GS01 0163
Analysis of Microarray Data

Keith Baggerly and Kevin Coombes
Section of Bioinformatics
Department of Biostatistics and Applied Mathematics
UT M. D. Anderson Cancer Center
kabagg@mdanderson.org
kcoombes@mdanderson.org

12 September 2005

Lecture 4: Introduction to R

- Comments on HWK 1

- Limits of Canned Packages

- Introducing R: Some Background

- Installing R

- Learning About R

- Obtaining extra R packages

- Graphics in R

Comments on HWK 1?

Was it easy to find the data?

Was it easy to understand the processing steps involved?

Was it easy to find the supplementary info used in their analysis?

Could you reproduce their analysis with the data provided?

Do you think the level of documentation for this paper is above or below average?

Was dChip easy to use?

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 R...

Why R?

- R is a powerful, general purpose language and software environment for statistical computing and graphics.

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).

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).

- There already exists an extensive package of microarray analysis tools, called BioConductor, written in R.

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).

- 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...)

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.

GS01 0163: ANALYSIS OF MICROARRAY DATA

Why Did S Catch On?

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

It had a C-like syntax, which was familiar to many of the people working on it to begin with.

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.

It allowed people to write their own modules.

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.

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

However, S is not free. R is. Many of the older modules written for S have been ported over to R. In addition, many new modules specifically for bioinformatics (and mostly microarray analysis) have been collected and centralized as part of the Bioconductor project.

Let's find R and Bioconductor, and set about using them (mostly R to start).

The Comprehensive R Archive Network

http://cran.r-project.org

The BioConductor Project
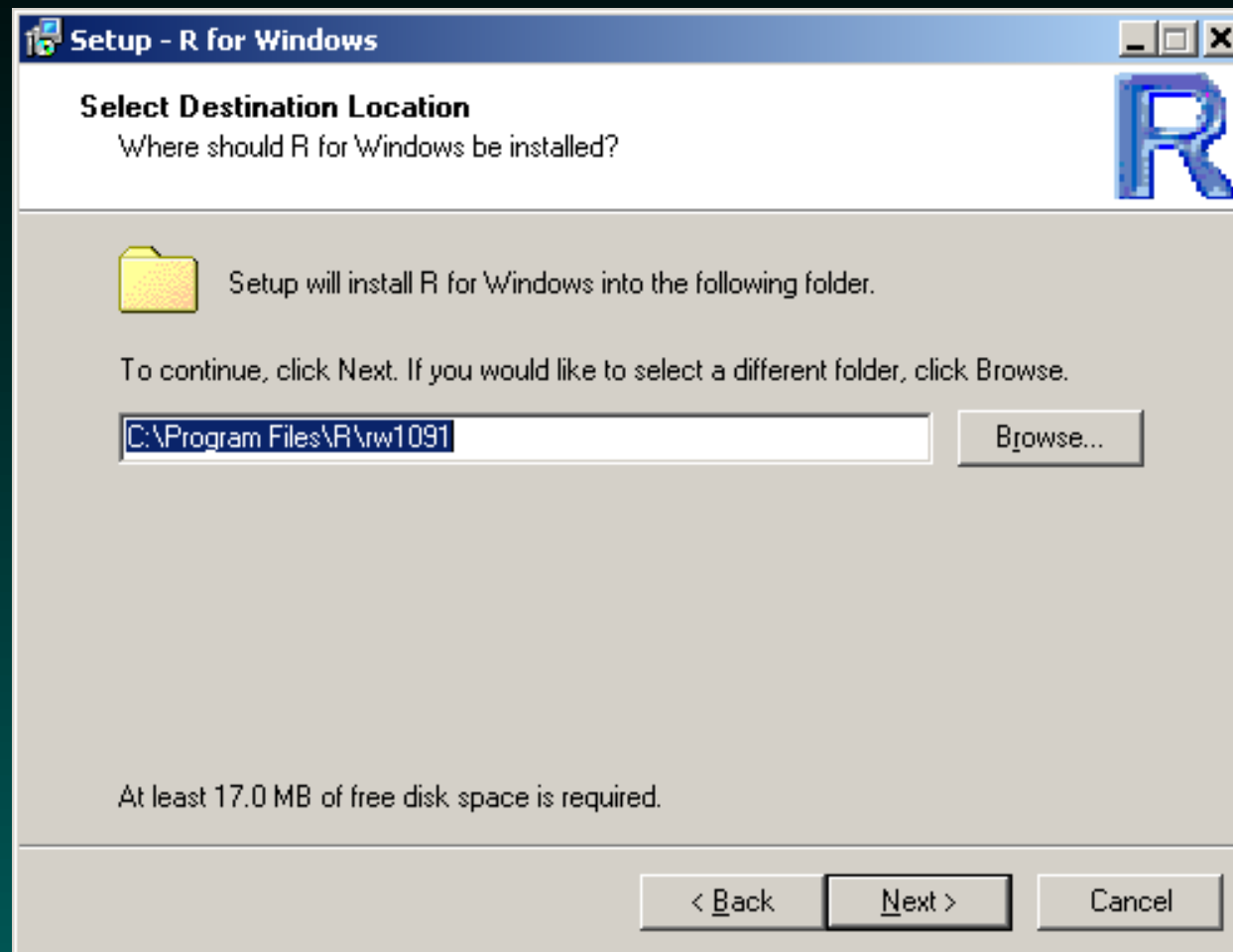

http://www.bioconductor.org
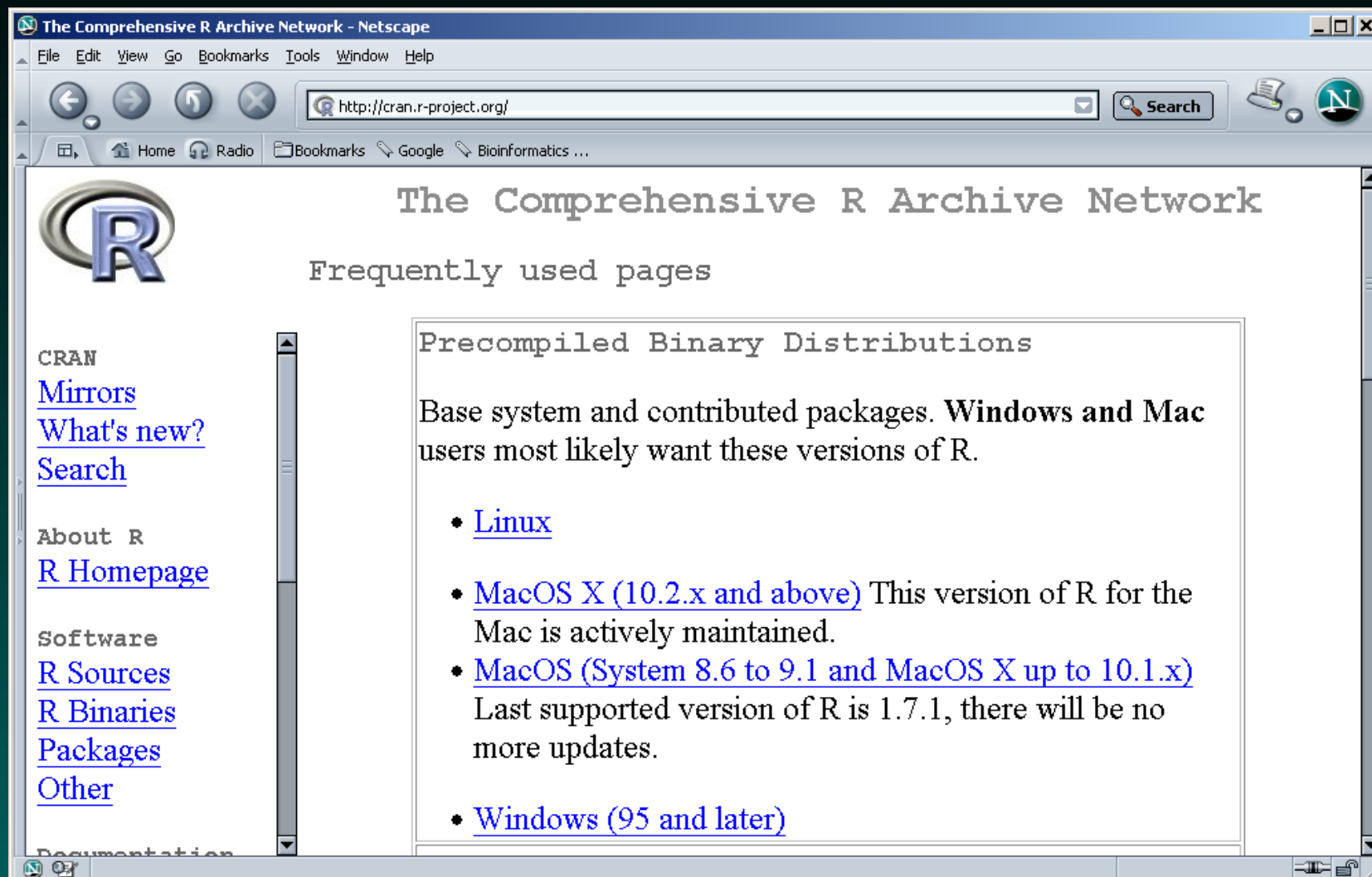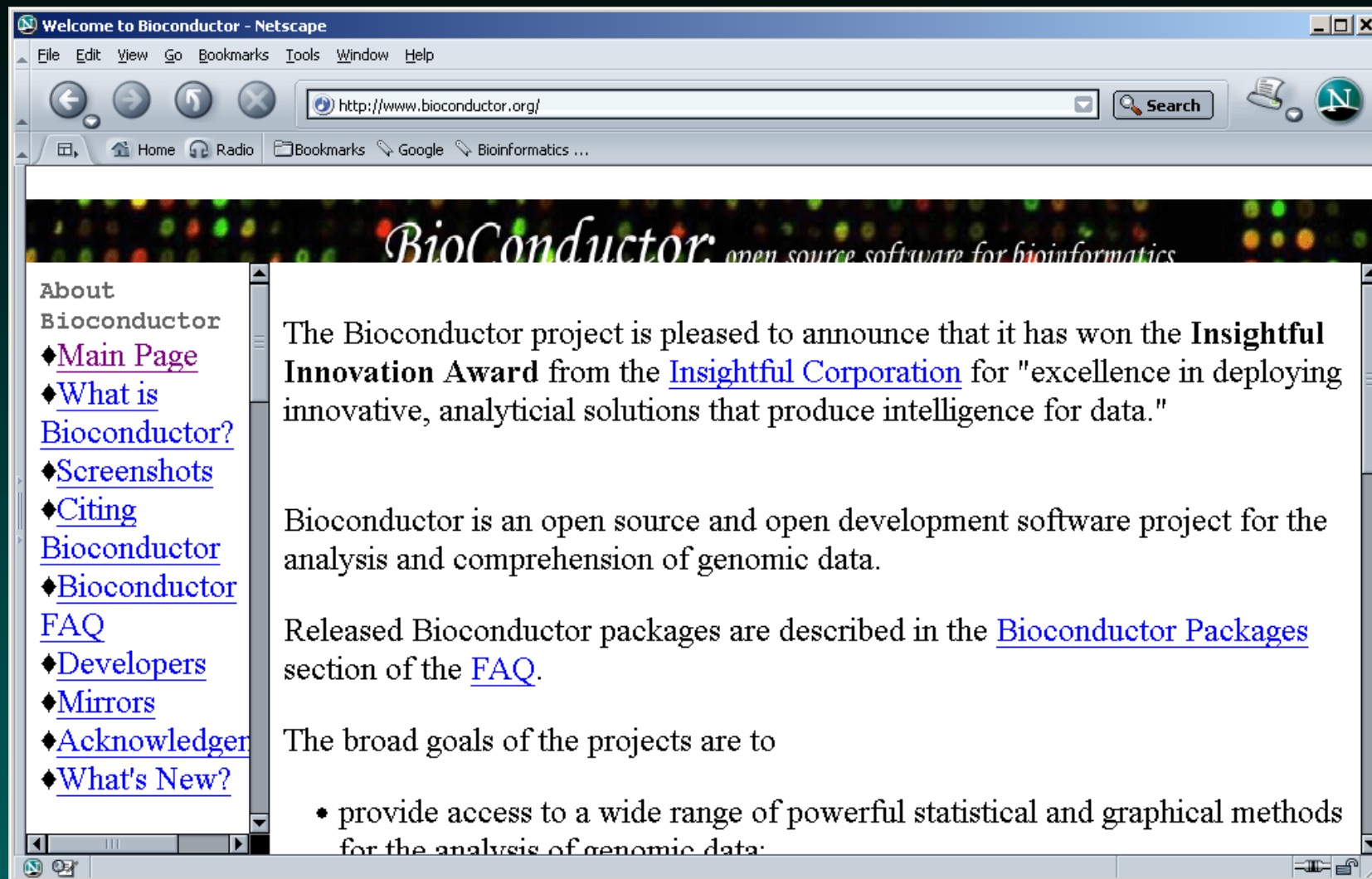
R Installation

