



Using R for Monte Carlo statistical analysis

Stanley Humphries, Ph.D.

Field Precision LLC

E mail: techinfo@fieldp.com

Internet: <https://www.fieldp.com>

Contents

1	Introduction	3
2	Working with the R console: vectors and data frames	5
3	Creating and reading CSV files	10
4	Linear curve fitting and plotting, Part A	16
5	Linear curve fitting and plotting, part B	20
6	Working with RStudio: multi-dimensional fitting	24
7	Nonlinear fitting and parameter estimation	31
8	Importing GamBet files into R	36
9	Creating GamBet SRC files with R	43
10	Generating arbitrary distributions	51

1 Introduction

Statistical analysis is the core of Monte Carlo codes like **GamBet**. Although our **GenDist** program performs basic calculations, high-power statistical packages are useful for intensive work. There are dozens of statistical programs, many of them available for free. A comprehensive list is available at

https://en.wikipedia.org/wiki/List_of_statistical_packages

In addition, mathematics programs like **SciDAVis** (<https://sourceforge.net/projects/scidavis>) perform many statistical functions.

In general, the interfaces of technical programs center on a console window where you may enter commands interactively. Commands range from simple arithmetic to complex statistical operations. The programs can read data from text files or spreadsheets and follow a sequence of commands from a script file. There are several options to export results and most packages have plotting capabilities. There are advantages to using standard programs rather than writing your own:

They feature extensive computational resources, programmed by experts.

There is less chance of error because routines have been thoroughly tested.

In the long run, the setup times are short compared to building your own programs.

On the other hand, the advantages come at a price.

Powerful technical packages have steep learning curves. Each program has its own command structure, so learning a program is comparable to acquiring a new computer language.

There are peculiarities of syntax and organization logic. Inexperienced users may suffer hours of frustration searching for errors.

The power and numerous options of the programs can initially be a drawback, overwhelming new users.

Matching program capabilities to your specific needs can be challenging.

Fortunately, perseverance pays off. After several days of effort, the moment arrives where everything starts to make sense. One goal of this tutorial is to help you reach that moment sooner. The other is to demonstrate how to handle **GamBet** data and how to perform analyses for Monte Carlo calculations.

The first issue is which mathematics package to choose. There are dozens available, each claiming to be the best. It is certainly not feasible to test them all. I am going to concentrate on a single resource, the **R** package, for the following reasons:

It is freely available for all computer platforms.

It has a large international user base.

R has powerful standard features with add-on packages for sophisticated work.

An integrated development environment is available.

Most important, there is good documentation including textbooks and online references.

You can obtain executables of the basic **R** package for most operating systems at:

<http://www.r-project.org/>

There are links on the site to online documentation. The **R Reference Card** is an invaluable resource:

<http://cran.r-project.org/doc/contrib/Short-refcard.pdf>

After installing the **R** package, I recommend downloading and installing the **RStudio** integrated development environment:

<https://www.rstudio.com/products/RStudio/>

The following sections are in the form of short lessons. They demonstrate essential **R** techniques for **GamBet** users. The emphasis is on curve fitting and parameter estimation. A topic of particular importance is how to read **GamBet** source (SRC) files directly into the **R** environment.

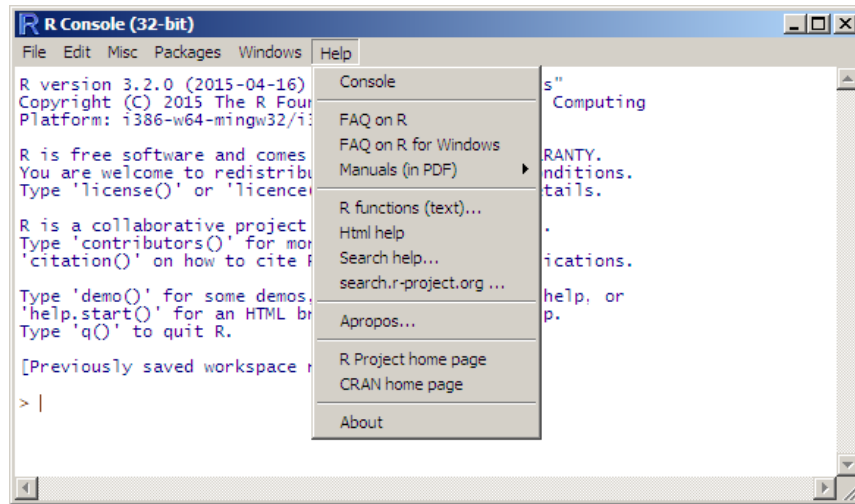


Figure 1: R console window.

2 Working with the R console: vectors and data frames

To begin, we'll work with the basic **R** console. Section 6 describes the enhancements available with the **RStudio** integrated development environment. Run **R** with the desktop shortcut to launch the window shown in Fig. 1. The program writes a stock message that you'll always want to erase. Click the important *Help* menu entry to display options for learning **R**. Clicking the *Console* option brings up a short window of information. The command to clear the window is *Control-L*.

Program output is shown in blue and things you enter are shown in red. The prompt `>` indicates that the program is ready for input. Type

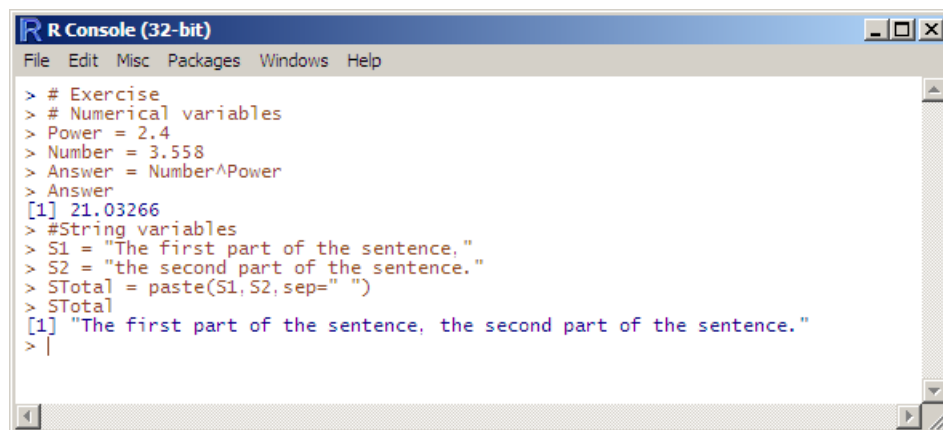
```
Power = 2.4
```

followed by *Enter*. This operation defines a simple numerical variable *Power* with the value 2.4. Anything you define is referred to as an *object*. Objects may include string variables, vectors and more complex data assemblies. To see a list of objects, type

```
objects()
```

To see the value of *Power*, type

```
Power
```

The image shows a screenshot of an R Console window titled "R Console (32-bit)". The window has a menu bar with "File", "Edit", "Misc", "Packages", "Windows", and "Help". The console displays the following commands and their outputs:

```
> # Exercise
> # Numerical variables
> Power = 2.4
> Number = 3.558
> Answer = Number^Power
> Answer
[1] 21.03266
> #String variables
> S1 = "The first part of the sentence,"
> S2 = "the second part of the sentence."
> STotal = paste(S1,S2,sep=" ")
> STotal
[1] "The first part of the sentence, the second part of the sentence."
> |
```

Figure 2: Combining string objects.

Because **R** is case sensitive, the capital *P* in *Power* is significant. The result is

```
[1] 2.4
```

The initial [1] is a display index, useful if the object contains many items.

To illustrate arithmetic operations, type

```
Number = 3.558
Answer = Number^Power
```

defining two new objects. Entering the name of the final variable gives the value

```
> Answer
[1] 21.03266
```

R also supports string variables and operations. For example,

```
S1 = "The first part of the sentence,"
S2 = "the second part of the sentence."
STotal = paste(S1,S2,sep=" ")
STotal
```

Here's a useful shortcut – it's not necessary to retype the examples in this document. Simply copy the text information above and paste it into the console. **R** executes the commands as they are transferred from the clipboard. The resulting console entries are shown in Fig. 2.

The **R** paste function combines two or more strings with the specified separation character. In the example, a space was used. No characters are added if `sep=""`.

Except for the simplest calculations, working in the console mode is impractical. For serious calculations, the best approach is to build and to test a sequence of commands with the option to save them. For this, we will work with **R** scripts (`filename.r`). Choose *File/New script* to open a new window for the script editor. Note that the console window remains active. Copy and paste the following statement into the script window:

```
rm(list=objects())
```

Nothing happens in the console window. In the passive mode, the script window acts as a basic text editor. Now, put the cursor on the first line and press *Control-R* (run). The command is transmitted to the console, which performs the action. Go back to the console and type

```
objects()
```

The response `character(0)` indicates that all defined objects have been removed. The **R** interface is quite flexible. You can move back and forth between the console and script editor, entering commands directly, editing the script or sending selected commands from the script.

Copy and paste the following content to the script window:

```
# Clear any objects loaded from a previous session
rm(list=objects())
# Define two new vectors
xvals = c(1.0,1.5,2.0,2.5,3.0,3.5,4.0)
yvals = c(3.4,2.5,4.1,3.6,5.2,4.9,6.3)
# Put them into a data frame
TestSet = data.frame(V1,V2)
# Show the objects that were created
objects()
# Calculate a function
mean(TestSet$yvals)
# Plot the data frame
plot(TestSet$xvals,TestSet$yvals,type="b")
```

The set of commands illustrates some critical **R** concepts and defines a practical calculation. Save the script for future reference. Let's discuss the meanings of the commands before running them.

The `rm()` command removes objects specified by the list of names in the argument. Here, a *list* is a collection of objects, a generalized vector. The output of the function `object()` is a list of the names of all defined objects. Hence, the form of the `rm()` command removes everything. The lines

```
xvals = c(1.0,1.5,2.0,2.5,3.0,3.5,4.0)
yvals = c(3.4,2.5,4.1,3.6,5.2,4.9,6.3)
```

create two vector objects named *xvals* and *yvals*. A vector is an ordered set of like items. The `c()` function combines a group of items into a vector, in this case a set of seven numbers. The expression `yvals[4]` returns the number 3.6..

The *data frame*¹ is an important object in **R**. It contains one or more vectors of any type (numbers, strings,...). The restriction is that all vectors must have the same length (number of entries). The data frame structure is appropriate for holding experimental results as well as the contents of **GamBet SRC** files. The command,

```
TestSet = data.frame(xvals,yvals)
```

combines the two vectors into a data frame named *TestSet*, which can be thought of as a matrix with two columns and seven rows. It is important to note that the names of things are important in **R**. By default, the column names in *TestSet* are set to the names of the component vectors, *xvals* and *yvals*². You can refer to columns by their names. For example, the expression *TestSet yvals* has the same content as the *yvals* vector. The command

```
mean(TestSet$yvals)
```

gives the average value of the components of the second vector, 4.285714.

The final command

```
plot(TestSet$xvals,TestSet$yvals,type="b")
```

calls for a simple plot of data points with *TestSet xvals* along the abscissa and *TestSet yvals* along the ordinate. As with almost all **R** functions, the user may set many options in `plot()` to over-ride defaults. In this case, the option `type="b"` specifies that the plot should contain both point symbols and connecting lines.

To step through each script command in sequence, put the script editor cursor in the first line and then press *Control-R* several times. To execute the full script, highlight all the lines and press *Control-R* once. Figure 3 shows the final state of the console window with the display of defined objects and the calculated mean. **R** has also opened up a graphics window to create the plot.

¹Some references use the term data set to refer to the content of the vectors and the term data frame to refer to the object that holds the content. This distinction seems unnecessary, so I will apply the term data frame to designate both the contents and the container.

²The `data.frame` command has an option to set the column names explicitly.

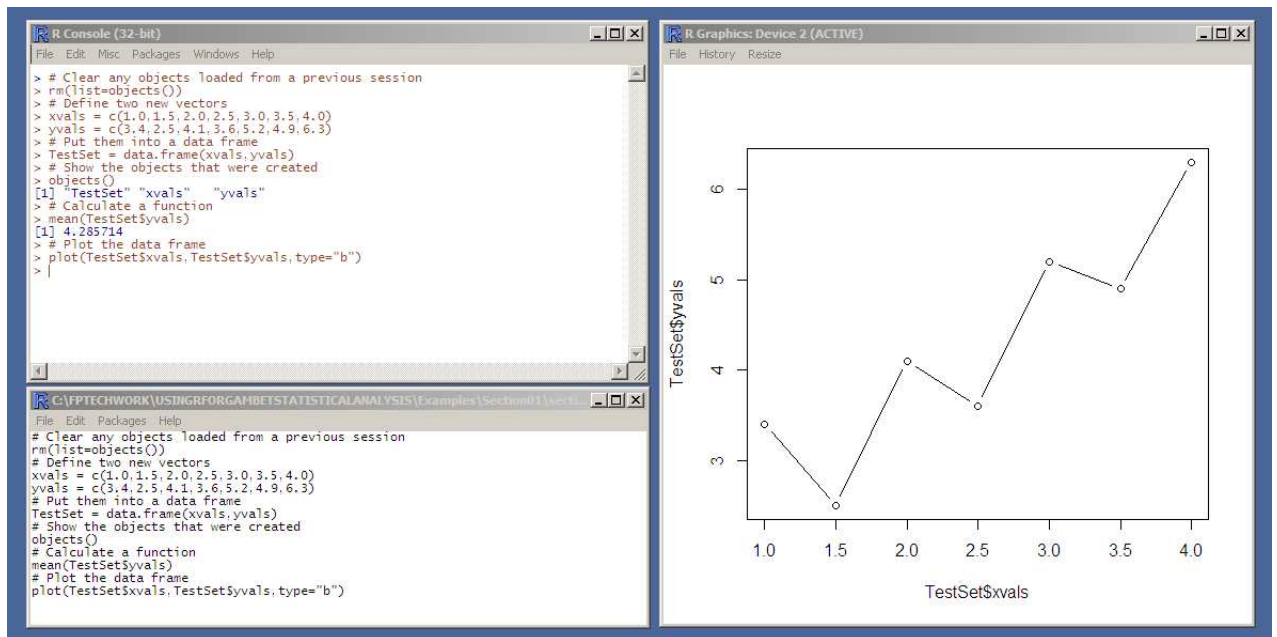


Figure 3: State of the console, script and plot windows after executing the script.

Finally, to run a script from the console without using the script editor, use a command of the form

```
source("C:/RExamples/Exercise0101.R")
```

Note that the path specification must contain forward slashes rather than the backslashes used in **Windows**. In the plot of Fig. 3, the data are crying out for an estimate of the slope and intercept. We'll tackle that issue after we learn about loading data into **R**.

3 Creating and reading CSV files

Data from experiments and computer outputs may include thousands of items, so it is clearly impractical to enter values in **R** by typing them. Generally, data are recorded in files and we use special **R** commands to read them. **R** recognizes some binary formats like **Excel**, but the most reliable approach is to record the data in a text file. Properly formatted text files act as a universal data medium that can be created or read by almost any program. There are no issues of software version changes and, most important, there are no mysteries. You can check the files with any text editor.

The most common format for storing data in a text file is comma-separated-variable, or **CSV**. In a general sense, **CSV** could apply to any file that uses commas as a delimiter. We'll use a more restrictive definition. A **CSV** file contains a number of data lines (rows). A line consists of text terminated by a carriage return and/or linefeed character. Each line is divided into the same number of text entries (columns) separated by commas. The text entries may represent strings or numbers. This organization is reminiscent of spreadsheet data or **R** data frames – it is not surprising that both types of programs can read and write directly to **CSV** files.

In this section we'll use a spreadsheet to create data for a statistical analysis, save the results as a **CSV** file, read the file into **R** and perform several useful calculations. Figure 4 shows a spreadsheet setup to generate points in the range $0.0 \leq x \leq 20.0$ from the formula

$$y = 22.67 + 4.71x - 0.30x^2 + 0.01x^3. \quad (1)$$

Random noise uniformly distributed over the range $-2.5 \leq \Delta y \leq 2.5$ is added to exercise the statistical analysis functions of **R**. Note that a header line containing the strings x and y has been included. The choice of names will be important in **R** – we will make consistent references to them when defining **R** operations.

