

# **GS01 0163**

## **Analysis of Microarray Data**

Keith Baggerly and Bradley Broom  
Department of Bioinformatics and Computational Biology  
UT M. D. Anderson Cancer Center

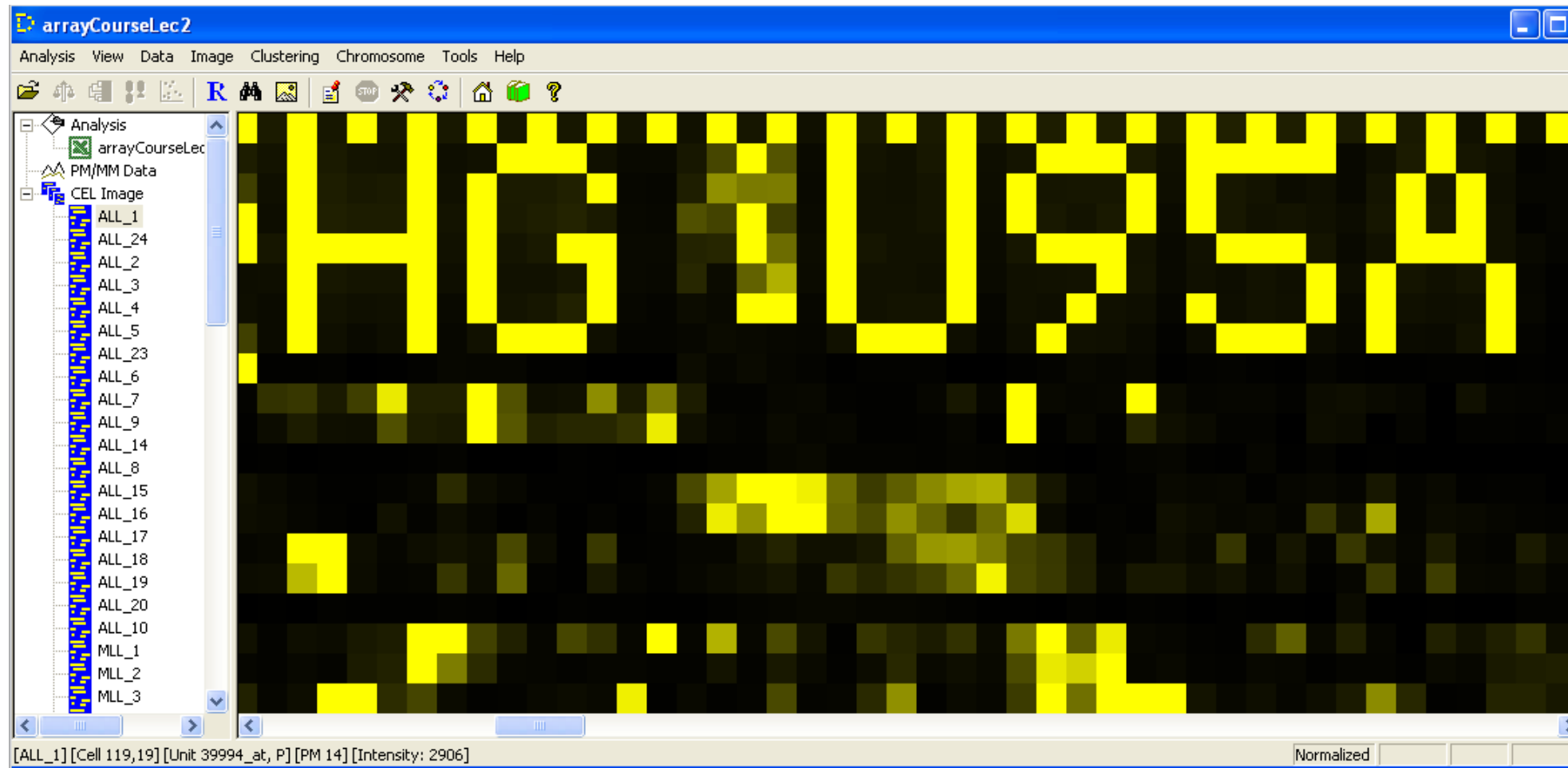
`kabagg@mdanderson.org`  
`bmbroom@mdanderson.org`

16 September 2010

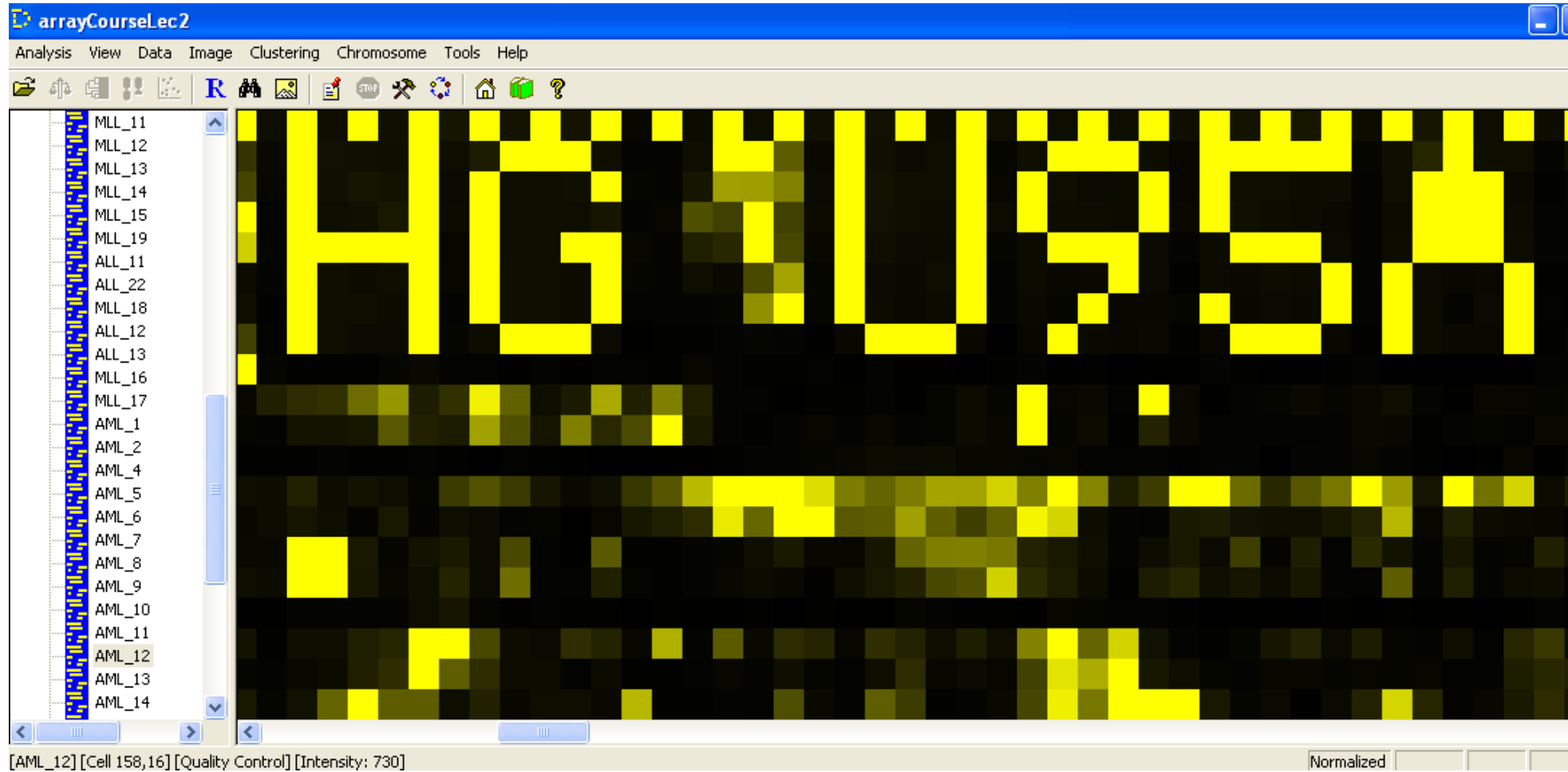
# Lecture 5: Introduction to R

- Limits of Canned Packages
- Introducing R: Some Background
- Installing R
- Learning About R
- Graphics in R
- Names and Attributes

# U95A



# U95Av2



# The Limits of dChip

Occasionally, we may not be happy with some results of a canned analysis package (even a good one).

We see something that looks suspicious.

We may want to ask more complex questions of the data than are allowed for in the context of the package.

In short, we want a general package for data analysis with some array structures built in that we can extend to deal with our specific quirks.

Enter programming...

# What is Programming?

Computers are DUMB.

Computers are incredibly, incredibly DUMB.

Computers do exactly what they are instructed to do.

Programming is creating a set of instructions for a computer to execute.

# Why is Programming Hard?

Computers are incredibly, incredibly DUMB.

Computers do exactly what they are instructed to do.

- Computers do not do what you want them to do.
- Computers do not do what you thought you told them to do.
- Computers do exactly what they've been instructed to do.

Because computers are so DUMB, we have to

- think of and write instructions for all possible contingencies,
- be incredibly precise when expressing those instructions.

# How are Computer Instructions Expressed?

Computers natively execute instructions encoded as patterns of 0's and 1's.

- Exceptionally hard for humans to follow
- Each instruction does very little
- Excessively time-consuming and expensive to write useful programs
- Only work on one kind of computer



# How are Computer Instructions Expressed?

Assembly languages use mnemonics to represent machine instructions.

- Existing program (assembler) converts mnemonics to machine instructions.
- Hard for humans to follow
- Each instruction does very little
- Very time-consuming and expensive to write useful programs
- Only work on one kind of computer

# How are Computer Instructions Expressed?

Third-generation languages use “high-level” commands that do not correspond exactly to machine instructions.

- Examples: Fortran, Cobol, Algol (60, 68, W), Snobol, BCPL, Bliss, C, C++, Java, Pascal, Modula (1, 2, 3), Lisp, Scheme, Prolog, ML, Haskell, ...
- Existing program (compiler) converts “high-level” commands to machine instructions.
- Relatively easy for humans to follow
- Somewhat time-consuming and expensive to write useful programs

- Can work many different kinds of computer (using appropriate compiler)
- Focus on execution efficiency

# How are Computer Instructions Expressed?

Scripting languages use “high-level” commands that do not correspond exactly to machine instructions.

- Examples: R, Perl, Python, Ruby, Matlab
- Existing program (interpreter) executes scripting language commands.
  - Usually provides an environment where commands can be entered and executed one-at-a-time
- Easiest for humans to follow
- Relatively fast and inexpensive to write useful programs
- Works on many different kinds of computer

- Focus on programming efficiency
- Often most useful for a specific kind of problem (e.g. string processing, statistical computation)

# What are Programming Languages?

Why not express instructions using English (or any other human language)?

- imprecise
- ambiguous

Programming languages are formal notations for expressing computer instructions.

Every programming language has detailed rules that specify what constitutes a valid program, and what that program will do.

These rules must be adhered to.

# Why R?

- R is a powerful, general purpose language and software environment for statistical computing and graphics.
- R runs on any modern computer system (including Windows, Macintosh, and UNIX/Linux).
- There already exists an extensive package of microarray analysis tools, called BioConductor, written in R.
- R and BioConductor are open source and free.

# Where Did R Come From?

Before R, there was S (lexicographers shudder...)

S was developed at AT&T by John Chambers and Richard Becker (brown book).

Extended to “New S” (blue book), and then to “S-Plus” (white book), with this last being commercialized.

More recently, there have been several versions of S-Plus, with more modifications of the code, and additional packages.



## Why Did S Catch On?

It had modules for several of the most common statistical operations.

It had a high-level syntax, which was familiar to many of the people working on it to begin with, and an interactive environment.

It incorporated options for basic graphics.

It allowed for the easy definition of relevant data structures, allowing analysts to group descriptive covariates so that the interrelationships could be easily explored, including access by name.

## Why Did S Catch On?

It arrived at the right time.

S became popular in the academic statistics community, in part as an alternative to SAS, and this popularity has continued.

It allowed people to write their own modules.

Repositories of modules that people had written became available (eg, statlib at Carnegie Mellon), extending functionality.

# Why R?

S is not free.