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

R Installation
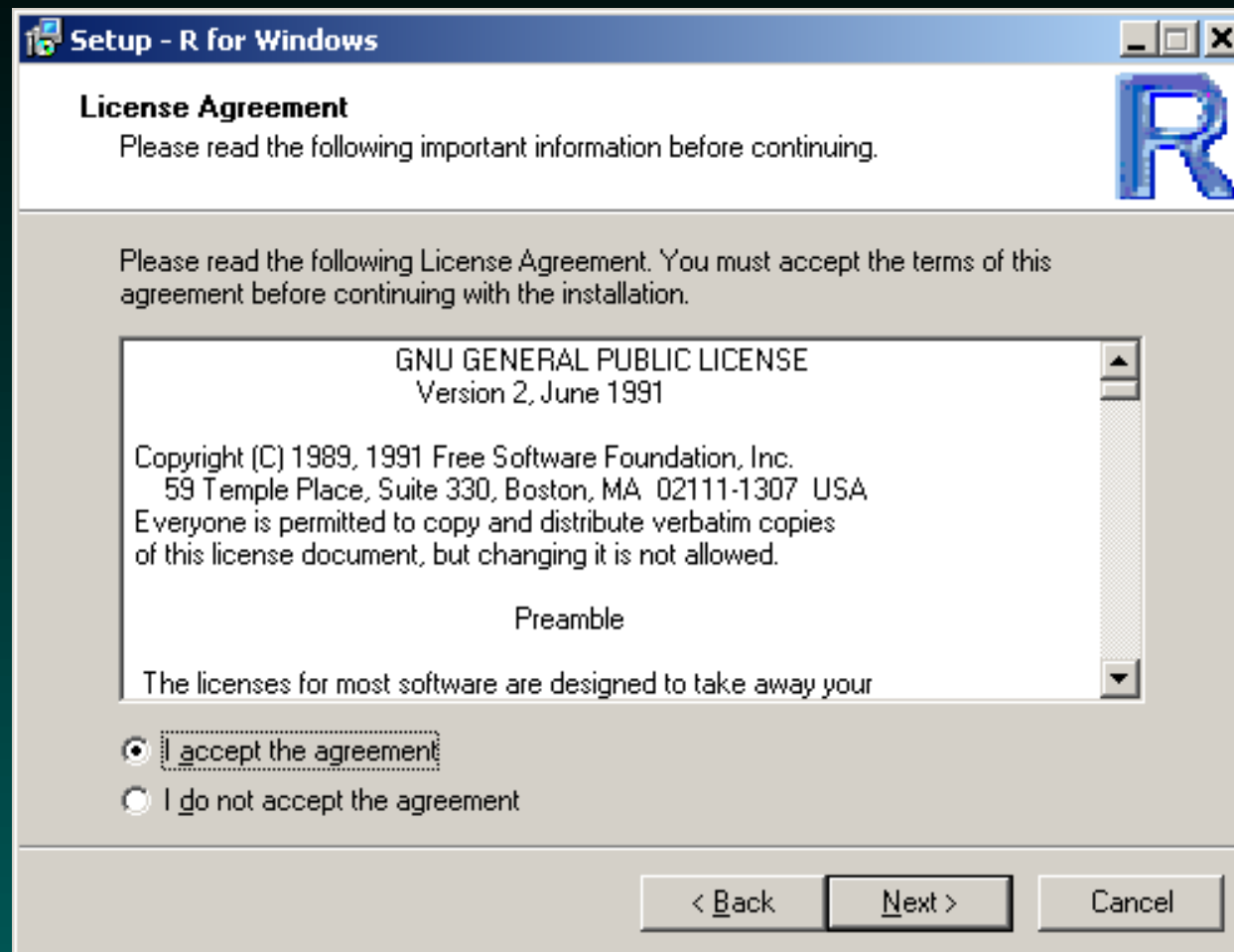
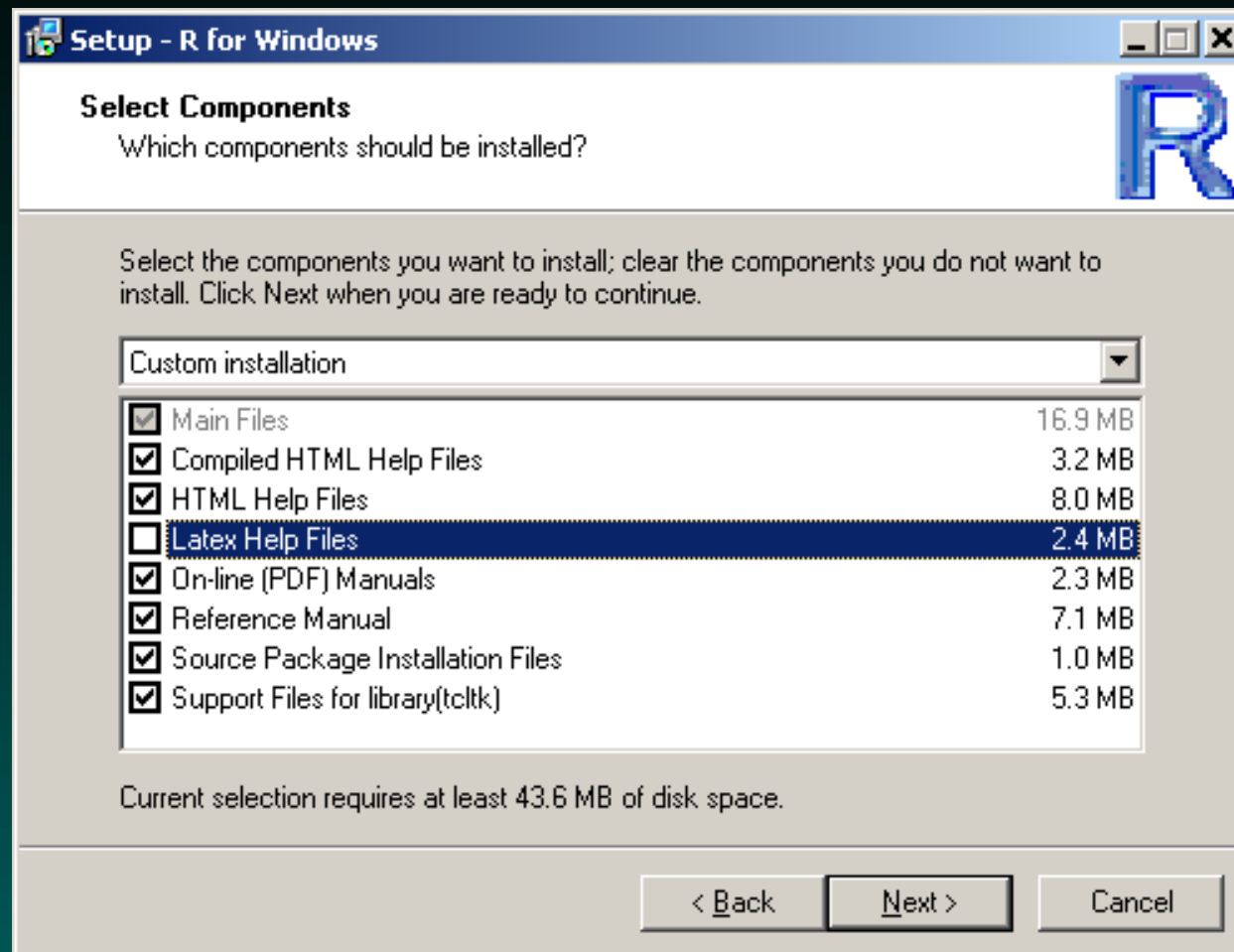You must click to accept the license agreement before you can proceed.