To create a **CSV** file from **OpenOffice Calc**, simply use *SaveAs* and choose the format option *Text CSV*. Here is a section of the resulting file:

```
x,y
0,24.8324297602
0.25,21.8520734699
0.5,24.207368359
...
```

There are 2 data columns and 81 rows (not counting the header). Most programs that read and write **CSV** files (including **Calc** and **R**) have flexible input parsers that recognize strings and numbers in any valid format, including scientific notation.

Here is the **R** script that we'll be using:

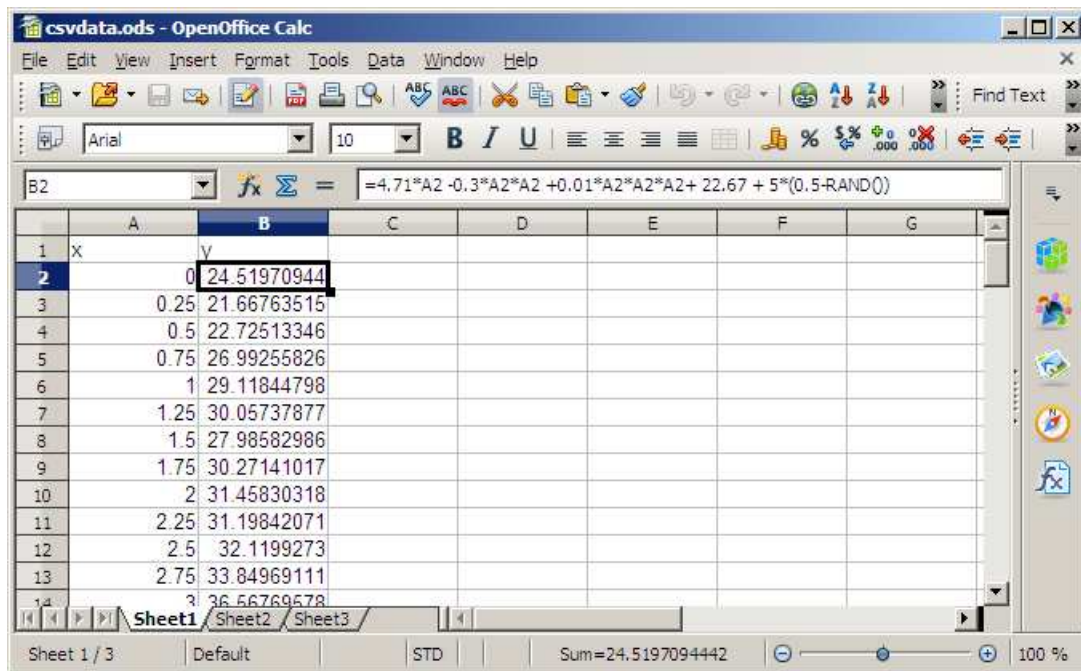


Figure 4: Spreadsheet to generate the test data.

```
setwd("C:/RExamples/CSVStudies")
TestData = read.csv("csvdata.csv",header=TRUE)
plot(TestData$x,TestData$y)
Check15 = subset(TestData,x >= 12.5 & x <= 17.5)
Check15$x = Check15$x - 15.0
Fit15 = lm(y~x,Check15)
summary(Fit15)
plot(Check15$x,Check15$y)
abline(coef(Fit15))
write.csv(Check15,file="points15.csv",row.names=FALSE)
```

Run **R** and choose *File/New script*. Copy and paste the lines above into the script editor window. (It's a good idea to save the script file for later reference.) We'll step through the lines to learn some useful **R** operations. Set the cursor in the script editor to the first line and press *Control-R* to advance through each step.

It's always a good idea to organize input/output files in a single directory. The following command sets the working directory so you don't have to enter full paths for file operations:

```
setwd("C:/RExamples/CSVStudies")
```

The next command (the core of this section) loads the CSV data file into a data frame named *TestData* with 2 columns and 81 rows.

```
TestData = read.csv("csvdata.csv",header=TRUE)
```

The columns have the names x and y . The option `header=TRUE` signals to **R** that the first line of the file contains the column names. If the file has no header, there are options in the `read.csv` command to set the names explicitly. Go to the console window and type `TestData` to confirm that the information was loaded. The next command

```
plot(TestData$x,TestData$y)
```

creates the plot of the input data shown as the top graph in Fig. 5. It is a noisy version of the curve of Eq. 1.

In a following article, we'll see how to get an estimate of all the polynomial coefficients in Eq. 1. For now, we'll concentrate on a local interpolation to estimate the value and slope at a single measurement point, $x = 15$. For this, we'll include only data points near the measurement point. This command,

```
Check15 = subset(TestData,x >= 12.5 & x <= 17.5)
```

produces a data frame *Check15* that is a subset of *TestData*. The selection criterion is that values in the column named x must be greater than or equal to 12.5 and less than or equal to 17.5. The range is shown by dashed red lines in Fig. 5. *Check15* contains 2 columns named x and y with 21 rows. Again, type *Check15* in the console to review the content.

We'll use a routine that finds the slope and intercept of the most probable straight line for the data. The intercept equals the desired interpolation if the measurement point corresponds to $x = 0.0$. This command shifts all values in the x column of the new data frame by -15.0:

```
Check15$x = Check15$x - 15.0
```

We're ready to do the fit, using the powerful and versatile `lm()` command:

```
Fit15 = lm(y~x,Check15)
```

The command has two arguments: 1) a specification of the fitting *formula* and 2) the data frame to be used. The formula specification $y \sim x$ is a subtle concept and merits a detailed explanation. The important point is that a formula is not the fitting equation, but rather an expression of dependencies in the fit.

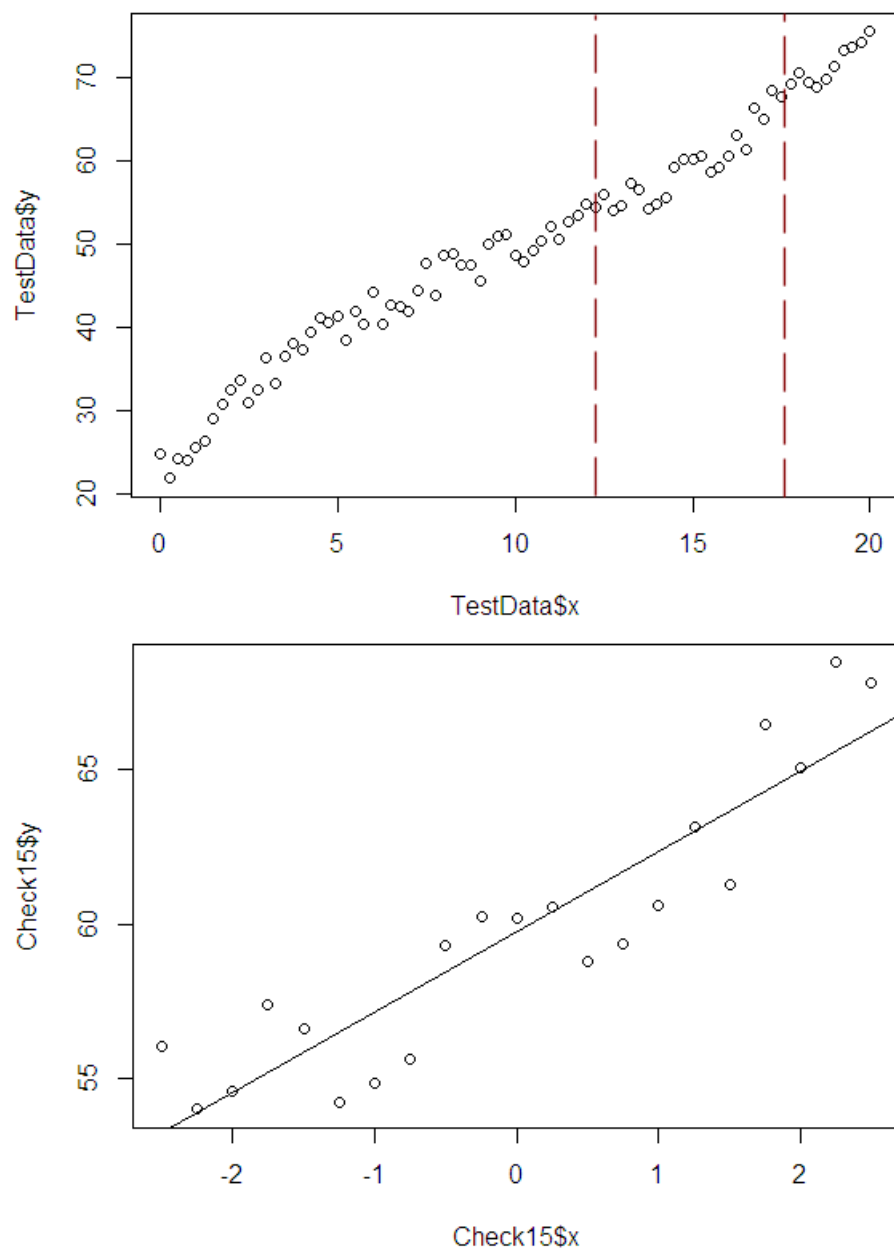


Figure 5: Top: input data, red lines show the range used for the interpolation. Bottom: the local interpolation data and the fitting line.

The \sim symbol designates a functional relationship rather than an equality.

The y to the left of \sim specifies that values in the column named y correspond to the dependent variable.

The x to the right of \sim specifies that values in the column named x correspond to the independent variable.

The appearance of the single x to the right of \sim indicates that the fit should set a linear relationship, $y = a + bx$.

We'll have a chance to discuss more complex formula specifications in following articles.

The `lm()` operation creates *Fit15*, a special object. It is neither a vector nor a data frame, but rather a set of fitting coefficients and tolerances in a format specific to **R**. There are special operations in **R** to display or to extract information from fitting objects. For example, the operation `summary(Fit15)` produces the following display:

Call:

```
lm(formula = y ~ x, data = Check15)
```

Residuals:

Min	1Q	Median	3Q	Max
-2.3474	-2.1410	0.1371	1.1435	2.8666

Coefficients:

	Estimate	Std. Error	t value	Pr(> t)
(Intercept)	59.7433	0.4041	147.851	< 2e-16 ***
x	2.5978	0.2669	9.732	8.13e-09 ***

Signif. codes:

0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 1.852 on 19 degrees of freedom

Multiple R-squared: 0.8329, Adjusted R-squared: 0.8241

F-statistic: 94.72 on 1 and 19 DF, p-value: 8.133e-09

This is probably more information than you'd want to know. The important results are that 1) the interpolated value at the point is $y = 59.7433 \pm 0.4041$ and 2) the slope is $dy/dx = 2.5978 \pm 0.2669$. The predictions from Eq. 1 are $y = 59.7557$ and $dy/dx = 4.71 - 0.6 \times 15 + 0.03 \times 15^2 = 2.4600$.

Another function to extract information from a fit is `coef()`. For a linear fit, the routine returns two numbers, the intercept and slope. It happens that

there is a plotting command in **R**, `abline()`, that adds a straight-line plots to an existing plot given an intercept and slope. We can therefore understand the following command set:

```
plot(Check15$x,Check15$y)
abline(coef(Fit15))
```

The `plot()` command displays y versus x values for the data over the range of x in *Check15*. The `abline` command adds the straight line determined by the fitting operation. The lower plot in Fig. 5 shows the result.

Finally, we can create a CSV file with **R**:

```
write.csv(Check15,file="points15.csv",row.names=FALSE,quote = FALSE)
```

The options indicate that row names and quotation marks should be excluded. Here is the initial section of the file `point15.csv`:

```
x,y
-2.5,56.0587661858
-2.25,54.0067949137
-2,54.5795616217
-1.75,57.4048454528
-1.5,56.623110328
...
```

If we didn't include the options, the output of the `write.csv()` function would look like this:

```
"","x","y"
"51",-2.5,56.0587661858
"52",-2.25,54.0067949137
"53",-2,54.5795616217
"54",-1.75,57.4048454528
"55",-1.5,56.623110328
...
```

4 Linear curve fitting and plotting, Part A

In this section and the next, we'll discuss linear curve fitting (or parameter estimation). Here, the term *linear* does not mean we are restricted to linear fitting functions, but rather that the parameters we seek are linear multipliers of the functions. To illustrate the distinction, suppose we have a set of measurements of a dependent quantity $y_i(x_i)$. We hypothesize that the measurements can be approximated by a variation of the form:

$$y = a_1 f_1(x) + a_2 f_2(x) + \dots, \quad (2)$$

where $f_1(x), f_2(x), \dots$, are known functions of any type. The task is to find the set of parameters a_1, a_2, \dots , such that Eq. 2 has the highest probability of matching the measured points. Given that Eq. 2 represents a linear fit, what is a nonlinear fit? To illustrate, suppose we wanted to find a resonance response buried in a noisy signal. In this case, the proposed function would be

$$y = a_1 \exp\left(-\frac{(x - a_2)^2}{2 a_3^2}\right). \quad (3)$$

We seek three parameters, only one of which is a linear multiplier of the function. The other two parameters are integrated within the function. While the linear problem is deterministic, nonlinear calculations require iterative methods that may or may not converge. Section 7 covers nonlinear fitting in **R**.

This section and the next one cover *univariate* linear fitting, $y = f(x)$. In a subsequent section, we'll advance to multivariate fitting, $z = g(x, y)$. To start, let's revisit the example from the previous section. The points in Figure 6 represent the raw data in the file `csvdata.csv`. The points follow a curve with two inflection points, so we suspect that a third-order polynomial would be a good guess:

$$y = a + bx + cx^2 + dx^3. \quad (4)$$

This function satisfies the definition of a linear fit (Eq. 2).

Copy and paste the following text into the script window of **R**:

```
setwd("C:/RExamples/LinFitStudies")
TestData = read.csv("csvdata.csv",header=TRUE)
FitPoly = lm(y~I(x)+I(x^2)+I(x^3),TestData)
summary(FitPoly)
CheckPoints = data.frame(x = c(0.0,5.0,10.0,15.0,20.0))
Pred = predict(FitPoly,newdata=CheckPoints)
Pred
```

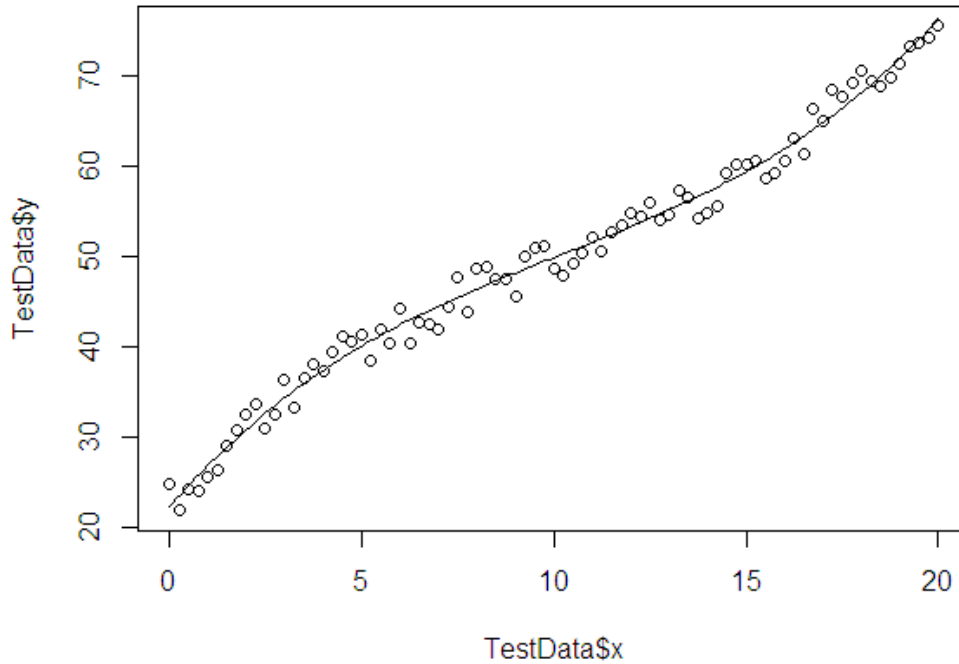



Figure 6: Test data with a third-order polynomial fit.

```
plot(TestData$x,TestData$y,type="p")
PlotSeq = seq(from=0.0,to=20.0,length.out=201)
PlotPos = data.frame(x=PlotSeq)
lines(PlotPos$x,predict(FitPoly,newdata=PlotPos))
```

As before, we set a working directory. It contains a copy of the test data prepared for the previous section. Save the script. Again we shall read the file contents to a data frame named *TestData* and apply the `lm()` command to produce the fitting object *FitPoly*:

```
FitPoly = lm(y~I(x)+I(x^2)+I(x^3),TestData)
```

The fitting formula is more complex than the first order function of the previous article:

```
y~I(x)+I(x^2)+I(x^3)
```

It implies that values in the column named *y* of *TestData* depend on the values in the column named *x*. The functional dependencies involve powers of *x* up to x^3 . Note the appearance of the command `I()`. It signals that the symbols inside the argument should be interpreted as arithmetic operations rather than formula specifications. There is some overlap between the

two. For example, in a formula specification the $+$ symbol does not designate addition, but rather signals that **R** should include another functional dependency³

The command `summary(FitPoly)` produces the following information:

	Estimate	Std. Error	Generating function
(Intercept)	22.278672	0.672675	22.67
I(x)	4.877190	0.293093	4.71
I(x^2)	-0.314646	0.034174	-0.30
I(x^3)	0.010301	0.001123	0.01

The coefficients of the original generating function have been included as the last column.

