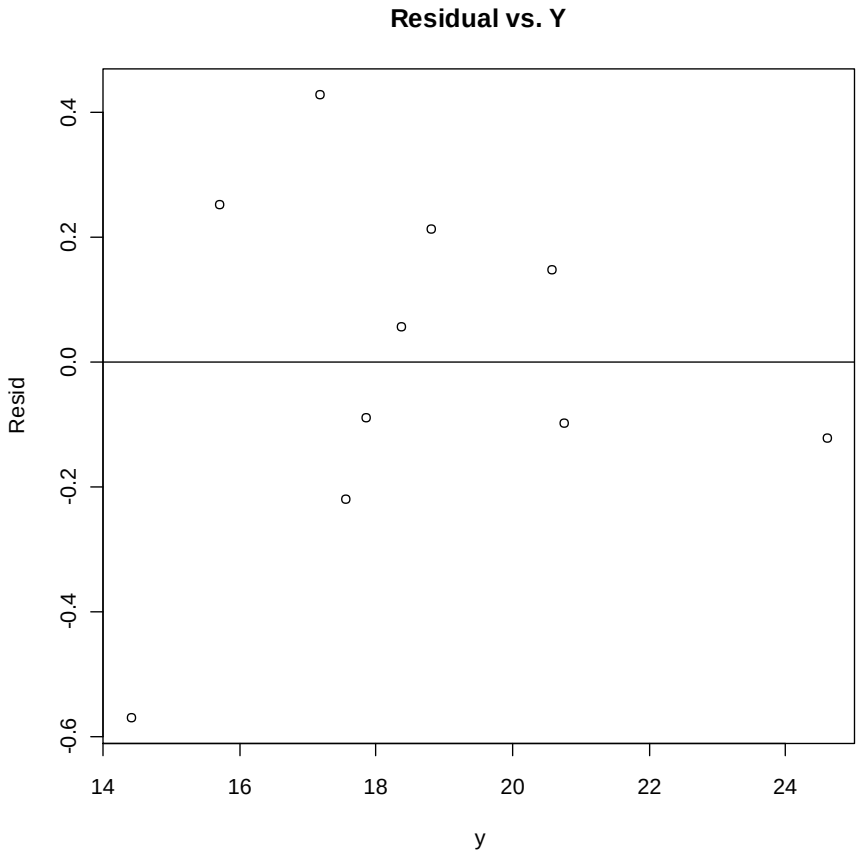


Figure 22. Residual plots.



The Polynomial Regression Function

The Function's Parameters

This section presents the polyreg() function which performs a polynomial regression. The heading for the declaration of function polyreg() is:

```
polyreg = function(x, y, order,
                   fy="function(x) { x }", ify = "function(x) { x }")
```

The function polyreg() has a parameter list that resembles that of function mlreg(). The only difference is that the parameter order (the order of the fitted polynomial) in function polreg() replaces the parameter formula in function mlreg(). The function uses the parameter order to dynamically create the formula for the fitted polynomial. The function also displays the string for the formula it creates. This output helps you confirm the order of the polynomial being fitted.

The Function's Tasks

The function polyreg() performs the following tasks:

1. Stores the number of elements of vector x in the variable n.
2. Sorts the values in vectors x and y, using the sort order of vector x. This task calls function order() and pplies the result of that function to both vectors x and y.
3. Copies the values of vector y into vector y0.
4. Transforms the values in vector y.
5. Performs the polynomial regression using the model in parameter formula. This task stores the result of calling function lm() in the variable plr.
6. Stores the regression residuals in variable Resid.
7. Calculates the untransformed y^ values.
8. Restores the original values in vector y.
9. Turns on the ask-user mode for graphs.
10. Plots the points for the observations in vectors x and y.
11. Sets the variable n2 to be 100 * n. This variable stores the number of points used to draw the curve for the polynomial.
12. Initializes the vector y2 with dummy values. The size of this vector equals the value in variable n2.
13. Initializes the vector x2 to have n2 points ranging from the smallest to the largest values in vector x. This task calls function which.min() and which.max() to obtain the range of values for x.
14. Calculates the values for vector y2 as points on the polynomial in the range specified by vector x2. This task uses nested for loops.
15. Draws the curve for the polynomial by calling function lines(x2, y2).
16. Plots the points for y^ vs. y by calling function plot(). The arguments for the x and y parameters are the vectors y and y.hat, respectively.
17. Draws a 45-degree line by calling the function abline(0,1).
18. Plots the points for residuals vs. y by calling function plot(). The arguments for the x and y parameters are the vectors y and Resid, respectively.
19. Draws the horizontal line through the zero value for the residuals.
20. Turns off the ask-user mode for graphs.
21. Returns the value in variable plr.