R Installation

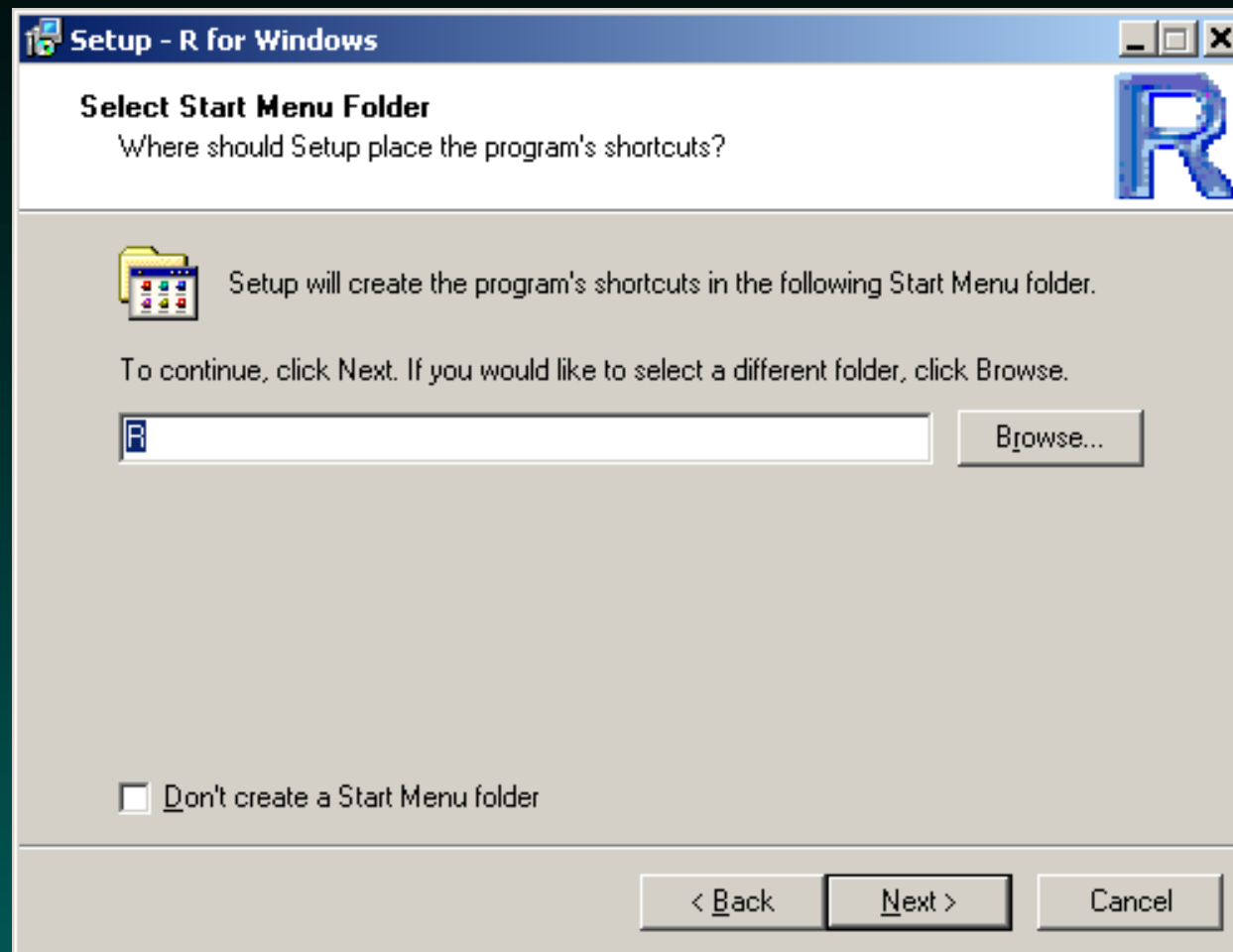You can change the installation path. It may be a good idea to choose a path name that does not include any spaces.
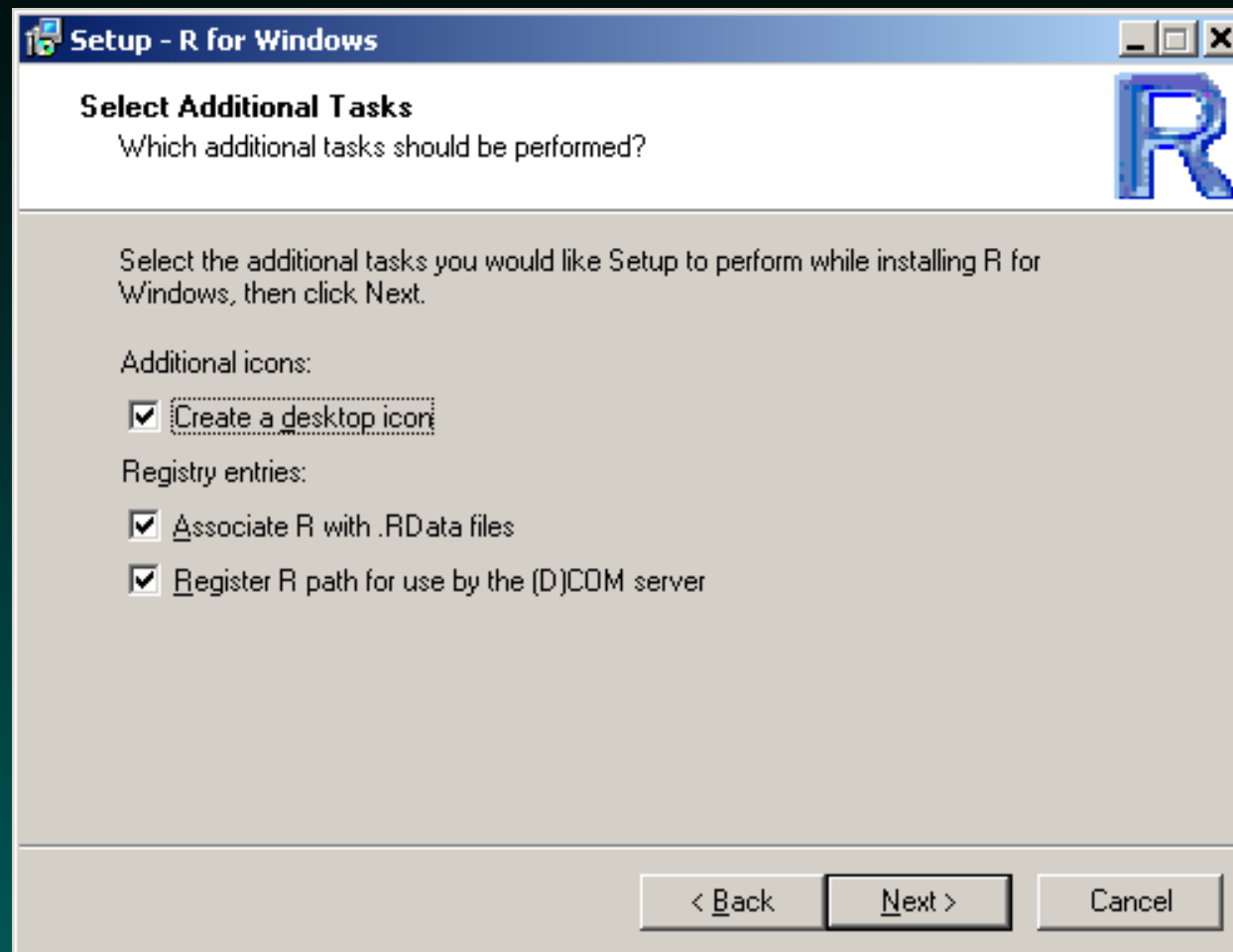
R Installation

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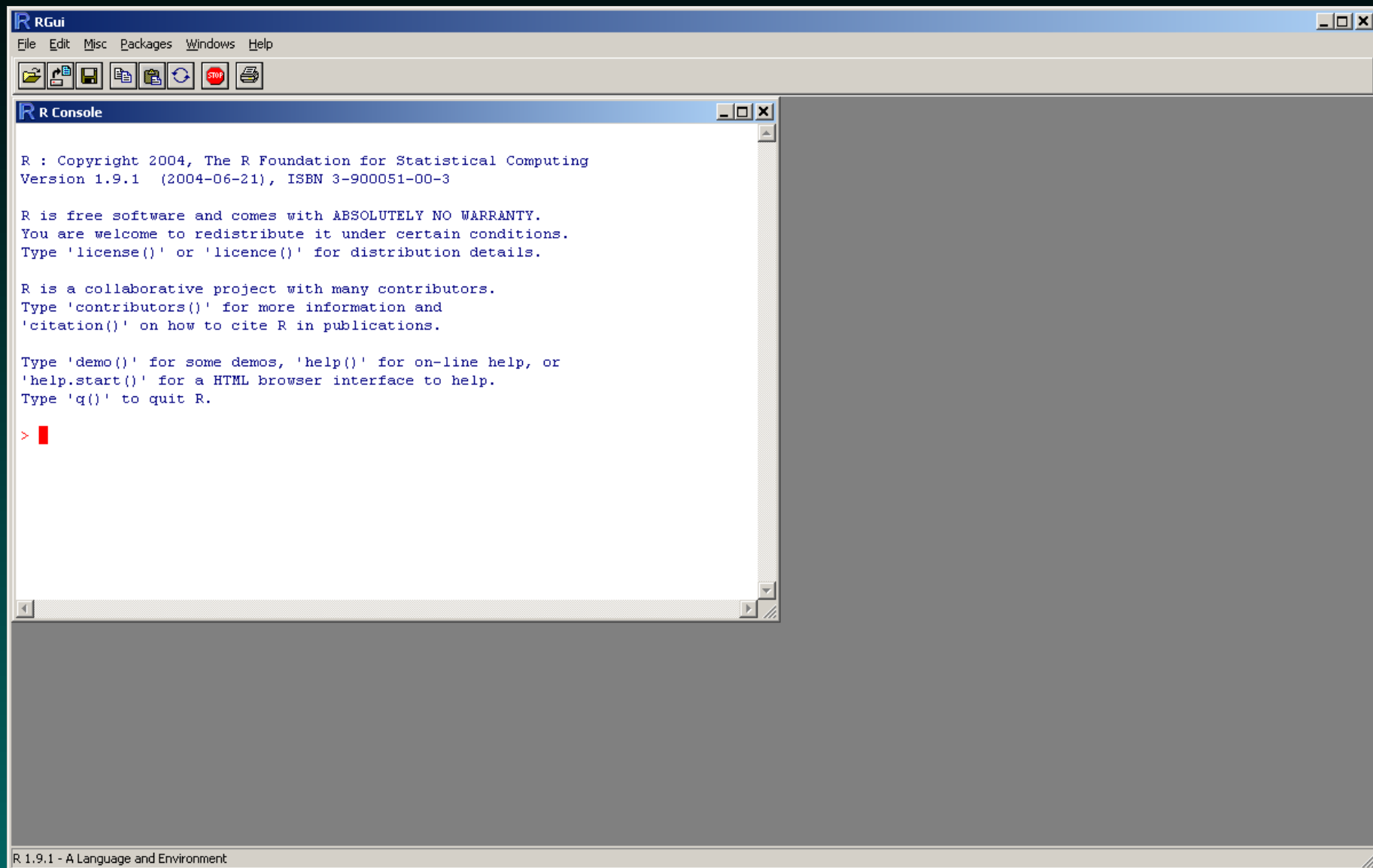
R Installation