There are two more tasks to complete this example:

Check the interpolated value returned by the fitting function for comparison to the linear interpolation of the previous article.

Superimpose a plot of the predicted curve on the original data for a visual validity check.

The following command lines accomplish the first task:

```
[1] CheckPoints = data.frame(x = c(0.0,5.0,10.0,15.0,20.0))
[2] Pred = predict(FitPoly,newdata=CheckPoints)
[3] Pred
```

Line [1] uses the `data.frame()` and `c()` commands that we have already discussed. The `c()` command combines several position values (including $x = 15.0$) in a vector. The `data.frame()` command incorporates the vector in a data frame named *CheckPoints* with one column named *x*. Line [2] contains the command `predict()` which requires two arguments: 1) an **R** fitting object and 2) a data frame with a named column that corresponds to the independent variable of the fitting object. The `predict()` command returns a value of *y* for each *x* value. The final line produces the following listing of information in the console window:

```
1      2      3      4      5
22.27867 40.08612 49.88714 59.40764 76.37351
```

The fourth value corresponds to $x = 15.0$.

The following lines create the plot of Fig. 6:

³In the previous article, we used a function specification $y \sim x$ without the `I()` command because the expression did not include any common symbols. To be safe, it is best to use `I()` for all functional expressions.

```
[1] plot(TestData$x,TestData$y,type="p")
[2] PlotSeq = seq(from=0.0,to=20.0,length.out=201)
[3] PlotPos = data.frame(x=PlotSeq)
[4] lines(PlotPos$x,predict(FitPoly,newdata=PlotPos))
```

We're already familiar with the `plot()` command of Line [1] – it opens the plot and creates the set of points. The next three lines provide a useful template for plotting any fitted line:

The `seq()` command creates a vector *PlotSeq* of uniformly-spaced position values over the plot range $0.0 \leq x \leq 20.0$. The vector contains 201 values with interval $\Delta x = 0.1$.

Line [2] incorporates *PlotSeq* into the data frame *PlotPos*. The single column has the name *x*.

The `lines()` command adds a number of connected lines to an existing plot. The arguments are two vectors of equal length giving values along the abscissa and ordinate. We use *PlotPos x* as the first vector and again use the `predict()` command to supply the second vector.

The result is the line in Fig. 6.

5 Linear curve fitting and plotting, part B

We'll continue the examination of linear fitting and plots with another example, a statistical Fourier analysis. It illustrates that 1) the `lm()` command can deal with any function and 2) data frame components need not always be called y and x . Here is the script to copy and paste in the **R** script window:

```
# Function to generate test data
TrigDemo = function(z) {
  Out = 1.4+2.30*sin(z)+0.75*cos(z)-1.80*sin(2*z)+2.15*cos(2*z)
        +0.65*sin(3*z)-0.25*cos(3*z)
  return(Out)
}
# Set up some variables
zmin = 0.0
zmax = 5.0
Noise = 0.55
# Plot the function without noise
curve(TrigDemo(z),zmin,zmax,xname="z")
# Create a data frame of input data
zvector = seq(from=zmin,to=zmax,length.out=51)
phivector = TrigDemo(zvector)
TestData = data.frame(z=zvector,phi=phivector)
# Add some noise
TestData$phi = TestData$phi + rnorm(length(TestData$phi),0,Noise)
# Plot the raw data
plot(TestData$z,TestData$phi)
# Fit the data and list the results
FitTrig = lm(phi~I(sin(z))+I(cos(z))+I(sin(2*z))+I(cos(2*z))
             +I(sin(3*z))+I(cos(3*z)),TestData)
summary(FitTrig)
# Add the fitted line to the plot
PlotSeq = seq(from=0.0,to=5.0,length.out=51)
PlotPos = data.frame(z=PlotSeq)
lines(PlotPos$z,predict(FitTrig,newdata=PlotPos))
```

We'll generate data by adding statistical noise to the test function

$$\phi = 1.4 + 2.3 \sin(z) + 0.75 \cos(z) - 1.8 \sin(2z) + 2.15 \cos(2z) + \quad (5) \\ 0.65 \sin(3z) - 0.25 \cos(3z),$$

defined over the interval $0.0 \leq z \leq 5.0$. Equation 5 is a lengthy expression, and it would be inconvenient to add it in command arguments. For efficiency, we'll define our own function to supplement the native functions of **R**. Here are the corresponding lines:

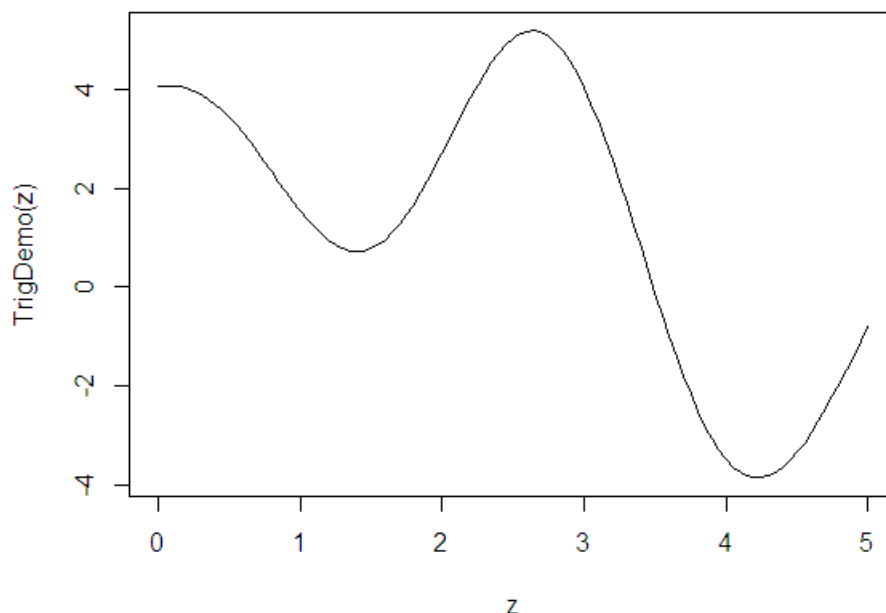


Figure 7: Plot of the generating curve for the Fourier example.

```
TrigDemo = function(z) {
  Out = 1.4+2.30*sin(z)+0.75*cos(z)-1.80*sin(2*z)+2.15*cos(2*z)
        +0.65*sin(3*z)-0.25*cos(3*z)
  return(Out)
}
```

The top definition line contains the name of the function, the keyword *function* and the names of the argument (or arguments) that will be used in the calculation. Any number of command lines for calculations with the argument may be included between the braces. **R** functions have flexible output. There is a single output value if the input argument is a scalar while, a vector argument gives a vector output. In other circumstances, the output may even be the functional dependence itself. The type of output depends on the context. To demonstrate the third case, we'll use the `TrigDemo()` function to create a plot of the variation of Eq. 5 using the `curve()` command:

```
curve(TrigDemo(z),zmin,zmax,xname="z")
```

The first argument is a definition of the mathematical expression to plot, such as $18.67 + z^2$, the next two arguments are numerical values that give the range of the independent variable and the last argument defines the name of the independent variable. Figure 7 shows the resulting plot.

The following script command lines demonstrate a case where the function produces vector output:

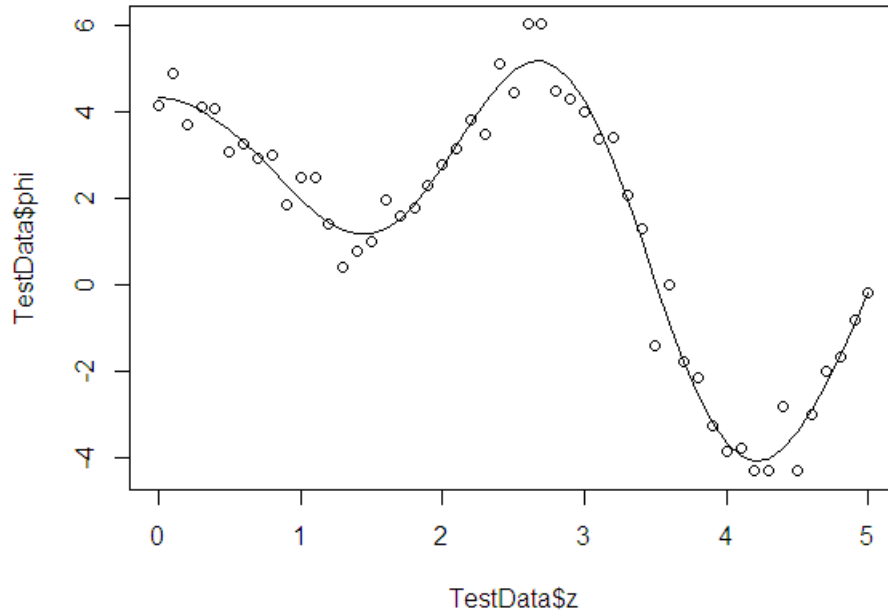


Figure 8: Points show the noisy input function. The line is the most probable fit to the data for the given formula.

```
[1] zvector = seq(from=zmin,to=zmax,length.out=51)
[2] phivector = TrigDemo(zvector)
[3] TestData = data.frame(z=zvector,phi=phivector)
```

As we saw in the previous article, the `seq()` operator generates a vector *zvector* that contains a set of *z* values over the desired range with interval $\Delta z = 0.1$. We apply the function `TrigDemo()` with argument *zvector* to generate *phivector*, a set of corresponding dependent values. The vectors are combined into a data frame with column names *z* and *phi*. Finally, we want to add some statistical noise:

```
TestData$phi = TestData$phi + rnorm(length(TestData$phi),0,Noise)
```

The function `rnorm()` generates random numbers in a normal distribution. **R** includes a related set of random-number functions for different distributions (`runif()`, `rpois()`,...). The output is a vector. The arguments in `rnorm()` are 1) the length of the output vector, 2) the mean of the distribution and 3) the standard deviation. The points in Fig. 8 show the resulting noisy representation of the generating function.

This form of the `lm()` command performs the fit:

```
FitTrig = lm(phi~I(sin(z))+I(cos(z))+I(sin(2*z))
             +I(cos(2*z))+I(sin(3*z))+I(cos(3*z)),TestData)
```

Finally, this set of commands plots the best-fit line in Fig. 8 following the method discussed in the previous section:

```
PlotSeq = seq(from=0.0,to=5.0,length.out=51)
PlotPos = data.frame(z=PlotSeq)
lines(PlotPos$z,predict(FitTrig,newdata=PlotPos))
```

Note how the column names of *TestData*, *z* and *phi* are used consistently throughout all the commands.

6 Working with RStudio: multi-dimensional fitting

RStudio is an integrated development environment (IDE) for working with **R**, similar to the IDEs available for most computer languages. Figure 9 shows a screenshot. There are work areas for the script editor, console and plots. The difference is that they are combined in a single convenient workspace rather than floating windows. **RStudio** is freely available and provides a wealth of features, so it is hard to imagine why anyone would choose not to use it. Here are some of the advantages:

RStudio remembers your screen setups, so it is not necessary to resize and reposition floating windows in every session.

The script editor (upper left in Fig. 9) features syntax highlighting and an integrated debugger. You can keep multiple tabbed documents active.

An additional area at upper-right shows all **R** objects currently defined. Double clicking a data frame opens it in a data editor tab in the script window.

The area at the lower right has multiple functions in addition to plot displays. Other tabs invoke a file manager, a package manager for **R** supplements and a *Help* page for quick access to information on commands.

I'll assume you're using **RStudio** for the example of this section and following ones.

We'll see how to use **R** to fit a three-dimensional polynomial function to noisy data. One application of the technique is making an accurate interpolation of radiation dose at a point, $D(x, y, z)$, in a **GamBet** calculation. Here, the dose in any single element is subject to statistical variations that depend on the number of model input particles. The idea is to get an estimate at a test point $[x_0, y_0, z_0]$ by finding the most probable smooth function using values from a number of nearby elements. The approach is valid when the quantity varies smoothly in space. The present calculation is the three-dimensional extension of the interpolation method discussed in a previous section.

