

Statistical Computing and Data Visualization in R

Lecture 1

Introduction to R

What is R?

- R is a **free** programming language and software environment for statistical computing and graphics.
- R is an open-source implementation (a “dialect”) of the S programming language developed at Bell Labs.
- To download R: <http://www.r-project.org> (more on this later)

A bit of history ...

- S was developed by John Chambers and others at Bell Lab starting in 1976 as an internal statistical analysis environment.
- StatSci (later Insightful Corp) starts developing a commercial implementation called S-plus in 1988.
- R was created by Ross Ihaka and Robert Gentleman in the Department of Statistics at the University of Auckland in 1991.
- The R Core Group (which controls the source code for R) was formed in 1997. One major release each year, along with more frequent minor updates.
- Subject to the GNU General Public License version 2.0.

The philosophy behind S

- From John Chambers:

“[W]e wanted users to be able to begin in an interactive environment, where they did not consciously think of themselves as programming. Then as their needs became clearer and their sophistication increased, they should be able to slide gradually into programming, when the language and system aspects would become more important.”

R as a programming language

- **R is an object-oriented language:** OOP is a programming paradigm that uses "objects" and their interactions to design computer programs.
 - Everything is an object having a class and attributes and methods which act on its attributes
- **R is also a functional language:** it treats computation as the evaluation of mathematical functions (*expressions* rather than *statements*).
 - Functions can be used as any other object (functions of functions).