Decide whether to make a folder on the start menu.
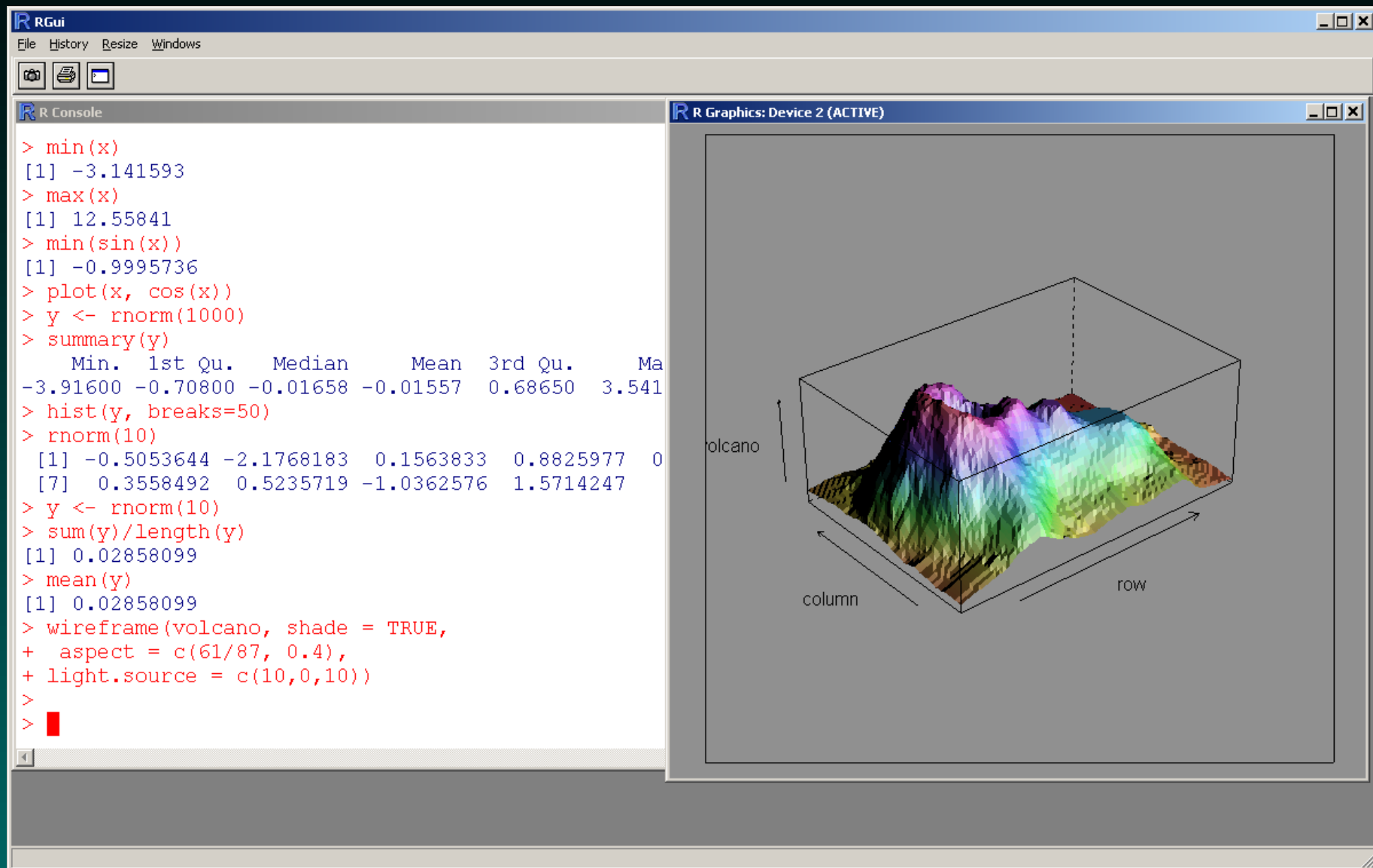
R Installation

Decide whether to put an icon on the desktop. After this step, the program installs itself fairly quickly.

The R Gui

GS01 0163: ANALYSIS OF MICROARRAY DATA

The R Gui

Learning About R

We'll try to cover many of the tools that we'll commonly use, but we can't cover all apsects 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.

For now, we will assume that you have acquired and started R.

Where Have You Started R?

(Mac) Under the main menu bar, there is an option for "Misc", under which there is the option to "Get Working Directory". Selecting this option issues a command:

```
> getwd()
[1] ``Users/kabagg''
```

Now, this directory may not be where you'll want to store the results of your analyses, so you may want to change this. Again going to Misc, we can choose the option "Change Working Directory" at which point we can browse the hierarchy to get where we want. You can guess where I moved by looking at the equivalent command:

```
> setwd(``MicroarrayCourse/DataSets/Testing'')
```

Notes on R

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 assign the value of a command to a variable, you use a "left arrow", made by typing $<$ (less than) followed by $-$ (minus), as in

```
x <- 2
```

This command produces no output; it simply stores the value "2" under the name "x". To retrieve the value, type the name of the variable.

GS01 0163: ANALYSIS OF MICROARRAY DATA

Notes on R

```
> x
[1] 2
```

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.

```
> y <- rnorm(10)
> y
 [1] -1.07521929 -1.15549677 -1.88800876 -0.89362362
 [5]  0.60838354 -2.11006124  0.41604637  0.52506983
 [9] -0.06416302 -0.22610929
```

The `rnorm` function generates random variables from the normal distribution.
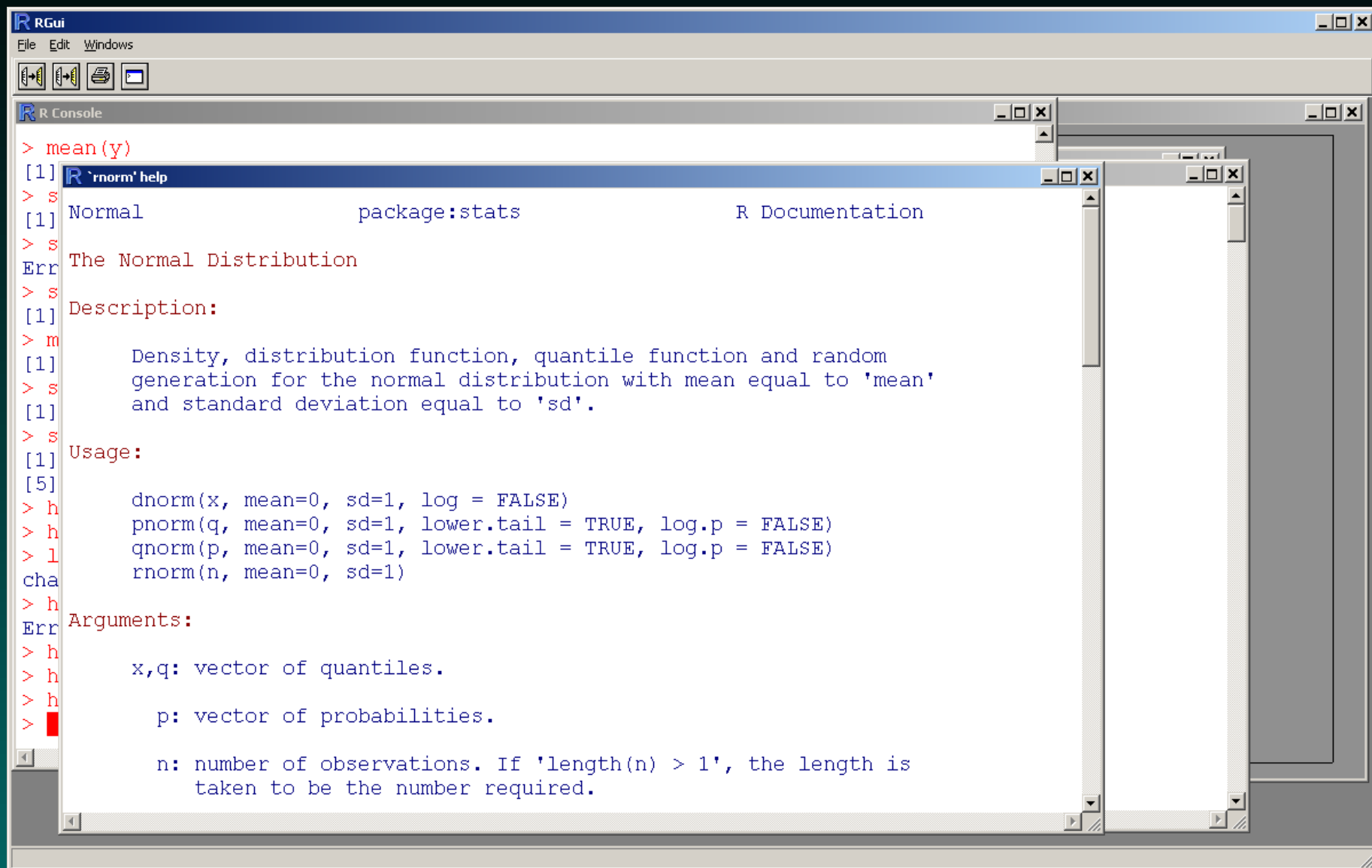
R Functions

R has lots of built-in functions.