As with previous examples, we'll need to create some test data. The measurement point is $X_{avg} = 23.45$, $Y_{avg} = 10.58$ and $Z_{avg} = -13.70$. The ideal variation of dose about the point is:

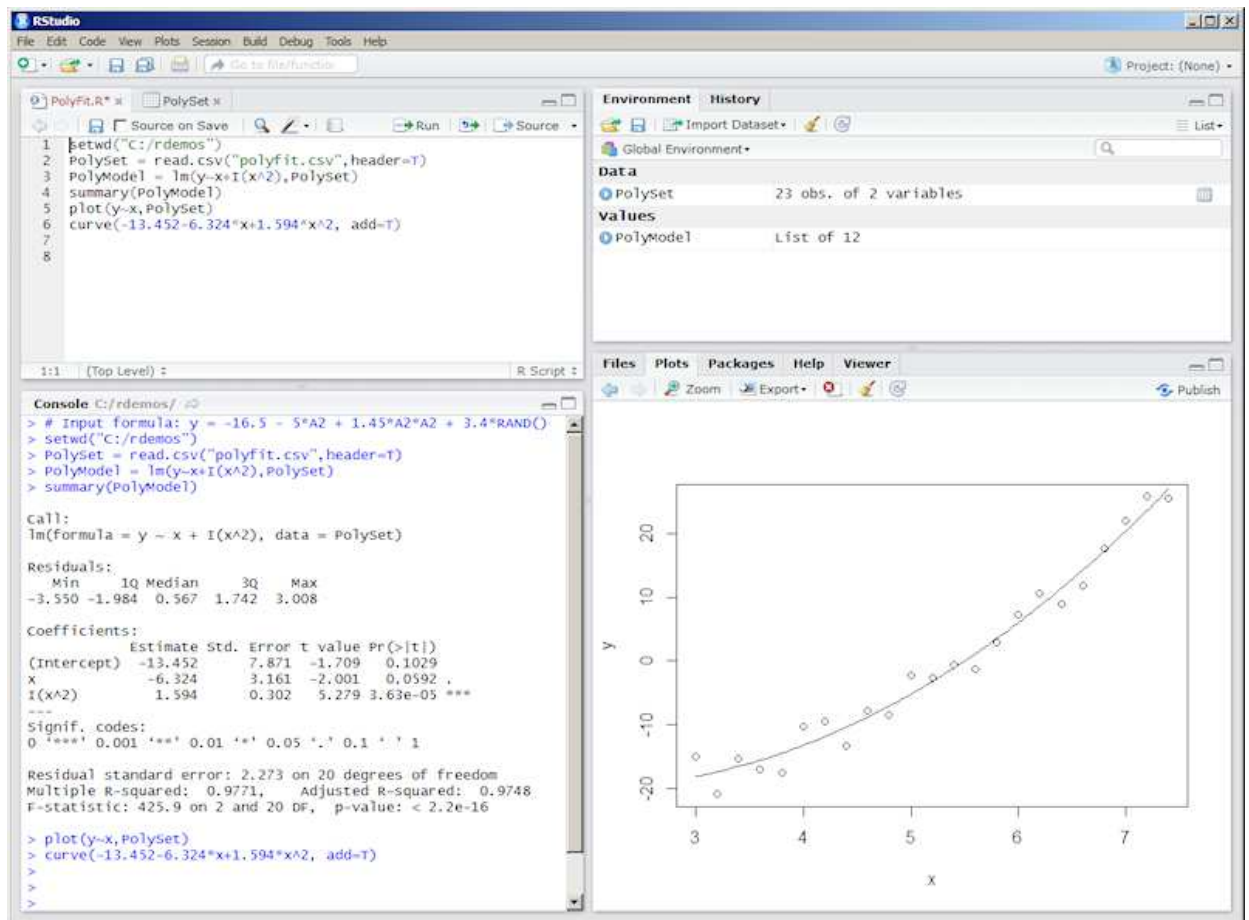


Figure 9: RStudio screenshot.

$$\begin{aligned}
D = & 19.45 + 2.52 (x - X_{avg}) - 0.25 (X - X_{avg})^2 \\
& -4.23 (y - Y_{avg}) + 0.36 (y - Y_{avg})^2 \\
& +1.75 (z - Z_{avg}) - 0.41 (z - Z_{avg})^2.
\end{aligned}
\tag{6}$$

The spacing of data points about the measurement point is $\Delta x = 1.00$, $\Delta y = 1.25$ and $\Delta z = 0.90$. In this case, we'll use a computer program to record the data in a **CSV** file. For reference, Table 6 shows the core of the FORTRAN program. It creates a header and then writes 1331 data lines in **CSV** format. Each line contains the position and dose, $[x, y, z, D]$. Here are the initial entries in `multiregress.csv`:

```

x,y,z,D
18.45000076,    4.32999992,   -18.19999886,    22.14448738
19.45000076,    4.32999992,   -18.19999886,    28.94458199
20.45000076,    4.32999992,   -18.19999886,    38.34004974
...
```

Run **R**, clear the console window (*Control-L*) and close any previous script. Copy and paste the following text into the script editor:

```

rm(list=objects())
setwd("C:/RExamples/MultiRegress")
DataSet = read.csv("multiregress.csv",header=T)
# Calculate the measurement point
XAvg=mean(DataSet$x)
YAvg=mean(DataSet$y)
ZAvg=mean(DataSet$z)
# Reference spatial variables to the measurement point
DataSet$x = DataSet$x - XAvg
DataSet$y = DataSet$y - YAvg
DataSet$z = DataSet$z - ZAvg
# Perform the fitting
FitModel = lm(D~x+y+z+I(x^2)+I(y^2)+I(z^2),DataSet)
summary(FitModel)
# Plot a scan along x through the measurement point
png(filename="XScan.png")
xscan = subset(DataSet,y < 0.001 & y > -0.001 & z < 0.001 & z > -0.001 )
plot(xscan$x,xscan$D)
lines(xscan$x,predict(FitModel,newdata=xscan))
dev.off()
```

The first three lines perform operations that we have previously discussed: clear all **R** objects, set a working directory and read information from the

Table 1: FORTRAN program to create a CSV file.

```

OPEN(FILE='multiregress.csv',UNIT=30)
WRITE (30,2000)
XAvg = 23.45
YAvg = 10.58
ZAvg = -13.70
Dx = 1.00
Dy = 1.25
Dz = 0.90
! Main data loop
DO Nz = -5,5
  z = ZAvg + REAL(Nz)*Dz
  DO Ny = -5,5
    y = YAvg + REAL(Ny)*Dy
    DO Nx = -5,5
      x = XAvg + REAL(Nx)*Dx
      CALL RANDOM_NUMBER(Zeta)
      Noise = 2.0*(0.5 - Zeta)
      D = 19.45 &
        + 2.52*(x-XAvg) - 0.25*(X-XAvg)**2 &
        - 4.23*(y-YAvg) + 0.36*(y-YAvg)**2 &
        + 1.75*(z-ZAvg) - 0.41*(z-ZAvg)**2 &
        + Noise
      WRITE (30,2100) x,y,z,D
    END DO
  END DO
END DO
CLOSE (30)
2000 FORMAT ('x,y,z,D')
2100 FORMAT (F14.8,',',',',F14.8,',',',',F14.8,',',',F14.8)

```

CSV file. The information is stored in data frame `DataSet`. Use *Control-R* in the script editor to execute the command lines. Note that *DataSet* appears in the *Environment* list of **RStudio**. Double-click on *DataSet* to open it in a data-editor tab. Here, you can inspect values and even modify them.

The first task in the analysis is to shift the spatial values so that they are referenced to the measurement point, as in Eq. 6:

```
XAvg = mean(DataSet$x)
YAvg = mean(DataSet$y)
ZAvg = mean(DataSet$z)
DataSet$x = DataSet$x - XAvg
DataSet$y = DataSet$y - YAvg
DataSet$z = DataSet$z - ZAvg
```

We use the `mean()` function under the assumption that the data points are centered about the measurement position. A single command performs the linear fit:

```
FitModel = lm(D~x+y+z+I(x^2)+I(y^2)+I(z^2),DataSet)
```

The `summary()` command gives the following information:

	Estimate	Std. Error	Generating
(Intercept)	19.540509	0.179142	19.45
x	2.560374	0.025733	2.52
y	-4.226011	0.020587	-4.23
z	1.737097	0.028593	1.75
I(x^2)	-0.265439	0.009214	-0.25
I(y^2)	0.358165	0.005897	0.36
I(z^2)	-0.404630	0.011375	-0.41

As with all statistical calculations, a visual indicator is a quick way to gauge the validity of the fit. The following commands create a two-dimensional plot: a scan of data points and the fitting function along x at $y = 0.0, z = 0.0$ (through the measurement point). The plot is saved as a PNG file in the working directory.

```
[1] png(filename="XScan.png")
[2] xscan = subset(DataSet,y < 0.001 & y > -0.001 & z < 0.001 & z > -0.001 )
[3] plot(xscan$x,xscan$D)
[4] lines(xscan$x,predict(FitModel,newdata=xscan))
[5] dev.off()
```

Line [1] opens a file for PNG output, while Line [5] flushes the plot information to the file and closes it. Line [2] uses the `subset()` function to create a data frame, *xscan*, that contains only data points where $y = 0.0$ and $z = 0.0$. The first argument is the source data frame and the second argument is the criterion for picking included row. There are two noteworthy features in the command expression:

The symbol `&` denotes the logical and operation.

The inclusion criterion has some slack to account to floating point round-off errors. An inspection of *xscan* with the data editor shows some values on the order of 10^{-8} rather than 0.0

Line [3] plots the raw data points while the `lines()` command in Line [4] applies the method discussed in a previous section to plot the fitting function. Similar operations were applied to create scans through the measurement point along y and z to produce the plots of Fig. 2.

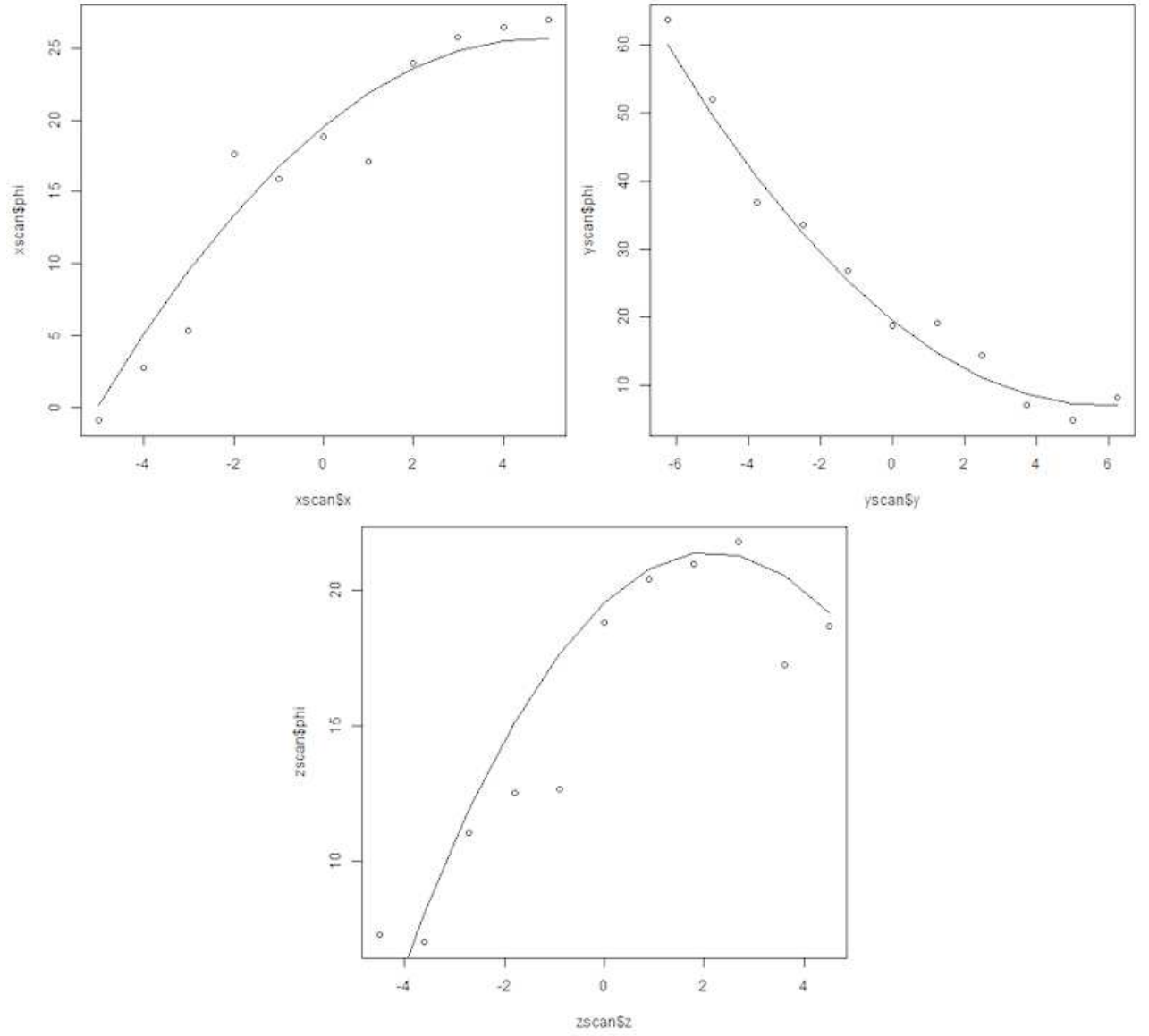


Figure 10: Scans of the data and fitting function along x, y and z passing through the measurement point.

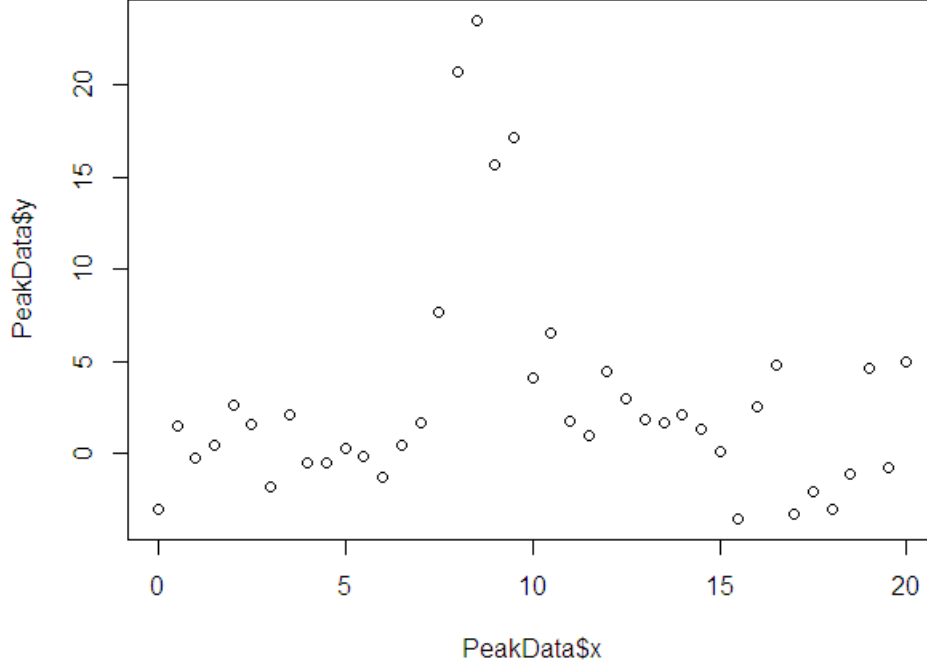


Figure 11: Test data: resonance in a noisy signal.

7 Nonlinear fitting and parameter estimation

To this point, we have carried out basic statistical calculations with **R**. With some effort, we could perform the operations in spreadsheets or our own programs. In this article, we'll turn to a much more sophisticated calculation that would be difficult to reproduce: nonlinear parameter fitting. Despite the complexity of the underlying calculations, it is simple to set up the **R** script.

Figure 11 illustrates the test calculation. Given a noisy signal, we want to determine if there is a resonance hidden within it. Such a response usually has the form of a Gaussian function

$$A \exp \left[-\frac{(x - \mu)^2}{2\sigma^2} \right]. \quad (7)$$

We want to determine values of A , μ and σ (the resonance width) such that the function gives the most probable fit to the data of Fig. 11. This is clearly a nonlinear problem because the parameters μ and σ are integrated in the function.

The spreadsheet of Fig. 12 is used to generate data over the range $0.0 \leq x \leq 20.0$. Values of y are determined from Eq. 7 with $A = 20.0$, $\mu = 8.58$ and $\sigma = 1.06$. The noise level is set as an adjustable parameter. Rather

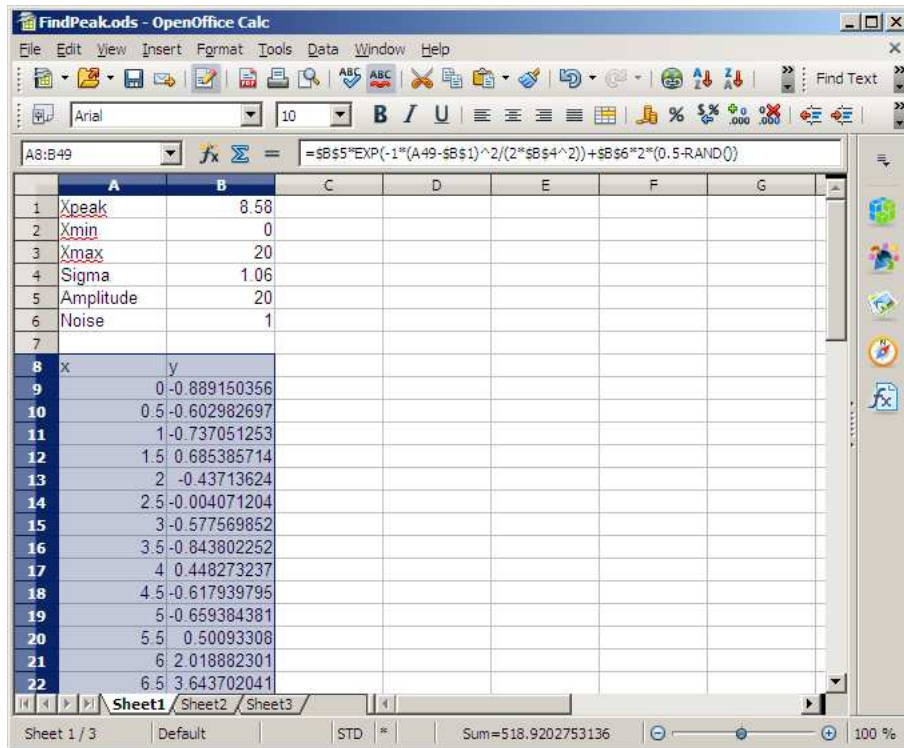


Figure 12: Spreadsheet to generate data pasted into a tab-delimited file.

than save the spreadsheet as a CSV file, we copy the data section as shown and paste it into a text file file, `peakdata.tab`. In this case, the entries on a line are separated by tab characters instead of commas. Therefore, we use a different command form to read the file:

```
PeakData = read.table("peakdata.tab",header=T,sep="")
```

The file header sets the column names of the data frame *PeakData* to *y* and *x*. The option `sep=""` means that any white-space characters (including tabs) act as delimiters.