Why R ... or not

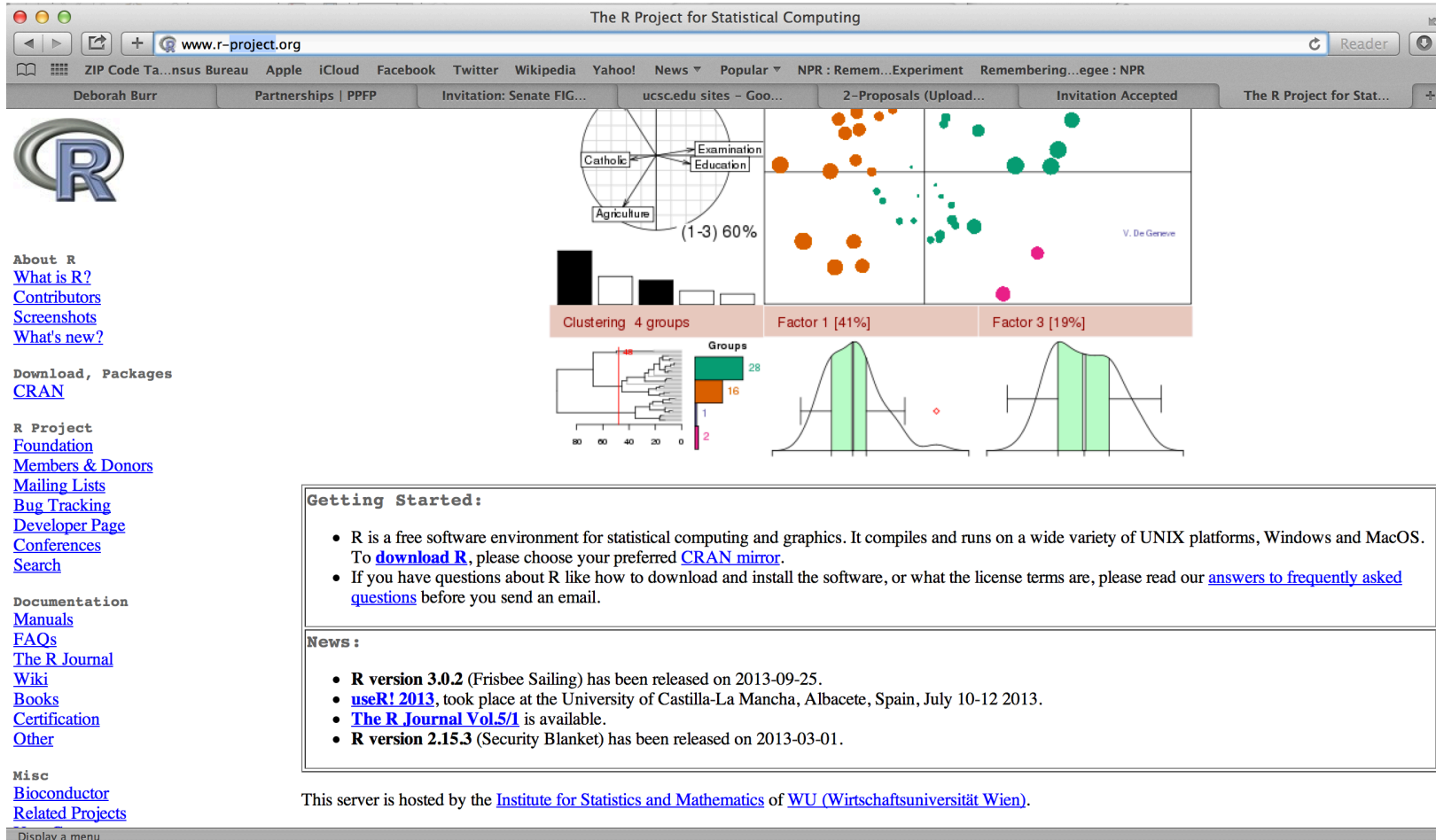
Advantages

- Extremely flexible and easy to expand.
- Powerful graphics.
- Intuitive syntax.
- Object oriented.

Disadvantages

- Relatively steep learning curve.
- Memory hog.
- Relatively slow.
- Some funny object coercion.

Obtaining the R package



The screenshot shows the homepage of the R Project for Statistical Computing. The browser window title is "The R Project for Statistical Computing" and the address bar shows "www.r-project.org". The page features a navigation bar with links to various resources, a sidebar with links to "About R", "Download, Packages", "R Project", "Documentation", and "Misc", and a main content area with a large R logo and several statistical plots. A "Getting Started" section provides information on how to obtain and use R, and a "News" section lists recent releases and events.

About R

- [What is R?](#)
- [Contributors](#)
- [Screenshots](#)
- [What's new?](#)

Download, Packages

- [CRAN](#)

R Project

- [Foundation](#)
- [Members & Donors](#)
- [Mailing Lists](#)
- [Bug Tracking](#)
- [Developer Page](#)
- [Conferences](#)
- [Search](#)

Documentation

- [Manuals](#)
- [FAQs](#)
- [The R Journal](#)
- [Wiki](#)
- [Books](#)
- [Certification](#)
- [Other](#)

Misc

- [Bioconductor](#)
- [Related Projects](#)

Getting Started:

- R is a free software environment for statistical computing and graphics. It compiles and runs on a wide variety of UNIX platforms, Windows and MacOS. To [download R](#), please choose your preferred [CRAN mirror](#).
- If you have questions about R like how to download and install the software, or what the license terms are, please read our [answers to frequently asked questions](#) before you send an email.

News:

- R version 3.0.2** (Frisbee Sailing) has been released on 2013-09-25.
- [useR! 2013](#), took place at the University of Castilla-La Mancha, Albacete, Spain, July 10-12 2013.
- [The R Journal Vol.5/1](#) is available.
- R version 2.15.3** (Security Blanket) has been released on 2013-03-01.

This server is hosted by the [Institute for Statistics and Mathematics](#) of [WU \(Wirtschaftsuniversität Wien\)](#).