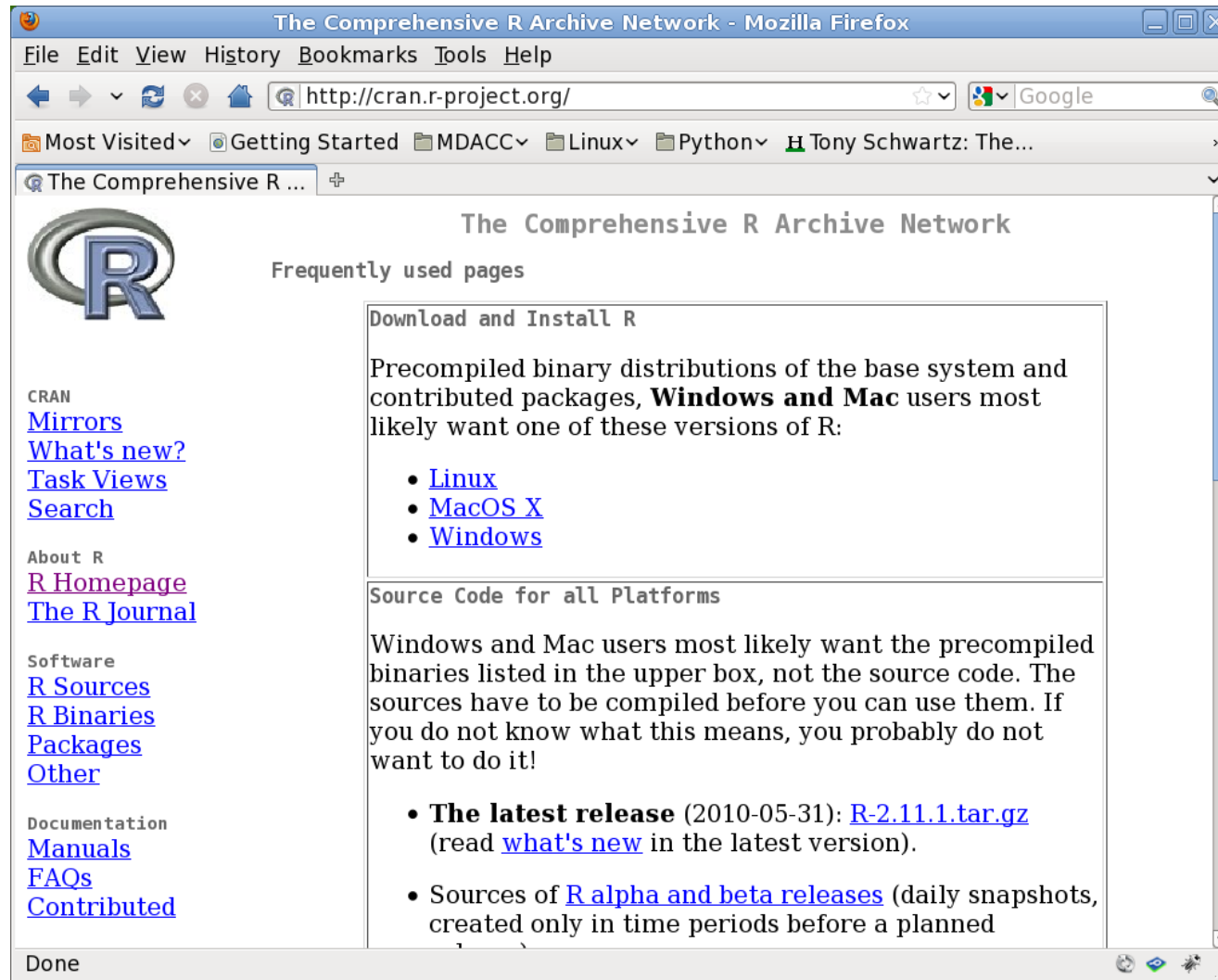
R is a free implementation of the S language.

- free in both price and liberty.
- there are some differences in the details.

Many of the older modules written for S have been ported over to R.

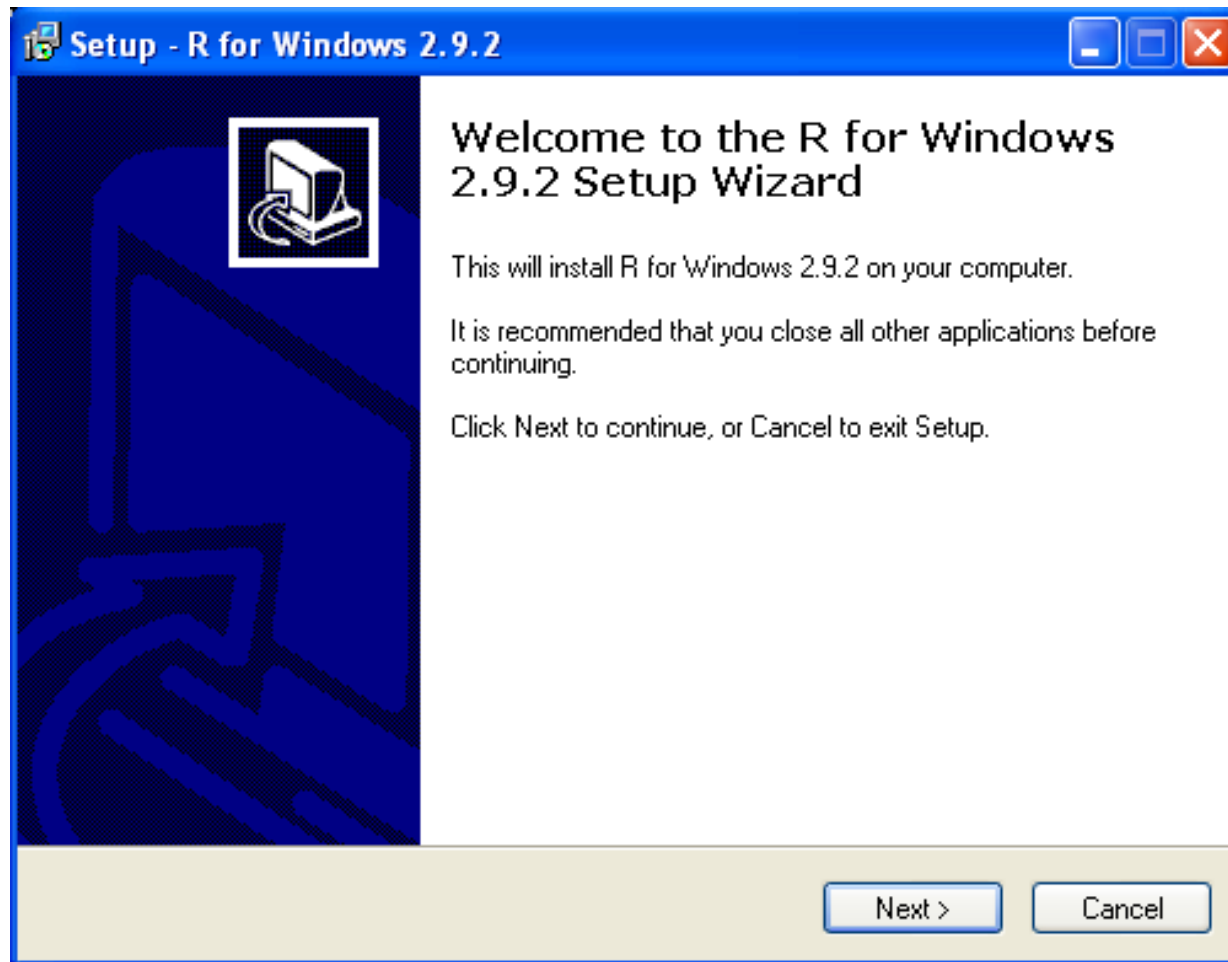
Many new modules specifically for bioinformatics (and mostly microarray analysis) have been collected and centralized as part of the Bioconductor project.