To run the example, copy and paste this text into the **RStudio** script editor:

```
PeakData = read.table("peakdata_high.tab",header=T,sep="")
fit = nls(y ~ I(Amp*exp(-1.0*(Mu-x)^2/(2.0*Sigma^2))),data=PeakData,
  start=list(Amp=5.0,Mu=5.0,Sigma=5.0))
summary(fit)
plot(PeakData$x,PeakData$y)
new = data.frame(x = seq(min(PeakData$x),max(PeakData$x),len=200))
lines(new$x,predict(fit,newdata=new))
confint(fit,level=0.95)
```


The critical command line for the nonlinear fit is

```
fit = nls(y ~ I(Amp*exp(-1.0*(Mu-x)^2/Sigma^2)),data=PeakData,  
          start=list(Amp=5.0,Mu=5.0,Sigma=5.0))
```

The first argument of the `nls()` command is a defining formula:

```
y ~ I(Amp*exp(-1.0*(Mu-x)^2/(2.0*Sigma^2)))
```

It implies that the variable y depends on the independent variable x as well as parameters named *Amp*, *Mu* and *Sigma*. The dependence is declared explicitly and enclosed in the `I()` function. The second argument `data=PeakData` defines the input data source. The third argument,

```
start=list(Amp=5.0,Mu=5.0,Sigma=5.0)
```

needs some explanation. The iterative calculation involves varying parameter values and checking whether the overall error is reduced. The user must set reasonable initial values for the search. The `start` option requires an **R** *list*, a general set of values. The `list()` operator combines several objects into a list. In the form shown, the first item in the list has name *Amp* and value 5.0. Be aware that for some choices of parameters, the process may not converge or may converge to an alternate solution.

The `summary()` command produces a console listing like this:

```
Formula: y ~ I(Amp * exp(-1 * (Mu - x)^2/(2 * Sigma^2)))
```

```
Parameters:
```

```
Estimate Std. Error t value Pr(>|t|)
```

```
Amp    22.45079      1.86245    12.05 1.49e-14 ***
```

```
Mu      8.63087      0.08361   103.22 < 2e-16 ***
```

```
Sigma  -0.87288      0.08361   -10.44 1.02e-12 ***
```

```
---
```

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

```
Residual standard error: 2.675 on 38 degrees of freedom
```

```
Number of iterations to convergence: 13
```

```
Achieved convergence tolerance: 5.277e-06
```

The next three commands plot the raw data and the fitting line following methods that we discussed in previous sections. Finally, there is a new command that gives the confidence interval:

```
confint(fit,level=0.95)
```

The output is a listing like this:

	2.5%	97.5%
Amp	18.738008	26.3724502
Mu	8.460237	8.8129274
Sigma	-1.071273	-0.7146478

The results have the following interpretation. What is the range where we can have 95% confidence that the parameter value lies within it? In other words, for the above example we can have 95% confidence that the peak amplitude is between 18.738008 and 26.3724502.

With an understanding of the mechanics of the calculation, we can check out some results. To begin, a high level of noise in the spreadsheet ($Noise = 5.0$) leads to the data points and fitting line of Fig. 13a. Here is a summary of numerical results:

Noise level: 5.0

	Estimate	Std. Error	2.5%	97.5%	Generating
Amp	22.45079	1.86245	18.738008	26.3724502	20.0
Mu	8.63087	0.08361	8.460237	8.8129274	8.5
Sigma	-0.87288	0.08361	-1.071273	-0.7146478	1.06

The negative sign for *Sigma* is not an issue because the parameter always appears as a square. Because the peak is barely discernible in the noise, there is some error in the estimate of *Sigma*. Rerunning the example with reduced noise level ($Noise = 0.5$) provides a check that the `nls()` command is operating correctly. Figure 13b shows the data points and fitting line. The results are much closer to the ideal values with no signal noise.

Noise level: 0.5

	Estimate	Std. Error	2.5%	97.5%
Amp	20.02893	0.17535	19.673012	20.386997
Mu	8.57904	0.01084	8.557082	8.600996
Sigma	1.07258	0.01084	1.050487	1.095135

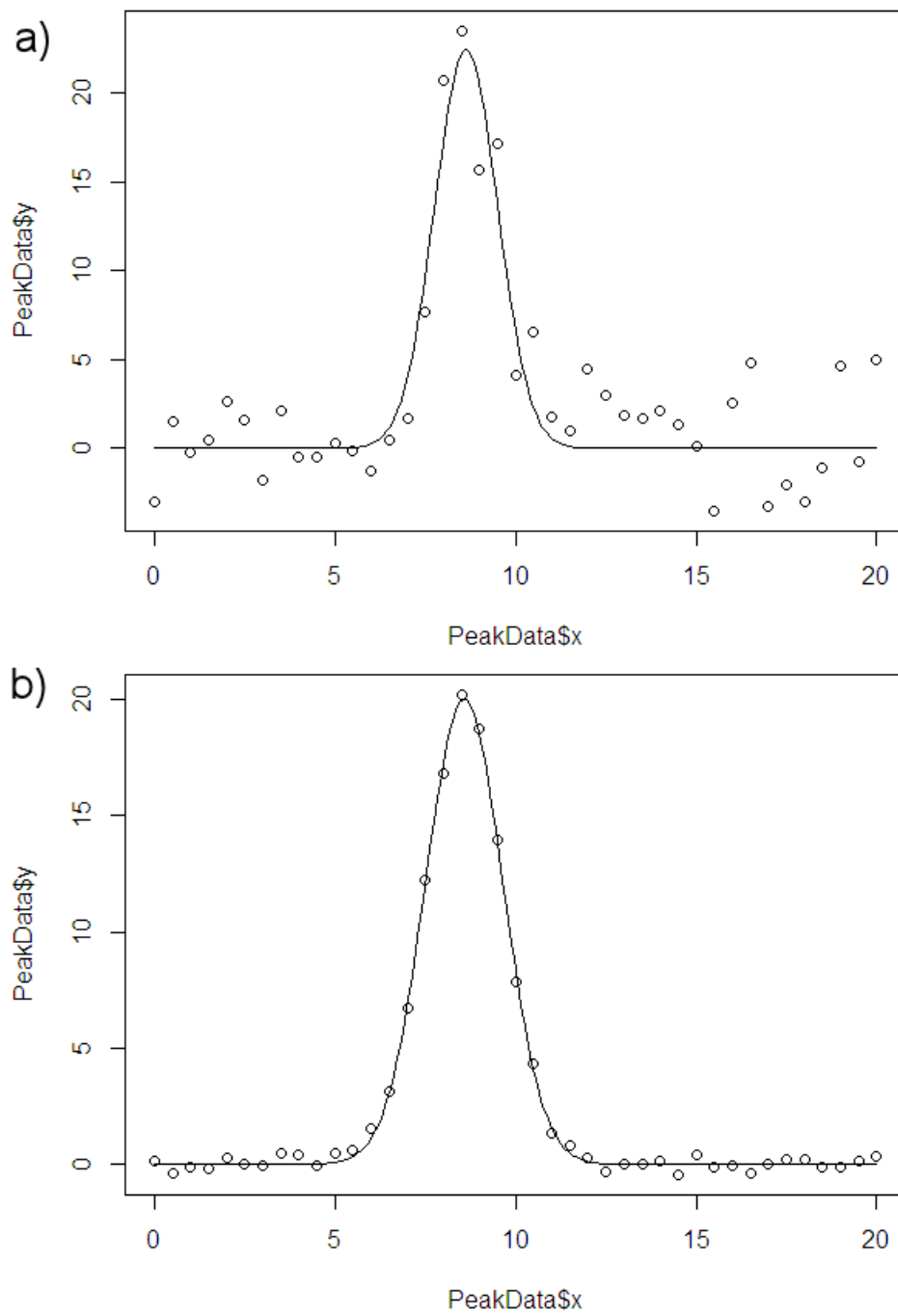


Figure 13: Results with $Noise = 5.0$.

8 Importing GamBet files into R

In the previous sections, we studied **R** techniques that would be useful to supplement most technical programs. In this section and the next, we'll concentrate on integrating **R** with **GamBet**. The **GamBet** program produces two types of text data files⁴ that can be analyzed with **R**:

Escape files have a name of the form **FName.SRC**. They record the parameters of electrons, photons and positrons that escape from the solution volume. An example is the distribution of bremsstrahlung photons produced by an electron beam striking a target. Escape files may serve as the particle source for a subsequent simulation. You can also use **R** to prepare source files with mathematically-specified distributions, the topic of the next section.

Files of the spatial distribution of deposited dose produced by the **MATRIX** commands in **GBView2** and **GBView3**.

We'll begin with a discussion of the **GamBet SRC** file. Here's an example, the output from a bremsstrahlung target:

```
* GamBet Particle Escape File (Field Precision)
* Output from run: BremGen
* DUnit: 1.0000E+03
* NPrimary: 1
* NShower: 500
*
* Type      Energy      X      Y      Z      ux      uy      uz
* =====
E  1.8276E+07  1.0000E+00 -2.2093E-01 -9.5704E-03  0.95867  -0.28441  -0.00726
P  1.0476E+06  1.0000E+00 -2.2099E-01 -9.6599E-03  0.96046  -0.27838  0.00421
P  1.4681E+07  1.0000E+00 -2.2079E-01 -9.4444E-03  0.96751  -0.24988  0.03846
P  1.1157E+05  1.0000E+00 -2.2125E-01 -9.0243E-03  0.93991  -0.33167  0.08099
...
P  3.7167E+06  1.0000E+00 -1.1189E-01 -6.2985E-02  0.99157  -0.11335  -0.06279
P  8.2996E+05  1.0000E+00 3.3150E-01 -2.6303E-01  0.91975  0.30757  -0.24386
ENDFILE
```

There is a file header consisting of eight comment lines marked by an asterisk. The header is followed by a large number of data lines. The file terminates with an **ENDFILE** marker. Each data line may contain 8 or 9 entries separated by space delimiters. A data line contains the following components:

⁴The `read.fortran()` command of **R** provides a path to import information from the binary output files of any Field Precision technical program.

The marker in the first column gives the type of particle, electron (E or $E-$), photon (P) and positron ($E+$). In the example, the output is a mixture of the primary electron beam and the secondary photons.

The kinetic energy in eV.

The position at the exit, (x, y, z) .

Components of a unit vector giving the particle direction (u_x, u_y, u_z) .

A ninth column listing current or flux is present in runs where weighting is assigned to the model input particles.

If we are going to perform a standard analysis on many files, it would be an advantage to create an **R** script where the user could choose the working directory and the SRC file interactively. Here is the section of the script to specify and to load a file. It introduces several new concepts and commands. Copy and paste the text to a script file window in **RStudio**:

```
library(utils)
if (!exists("WorkDir")) {
  WorkDir = choose.dir(default = "", caption = "Select folder")
}
setwd(WorkDir)
FDefault = paste(WorkDir, "\\*.src", sep="")
FName = choose.files(default=FDefault, caption="Select GamBet SRC file", multi=FALSE)
CheckFile = read.table(FName, header=FALSE, sep=" ", comment.char="*", fill=TRUE, nrow=1)
NColumn = ncol(CheckFile)
if(NColumn==8) {
  # Standard column names
  cnames = c("Type", "Energy", "X", "Y", "Z", "ux", "uy", "uz")
} else {
  cnames = c("Type", "Energy", "X", "Y", "Z", "ux", "uy", "uz", "Flux")
}
SRCFile = read.table(FName, header=FALSE, sep=" ", comment.char="*",
  col.names=cnames, fill=TRUE)
NLength = nrow(SRCFile)
SRCFile = SRCFile[1:(NLength-1),]
```

The first command

```
library(utils)
```

occurs often in work with **R**. Many default commands are loaded when you start the **R** console. Although they have been sufficient for our previous work, they represent only a fraction of the available features. In this case, we load a library *utils* that supports interactive file operations. The command lines constitute an if statement:

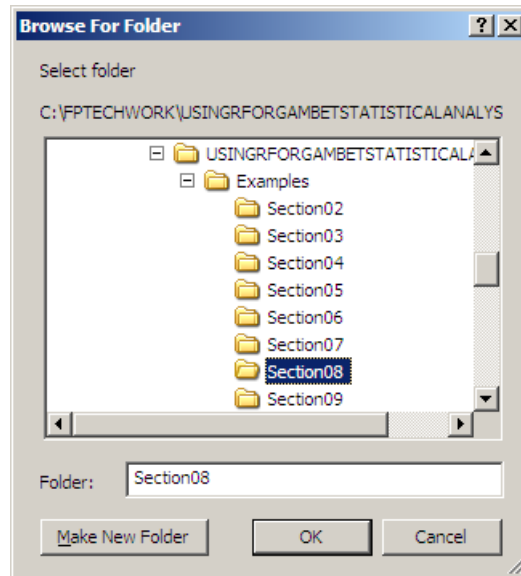


Figure 14: Dialog for the `choose.dir()` operation.

```
if (!exists("WorkDir")) {
  WorkDir = choose.dir(default = "", caption = "Select folder")
}
```

Simple `if` statements consist of a conditional line followed by any number of commands in braces. In this case, the commands are executed only if the object *WorkDir* does not exist (the exclamation point designates the logical *not* operation). If we had a number of **SRC** files to analyze in the same directory, we would not want to reset the working directory every time. The `choose.dir()` operation brings up the standard **Windows** selection dialog of Fig. 14 and returns the path as *WorkDir*. The path is then set as the working directory.

After picking a directory, we'll use the `choose.files()` command to pick a file. One of the command parameters is a default file name. We again use the paste operation to concatenate the path name and the default file name:

```
FDefault = paste(WorkDir, "\\*.src", sep="")
```

The result looks like this:

```
FDefault = "C:\\USINGRFORGAMBETSTATISTICALANALYSIS\\Examples\\Section08\\*.src"
```

The double backslash represents the forward slash used in **R**. The command:

```
FName = choose.files(default=FDefault,caption="Select GamBet SRC file",multi=FALSE)
```

opens a standard dialog to return the name of a single file in the working directory of type `*.SRC` (Figure 2). The example file contains 47,892 entries.

The `read.table()` command provides a simple option for reading the file, but there are two challenges:

We don't know whether there will be 8 or 9 data columns.

The line with `ENDFILE` does not contain any data.