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

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

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

R Help on `rnorm`

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.

When R starts, it loads the packages "base", "utils", "graphics", and "stats". Other packages must be loaded using the `library` command. Alternatively, you can use the menu item "Packages", then "Load packages...", which is available when the cursor is in the console window. (Note: Menu items in the R GUI change depending on the active subwindow.) You get a dialog box with a list of packages.

GUI Loading Library Packages
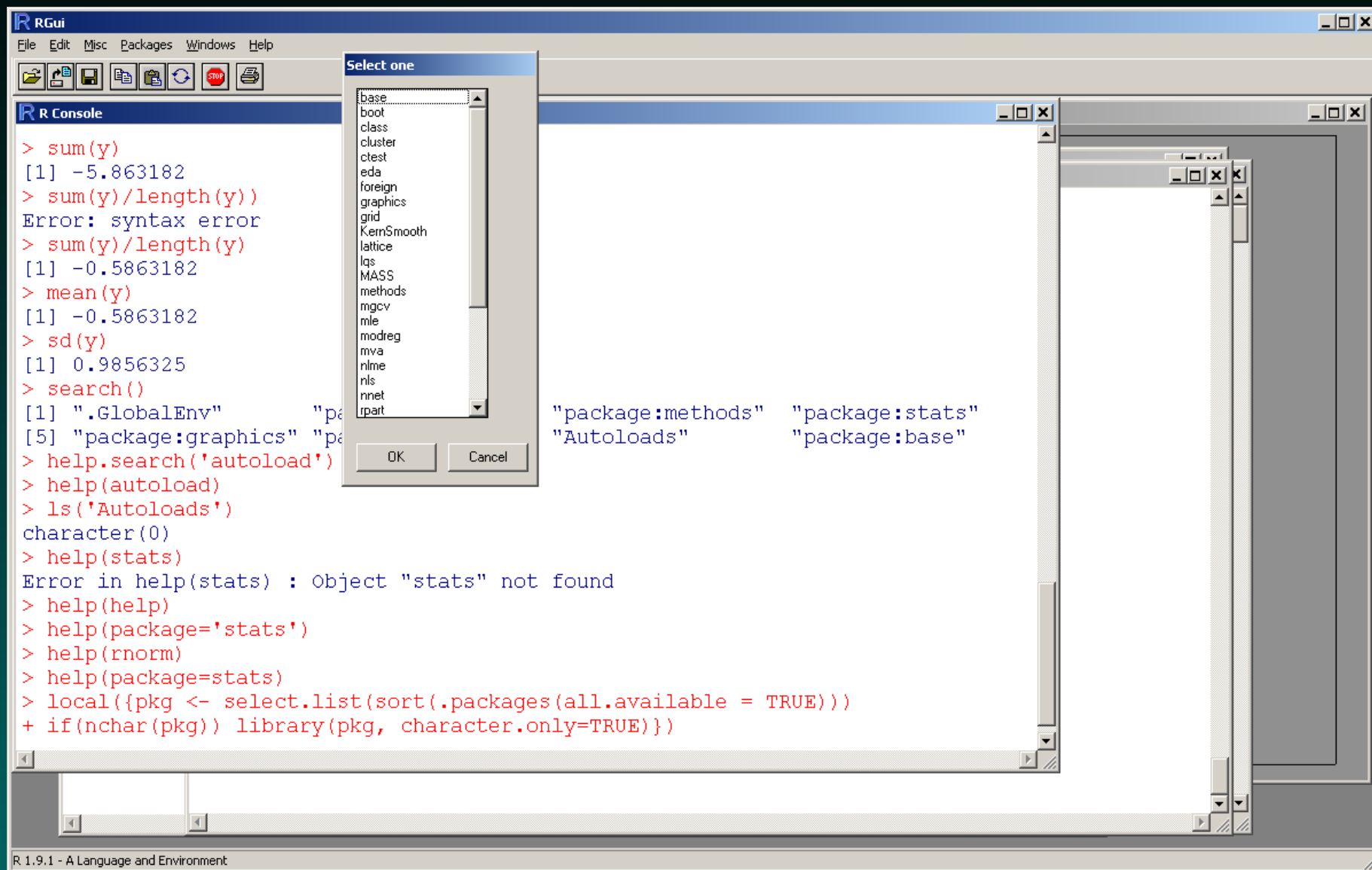
Browser-based R Help

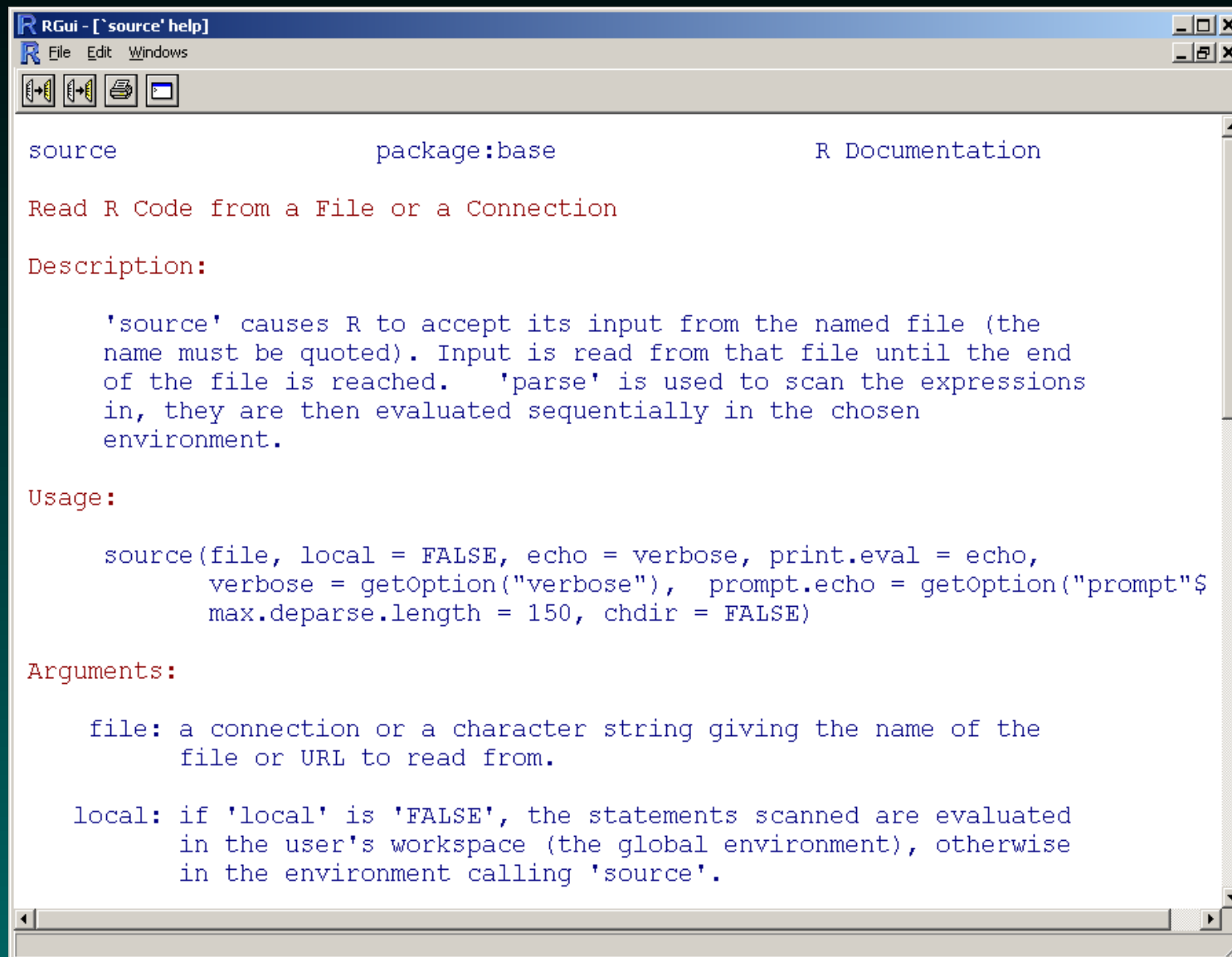You can also use the GUI menu item "Help" followed by "Html help" to open a web browser with help information.
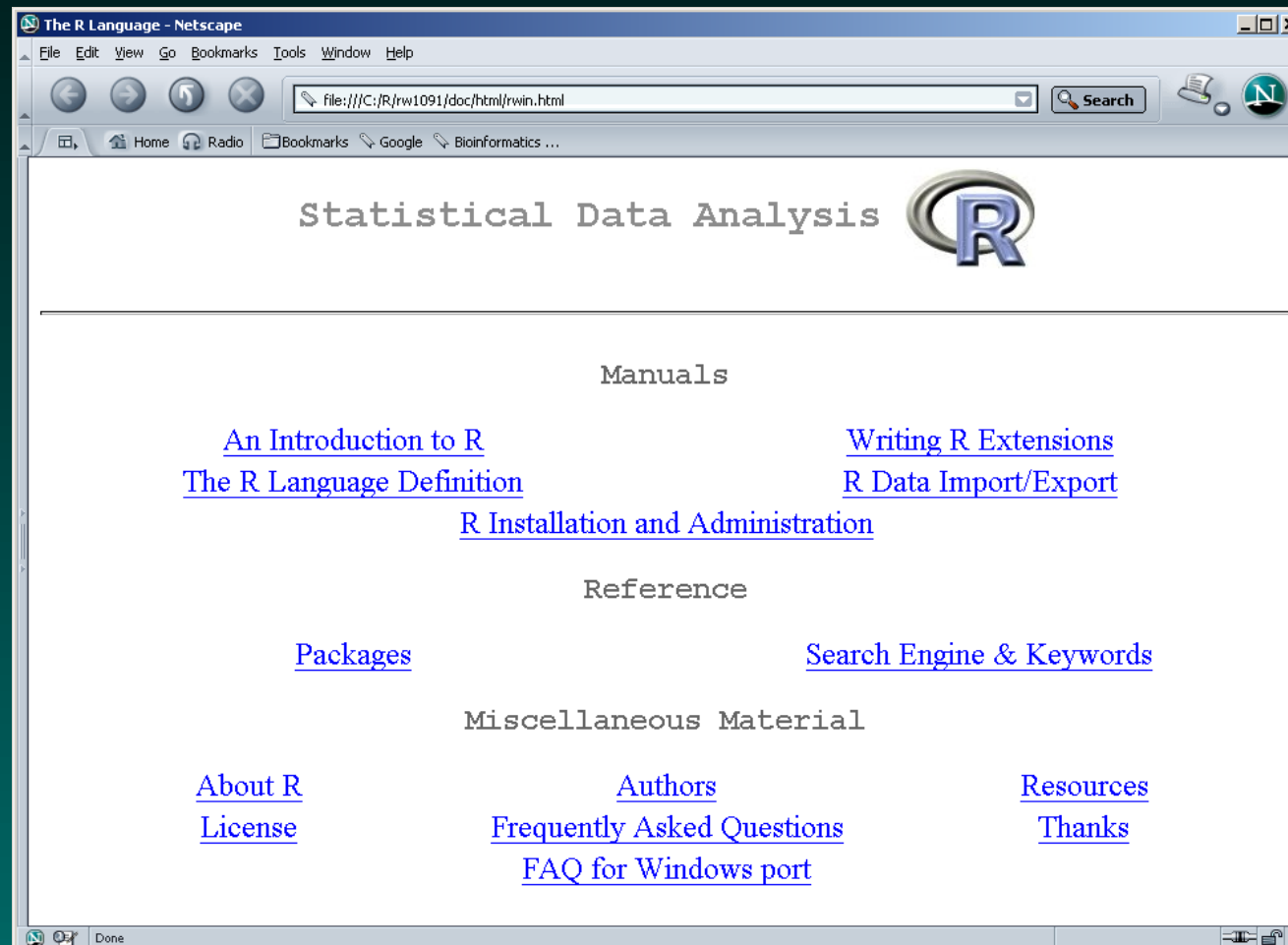
Homework Submission