# The Comprehensive R Archive Network



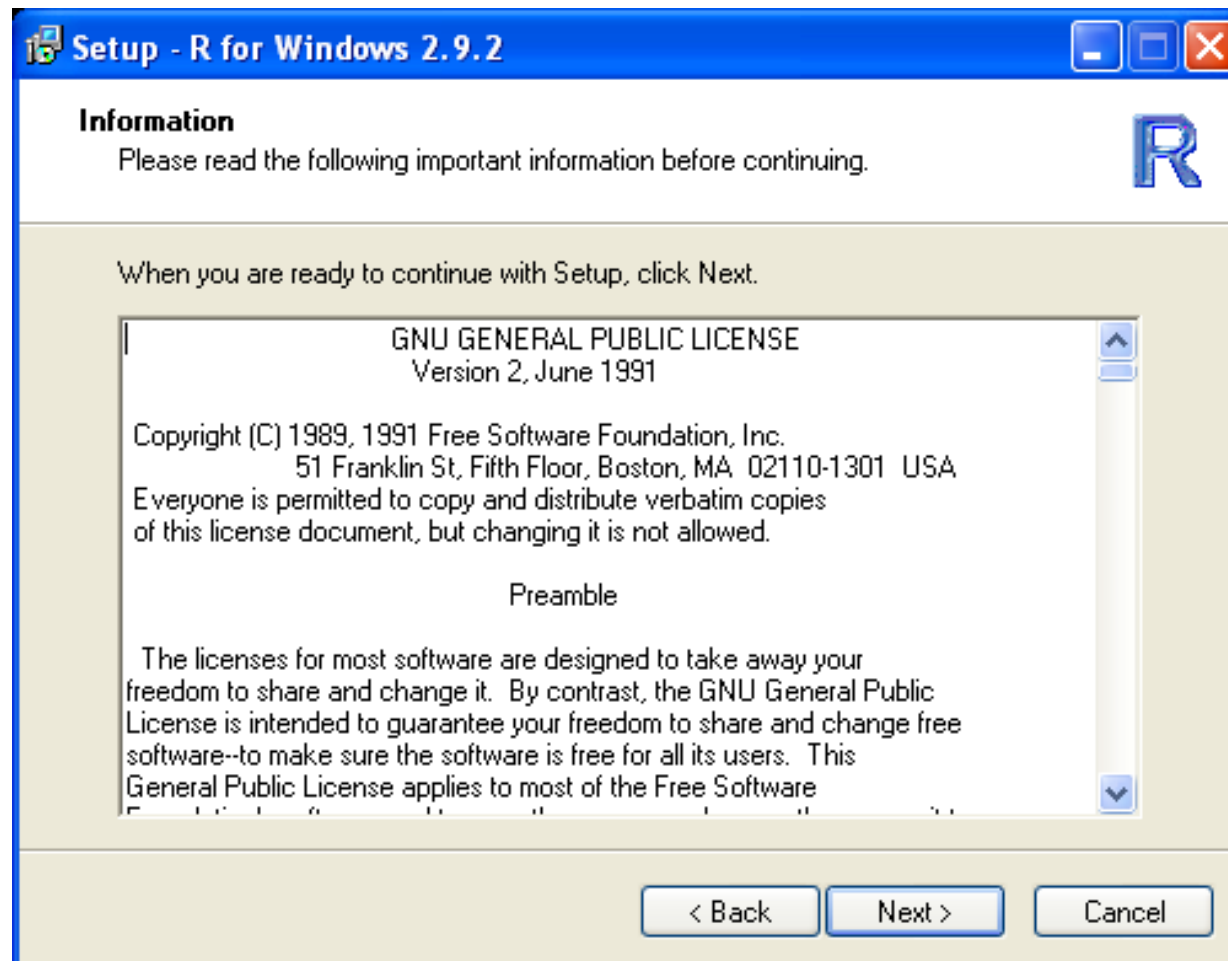
<http://cran.r-project.org>

# R Installation



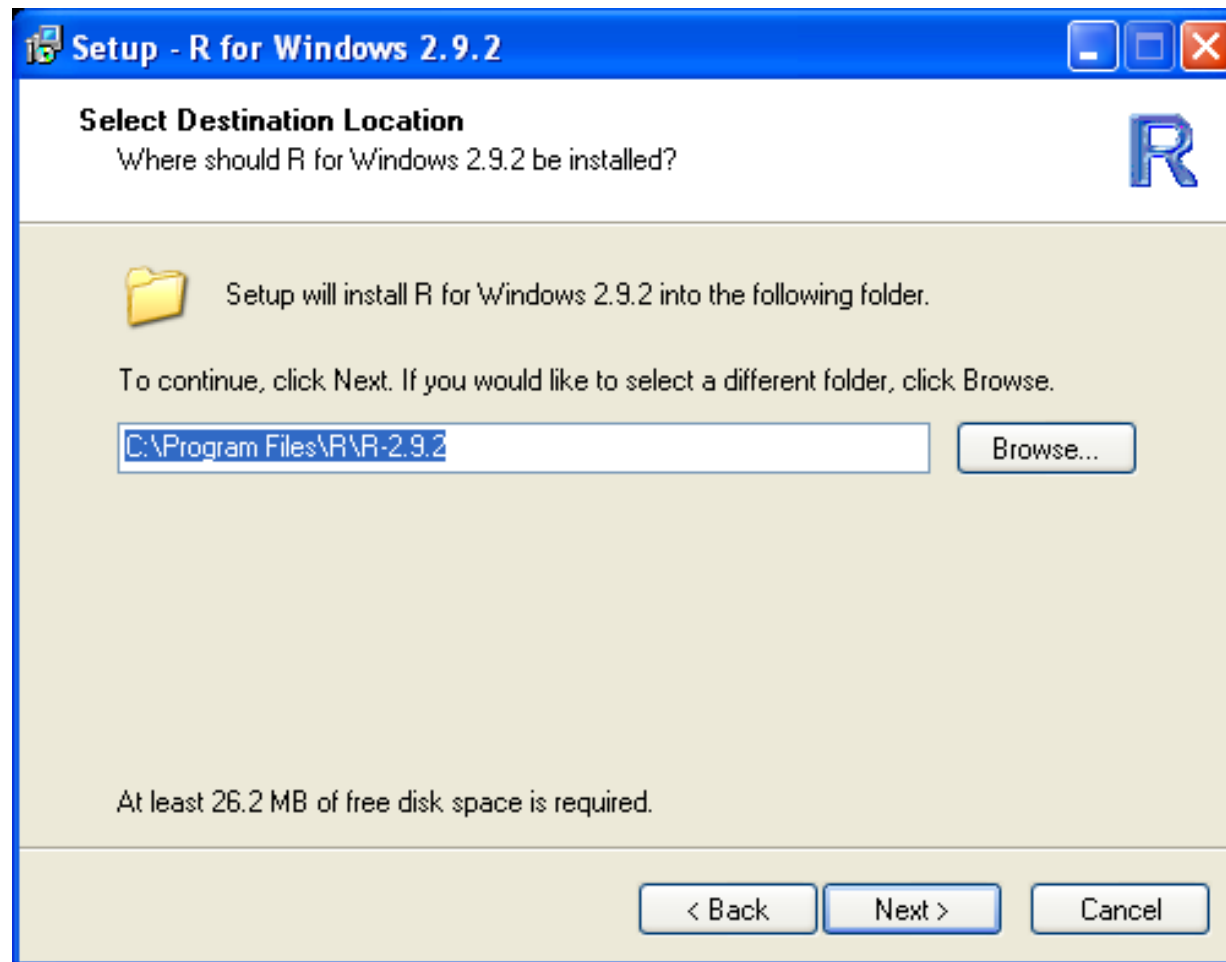
After downloading R from CRAN, you start the installation program and see this screen. Press “Next”.

# R Installation



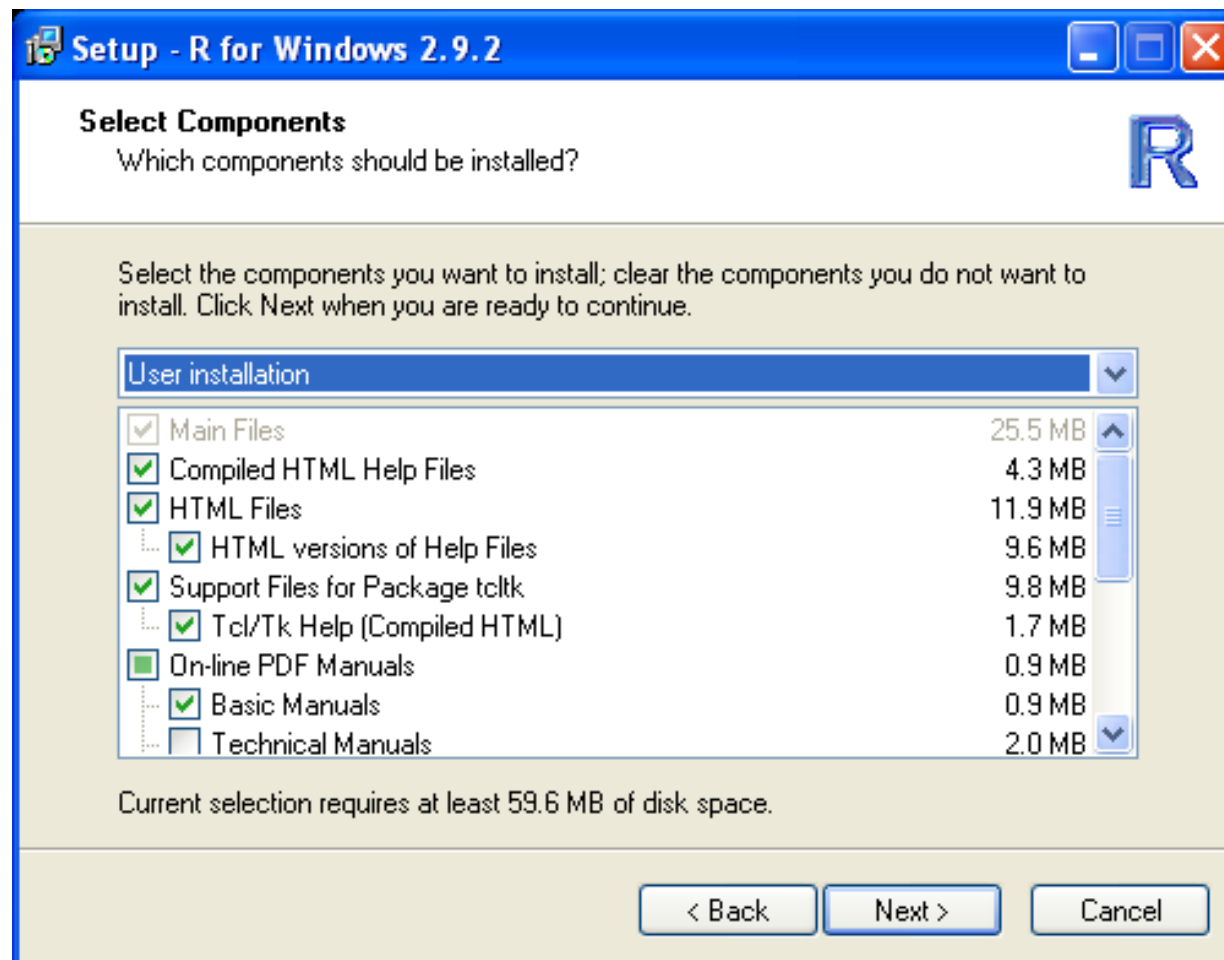
Click next to proceed.

# R Installation



You can change the installation path. It may be a good idea to choose a path name that does not include any spaces.

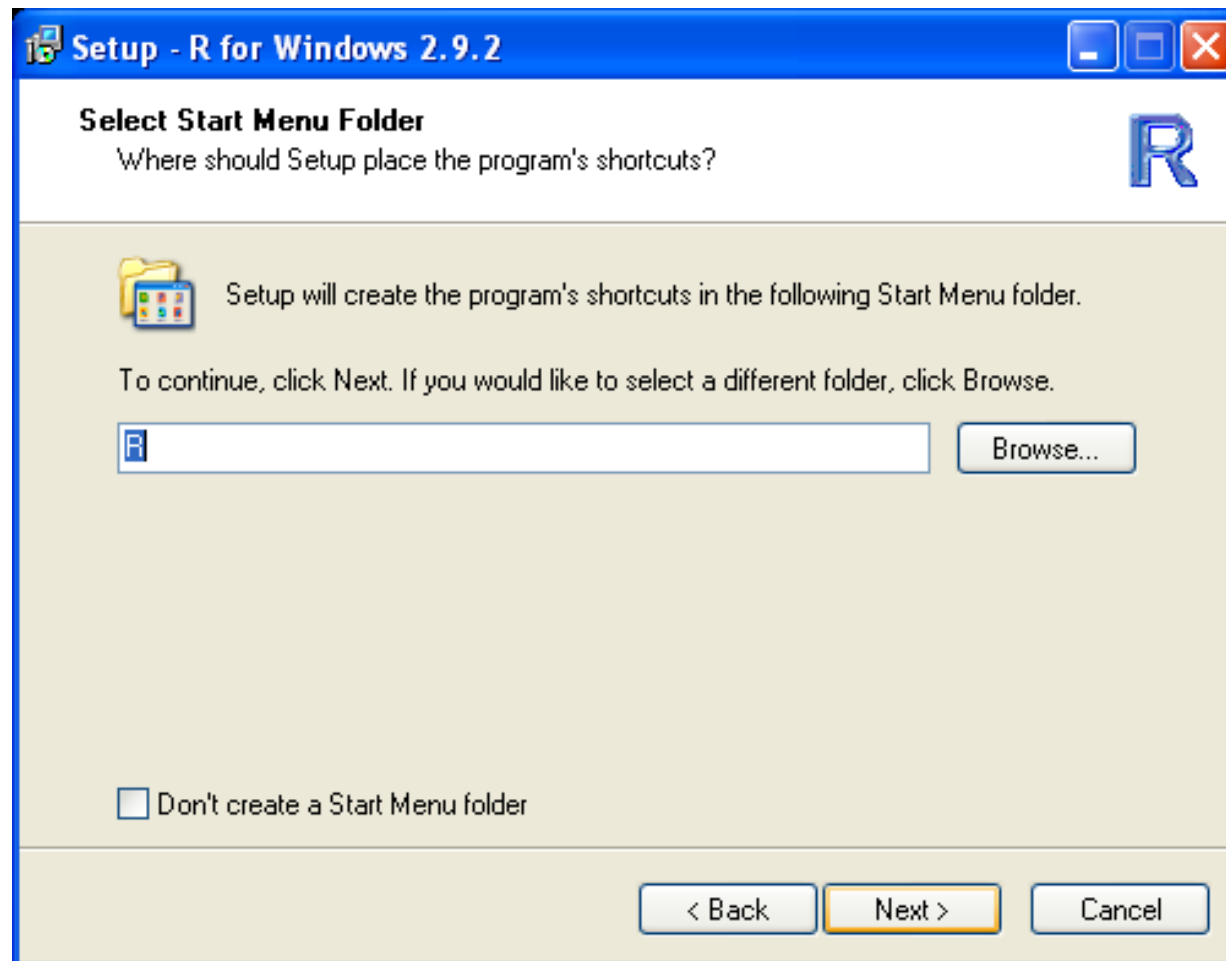
# R Installation



You can choose which pieces to install. In general, installing documentation and help files is a good idea.