Obtaining the R package

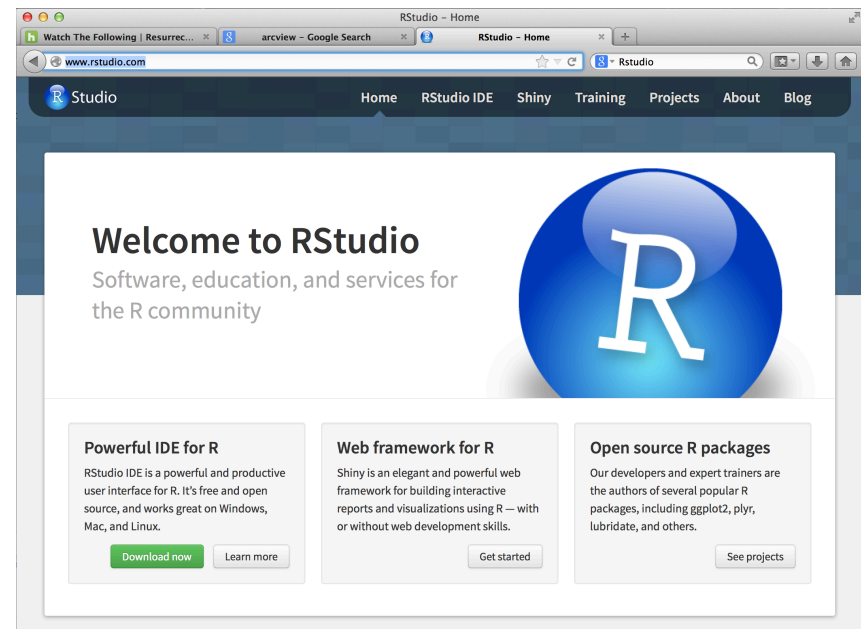
- The R system consists of the “base” R distribution (which provides basic functionality) and a suite of packages that augment the capabilities in the “base” R.
- You can develop your own packages, and make them available (under GNU General Public License version 2.0) in CRAN.

R implementations

- There are a number of implementations and GUIs for R.
 - Examples of implementations that use menus include Deducer and R Commander: Easier to use, but they are somewhat inflexible.
 - There is at least one implementation (Rexcel) that works directly from within Excel: Easier to use, but useful mostly for computing (not graphics).
 - I recommend RStudio, which is command-based (key for flexibility) but has a lot of additional functionality.

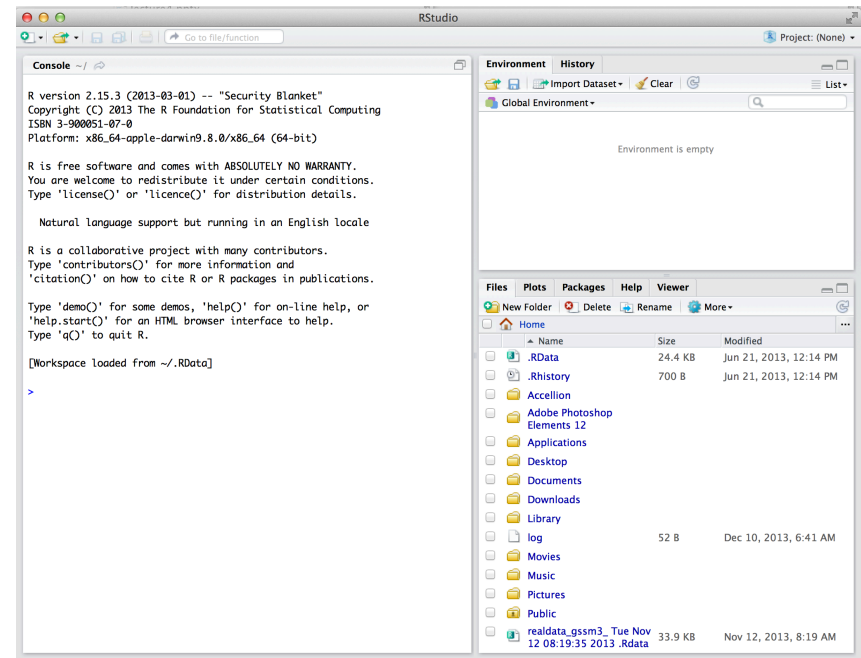
RStudio

- You can download RStudio from <http://www.rstudio.com/>
- Click on the green “Download now” button.
- In the next window click “Download RStudio Desktop”.
- Select the right download depending on whether you are a Windows or Mac user.
- You need to install R itself separately (see previous slides).



RStudio

- The image on the right shows the RStudio interface you should see once you have installed and opened RStudio.
 - Left window is the command window: you type instructions there.
 - The top right box shows you the list of objects you have created.
 - The bottom right box is a browser for files, graphs, etc.