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.

Comments can be include in the file by prefacing them with a "hash" or "pound" sign (#), as in the example on the next slide.

A Simple Script

```
# 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')
```

Obtaining extra R packages

The R GUI makes it easy to get additional packages via the internet. From the "Packages" menu, you simply select either "Install package(s) from CRAN..." or "Install package(s) from Bioconductor". Both menu items present you with a dialog box containing a list of the available packages. You then select one or more (by holding the control key while clicking with the mouse) and press the "OK" button. R then downloads the package, installs it, and updates the help files. It finishes by asking if you want to delete the downloaded files; unless you want to save them to install them on another computer without an internet connection, the usual answer is "yes". We'll come back to this point later when we start working with Bioconductor.

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
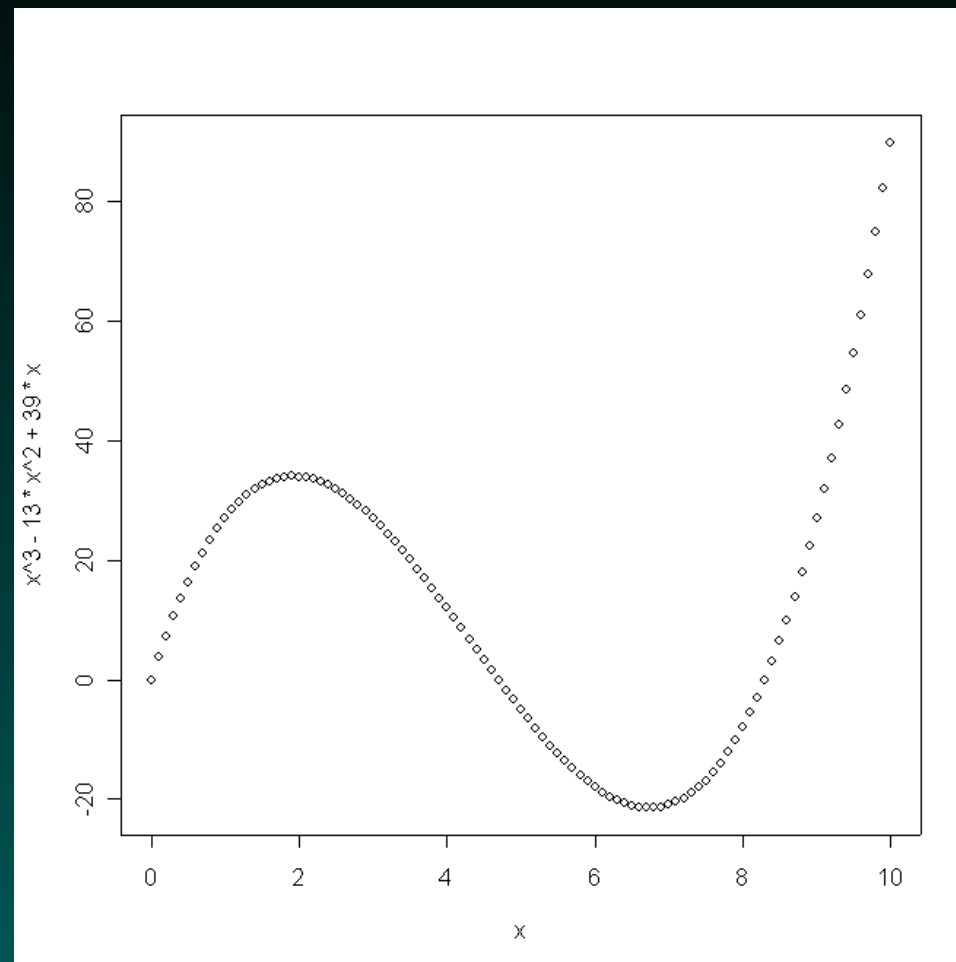

```
> x <- (0:100)/10  # from 0 to 10, increment of 0.1
> plot(x, x^3-13*x^2+39*x)
```

Plotting curves instead of points
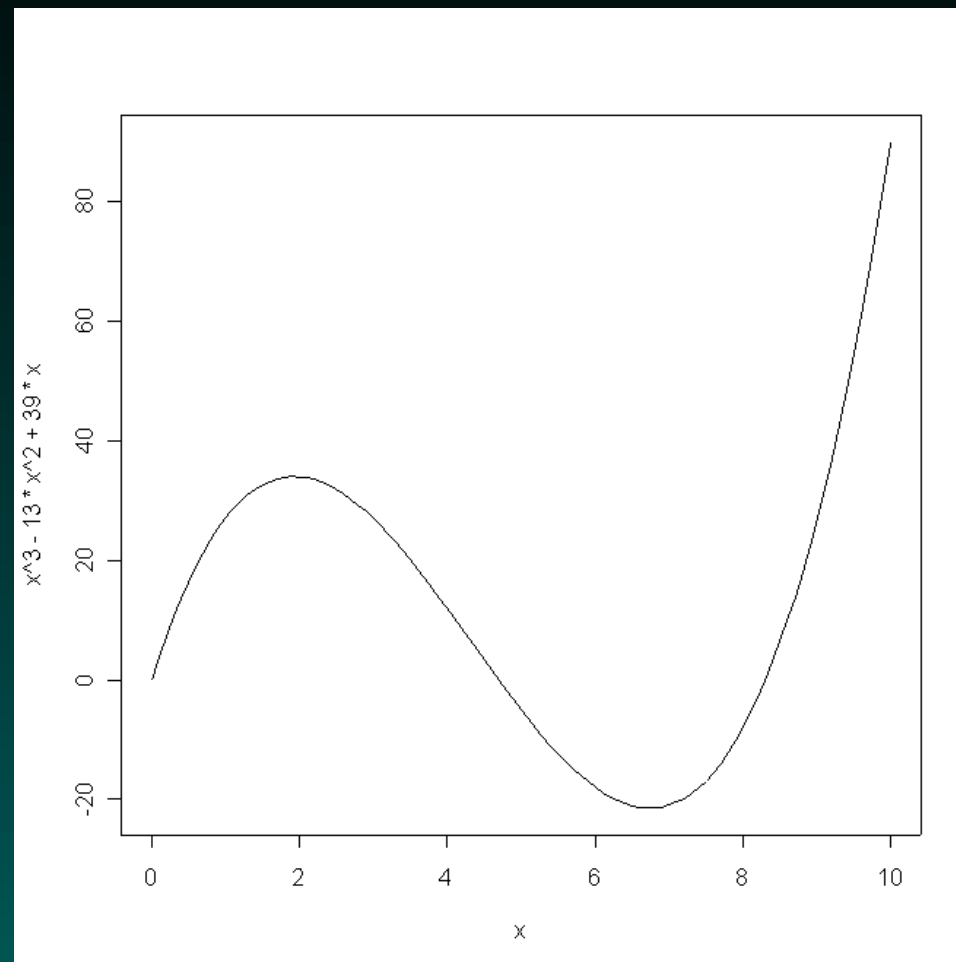

```
> plot(x, x^3-13*x^2+39*x, type='l')
```