These commands address the first problem:

```
CheckFile = read.table(FName,header=FALSE,sep="",comment.char="*",fill=TRUE,nrows=1)
NColumn = ncol(CheckFile)
if(NColumn==8) {
  cnames = c("Type","Energy","X","Y","Z","ux","uy","uz")
} else {
  cnames = c("Type","Energy","X","Y","Z","ux","uy","uz","Flux")
}
```

The `read.table()` command ignores the header comment lines and reads a single data line (`nrows=1`) to the dummy data frame `CheckFile`. The `ncol()` function returns the number of data columns as `NColumn`. Depending on the value, we define a vector `cnames` of column names containing 8 or 9 components. The next `read.table()` command inputs the entire set of data lines plus the `ENDFILE` line:

```
SRCFile = read.table(FName,header=FALSE,sep="",
  comment.char="*",col.names=cnames,fill=TRUE)
```

Note that the number of columns and their names is set by `col.names=cnames`. In the absence of the `fill=TRUE` option, the operation would terminate with an error because the `ENDFILE` line has only one entry. The option specifies that a line with fewer entries than specified should be filled out with `N/A` values. The last line is meaningless, so we delete it with the commands:

```
NLength = nrow(SRCFile)
SRCFile = SRCFile[1:(NLength-1),]
```

Clearly, the procedure for loading a **GamBet** source file has several tricky features. Discovering them takes some trial-and-error. The advantage of a script program like **R** is that the effort is required only once. The script we have discussed provides a template to load all **SRC** files.

Now that the file has been loaded as the data frame `SRCFile`, we can do some calculations. Copy and paste the following information below the load commands:

```

electrons = subset(SRCFile,Type=="E" | Type=="E-")
photons = subset(SRCFile,Type=="P")
elecavg = mean(electrons$Energy)
photavg = mean(photons$Energy)
hist(electrons$Energy,breaks=20,density=15,
     main="Electron energy spectrum",xlab="T (eV)",ylab="N/bin")
hist(photons$Energy,breaks=20,density=15,
     main="Photon energy spectrum",xlab="T (eV)",ylab="N/bin")

```

This command:

```

electrons = subset(SRCFile,Type=="E" | Type=="E-")

```

creates the data frame *electrons* that contains only rows where the *Type* is *E* or *E−*. We calculate the mean kinetic energy of electrons emerging from the target and create a histogram. Figure 15 shows the result. At this point, you should be able to figure out the meanings of options in the `hist()` command. Use the *Help* tab in **RStudio** and type in `hist` for more detailed information.

To conclude, we'll discuss importing a matrix file from the two-dimensional **GBView2** post-processor. A matrix file is a set of values computed over a regular grid (uniform intervals in *x* and *y* or *z* and *r*). In this case, the quantity is total dose (deposited energy/mass), dose from primary electrons, dose from primary photons, etc. Here is a sample of the first part of a file:

Matrix of values from data file alumbeam.G2D

```

XMin:  0.0000E+00   YMin:  -5.0000E-02
XMax:  1.0000E-01   YMax:   5.0000E-02
NX:  20   NY:  20
X           Y           NReg  DoseTotal   DoseElecP   DosePhotP
=====
0.0000E+00 -5.0000E-02     1  3.3101E+04  2.6251E+04  0.0000E+00
5.0000E-03 -5.0000E-02     1  3.5977E+05  3.0285E+05  0.0000E+00
1.0000E-02 -5.0000E-02     1  3.5977E+05  3.0285E+05  0.0000E+00
1.5000E-02 -5.0000E-02     1  3.9705E+05  3.1705E+05  0.0000E+00
...

```

```

           DosePosiP   DoseElecS   DosePhotS   DosePosiS
=====
0.0000E+00  6.8504E+03  0.0000E+00  0.0000E+00
0.0000E+00  5.6925E+04  0.0000E+00  0.0000E+00
0.0000E+00  5.6925E+04  0.0000E+00  0.0000E+00
0.0000E+00  8.0007E+04  0.0000E+00  0.0000E+00
...

```

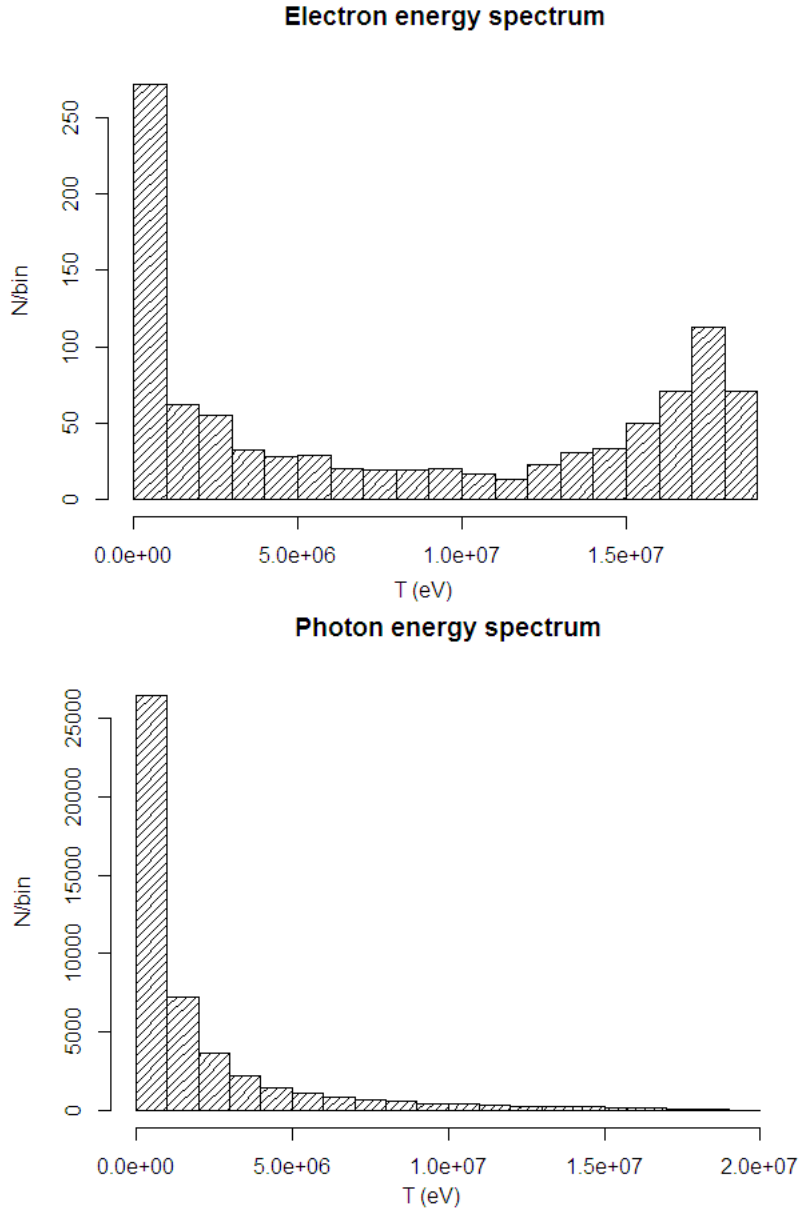



Figure 15: Energy distributions of primary electrons and bremsstrahlung photons determined from the test SRC file.

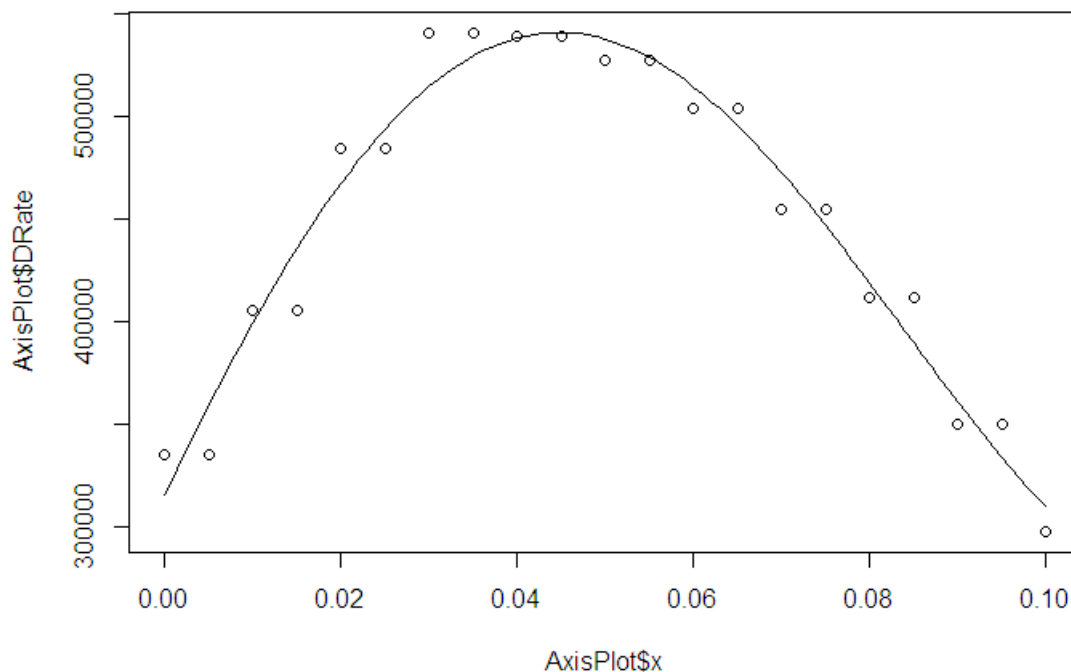


Figure 16: Fitted dose distribution determined from a **GBView2** matrix file.

The data lines are space delimited and can easily be loaded with the `read.table()` command. The challenge is the header, where the lines are not comments. Because the header information is not required for the **R** analysis, we can simply omit it by using the `skip` option. The following code loads matrix file information, limits data to a scan along x at $y = 0.0$, carries out a fourth-order polynomial fit and plots the results (Figure 4).

```
cnames = c("x","y","NReg","DRate","DoseElecP","DosePhotP","DosePosiP",
           "DoseElecS","DosePhotS","DosePosiS")
MatrixData = read.table(file="Demo.MTX",header=FALSE,sep=" ",col.names=cnames,skip=6)
AxisPlot = subset(MatrixData,y > -0.00001 & y < 0.00001)
plot(AxisPlot$x,AxisPlot$DRate)
DFit = lm(DRate~I(x)+I(x^2)+I(x^3)+I(x^4),AxisPlot)
PlotSeq = seq(from=0.0,to=0.10,length.out=101)
PlotPos = data.frame(x=PlotSeq)
lines(PlotPos$x,predict(DFit,newdata=PlotPos))
```

9 Creating GamBet SRC files with R

In this section, we'll learn how to generate particle distributions with **R** and how to export them in a file that can be used directly as input to **GamBet**. A second topic is how to run **R** scripts directly from the **Windows** command prompt or from batch files. With this capability, you can write scripts for automatic analyses of output from **GamBet** and other technical programs.

The parameters of primary particles for Monte Carlo simulations in **GamBet** are specified in source (SRC) files. Output escape files have the same format, so the output from one **GamBet** calculation can be used as the input to a subsequent one. For initial **GamBet** simulations, the user usually prepares a source file representing specific distributions in position, velocity and energy. Although the **GenDist** utility can create several useful distributions, the possibilities with **R** are greatly expanded.

As a test case, we'll generate an input electron beam for a 3D calculation with current $CurrTotal = 2.5$ A and average energy $E_0 = 20.0$ keV. The beam has a circular cross section with a Gaussian distribution in x and y of width $X_w = Y_w = 1.4$ mm. The average position is $X_0 = 0.0$ mm, $Y_0 = 0.0$ mm and $Z_0 = 0.0$ mm. The parallel electrons move in z . There is a Gaussian energy spread of $E_w = 500$ eV.

First, we set up a script that can be tested in the interactive environment of **RStudio** and then convert it to an autonomous program that can be run from a batch file. The first set of commands clears the workspace, sets a working directory and defines parameters:

```
rm(list=objects())
WorkDir = "C:/Examples/Section09/"
setwd(WorkDir)
CurrTotal = 2.5
X0 = 0.0
Y0 = 0.0
Z0 = 0.0
Xw = 1.4
Yw = 1.4
E0 = 20000.0
Ew = 500.0
Ux0 = 0.0
Uy0 = 0.0
Uz0 = 1.0
NPart= 100000
```

The previous section discussed the SRC file format. The strategy will be to set up a vector of length $NPart$ for each quantity in the file data lines,

combine them into a data frame and then use the `write.table()` command to make the file. These commands create the vector for the *Type* column:

```
Type = character(length=NPart)
for (n in 1:NPart) {
  Type[n] = "E"
}
```

The first command creates a character vector called *Type* with *NPart* blank entries. The loop fills the vector with the character *E*.

The quantities *Z*, *Ux*, *Uy*, and *Uz* are numerical vectors of length *NPart* that contain identical values. it is convenient to create and to fill the vectors with the `seq()` and `c()` commands:

```
Z = c(seq(from=Z0,to=Z0,length.out=NPart))
Ux = c(seq(from=Ux0,to=Ux0,length.out=NPart))
Uy = c(seq(from=Uy0,to=Uy0,length.out=NPart))
Uz = c(seq(from=Uz0,to=Uz0,length.out=NPart))
```

The position vectors *X* and *Y* and the *Energy* vector are created using the `rnorm()` function. It generates *NPart* values following a specified normal distribution:

```
X = rnorm(NPart,mean=X0,sd=Xw)
Y = rnorm(NPart,mean=Y0,sd=Yw)
Energy = rnorm(NPart,mean=E0,sd=Ew)
```

The set of vectors is assembled into a data frame. Note that the column names are the same as the vector names, *Type*, *Energy*, *X*,...:

```
SRCRaw = data.frame(Type,Energy,X,Y,Z,Ux,Uy,Uz)
```

We must take some precautions using the normal distribution. There is a small but non-zero probability of extreme values of the position and energy. They could result in electrons outside the **Gambet** solution volume or negative energy values. Both conditions would lead to a program error. We create a subset of the raw data such that no electron has $r > 7.0$ mm or *Energy* = 0.0:

```
SRCFile = subset(SRCRaw,((X^2+Y^2)<=25.0*Xw^2) & (Energy > 0.0))
```

We need to add the current per electron to complete the *SRCFile* data frame. Note the use of *NLength*, the number of electrons in the modified data frame, which may be less than *NPart*:

```
NLength = length(SRCFile$Type)
dCurr = CurrTotal/NLength
```

A column vector of identical current values is constructed and appended to the *SRCFile* data frame with the `cbind()` command:

```
Curr = c(seq(from=dCurr,to=dCurr,length.out=NLength))
SRCFile = cbind(SRCFile,Curr)
```

We also define two plotting vectors for latter use:

```
RVector = sqrt(SRCFile$X^2 + SRCFile$Y^2)
EVector = SRCFile$Energy
```

We're ready to write the file. Some variables are set up in preparation:

```
FNameOut = "TestSRCGeneration.SRC"
HLine1 = "* GamBet Particle Escape File (Field Precision)"
HLine2 = paste("* NPart:",NLength)
HLine3 = "* Type      Energy          X          Y          Z          Ux ...
HLine4 = "* =====
```

Note that the actual number of electrons was written to *HLine2*. The following commands open the file (over-writing any previous version) and write the header using the `cat()` command:

```
cat(HLine1,file=FNameOut,append=FALSE,fill=TRUE)
cat(HLine2,file=FNameOut,append=TRUE,fill=TRUE)
cat(HLine3,file=FNameOut,append=TRUE,fill=TRUE)
cat(HLine4,file=FNameOut,append=TRUE,fill=TRUE)
```

We could add the table in its current form, but it would be nice to have the fixed-width format illustrated in the previous section. This command adds three initial spaces to the *Type* column:

```
SRCFile$Type = paste("   ",SRCFile$Type,sep="")
```

These commands convert the numbers in the columns to character representations with width 12 in either scientific or standard notation:

```
SRCFile$Energy = format(SRCFile$Energy,scientific=TRUE,digits=5,width=12)
SRCFile$X = format(SRCFile$X,scientific=TRUE,digits=5,width=12)
SRCFile$Y = format(SRCFile$Y,scientific=TRUE,digits=5,width=12)
SRCFile$Z = format(SRCFile$Z,scientific=TRUE,digits=5,width=12)
SRCFile$Ux = format(SRCFile$Ux,scientific=FALSE,digits=6,width=12)
SRCFile$Uy = format(SRCFile$Uy,scientific=FALSE,digits=6,width=12)
SRCFile$Uz = format(SRCFile$Uz,scientific=FALSE,digits=6,width=12)
SRCFile$Curr = format(SRCFile$Curr,scientific=TRUE,digits=6,width=12)
```

Note that *RVector* and *EVector* for making plots were defined before these commands when the data entries were still numbers. Finally, this command writes the SRC file:

```
write.table(SRCFile,file=FNameOut,sep=" ",append=TRUE,
           col.names=FALSE,quote=FALSE,row.names=FALSE)
```

Here is a sample of the result:

```
* GamBet Particle Escape File (Field Precision)
* NPart: 100000
* Type      Energy      X      Y      Z
* =====
E  2.0448e+04 -6.8232e-01  8.4354e-01  0e+00
E  1.9949e+04 -4.8475e-02  1.5114e+00  0e+00
E  2.1422e+04 -6.9644e-01 -4.9890e-01  0e+00
E  1.9291e+04 -1.9870e+00  5.5780e-01  0e+00
E  2.0032e+04 -5.6534e-01 -2.0669e-01  0e+00
...
```

Ux	Uy	Uz	Curr
=====	=====	=====	=====
0	0	1	2.5e-05
0	0	1	2.5e-05
0	0	1	2.5e-05
0	0	1	2.5e-05
0	0	1	2.5e-05
...			