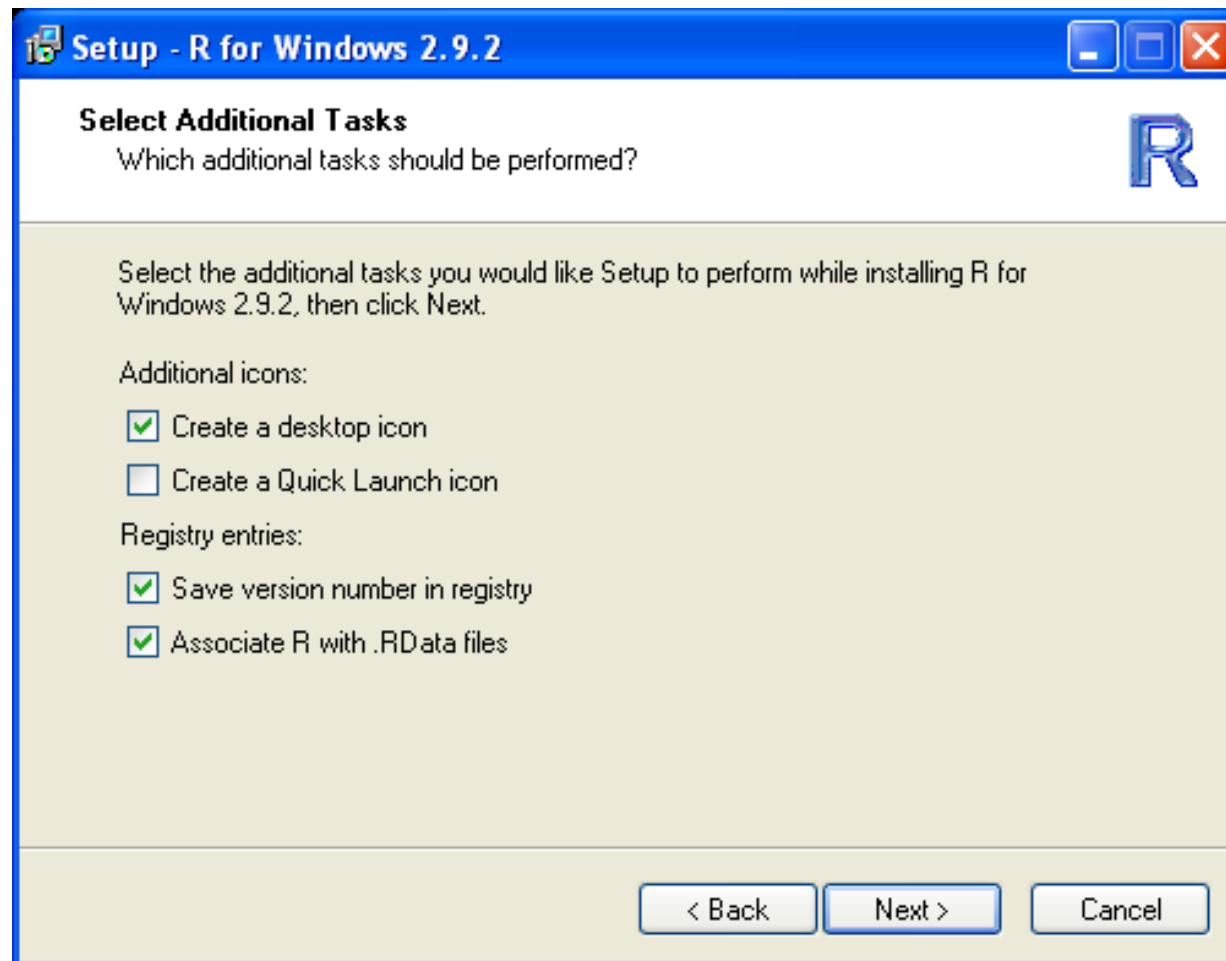


# R Installation



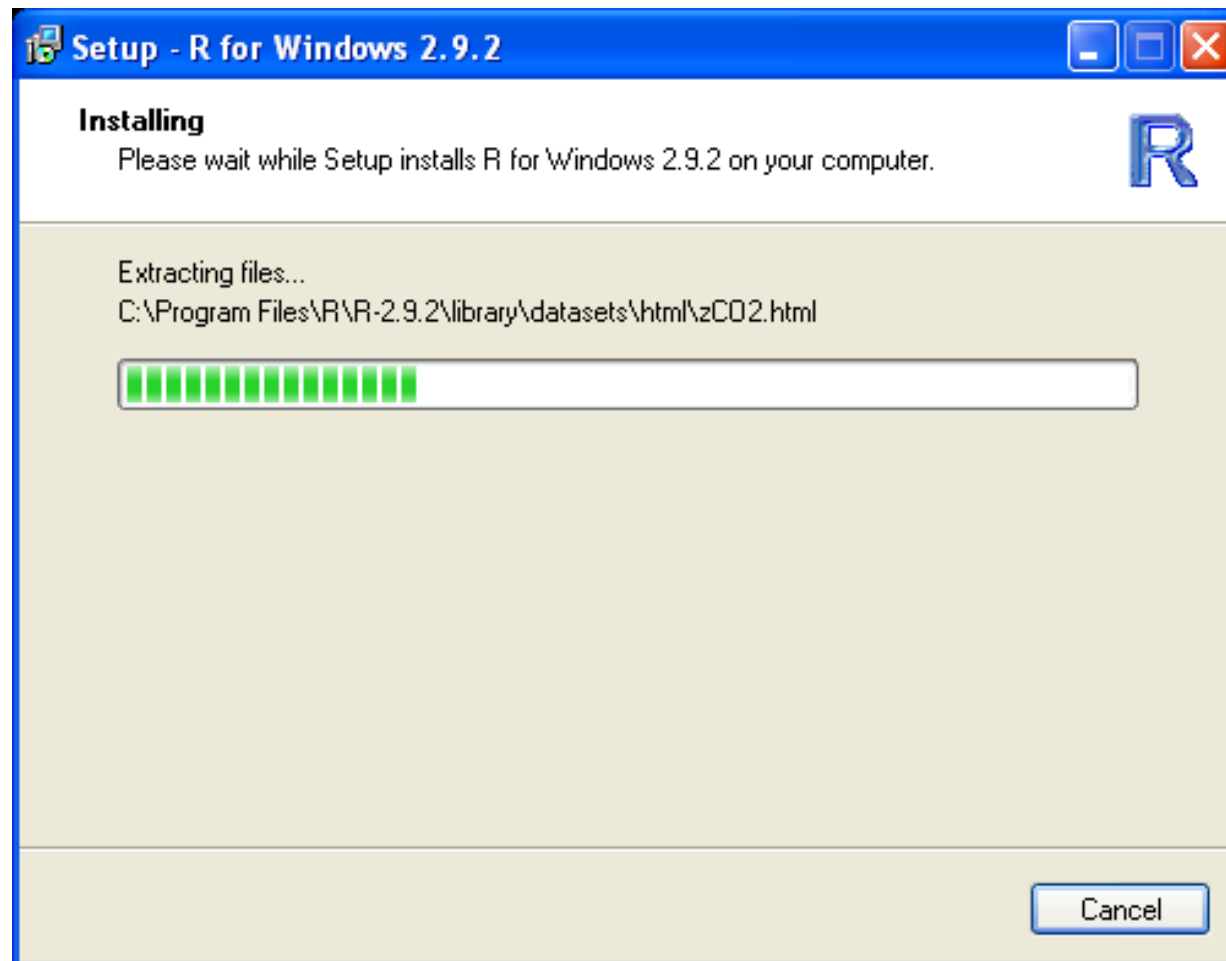
Decide whether to make a folder on the start menu.

# R Installation



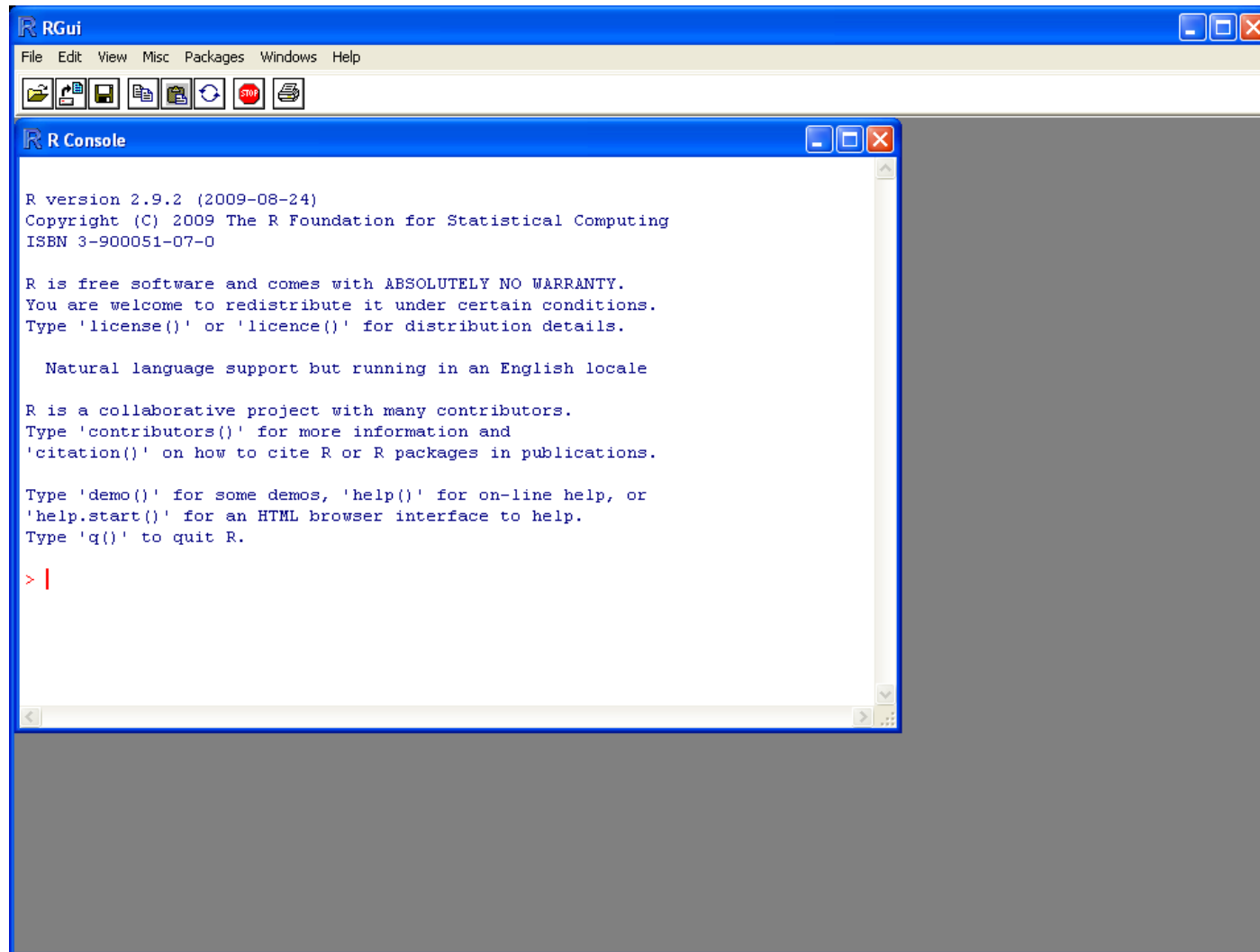
Decide whether to put an icon on the desktop.

# R Installation

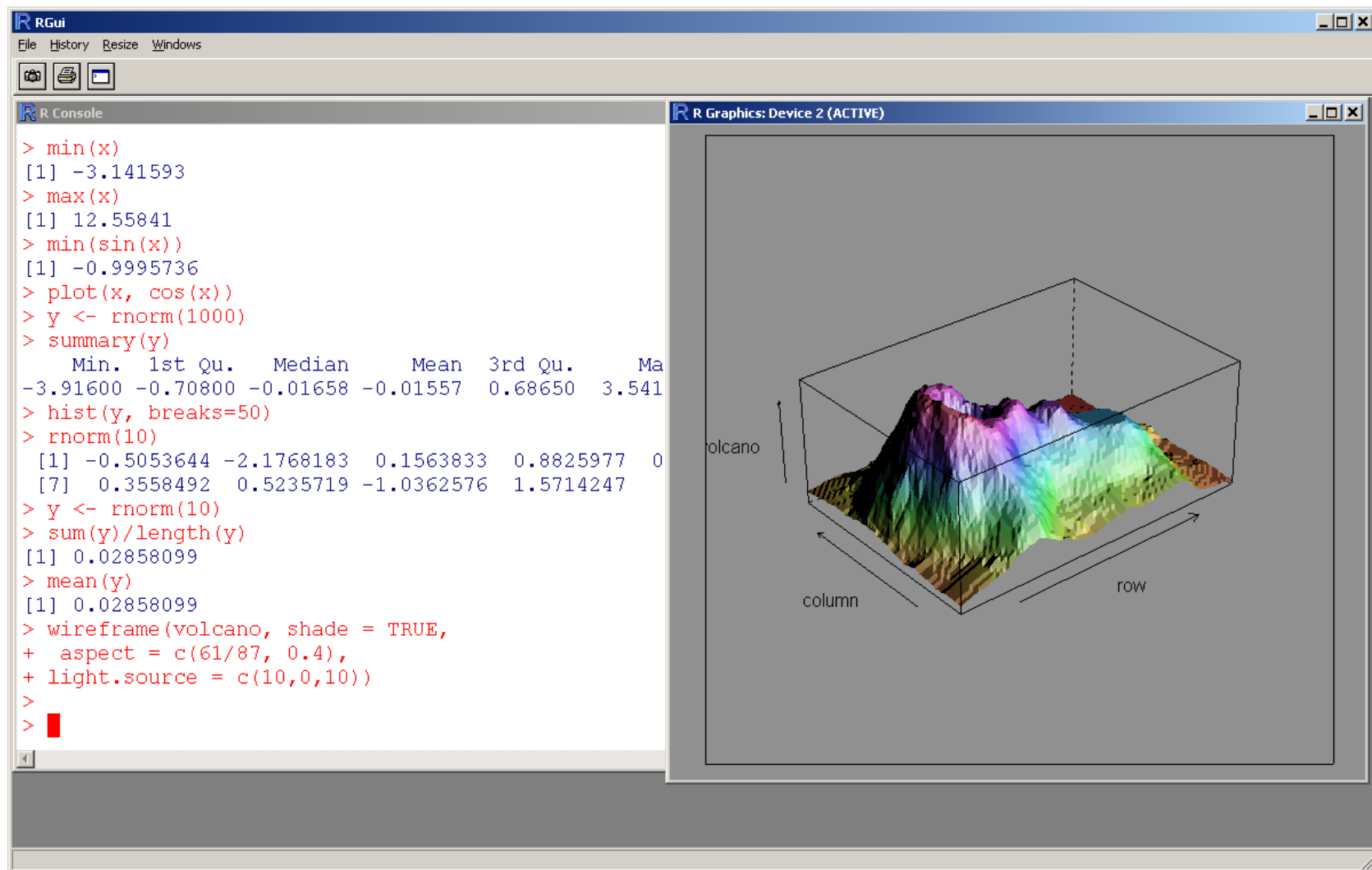


The program installs itself fairly quickly.