Installing packages

- One of the key advantages of R is its expandability.
- In Rstudio, R packages can be installed using the menu “Tools/Install Packages”. Alternatively,

```
> install.packages("swirl")
```
- Once a package is installed you need to load it before you can use it,

```
> library("swirl")  
> swirl()
```
- R also includes a number of sample datasets,

```
> data()  
> faithful
```

Getting help in R

- R has very detailed help files that are easy to use once you have gotten used to the system.
- To get help of a topic use the command `help()`. For example,

```
help(boxplot)
```

explains what the function `boxplot` does (spoiler alert: it draws a boxplot!) and how it works.

- The name of most functions is pretty obvious!
- Large user community: Search the internet for answers!
- Usually multiple ways to do the same thing ...

Getting help in R

- Official manuals:
 - [An Introduction to R.](#)
 - [R Data Import/Export.](#)
 - [Writing R Extensions.](#)
 - [R Installation and Administration.](#)
- Some good books:
 - Venables, William N., and Brian D. Ripley. *Modern applied statistics with S-PLUS*. Springer Science & Business Media, 2013.
 - Wickham, Hadley. *Advanced R*. CRC Press, 2014.
 - Other books in the *Use R!* sequence.
 - [The R inferno!](#)
- The package `swirl` offers R courses from inside R.

Before we start ...

- Make sure you document your analysis!
 - I strongly recommend that you always save the commands you use for your analysis in a separate file. **Rstudio has its own editor and is great for this!**
 - File -> New File -> R script
 - Add a lot of comments into your scripts (# is the symbol for comments).
 - Indent your code.
 - Limit the width of your code.
 - The package `knitr` could also be useful (more on packages later).
 - In most cases, it pays off in the long run if you save each figure in a multi-figure panel separately.
 - Use names for objects and graphs that are recognizable.

Basics of R

- At its most basic, R can be used as a soup-up calculator:

```
> 5 + 7
```

(You type this at the prompt)

```
[1] 12
```

(This is the answer R gives you)

- Try to compute the natural logarithm of 10 and the tangent of $\pi/4$.

```
> log(10)
```

```
[1] 2.302585
```

```
> tan(pi/4)
```

```
[1] 1
```


Basics of R

- Values can be stored into objects. The symbol `<-` is the assignment operator.

```
> a <- 3
> print(a)
[1] 3
> a
[1] 3
```

- You can do multiple assignments in a single command (but I typically do not recommend it)

```
> a <- b <- 5
> print(a)
[1] 5
> print(b)
[1] 5
```

- R is case sensitive!
- Never use an R function name as an object name.

Basics of R

- The symbol `=` can also be used as an assignment operator, but it can only be used in a restricted set of circumstances (like the command line!).

```
> a = 3
```

```
> print(a)
```

```
[1] 3
```

```
> a
```

```
[1] 3
```

- I tend to use them interchangeably.

Basics of R

- To see a list of R objects in the current environment,

```
> ls()  
[1] "a" "b"
```

- To remove an object (e.g., to free memory),

```
> rm(a)    # Unusually, rm("a") is also valid  
> ls()  
[1] "b"
```

- To remove all objects,

```
> rm(list=ls()) # Here = is not an assignment  
> ls()  
character(0)
```

Types of R objects

- Each object in R has a class. Basic classes in R:
 - Character
 - Numeric (real numbers) => Default class for numbers!!
 - Integer
 - Complex
 - Logical (TRUE/FALSE)
 - Factors
- Examples of objects in R:
 - Vectors
 - Matrices and Arrays
 - Data Frames
 - Lists
- Objects may have additional attributes beyond the class.

Vectors

- To create a vector with given entries:

```
> x <- c(1, 3, 4, 9, 5)
```

```
> x
```

```
[1] 1 3 4 9 5
```

```
> x <- c("CA", "OR", "WA", "NE")
```

```
> x
```

```
[1] "CA" "OR" "WA" "NE"
```

- To create an empty vector of a given length:

```
> x <- vector("logical", 5)
```

```
> x
```

```
[1] FALSE FALSE FALSE FALSE FALSE
```

```
> y <- vector("numeric", 8)
```

```
> y
```

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

- To create a vector of length 0, a shortcut to `x <- vector("numeric", 0)` is `x <- numeric()`.