The format isn't precisely the way we would like. For example, despite the setting `digits=6`, the entries in *Ux* are 0 rather than 0.00000. This is a quirk of **R**. The program will not write more significant figures than those in the defined values, no matter what you tell it. Perhaps there is a solution, but I haven't found it. **GamBet** and **GenDist** will recognize the above form. Figure 17 shows a **GenDist** scatter plot of model particles in the x-y plane. The following commands create histograms to display the distribution of electrons in radius and energy (Figure 18):

```
hist(RVector,breaks=100)
hist(EVector,breaks=100)
```

To conclude, we'll modify the script so it may be called from a batch file. Suppose we want to make a series of **GamBet** runs with incident beams of differing width. For each run, we regenerate the input SRC file with **R** to represent the new width, run **GamBet** and rename output files so they are not over-written. Here's a sample of a Windows batch file showing the first operation:

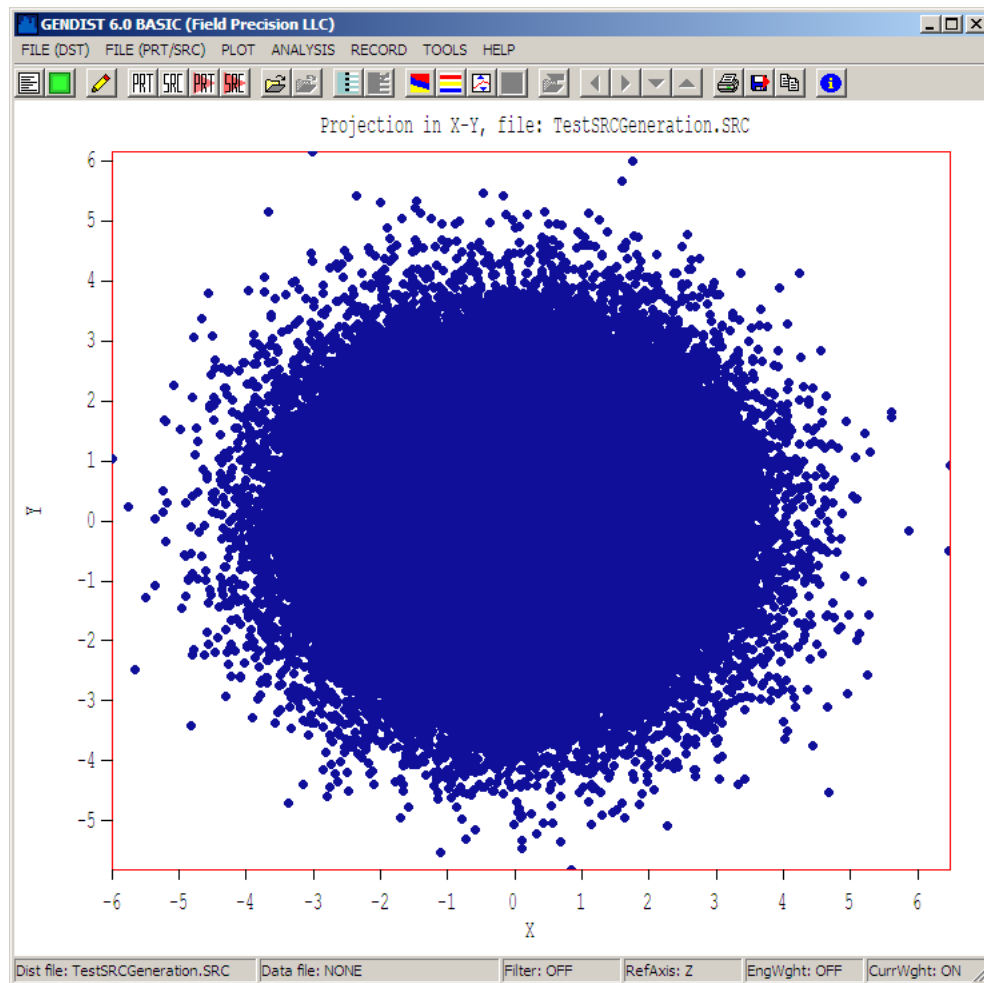


Figure 17: Scatter plot in x - y produced from the SRC file by **GenDist**.

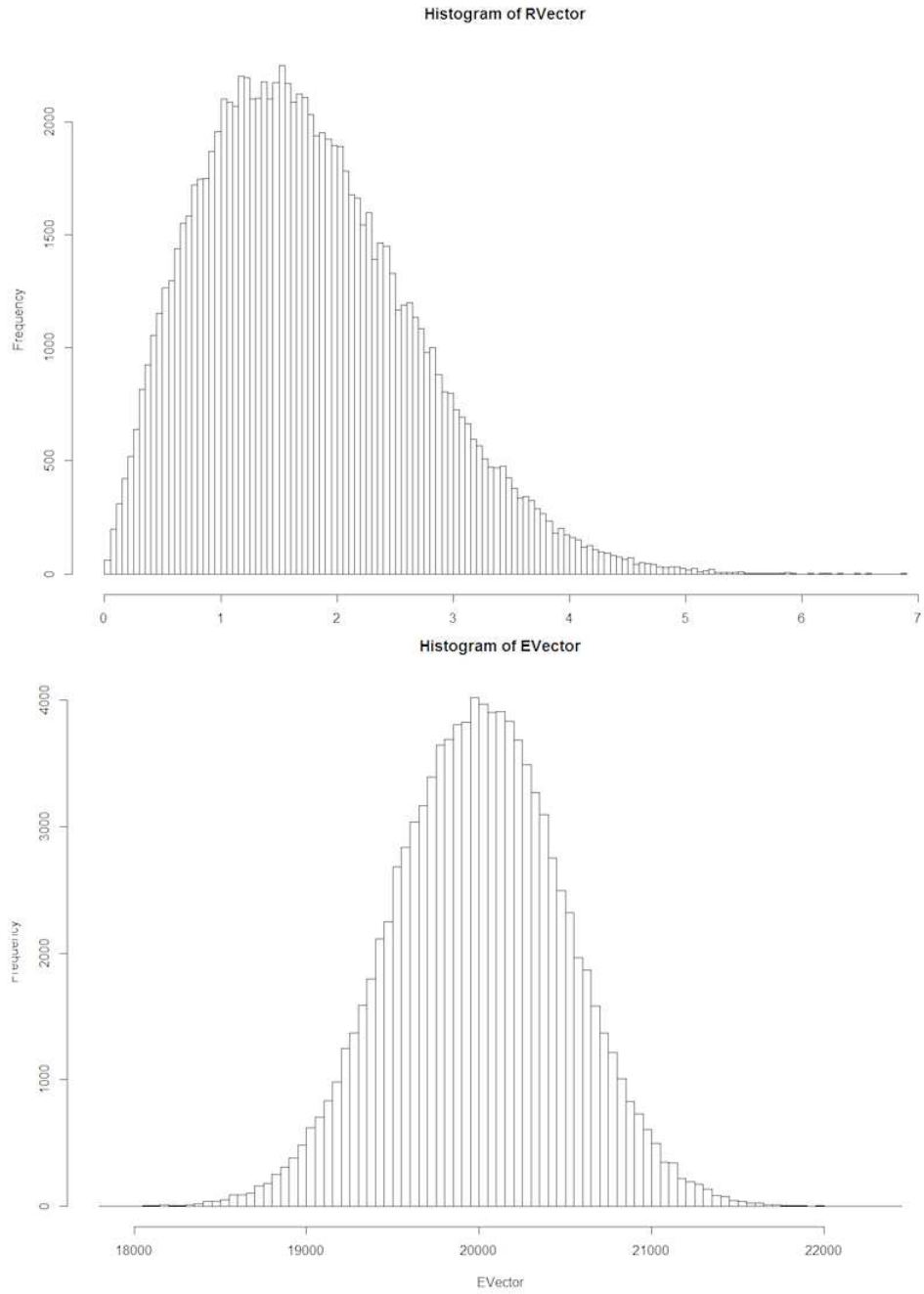


Figure 18: Variation of particle density with radius and the energy spectrum created with the `hist()` command.


```

START /WAIT RScript Sect09DemoScript.R "C:\Examples\Section09\" "1.4"
START /WAIT C:\fieldp_basic\gambet\gambet BatchDemo
RENAME BatchDemo.GLS BatchDemo01.GLS
RENAME BatchDemo.G3D BatchDemo01.G3D
...

```

The `/WAIT` option in the `START` commands ensures that the `SRC` file is available before starting **GamBet** and that there will not be a file access conflict when **GamBet** is running. Consider the first command. **RScript** is a special form of **R** to run scripts. To avoid typing the full path to the program every time, I modified the Windows path to include

```
C:\Program Files\R\R-3.2.0\bin\
```

The next command-line parameter is the name of the script followed by two character values that are passed to the script. The first is the working directory for the data and the second is the value of *Xw*.

Here's how the script we have discussed is modified to act as an autonomous program. The first set of commands becomes:

```

rm(list=objects())
args = commandArgs()
WorkDir = args[6]
setwd(WorkDir)

```

The second command stores the command line parameters in a character vector *args*. You may wonder, why start with `args[6]`? A listing of *args* gives the following output:

```

[1] C:\PROGRA~1\R\R-32~1.0\bin\i386\Rterm.exe
[2] --slave
[3] --no-restore
[4] --file=Sect09DemoScript.R
[5] --args
[6] C:\Examples\Section09\
[7] 1.4

```

The first argument is the name of the running program, a typical convention in **Windows**. The next four are the values of options that may be set in **RTerm**. The values of interest start at the sixth component. The next change is in the initialization section:

```
Xw = as.numeric(args[7])
```

Because command-line parameters are strings, we force the value to be interpreted as a number to define Xw . Finally, suppose we want to inspect the histograms, even though the program is running in the background. The ending statements are modified to:

```
pdf(file="Test.pdf")
hist(RVector,breaks=100)
hist(EVector,breaks=100)
dev.off()
```

The results of all graphics operations between `pdf()` and `dev.off()` are sent to the specified PDF file where they can be viewed after the run.

10 Generating arbitrary distributions

The **R** package includes functions to create a wide array of distributions of interest in mathematics. Nonetheless, there are many particle distributions of interest in physics (such as differential cross sections) that are not directly represented. This section introduces a method to sample from any probability function that can be approximated by a set of points.

Let's start by creating some data to work with. We'll model the kinetic energy distribution of positrons emitted in the β decay of ^{22}Na . The maximum value is 540 keV. To approximate the probability mass density, we'll use one half cycle of a sine function skewed toward higher energies to represent Coulomb repulsion from the product nucleus. Copy and paste the following commands to the **RStudio** script editor:

```
rm(list=objects())
xmax = 540.0
nmax = 50
MassDens = function(x) {
  Out = sin(x*pi/(xmax))*(0.35+0.4*x/xmax)
  return(Out)
}
xval = c(seq(from=0.0,to=xmax,length.out=(nmax+1)))
mdens = numeric(length=(nmax+1))
for (n in 1:(nmax+1)) {
  mdens[n] = MassDens(xval[n])
}
plot(xval,mdens)
curve(MassDens,0.0,xmax,xname="x",add=TRUE)
```

The commands

```
MassDens = function(x) {
  Out = sin(x*pi/(xmax))*(0.35+0.4*x/xmax)
  return(Out)
}
```

define a function that follows the curve of Fig. 19. The command

```
xval = c(seq(from=0.0,to=xmax,length.out=(nmax+1)))
```

creates a vector of 51 points equally spaced along the energy axis from 0.0 to 540.0 keV. The commands

```
mdens = numeric(length=(nmax+1))
for (n in 1:(nmax+1)) {
  mdens[n] = MassDens(xval[n])
}
```

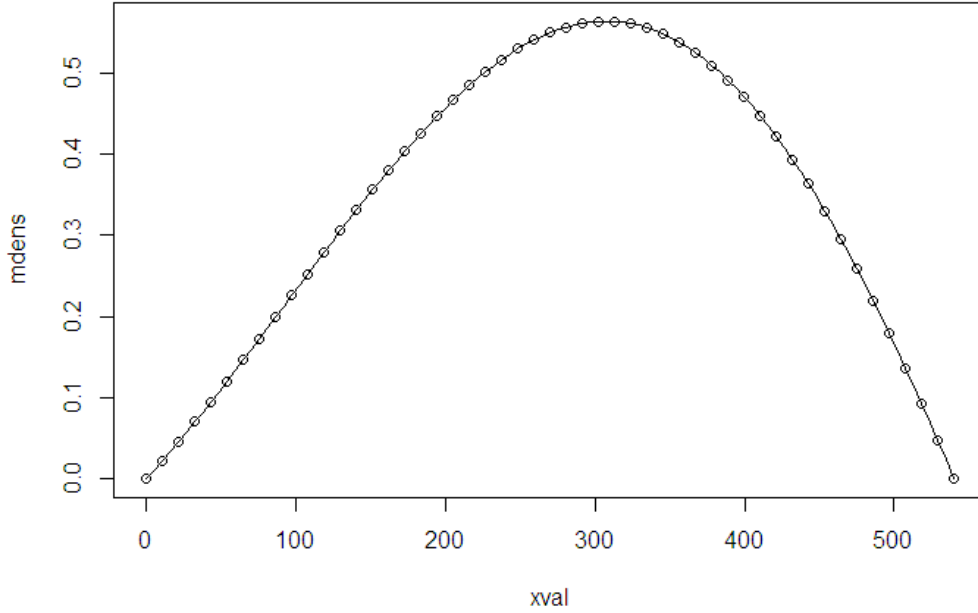


Figure 19: Positron spectrum. The relative probability mass density is represented by 51 points.

create an empty vector to hold the probability mass density $p(x)$ and fills the values with a loop that uses the *MassDens* function. At this point, the relative probability is sufficient – it is not necessary to worry about normalization.

To assign energy values for a particle distribution, we need the cumulative probability distribution, defined as

$$P(x) = \int_{x_{min}}^x p(x') dx'. \quad (8)$$

The function $P(x)$ equals the probability that the energy is less than or equal to x . It has a value of 0.0 at x_{min} and 1.0 at x_{max} . The following commands use the trapezoidal rule to perform the integral of Eq. 8:

```
dx = xmax/nmax
cdens = numeric(length=(nmax+1))
cdens[1] = 0.0
for (n in 2:(nmax+1)) {
  cdens[n] = cdens[n-1] + (mdens[n-1]+mdens[n])*dx/2.0
}
cdens = cdens/cdens[51]
```

Note that the final command normalizes the function. The result is a vector *cdens* of 51 points to represent the cumulative distribution.