The Source Code

Here is the source code for the function polyreg() and the demo statements:

```
polyreg = function(x, y, order,
                   fy="function(x) { x }", ify = "function(x) { x }")
{
  #-------------------------- Parameters -------------------
  #
  # x is a vector of independent variables
  # y is a vector of the dependent variable
  #
  # fy is the function for transforming y values. The default function is
  #   for no transformation.
```



```
  # fy is the function for the inverse transformation of y values. The
  # default function is for no transformation.

  #
  # Function plots y^ vs. y and then Residuals vs. y.
  # Function returns the lm() regression object.

  # store the number of observations
  n = length(x)
  # sort the x and y vectors
  sort.order = order(x)
  x = x[sort.order]
  y = y[sort.order]
  # make a copy of vector y
  y0 = y
  # transform vector y
  y = fy(y)
  formula = "y ~ x"
  for (i in 2:order)
    formula = paste(formula, " + I(x^", as.character(i), ")", sep="")
  # perform multiple linear regression
  cat("Formula is: ", formula, "\n")
  plr = lm(formula)
  # get the residuals
  Resid = plr$residuals
  # calculate Y^ for untransformed y values
  y.hat = ify(y - Resid)
  y = y0 # restore vector y to untransformed values
  par(ask=TRUE)
  # plot the observations
  plot(x, y, type="p", main=paste("Y = polynomial of X of order",
       as.character(order)))
  # calculate a higher number of samples for drawing the polynomial curve
  n2 = 100 * n
  # initialize the y2 vector
  y2 = rep(0,n2)
  # initialize the x2 vector
  x2 = seq(which.min(x), which.max(x), length.out=n2)
  # calculate vectors for the polynomial curve
  for (i in 1:n2) {
    y2[i] = plr$coefficients[1]
    for (j in 2:(order+1)) {
      y2[i] = y2[i] + plr$coefficients[j] * x2[i]^(j-1)
    }
  }
  # draw polynomial
  lines(x2, y2)
  plot(y, y.hat, type = "p", main = "Y^ vs. Y")
  abline(0,1)
  plot(y, Resid , type = "p", main = "Residual vs. Y")
  abline(h=0)
  par(ask=FALSE)
  return (plr)
}

if (exists("TEST.FX")) {
```



```
if (TEST.FX) {
x = runif(10,1,10)
y = 10 + x + 2*x^2 + 0.1*x^3
for (i in 1:length(y))
 y[i] = y[i] + 50 * runif(1,-1,1)
fy = function(x) { x }
ify = function(x) { x }
plr=polyreg(x, y, 3)
show(plr)
show(summary(plr))
}
}
```

Sample Run

The test code erforms a cubic fit. Save the above code in file polyreg.r. Then set the variable TEST.FX to TRUE and load the function polyreg() by executing the following command (which shows the following text output):

```
Formula is:  y ~ x + I(x^2) + I(x^3)
Waiting to confirm page change...
Waiting to confirm page change...
Waiting to confirm page change...

Call:
lm(formula = formula)

Coefficients:
(Intercept)            x       I(x^2)       I(x^3)
    504.969     -259.181       45.243       -2.196


Call:
lm(formula = formula)

Residuals:
    Min      1Q  Median      3Q     Max
-39.578 -14.016   7.794  16.779  23.793

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)  504.969    378.979   1.332    0.231
x           -259.181    214.036  -1.211    0.271
I(x^2)        45.243     37.808   1.197    0.277
I(x^3)        -2.196      2.104  -1.043    0.337

Residual standard error: 26.7 on 6 degrees of freedom
Multiple R-squared: 0.9194,     Adjusted R-squared: 0.8791
F-statistic: 22.81 on 3 and 6 DF,  p-value: 0.001111
```

Figure 23 shows the polynomial curve. Figure 24 shows the Y^ vs. Y plot. Figure 25 shows the residuals plot.