Vectors

- You can create vectors that contain linear sequences:

```
> x <- seq(1,8, by=2)
```

```
> x
```

```
[1] 1 3 5 7
```

```
> x <- 3:8
```

```
> x
```

```
[1] 3 4 5 6 7 8
```

- You can also create vectors with repeated elements:

```
> x <- rep(TRUE, 6)
```

```
> x
```

```
[1] TRUE TRUE TRUE TRUE TRUE TRUE
```

```
> y <- rep(seq(1,3), 2)
```

```
> y
```

```
[1] 1 2 3 1 2 3
```

```
> y <- rep(seq(1,3), each=2)
```

```
> y
```

```
[1] 1 1 2 2 3 3
```

Vectors

- All elements of a vector must belong to the same class (integers, numeric, logical, etc).
- To check the class of the elements of a vector:

```
> x <- rep(TRUE, 6)
> is.numeric(x)
[1] FALSE
> is.logical(x)
[1] TRUE
```

- If you mix classes, automatic coercions will happen

```
> x <- c("CA", 1, TRUE)
> is.numeric(x)
[1] FALSE
> is.logical(x)
[1] FALSE
> is.character(x)
[1] TRUE
> x <- c(1.5, 3.7, 2, TRUE)
> is.numeric(x)
[1] TRUE
```

Vectors

- To check the length of a vector

```
> x <- c(0, 1, 2, 0, 4, 1)
> length(x)
[1] 6
```

- You can force coercions, but some do not work well ...

```
> x <- c(0, 1, 2, 0, 4, 1)
> is.numeric(x)
[1] TRUE
> as.logical(x)
[1] FALSE TRUE TRUE FALSE TRUE TRUE
> y <- c(TRUE, FALSE, TRUE, TRUE)
> as.numeric(y)
[1] 1 0 1 1
> z <- c("CA", "NE", "OR")
> as.numeric(z)
[1] NA NA NA
```

```
Warning message:
NAs introduced by coercion
```


Vectors

- Missing values (NA) and not-a-number (NaN) are special values that might result from undefined operations.

```
> x <- c(NA, 4, 3, NA, 1)
> is.numeric(x)
[1] TRUE
> is.na(x)
[1] TRUE FALSE FALSE TRUE FALSE
> 0/0
[1] NaN
```

- R also has a special “number” for infinity, Inf

```
> 3/0
[1] Inf
> 4 + Inf
[1] TRUE
> 3/Inf
[1] 0
> Inf/Inf
[1] NaN
```

Vectors

- You can sort and rank the elements of a vector:

```
> x <- c(1.3, -2.2, 1.15, 0.23, -1.1)
> sort(x)
[1] -2.20 -1.10  0.23  1.15  1.3
> x[order(x)]
[1] -2.20 -1.10  0.23  1.15  1.3
> order(x)
[1] 2 5 4 3 1
> rank(x)
[1] 5 1 4 3 2
```

- Many other functions:

```
> x <- c(1.3, -2.2, 1.15, 0.23, -1.1)
> sum(x)
[1] -0.62
> prod(x)
[1] 0.832117
> which(x < 0)
[1] 2 5
```

Some useful functions for vectors

Summary statistic

Number of elements

Mean

Median

Maximum

Minimum

Midrange

Range

10% quantile

Variance

Standard Deviation

Logical operators

All entries true?

At least one entry true?