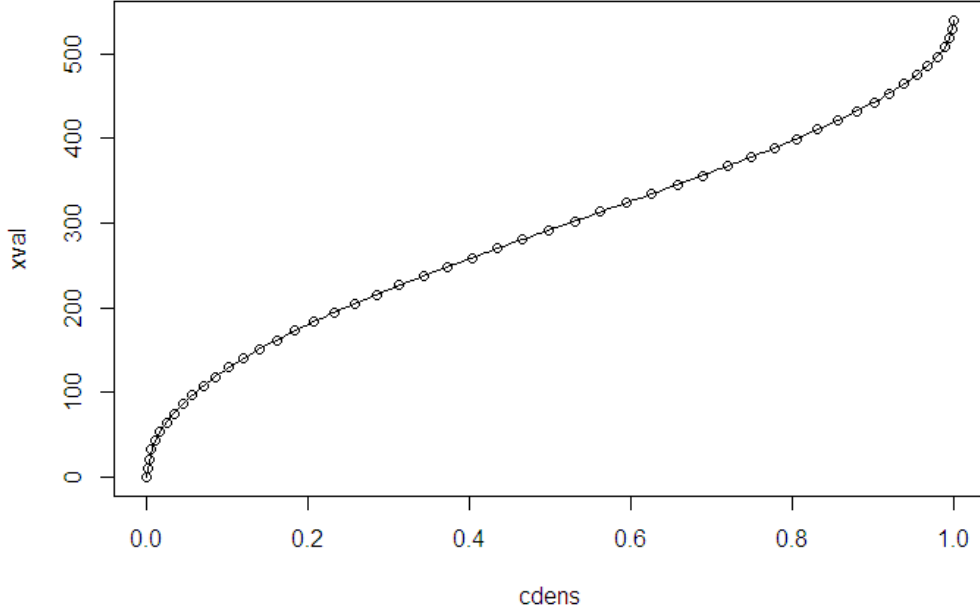


Figure 20: Plot of x versus $P(x)$ (the cumulative distribution function) showing data points and a spline interpolation.

The cumulative probability $P(x)$ is a monotonically-increasing function of x – there is a unique value of x for every value of $P(x)$. Therefore, we can view x as a function of $P(x)$, as illustrated by Fig. 20 created with the command:

```
plot(cdens,xval)
```

We can also make an accurate calculation of x for any $P(x)$ using the spline interpolation functions of **R**. The line in Fig. 20 was created with the command:

```
lines(spline(cdens,xval))
```

Figure 21 illustrates the principle underlying of the sampling method. Suppose we want to create 10,000 particles. By the definition of the cumulative probability distribution, a total of 1000 of the particles should be contained within the interval $0.6 \leq P \leq 0.7$. The graph show that these particles should be assigned energies in the range $320 \text{ keV} \leq x \leq 360 \text{ keV}$. In other words, if ζ represents a uniform sequence of 10,000 numbers from 0.0 to 1.0, then the desired distribution will result if the energy values are assigned according to

$$x_n = P^{-1}(\zeta_n) \quad (9)$$

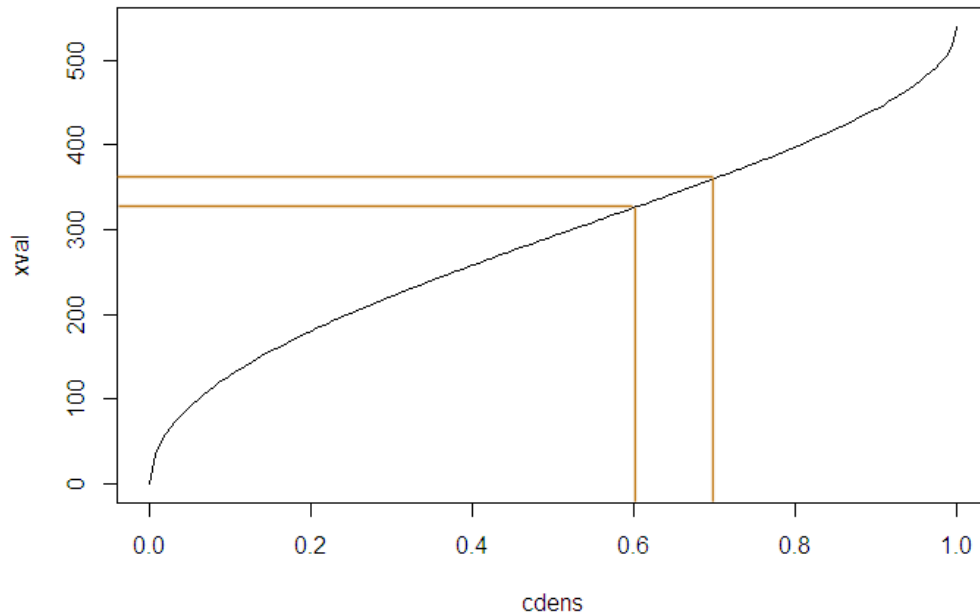


Figure 21: Energy values associated with intervals of the cumulative distribution function.

The **R** expression for the assignment operation uses a form of the `spline()` function that creates values at specified points:

```
NMax = 10000
dpoints = numeric(length=NMax)
zetavals=c(seq(from=0.0,to=1.0,length.out=NMax))
ztemp = spline(cdens,xval,xout=zetavals)
dpoints = ztemp$y
hist(dpoints,breaks=35)
```

In this case, the `spline()` function returns a data frame containing both the independent and dependent values. The assigned energies are set equal to the dependent values, `dpoints = ztemp$y`. The top graph in Figure 22 shows a histogram of the resulting distribution.

The routine creates the same distribution each time the script is run. In some circumstances, you may want to add variations so that runs are statistically independent. In this case, the uniform sequence of ζ values may be replaced with a random-uniform distribution:

```
zetavals=runif(NMax,0.0,1.0)
```

The lower graph in Fig. 22 shows the result.

In conclusion, the **R** package has a vast set of available commands and options that could occupy several textbooks. In this tutorial, I've tried to cover the fundamental core. My goal has been to clarify the sometimes arcane syntax of **R** so you have the background to explore additional functions. A compendium of scripts and data files for the examples is available. To make a request, please contact us at `techinfo@fieldp.com`.

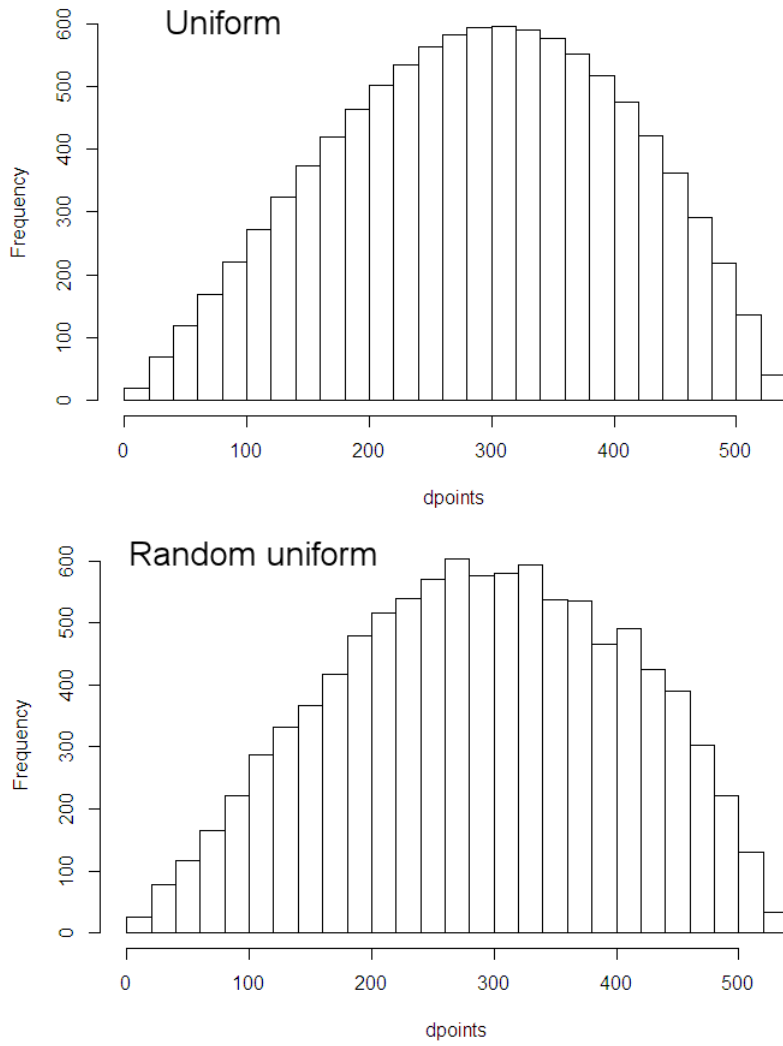


Figure 22: Histograms of the energy spectra of 10,000 particles. Top: uniform distribution of ζ . Bottom: Random-uniform distribution of ζ .

Index

- batch file, [46](#)
- batch file START command, [46](#)
- beta decay, [51](#)

- command line variables, [49](#)
- confidence interval, [34](#)
- criterion, [12](#), [29](#), [39](#), [44](#)
- CSV file, [10](#)
 - header, [10](#)
 - reading, [11](#)
 - write from FORTRAN, [26](#)
 - writing from R, [15](#)
- cumulative probability distribution, [52](#)
- curve fitting, [16](#)

- data frame, [10](#)
 - column names, [8](#), [10](#), [12](#), [20](#), [39](#), [44](#)
 - create a subset, [12](#), [28](#)
 - definition, [8](#)
 - number of columns, [37](#)
 - number of rows, [37](#)
 - reference column values, [8](#)
 - truncate, [37](#)
- differential cross section, [51](#)
- distribution generation, [43](#), [51](#)
- dose, [24](#)

- filtering by particle type, [40](#)
- fitting object, [14](#), [18](#), [28](#)
- for loop, [44](#), [52](#)
- formula syntax, [12](#), [17](#), [28](#), [33](#)
- Fourier analysis, [20](#)
- functions in R, [20](#), [51](#)

- GamBet, [3](#), [24](#), [36](#), [43](#), [46](#)
 - escape file, [36](#), [43](#)
 - matrix file, [36](#), [40](#)
 - output file types, [36](#)
 - SRC file, [36](#), [43](#)
- Gaussian function, [31](#), [43](#)
- GBView2, [36](#)
- GBView3, [36](#)
- GenDist, [3](#), [43](#), [46](#)

- how to
 - add a column to a data frame, [45](#)
 - calculate fitted values, [18](#)
 - clear the R console, [5](#)
 - combine vectors in a data frame, [8](#)
 - concatenate strings, [6](#)
 - create a sequence of numbers, [19](#)
 - create a vector, [8](#)
 - create an empty vector, [44](#), [51](#)
 - define a fitting formula, [14](#)
 - define a function, [20](#), [51](#)
 - delete all objects, [7](#)
 - find confidence intervals, [33](#)
 - find the size of a data frame, [37](#)
 - force string to numerical value, [49](#)
 - format numbers for output, [45](#)
 - generate any distribution, [51](#)
 - list objects, [5](#)
 - make a linear fit, [14](#)
 - make a nonlinear fit, [31](#), [32](#)
 - make a normal distribution, [22](#)
 - make a polynomial fit, [17](#), [24](#)
 - make plots in background operation, [50](#)
 - obtain example files, [55](#)
 - pick a directory interactively, [38](#)
 - pick a file interactively, [39](#)
 - plot a fitting line, [19](#), [23](#)
 - plot a histogram, [54](#)
 - plot a mathematical curve, [21](#)
 - plot a spline fit, [53](#)
 - read a CSV file, [11](#)
 - read command-line variables, [49](#)
 - renormalize a vector, [12](#)
 - run a script from the console, [9](#)

- run a script line, [7](#)
- run R from a batch file, [46](#)
- see the values of objects, [5](#)
- set the working directory, [11](#)
- set up a for loop, [44](#), [52](#)
- set up an if statement, [37](#)
- write a CSV file, [15](#)
- write a data frame to a file, [46](#)
- write a line to a file, [45](#)

if structures, [37](#)

integration, numerical, [52](#)

interactive script

- choose directory, [37](#)
- choose files, [37](#)

inverse function, [53](#)

linear fitting, [12](#)

- definition, [16](#)

logical and, [12](#), [29](#), [44](#)

logical not, [38](#)

logical or, [39](#)

Monte Carlo codes, [3](#)

multivariate, [24](#)

nonlinear fitting, [31](#)

- definition, [16](#)
- starting conditions, [33](#)

normal distribution, [22](#), [44](#)

OpenOffice Calc, [10](#)

parameter estimation, [16](#), [31](#)

plot, [12](#)

- connected lines, [19](#)
- fitting curve, [19](#), [23](#), [33](#)
- histogram, [39](#)
- mathematical curve, [21](#)
- set style, [8](#)
- simple y versus x, [8](#)
- slope/intercept line, [15](#)
- to PNG file, [28](#)

plot fitting curve, [28](#)

PNG file, [28](#)

polynomial fit, [16](#), [24](#)

R

- advantages, [4](#)
- basic arithmetic, [5](#)
- case sensitive, [6](#)
- console, [5](#)
- data frame, [8](#)
- file input, [10](#)
- help, [5](#), [24](#)
- libraries, [24](#)
- object definition, [5](#)
- objects listing, [24](#)
- reference card, [4](#)
- script, [7](#)
- string variables, [6](#)
- where to obtain, [4](#)

R command

- lines(), [28](#)
- plot(), [28](#)
- subset(), [28](#)

R commands

- abline(), [15](#)
- as.numeric(), [49](#)
- c(), [8](#), [18](#), [44](#)
- cat(), [45](#)
- cbind(), [45](#)
- character(), [44](#)
- coef(), [15](#)
- commandArcs(), [49](#)
- confint(), [32](#)
- curve(), [21](#)
- data.frame, [8](#)
- data.frame(), [18](#), [22](#), [44](#)
- dev.off(), [28](#), [50](#)
- for(), [44](#)
- format(), [45](#)
- function, [20](#)
- hist(), [39](#), [46](#), [50](#)
- I(), [17](#)
- if(), [37](#)

- length(), 44
- library(), 37
- lines(), 19
- list(), 33
- lm(), 12, 17, 22, 28
- mean(), 8, 28, 39
- ncol(), 37
- nls(), 32
- nrow(), 37
- numeric(), 51
- objects(), 5
- paste(), 6, 37, 45
- pdf(), 50
- plot(), 8, 12, 19
- png(), 28
- predict(), 18
- read.csv(), 11, 28
- read.table(), 32, 37, 42
- rm(), 7, 28
- rnorm(), 22, 44
- seq(), 19, 22, 44
- setwd(), 11, 28
- source(), 9
- spline(), 53
- sqrt(), 45
- subset(), 12, 39, 44
- summary(), 14, 18, 28, 33
- write.csv, 15
- write.table(), 46
- random noise, 20
 - spreadsheet, 10
- round-off errors, 29
- RScript, 46
- RStudio, 43
 - features, 24
 - screen layout, 24
 - where to obtain, 4
- scientific notation, 45
- script
 - as template, 39
 - automatic analysis, 43
 - run all lines, 8
 - run from console, 9
 - starting, 7, 11
 - step through lines, 8, 11
- slope/intercept line, 14
- splines, 53
- spreadsheet, 10, 32
- square root function, 45
- statistical software
 - choosing, 4
 - list link, 3
- straight line fit, 14
- summary of fitting object, 14
- tab/space delimited file
 - reading, 32
- technical software
 - advantages and drawbacks, 3
 - interface, 3
- trapezoidal rule, 52
- univariate, 16
- utils library, 37
- vector, 8
 - empty, 44
 - shift values, 12, 28
- Windows
 - batch file, 46
 - command line variable, 49
 - path environmental variable, 49