Labeling axes
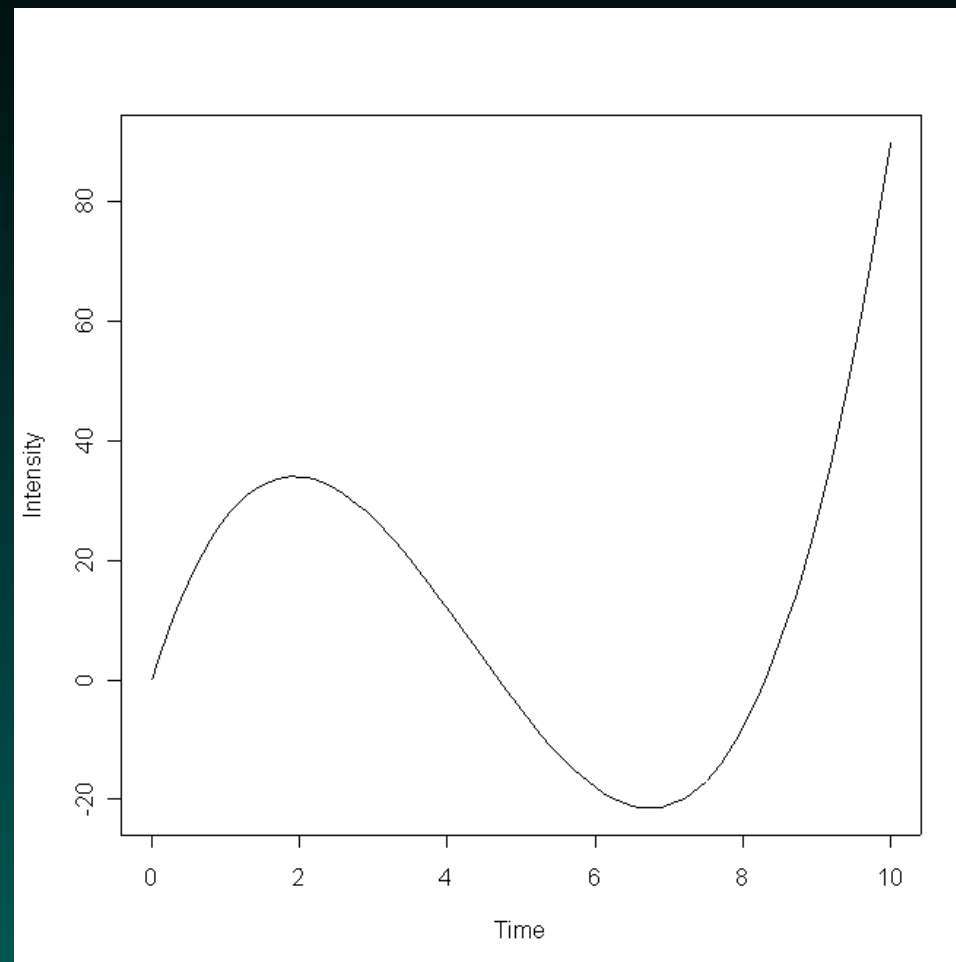

```
> plot(x, x^3-13*x^2+39*x, type='l',
+       xlab='Time', ylab='Intensity')
```

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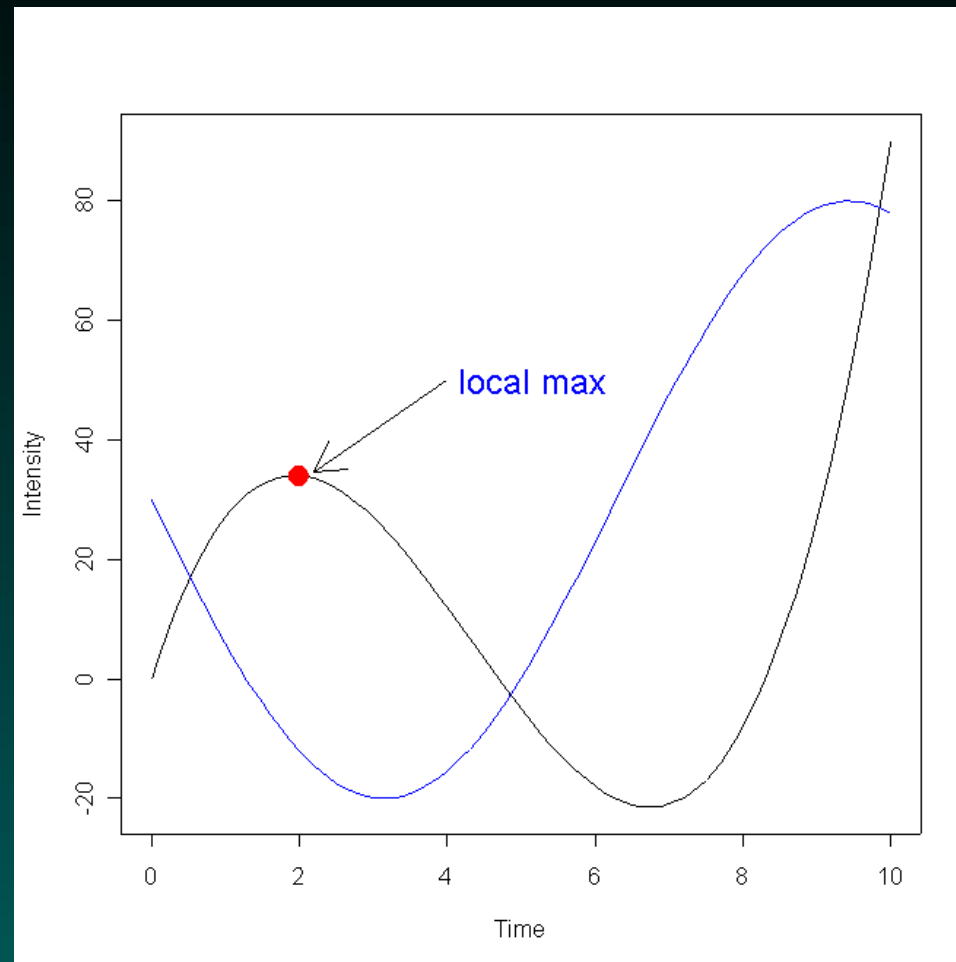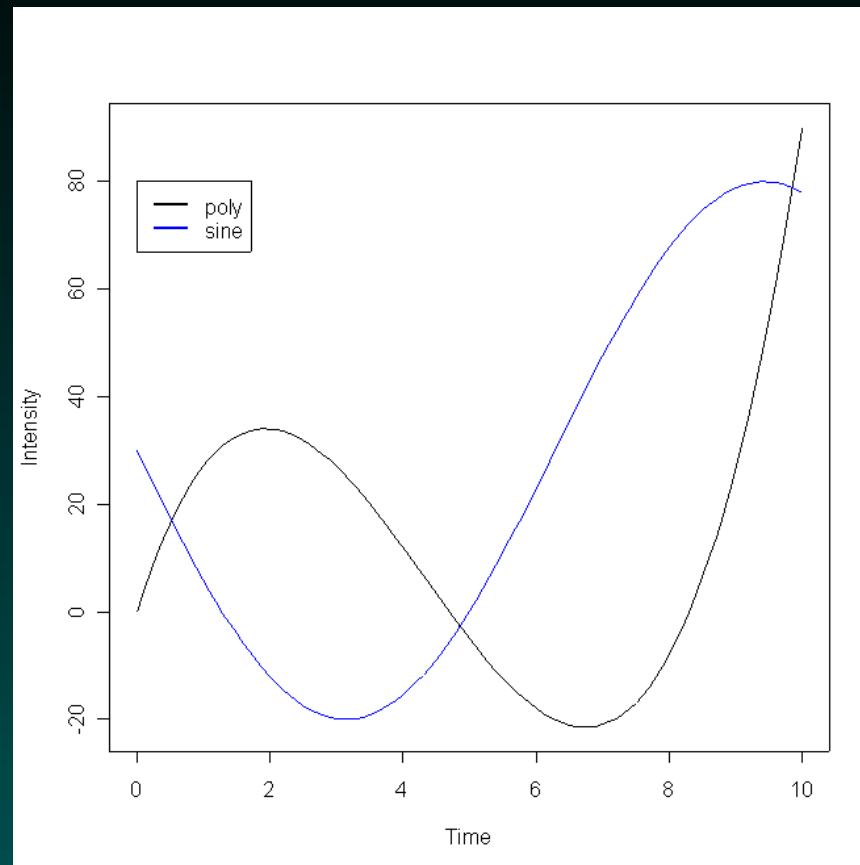
```
> plot(x, x^3-13*x^2+39*x, type='l')
> lines(x, 30-50*sin(x/2), col='blue')
> legend(0, 80, c('poly', 'sine'),
+       col=c('black', 'blue'), lwd=2)
```

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 two most useful formats are probably

- PNG; useful for including figures in PowerPoint or Word

- Postscript; 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`. 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
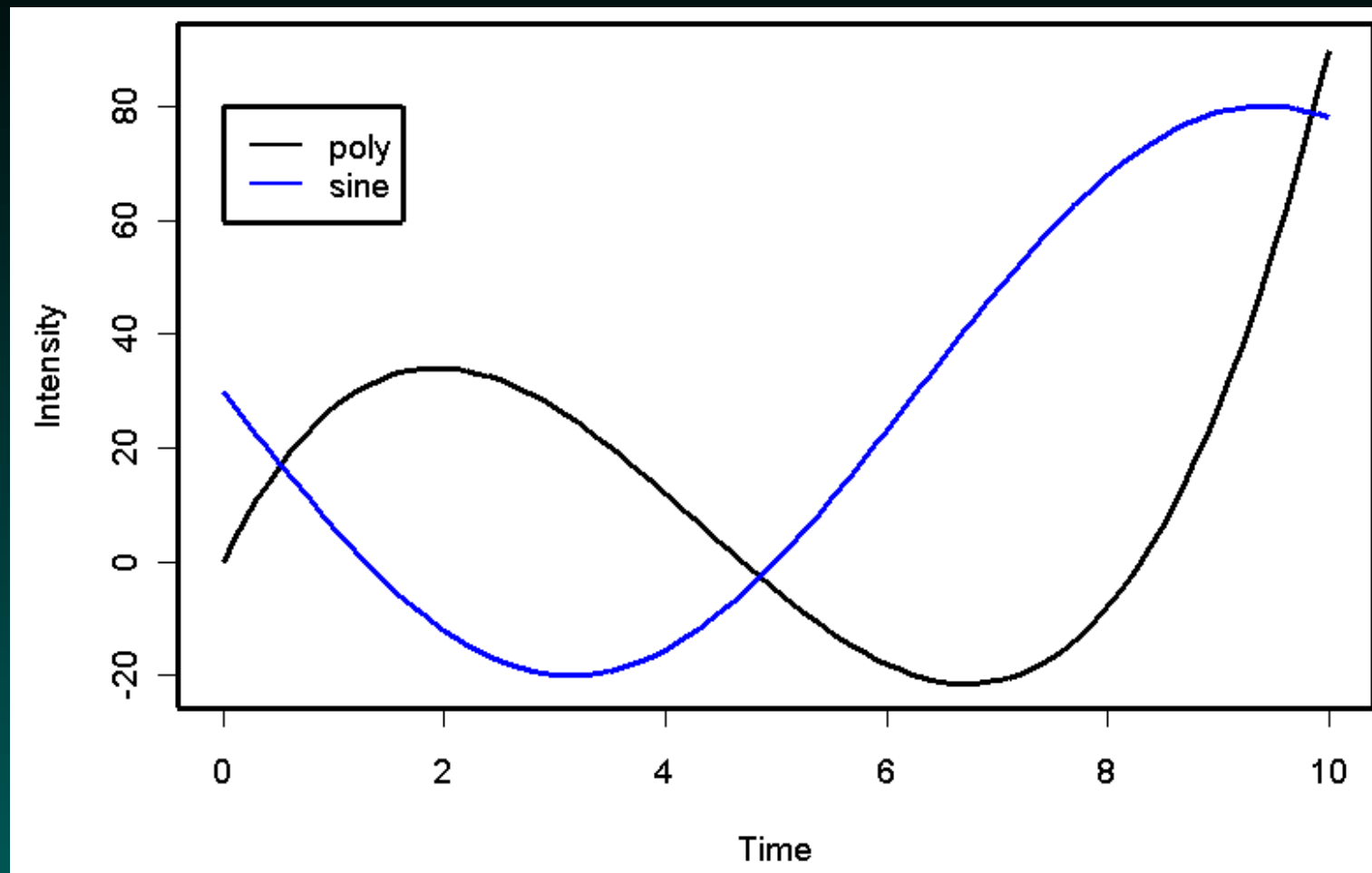
Additional graphics commands