# The R Gui



# The R Gui



The screenshot displays the R GUI interface. The R Console window on the left shows the following R code and its output:

```
> min(x)
[1] -3.141593
> max(x)
[1] 12.55841
> min(sin(x))
[1] -0.9995736
> plot(x, cos(x))
> y <- rnorm(1000)
> summary(y)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
-3.91600 -0.70800 -0.01658 -0.01557  0.68650  3.541
> hist(y, breaks=50)
> rnorm(10)
 [1] -0.5053644 -2.1768183  0.1563833  0.8825977  0.
 [7]  0.3558492  0.5235719 -1.0362576  1.5714247
> y <- rnorm(10)
> sum(y)/length(y)
[1] 0.02858099
> mean(y)
[1] 0.02858099
> wireframe(volcano, shade = TRUE,
+ aspect = c(61/87, 0.4),
+ light.source = c(10,0,10))
>
>
> █
```

The R Graphics: Device 2 (ACTIVE) window on the right displays a 3D surface plot of a volcano. The plot is a wireframe with shading, showing a central peak. The axes are labeled 'column' (horizontal), 'row' (depth), and 'volcano' (vertical). The surface is colored with a gradient from blue at the base to red at the peak.

# Learning About R

We'll try to cover many of the tools that we'll commonly use, but we can't cover all aspects of the language or the packages. Fortunately, we don't have to, because many people have already kindly written documentation for us.

There are the manuals (if you must know all):

<http://cran.r-project.org/manuals.html>

and there is contributed documentation

<http://cran.r-project.org/other-docs.html>

The “R for Beginners” document by Emmanuel Paradis is a very good starting point.

## Notes on R: expressions

At heart, R is a command line program. You type commands in the console window. Results are displayed there, and plots appear in associated graphics windows.

R always prints a prompt (usually `>`) where you can type commands. If a line does not contain a complete command, then R prints a continuation prompt (usually `+`).

To evaluate an expression, enter it after the command prompt:

```
> 1+3*4  
[1] 13
```

Note that operators have precedence.

You can change the order of evaluation using parentheses:

```
> (1+3) * 4  
[1] 16
```

Note that the output is prefaced by the number “1” in brackets. Output often consists of vectors, and R tells you which item of the vector starts the output.

```
> 1:10+0.123  
[1] 1.123 2.123 3.123 4.123 5.123  
[6] 6.123 7.123 8.123 9.123 10.123
```

The `:` operator generates a sequence of numbers.

R has many operators and functions that accept and produce vectors, matrices, and other types of data collections as single entities.



## Notes on R: variables

You can save the result of any expression in a variable, which is similar to a “memory” in a calculator, except that

- you can have as many variables as you like, and
- you call each variable using a name of your choice.
  - names must begin with a letter (a–z or A–Z),
    - note that case is significant: `x` is not `X`.
  - names can also contain digits (0–9), period (`.`), and underscore (`_`)

Valid names include `x0`, `X0`, `p1`, `p1`

Invalid names include `_var1`, `my-last-$`, `2fast`.

## Notes on R: variables

To assign the value of an expression to a variable, you use the “assign operator”, made by typing `<` (less than) followed by `-` (minus), as in

```
x <- 2
```

which is pronounced “`x` is assigned 2”.

This command produces no output; it simply stores the value “2” under the name “`x`”. To retrieve the value, simply use it in an expression.

```
> x
```

```
[1] 2
```

# R Functions

R has lots of built-in functions.

```
> y <- rnorm(10)
> sum(y)
[1] -5.863182
> sum(y)/length(y)
[1] -0.5863182
> mean(y)
[1] -0.5863182
> sd(y)
[1] 0.9856325
```

A function call consists of the function name followed by a list of its parameters enclosed in parentheses. The parentheses are required even if there are no parameters.

## More About R Functions

R uses many “functions” to do things via side effects.

For example, to quit R use the `q` function:

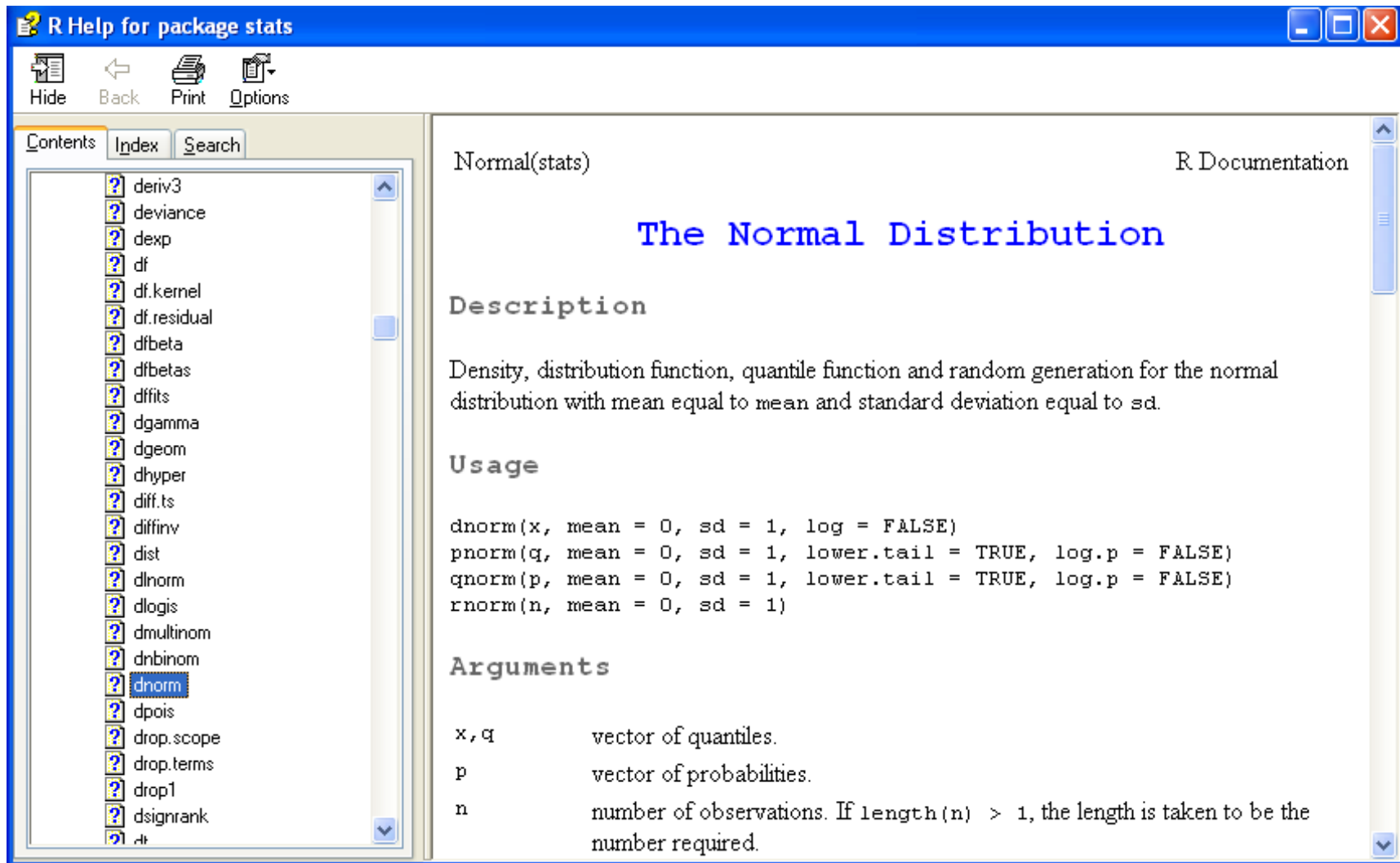
```
> q()
```

You can get help on functions using (surprise) the `help` function. For example,

```
> help(rnorm) # opens a help window
```

```
>
```

# R Help on rnorm



The screenshot shows the R Help window for the 'stats' package. The window title is 'R Help for package stats'. The left pane contains a list of functions, with 'dnorm' selected. The right pane displays the help page for 'dnorm', titled 'The Normal Distribution'. The page includes a description, usage examples, and arguments.

Normal(stats) R Documentation

## The Normal Distribution

### Description

Density, distribution function, quantile function and random generation for the normal distribution with mean equal to mean and standard deviation equal to sd.

### Usage

```
dnorm(x, mean = 0, sd = 1, log = FALSE)
pnorm(q, mean = 0, sd = 1, lower.tail = TRUE, log.p = FALSE)
qnorm(p, mean = 0, sd = 1, lower.tail = TRUE, log.p = FALSE)
rnorm(n, mean = 0, sd = 1)
```

### Arguments

x, q	vector of quantiles.
p	vector of probabilities.
n	number of observations. If <code>length(n) &gt; 1</code> , the length is taken to be the number required.

# Packages

How do you find out which functions are available? Every function in R is in a package, and packages come with documentation. To get help on the “stats” package, you would type

```
help(package=stats)
```

This will open a help window containing one-line descriptions of all functions in the package.

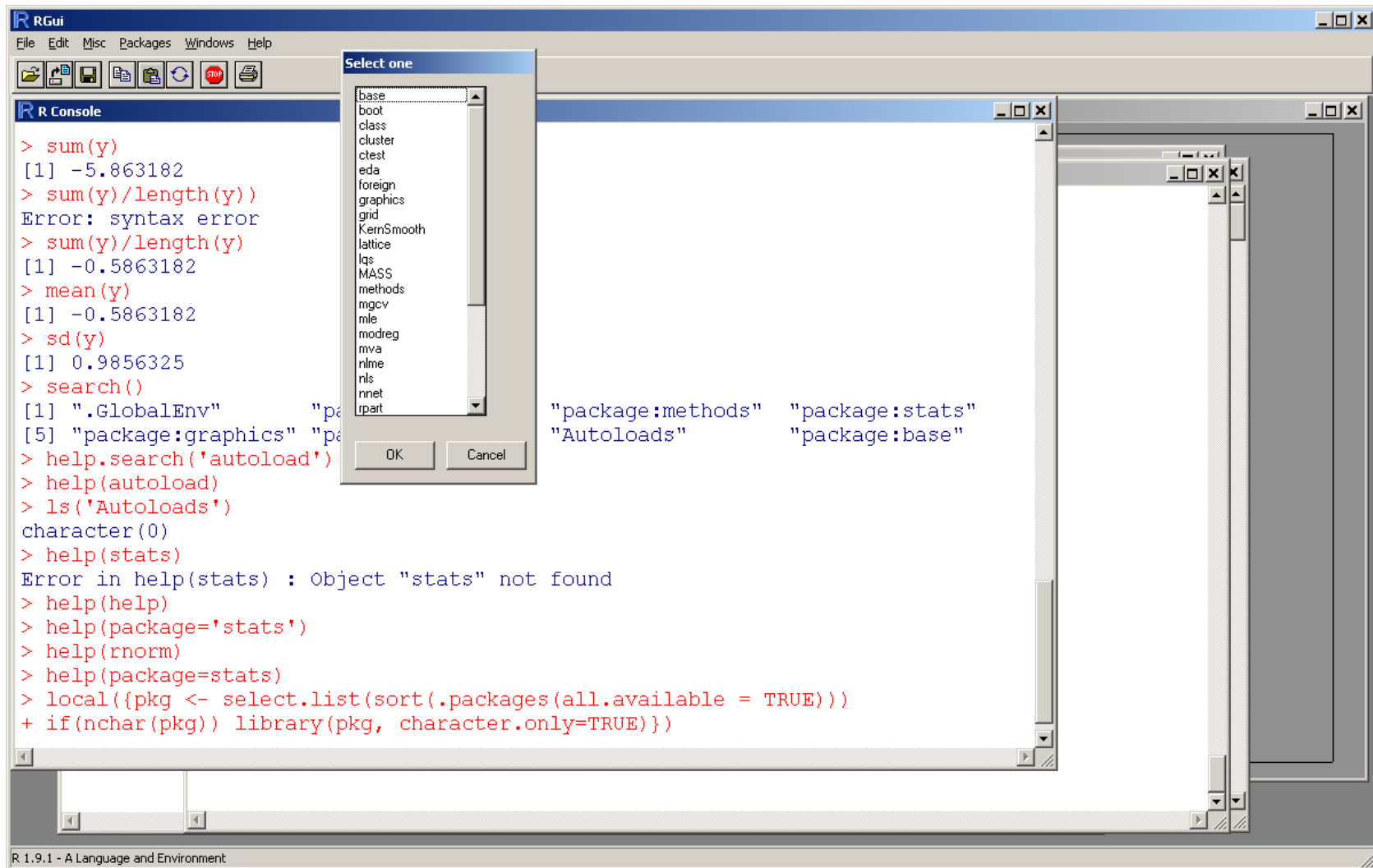
# Loading Installed Packages

When R starts, it loads the packages “base”, “utils”, “graphics”, “grDevices”, “datasets”, “methods”, and “stats”.