How to compute in R ...

```
length(x)
```

```
mean(x)
```

```
median(x)
```

```
max(x)
```

```
min(x)
```

```
(max(x)+min(x))/2
```

```
max(x)-min(x)
```

```
quantile(x, 0.10)
```

```
var(x)
```

```
sd(x)
```

How to compute in R ...

```
all(x)
```

```
any(x)
```

Vectors

- R likes vectors. By default all operators are “vectorized”, i.e., are applied to each element of the vector (note differences with MATLAB)

```
> x <- c(3, -1, 2, 4)
```

```
> x + 3
```

```
[1] 6 -1 5 7
```

```
> y <- c(2, 1, 4, 3)
```

```
> x*y
```

```
[1] 6 -2 8 12 (Elementwise operations, not a scalar or matrix!)
```

```
> x >= 3
```

```
[1] TRUE FALSE FALSE TRUE
```

```
> x == -1
```

```
[1] FALSE TRUE FALSE FALSE
```

- More on vectorization later!
- BTW, be careful, $x < -1$ and $x < -1$ are different!

Comparing two vectors

- There are at least three ways to test whether two vectors are identical:

```
> x <- c(3,1,2,4)
> y <- x + 3
> identical(x+3,y)
[1] TRUE
> all((x+3)==y)
[1] TRUE
> all.equal(x,y)
[1] TRUE
```

- However, there is a subtle difference between them

```
> y <- exp(log(x))
> identical(x+3,y)
[1] FALSE
> all((x+3)==y)
[1] FALSE
> all.equal(x,y)
[1] TRUE
```

Recycling

- If the length of two vectors/matrices involved in vectorized operations do not match, the values of the smaller one are recycled

```
> x <- c(0,1,2,3)
```

```
> y <- c(2,1)
```

```
> x + y
```

```
[1] 2 2 4 4 #As if you had defined y = c(2,1,2,1)
```

- You are not warned about recycling except when dimensions are not multiple of each other

```
> x <- c(0,1,2,3)
```

```
> y <- c(2,1,4)
```

```
> x + y
```

```
[1] 2 2 6 5 #As if you had defined y = c(2,1,4,2)
```

```
Warning message:
```

```
In x + y : longer object length is not a multiple  
of shorter object length
```

Sub-setting vectors

- You can access specific elements of a vector. One option is to specify specific entries using a numeric vector of indexes:

```
> x <- c(3, -1, 2, 4, 7, -1)
```

```
> x[c(1, 3, 6)]
```

```
[1] 3 2 -1
```

```
> x[2:5]
```

```
[1] -1 2 4 7
```

- Another option is to use a logical vector (if shorter than vector being sub-set, it is recycled and no error is generated!):

```
> x <- c(3, -1, 2, 4, 7, -1)
```

```
> x[x != -1]
```

```
[1] 3 2 4 7
```

```
> x[c(TRUE, FALSE, TRUE)]
```

```
[1] 3 2 4 -1
```

Sub-setting vectors

- When multiple conditions need to be checked you can use logical operators:

```
> x <- c(3,-1,2,4,7,-1)
> x[x== -1 | x==4 | x==5]
[1] -1  4 -1
> x[x <= 7 & x>3]
[1] 4 7
```

- When multiple or operators are required there are streamlined options

```
> x <- c(3,-1,2,4,7,-1)
> x[is.element(x, c(-1,4,5))]
[1] -1  4 -1
> x[x %in% c(-1,4,5)]      # Alternative syntax
[1] -1  4 -1
```

- Set operators can be used to define the second argument

```
> x <- c(3,-1,2,4,7,-1)
> z <- c(2,-1)
> x[is.element(x, union(c(-1,4,5), c(2,-1)))]
[1] -1  2  4 -1
> x[is.element(x, intersect(c(-1,4,5), c(2,-1)))]
[1] -1 -1
```


Add/remove elements from a vector

- The `[]` operator can also be used to remove elements from a vector:

```
> x <- c(3, -1, 2, 4, 7, 1)
> x[-c(1, 5)]
[1] -1  2  4  1
```

- To add new elements at the end of a vector we can either use the concatenation operator, or assign an extra element

```
> x <- c(3, -1, 2, 4, 7, 1)
> x <- c(x, 2)
> x
[1] 3 -1  2  4  7  1  2
> x[8] <- -4
[1] 3 -1  2  4  7  1  2 -4
```

- To add elements somewhere in the middle

```
> x <- c(3, -1, 2, 4, 7, 1)
> x <- append(x, 78, 3) # Adds 78 after the 3rd element
> x
[1]  3 -1  2 78  4  7  1  2
```