Figure 23. The fitted polynomial

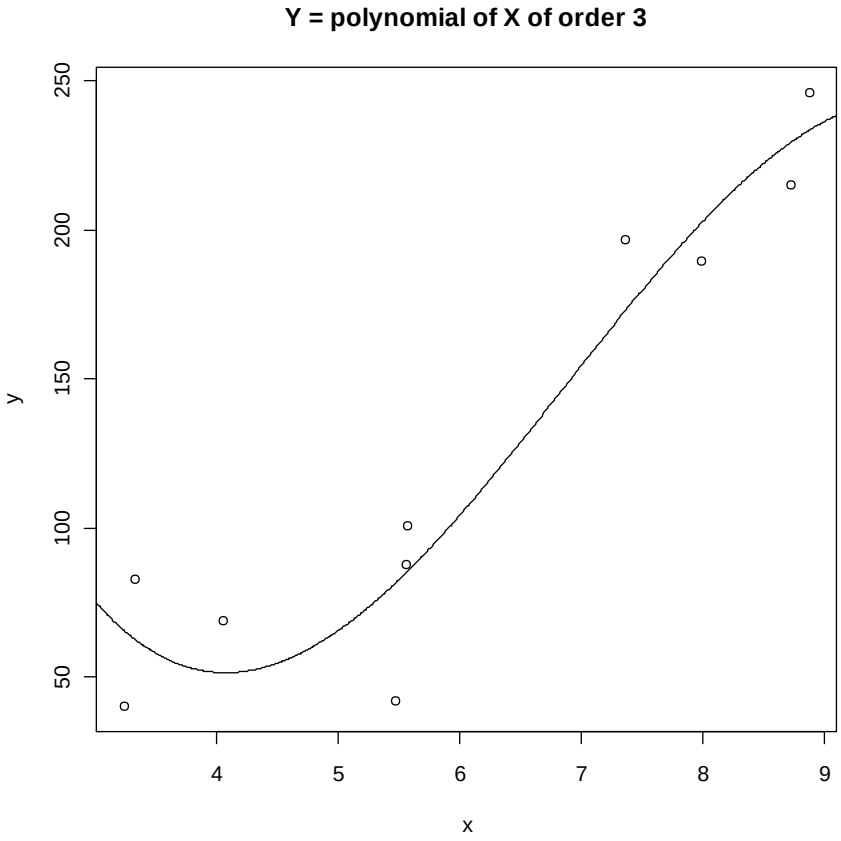


Figure 24. The Y^ vs. Y plot for a polynomial fit.

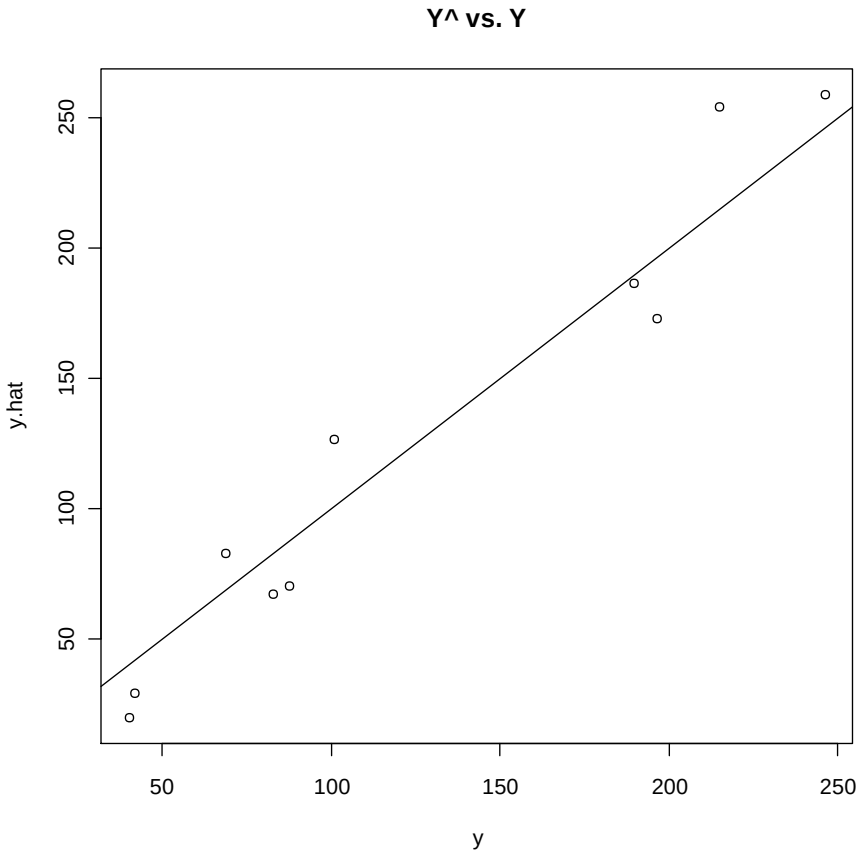


Figure 25. The residuals plot for a polynomial fit.