Other packages must be loaded explicitly, using either

- the `library` command, or
- make the console subwindow active, then select the menu item “Packages”, then “Load packages...”,
  - Note: Menu items in the R GUI change depending on the active subwindow.
- A dialog box with a list of installed packages is displayed, from which you can choose packages to load.

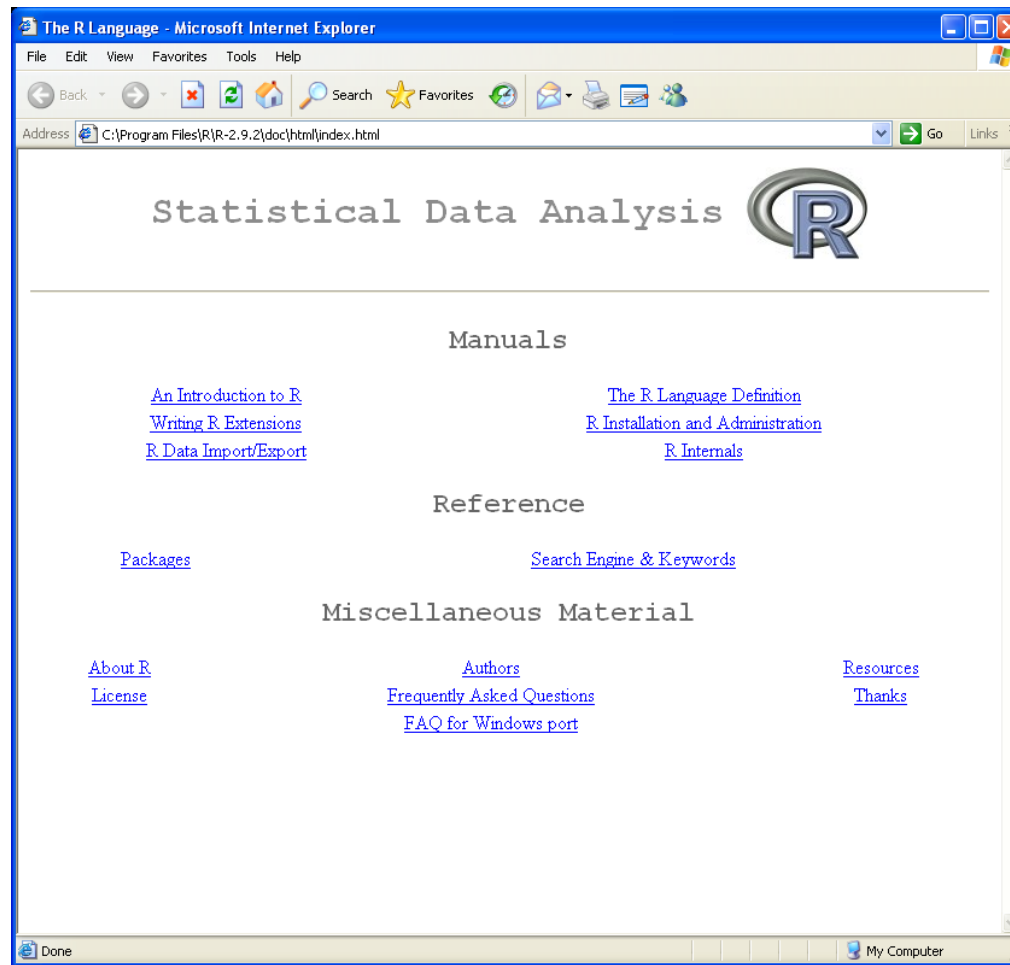
# GUI Loading Library Packages





# Browser-based R Help

You can also use the GUI menu item “Help” followed by “Html help” to open a web browser with help information.



## Where Have You Started R?

On the main menu bar, there is an option for “File”, under which there is the option to “Change dir...”. Selecting this option will open a new window that shows the location of R’s current working directory and allows you to change it.

Now, the initial working directory is probably not be where you’ll want to store the results of your analyses, so you will likely want to change this.

You can also find and change R’s working directory using the equivalent R functions:

```
> getwd()  
[1] "Users/bmbroom"  
> setwd("MicroarrayCourse/DataSets/Testing")
```

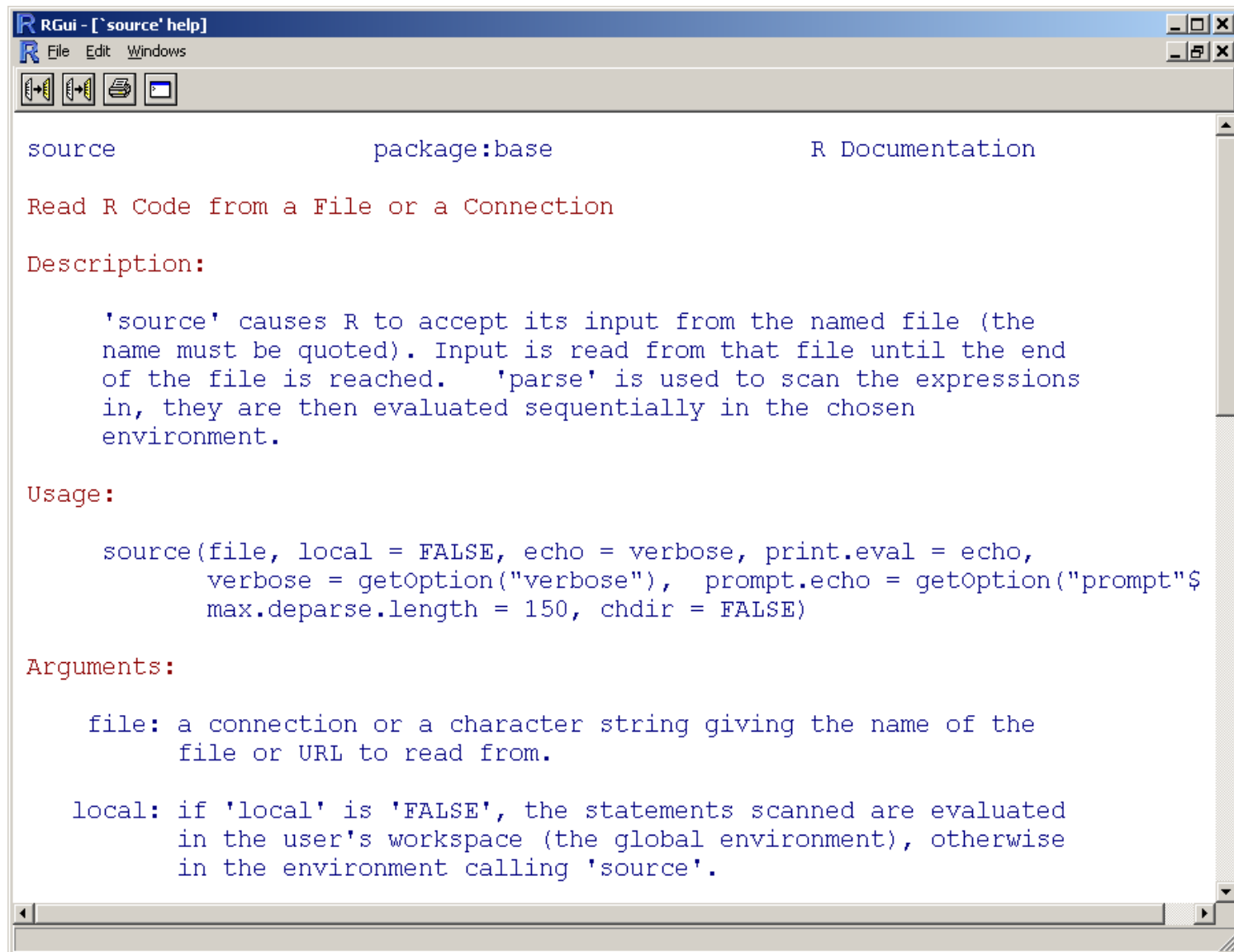
# Scripting

There are both good and bad aspects of R's interactive command-line interface. On the good side, it is very flexible. It encourages exploration, allowing you to try things out and get rapid feedback on what works and what doesn't.

On the bad side, record-keeping and documentation can be difficult. If you just type merrily away, you may have a hard time reconstructing exactly how you solved a problem. You'll need to devise a method for keeping better records than are possible just by typing things at the command line.

The critical R command that makes this possible is `source`.

# R Help for source



## Using source

One method for keeping track of how you solve a problem is to create a file containing the commands with the solution. You then `source` this file to produce the answer. You can use a “plain text” editor (like Notepad, but not Microsoft Word) to modify the commands if they don’t work correctly the first time around.

In newer versions of R, you can also use the built-in editor. You create a script using the menu item “File – > New script”, and you load an old script with the menu item “File – > Open script...”. You can execute selected text in the script window by typing Ctrl-R or by using the “Run line or selection” icon. Make sure to save the script frequently!

## A Simple Script

```
# Anything following a hash or pound sign is a
# comment. Good comments are essential.

# Partial solution to a homework problem

# Create a vector of x-values
x <- seq(0, 3*pi, by=0.1)

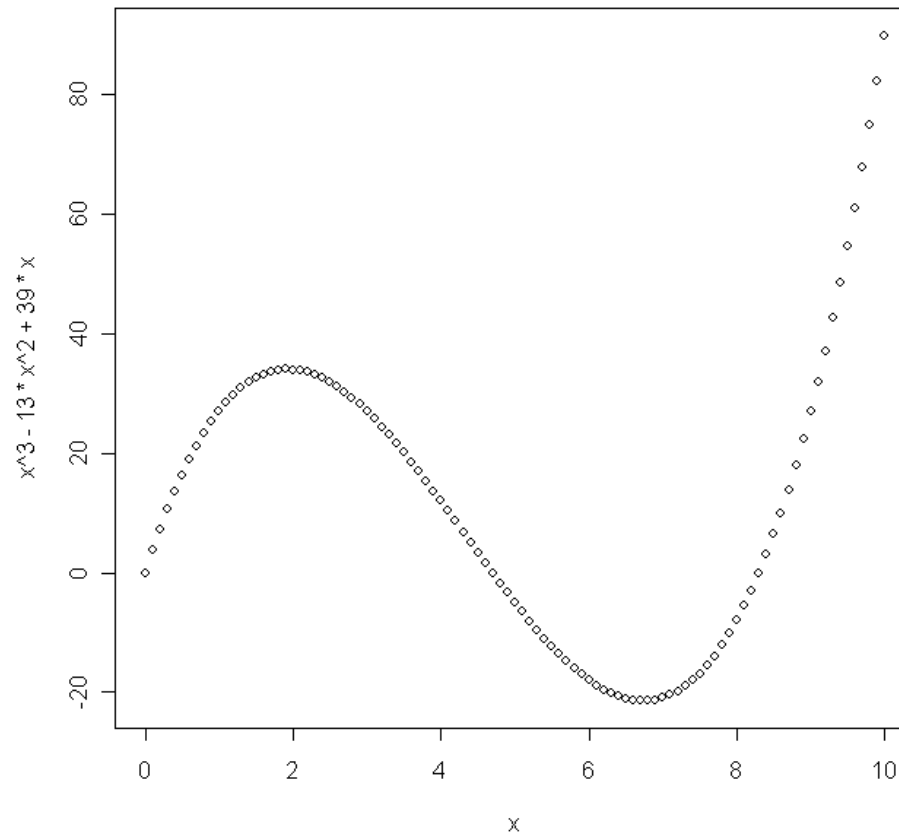
# Plot the sine of x as a curve instead of as a
# bunch of unconnected points.
plot(x, sin(x), type='l')
```

# Graphics in R

R includes a fairly extensive suite of graphics tools. There are typically three steps to producing useful graphics.