Probability distributions and pseudorandom number generation

- R provides functions to compute the pdf/pfm, cdf, inverse cdf of many standard distributions (these functions are vectorized and values are recycled).

```
> dnorm(1, mean=0, sd=1)      #Density at 1
[1] 0.2419707
> pnorm(1, mean=0, sd=1)      #p(X ≤ 1)
[1] 0.8413447
> qnorm(0.5, mean=0, sd=1)    #x s.t. p(X ≤ x)=0.5
[1] 0
```

- R also provides random number generators for many of the same distributions (also vectorized and use recycled values)

```
> rnorm(4, mean=3, sd=1)
[1] 4.024884 5.293606 2.172033 2.425855
> rbinom(10, size=12, prob=0.2)
[1] 2 1 1 0 3 4 1 3 5 0
```

Probability distributions and pseudorandom number generation

- To sample from a discrete distribution (with or without replacement):

```
> x <- c("CA", "NE", "OR", "WA", "UT")
> sample(x, 3, replace=TRUE)
[1] "WA" "NE" "NE"
> sample(x, 3, replace=FALSE)
[1] "WA" "UT" "NE"
```

- Sometimes useful to get the results in log scale directly (rather than taking the log yourself)

```
> exp(pnorm(1, mean=0, sd=1, log.p=T))
[1] 0.8413447
> pnorm(1, mean=0, sd=1)
[1] 0.8413447
```

Probability distributions and pseudorandom number generation

- You can set the seed of the underlying uniform random variate generator:

```
> set.seed(212957)
```

This is specially useful if you want to be able to reproduce your results later (e.g., for debugging purposes)

- You can also change the underlying uniform random variate generator

```
> RNGkind(kind = "Marsaglia-  
Multicarry", normal.kind = NULL)
```

Matrices

- A matrix is a two-dimensional array of elements of the same type. To define an empty one:

```
> x <- matrix(0, nrow=2, ncol=3)
```

```
> x
```

	[,1]	[,2]	[,3]
[1,]	0	0	0
[2,]	0	0	0

- If the first argument is a vector, the matrix is filled with it (by default columnwise, and values recycled).

```
> x <- matrix(c(1,2,3,4,5), nrow=2, ncol=3)
```

```
> x
```

	[,1]	[,2]	[,3]
[1,]	1	3	5
[2,]	2	4	1

Matrices

- The default direction in which the array is filled can be changed:

```
> x <- matrix(c(1,2,3,4,5,6), nrow=2, ncol=3, byrow=T)
> x
```

	[,1]	[,2]	[,3]
[1,]	1	2	3
[2,]	4	5	6

- You can attach names to the rows and columns of the matrix.

```
> x <- matrix(c(1,2,3,4,5,6), nrow=2, ncol=3,
dimnames=list(c("Row1", "Row2"),c("Col1", "Col2", "Col3")))
> x
```

	Col1	Col2	Col3
Row1	1	3	5
Row1	2	4	6

```
> y <- rownames(x)
> y
```

```
[1] "Row1" "Row2"
```

```
> colnames(x) <- c("AA", "BB", "CC")
> x
```

	AA	BB	CC
Row1	1	3	5
Row2	2	4	6

Matrices

- You can also create matrices by “binding” vectors together

```
> x <- c(1,2,3,4,5,6)
> y <- c(11,12,13,14,15,16)
> z1 <- rbind(x,y)
> z1
```

	[,1]	[,2]	[,3]	[,4]	[,5]	[,6]
x	1	2	3	4	5	6
y	11	12	13	14	15	16

- If you want to generate a 6 by 2 matrix instead, use `cbind`.