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
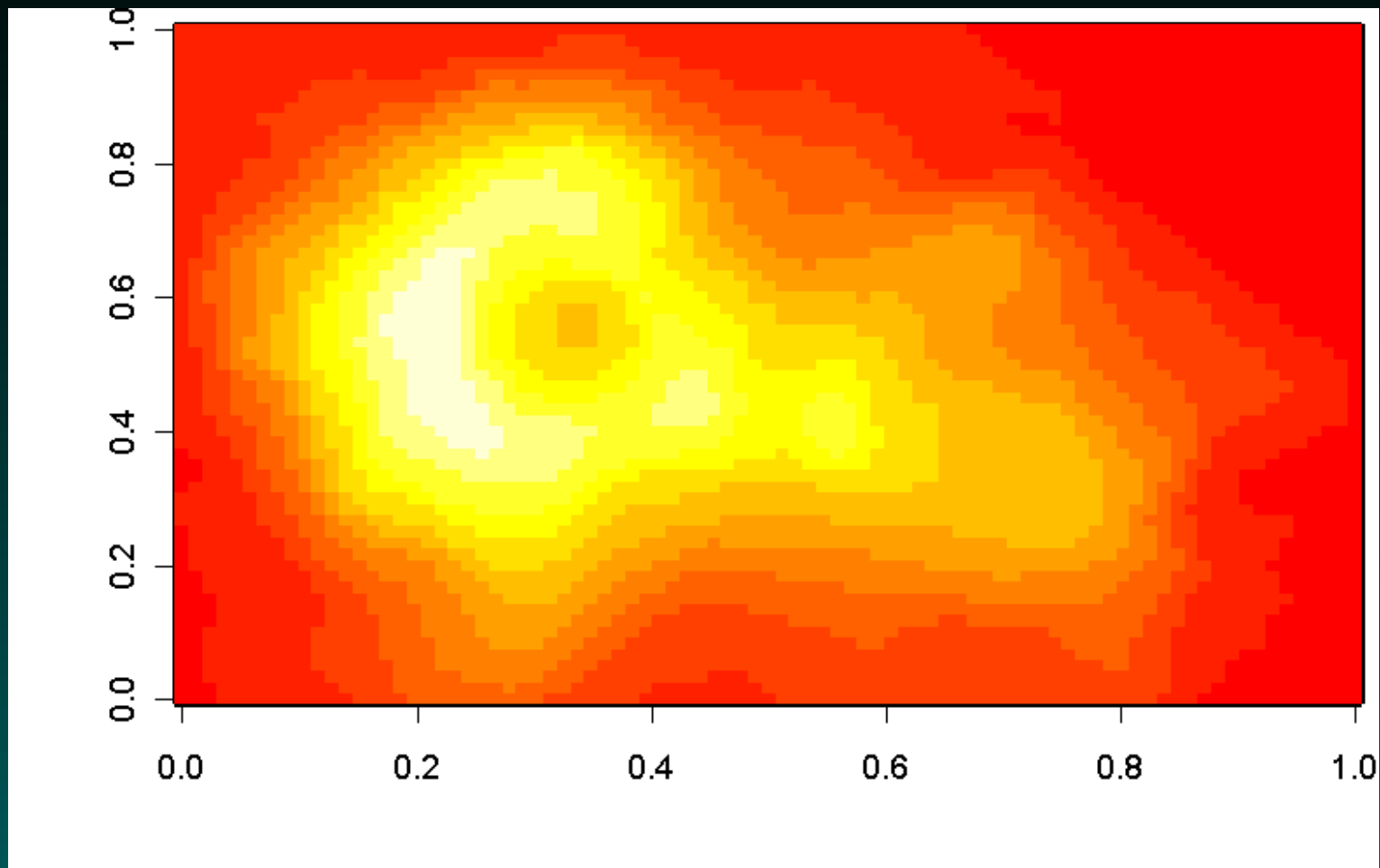

```
> data(volcano)
> image(volcano)
```

Volcano
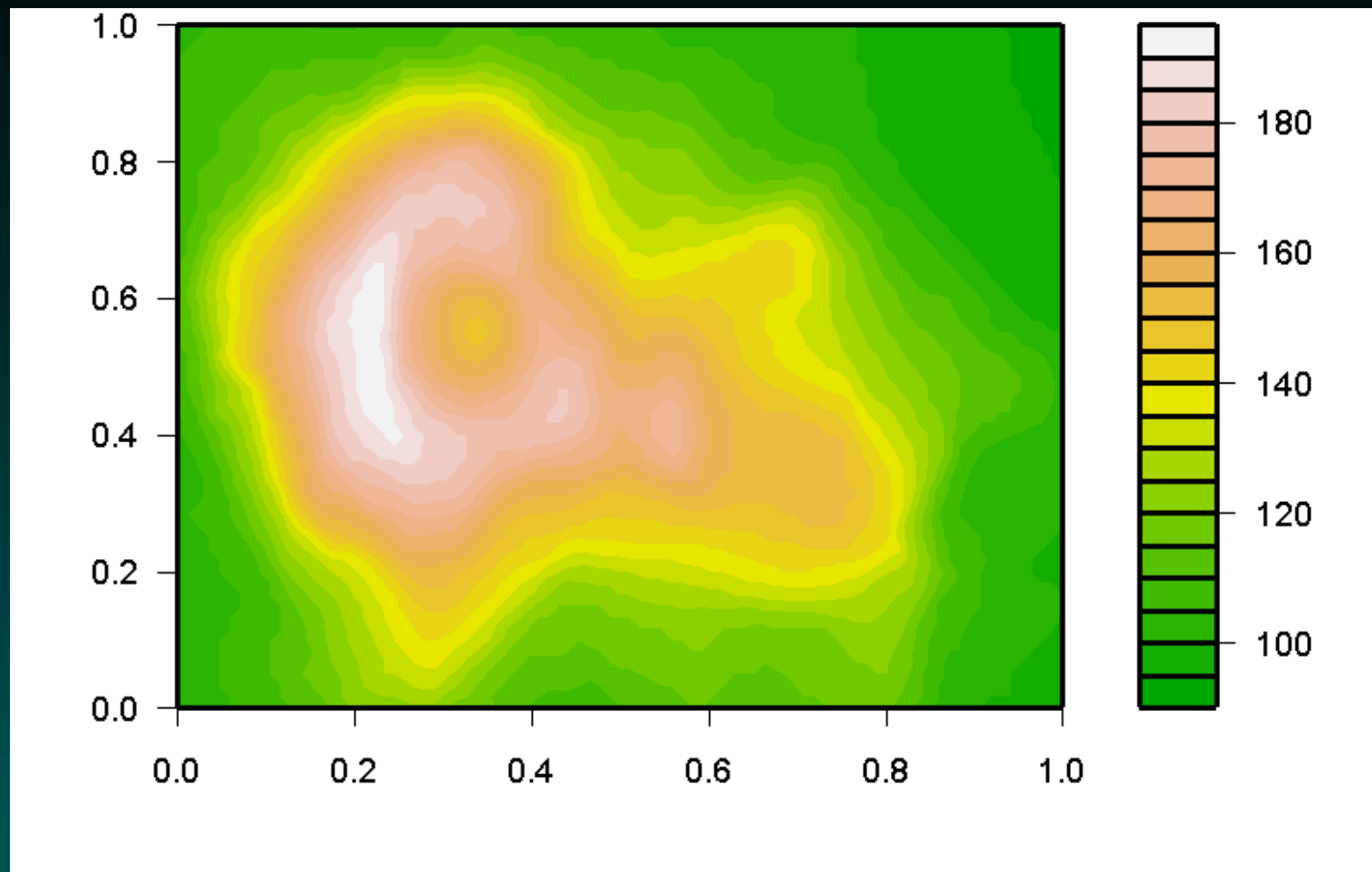

```
> filled.contour(volcano, color=terrain.colors)
```

So, What's New?

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? Well, there may be a function called matrix. Let's find out.

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(1:12)
> 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
```

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!

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
```

Reading Data Into R

While we can simply type stuff in, or use source() to pull in small amounts of data we've typed into a file, what we really want to do is to read a big table of data. R has several functions that allow us to do this, including `read.table(), read.delim(),` and `scan().`

We can experiment by using some of the files that we generated in dChip for the first HWK.

We could load the sample info file, and the list of filtered genes. Then we could use the sample info values to suggest how to contrast the expression values in the filtered gene table.