- Creating the basic plot
- Enhancing the plot with labels, legends, colors, etc.
- Exporting the plot from R for use elsewhere

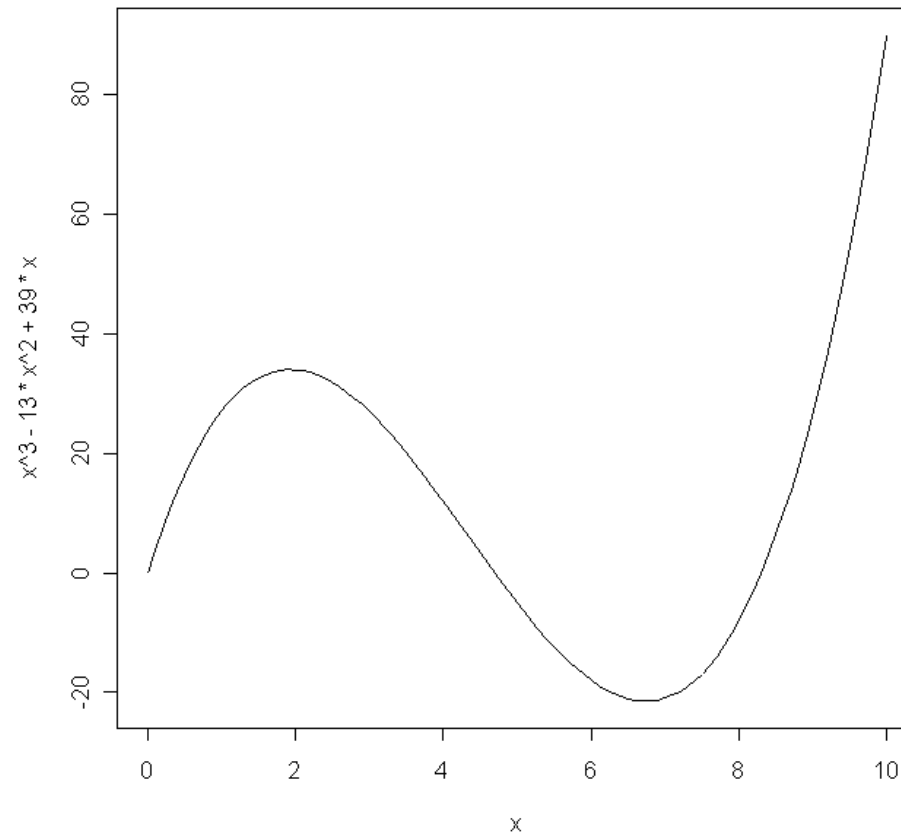
# Basic plot



```
> x <- 0:100/10 # from 0 to 10, increment of 0.1  
> plot(x, x^3-13*x^2+39*x)
```

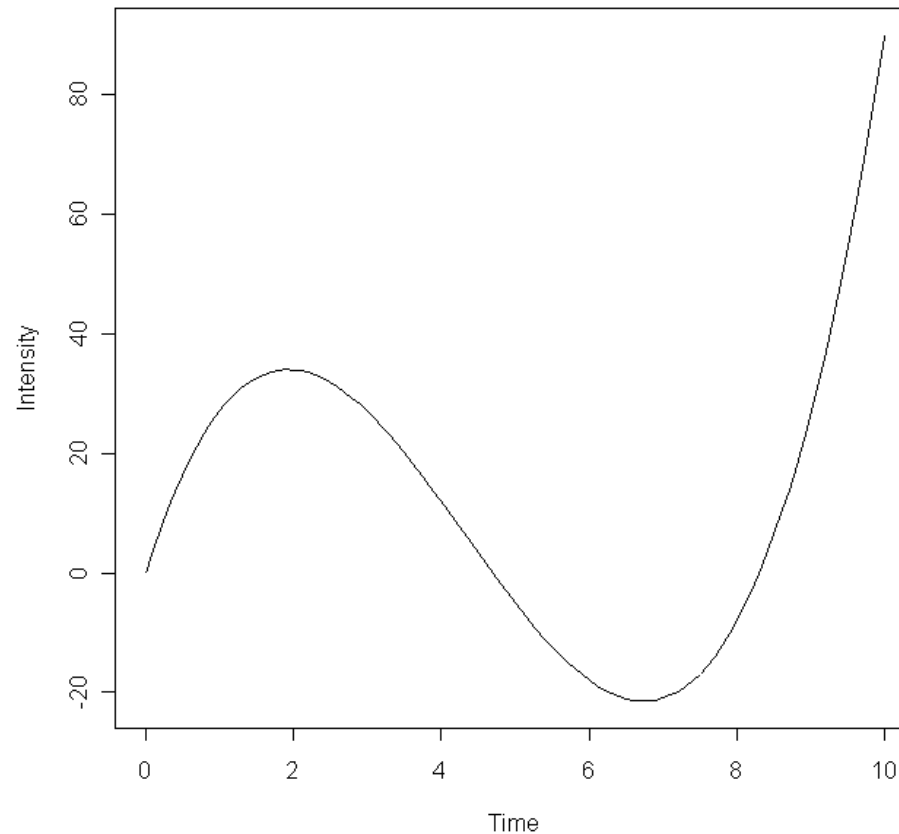


# Plotting curves instead of points



```
> plot(x, x^3-13*x^2+39*x, type='l')
```

# Labeling axes



```
> plot(x, x^3-13*x^2+39*x, type='l',  
+      xlab='Time', ylab='Intensity')
```

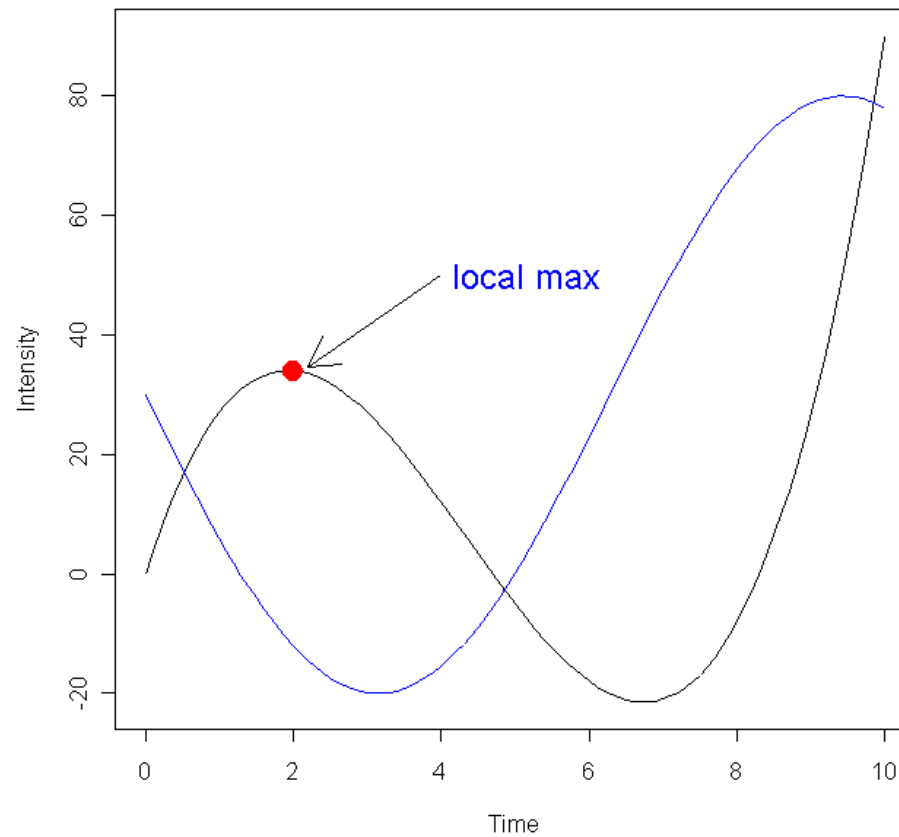
## Repeating yourself ...

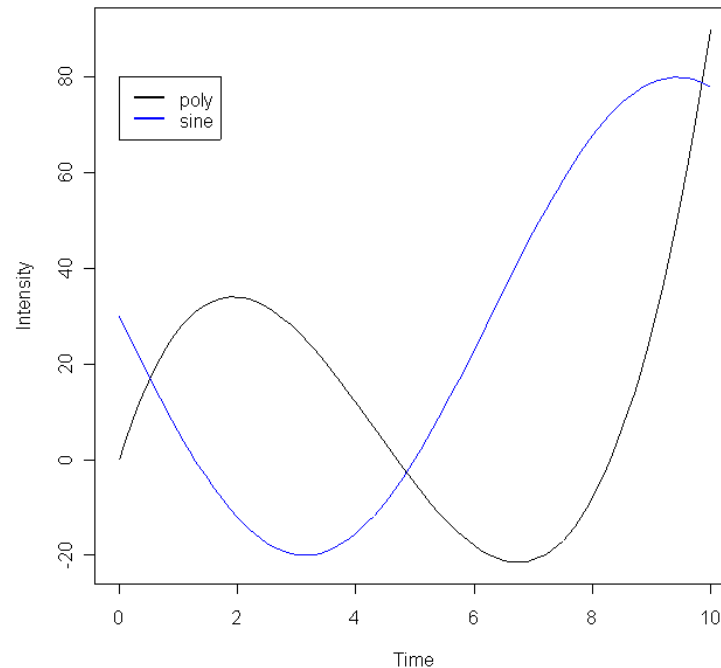
If you change your mind about how you want things like curves or axes displayed, you often have to regenerate the plot from scratch. There are very few things that can be changed after the fact.

You can, however, add points, arrows, text, and lines to existing plots.

```
> points(2, 34, col='red', pch=16, cex=2)
> arrows(4, 50, 2.2, 34.5)
> text(4.15, 50, 'local max', adj=0,
+      col='blue', cex=1.5)
> lines(x, 30-50*sin(x/2), col='blue')
```

# Annotated plot





```
> plot(x, x^3-13*x^2+39*x, type='l')
> lines(x, 30-50*sin(x/2), col='blue')
> legend(0, 80, c('poly', 'sine'),
+       col=c('black', 'blue'), lwd=2)
```

# Saving plots to use elsewhere

In the R GUI:

- first activate the window containing a plot that you want to save,
- on the “File” menu, choose “Save As ->”, which gives you several choices of file format.

The most useful formats are probably:

- PNG; useful for including figures in PowerPoint or Word
- Postscript; often useful for submitting manuscripts.
- PDF; often useful for submitting manuscripts.

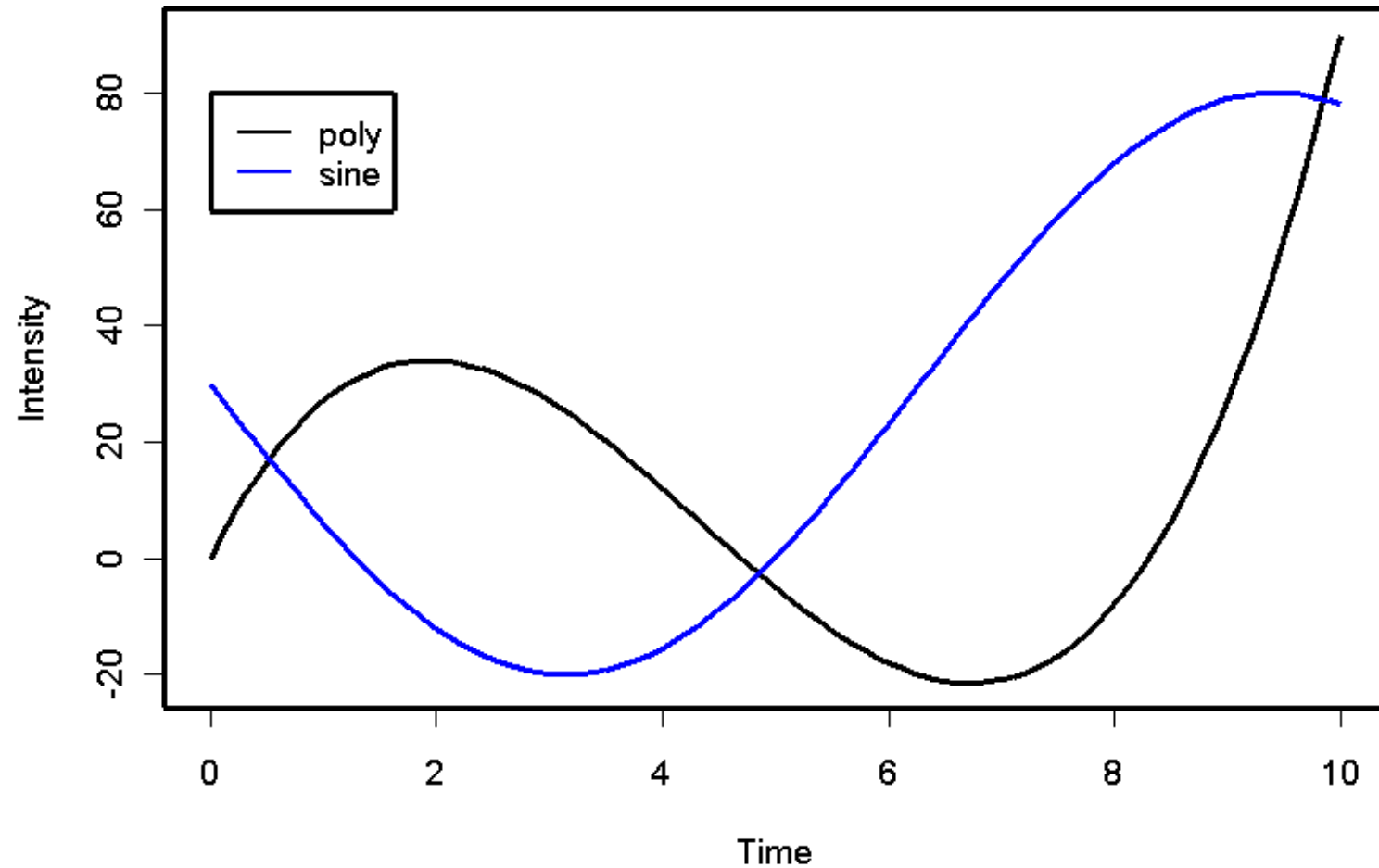
## Graphics parameters

R includes a large number of additional parameters that can be used to control the layout of a graphics window. For a complete list, read the help pages on `par` and `windows` (Mac: `quartz`). The figures included here so far have been produced using the default settings. Remaining figures will be produced after running the commands

```
> windows(width=8, height=5, pointsize=14)
> par(mai=c(1, 1, 0.1, 0.1), lwd=3)
```

which will change the default window size, the size of characters used in the window, and the margin areas around the plot. Rerunning the last set of plot commands will then produce the following figure:

# Same figure with new defaults



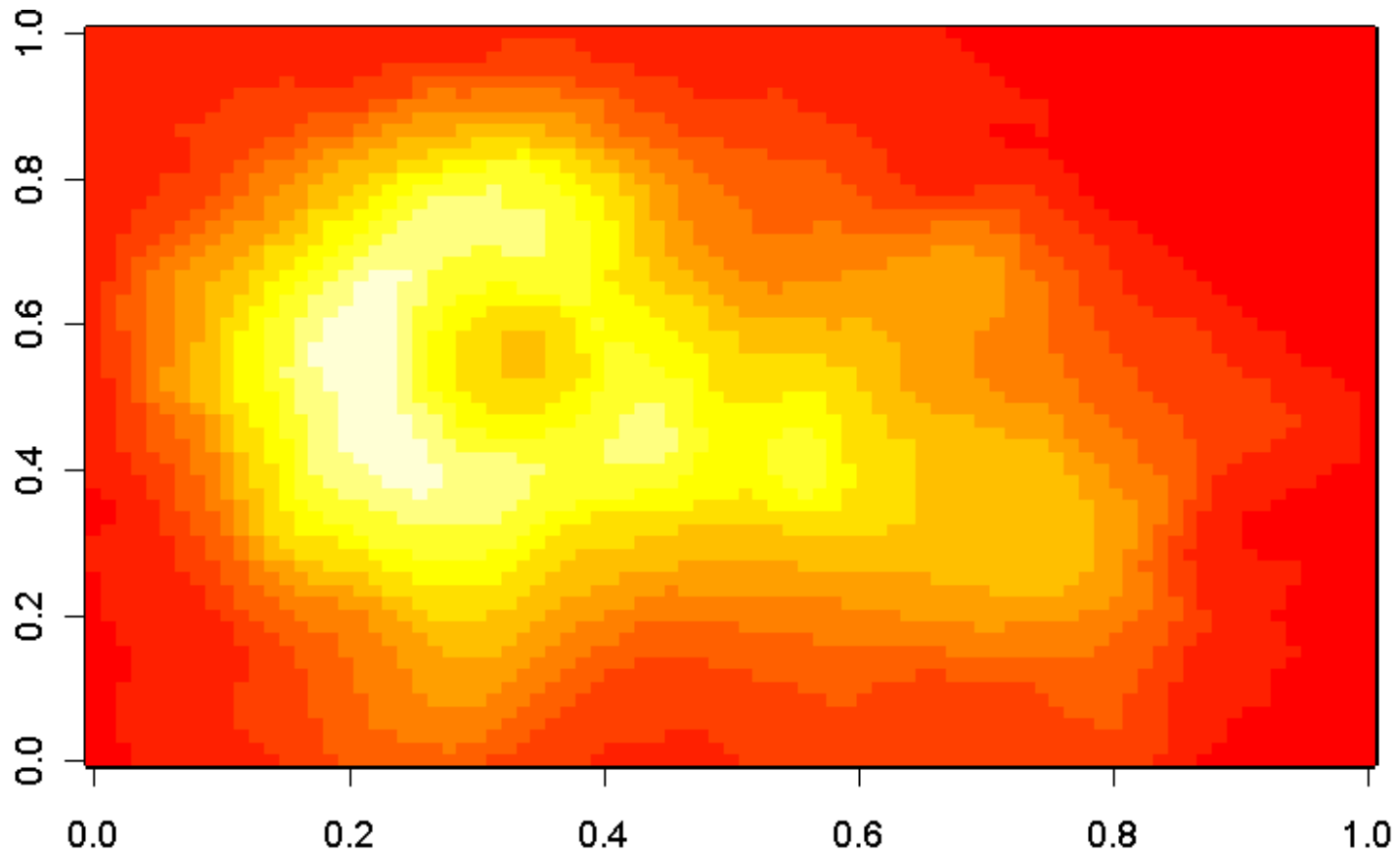


## Additional graphics commands

R includes commands to generate a large number of different kinds of plots, including histograms (`hist`), box-and-whisker plots (`boxplot`), bar charts (`barplot`), dot plots (`dotplot`), strip charts (`stripchart`), and pie charts (`pie`).

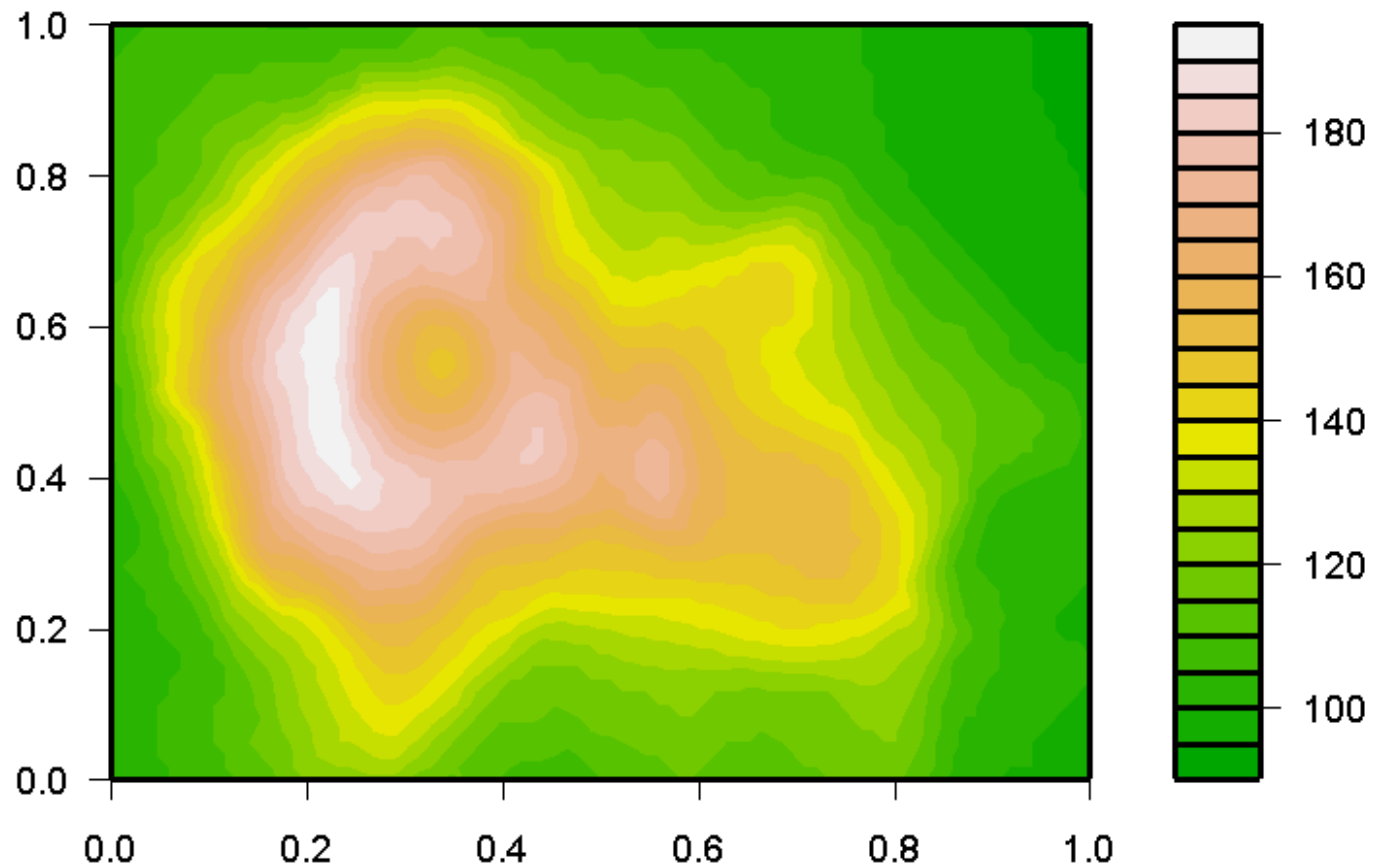
R also includes a number of commands to visualize matrices. On the next slide, we use the `data` command to load a sample data matrix that comes with R. We then produce an image of the matrix, treating the rows and columns as  $x$ - $y$  coordinates and the matrix entries as intensities or heights.

# Volcano



```
> data(volcano)
> image(volcano)
```

# Volcano



```
> filled.contour(volcano, color=terrain.colors)
```

## Names and Attributes

We've seen that R has a bunch of useful functions, and we can see how these would have helped S (and then R) to catch on. But there was more; we remarked on how S allowed one to think about data in a more coherent fashion. Let's think about that a bit more.

Consider `x`. Initially, this symbol has no meaning; we must assign something to it.

```
x <- 2
```

In the process of assignment, we have created an object with the name of `x`. This object has the value 2, but there are other things about objects: they have properties, or attributes.

## Some Basic Attributes

```
> mode(x)
"numeric"
> storage.mode(x)
"double"
> length(x)
[1] 1
```

These are attributes that  $x$  has by default, but we can give it others.

# What's In A Name?

Initially, nothing:

```
> names(x)
```

```
NULL
```

but we can change this

```
> names(x) <- c("A")
```

```
> x
```

```
A
```

```
2
```

This element of `x` now has a name! If something has a name, we can call it by name.

# Calling Names

```
> x [ 1 ]
```

```
A
```

```
2
```

```
> x [ "A" ]
```

```
A
```

```
2
```

Admittedly, this isn't that exciting here, but it can get more interesting if things get bigger and the names are chosen in a more enlightening fashion.

Let's assign a matrix of values to  $x$ , and see if we can make the points clearer.

So, how do we assign a matrix?

## Matrix X: TMTOWTDI!

There's more than one way to do it...

```
> help(matrix)
```

Usage

```
matrix(data = NA, nrow = 1, ncol = 1,  
        byrow = FALSE, dimnames = NULL)
```

even the arguments to the function have names!

Arguments to a function can be supplied by position, or by name.



## Matrix X: TMTOWTDI!

We're going to assign the numbers 1 through 12. That means we need to get these numbers. Some ways to do that:

```
> 1:12
> c(1, 2, 3, 4, 5:12)
> c(1:6, c(7:12))
> 1:12.5
> seq(from=1, to=12, by=1)
> seq(1, 12, 1)
> seq(1, 12)
> seq(by=1, to=12, from=1)
```

## Matrix X: TMTOWTDI!

```
> x <- matrix(1:12, 3, 4)
> x <- matrix(data = 1:12, nrow = 3, ncol = 4)
> x
      [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    2    5    8   11
[3,]    3    6    9   12
```

The numbers in brackets suggest how things should be referred to now:

```
> x[2, 3]
[1] 8
> x[2, ]
[1] 2  5  8 11
```

## Matrix X: TMTOWTDI!

```
> x[,3]
[1] 7 8 9
> x[3,1:2]
[1] 3 6
> x[3,c(1,4)]
[1] 3 12
> x[2, x[2,] > 6]
[1] 8 11
```

But what about names?

# Naming x

```
> rownames(x)
```

```
NULL
```



```
> rownames(x) <- c("Gene1", "Gene2", "Gene3")
```

```
> x
```

	[, 1]	[, 2]	[, 3]	[, 4]
Gene1	1	4	7	10
Gene2	2	5	8	11
Gene3	3	6	9	12

## Naming x

```
> colnames(x) <- c("N01", "N02", "T01", "T02")
```

```
> x
```

	N01	N02	T01	T02
Gene1	1	4	7	10
Gene2	2	5	8	11
Gene3	3	6	9	12

One more thing – names can be inherited!



```
> x["Gene2", ]
```

	N01	N02	T01	T02
Gene2	2	5	8	11