Matrices

- Matrices are sub-set in the same way as vectors:

```
> S <-  
matrix(c(1,0.5,1.3,0.4,-1,-0.2,0.5,-1,0.6), 3,  
3)  
> S[1:2,c(1,3)]  
      [,1] [,2]  
[1,]  1.0  0.5  
[2,]  0.5 -1.0
```

- You can apply functions to rows and columns (more on this later):

```
> S <-  
matrix(c(1,0.5,1.3,0.4,-1,-0.2,0.5,-1,0.6), 3,  
3)  
> apply(S, 2, sum)  
[1,]  2.8 -0.8  0.1
```

Retain second dimension (columns)
collapse over all others

Function to be applied

Matrices

- You can access matrices as if they were vectors

```
> S <-  
matrix(c(1,0.5,1.3,0.4,-1,-0.2,0.5,-1,0.6), 3,  
3)  
> S  
      [,1] [,2] [,3]  
[1,]  1.0  0.4  0.5  
[2,]  0.5 -1.0 -1.0  
[3,]  1.3 -0.2  0.6  
> S[6]  
[1,] -0.2  
> S[6:8]  
[1,] -0.2  0.5 -1.0
```

- Matrices get converted into vectors by column

Matrices

- To figure out the size of a matrix:

```
> S <-  
matrix(c(1, 0.5, 1.3, 0.4, -1, -0.2, 0.5, -1, 0.6), 3, 3)  
> dim(S)  
[1] 3 3  
> length(S)  
[1] 9
```

- Unlike Matlab, R “drops” unnecessary dimensions by default. This can be problematic, but can be avoided:

```
> S[, 3]  
[1] 0.5 -1.0 0.6  
> S[, 3, drop=FALSE]  
[,1]  
[1,] 0.5  
[2,] -1.0  
[3,] 0.6
```

Matrices

- Linear algebra operations are easy in R.

```
> S <- matrix(c(1,0.4,0.5,-1), 2, 2)
> x <- c(0, 1)
> S%*%x                                # matrix product
      [,1]
[1,]  0.5
[2,] -1.0
> eigen(S)                             # Eigendecomposition of S
$values
[1]  1.095445 -1.095445

$vectors
      [,1]      [,2]
[1,] 0.9822637 -0.2320969
[2,] 0.1875046  0.9726927
```

Matrices

- Linear algebra operations are easy in R.

```
> S <- matrix(c(1,0.4,0.5,-1), 2, 2)
```

```
> x <- c(0, 1)
```

```
> t(S) # Transpose
```

```
      [,1] [,2]  
[1,]  1.0  0.4  
[2,]  0.5 -1.0
```

```
> solve(S) # Inverse
```

```
      [,1] [,2]  
[1,] 0.8333333 0.4166667  
[2,] 0.3333333 -0.8333333
```

```
> solve(S,x) # Solution of Sb=x
```

```
[1] 0.4166667 -0.8333333
```

Higher order arrays

- R supports higher order arrays:

```
> S <- array(1:24, dim=c(4,2,3))
```

```
> S
```

```
, , 1
```

	[,1]	[,2]
[1,]	1	5
[2,]	2	6
[3,]	3	7
[4,]	4	8

```
, , 2
```

	[,1]	[,2]
[1,]	9	13
[2,]	10	14
[3,]	11	15
[4,]	12	16

```
, , 3
```

	[,1]	[,2]
[1,]	17	21
[2,]	18	22
[3,]	19	23
[4,]	20	24

Characters

- Strings are denoted by quotation marks.
- You can easily concatenate strings.

```
> x <- "Hello"
> y <- "Bye"
> z <- "world"
> paste(x, z)
[1] "Hello world"
> w <- c(x,y)
> paste(w, z, sep=" cruel ")
[1] "Hello cruel world" "Bye cruel world"
> paste(w, z, sep=" cruel ", collapse=" and ")
[1] "Hello cruel world and Bye cruel world"
```

- You can also trim a string

```
> substr(x, 2, 4)
[1] "ell"
> substr(x, nchar(x)-2, nchar(x)) #Pick last 3 chars
[1] "llo"
```

Factors

- Factors are meant to store nominal variables.

```
> z <- factor(c("CA", "NE", "OR", "CA", "CA", "OR", "OR",  
"CA", "NE"))
```

```
> z      #Categorical variable with 3 levels
```

```
[1] CA NE OR CA CA OR OR CA NE
```

```
Levels: CA NE OR
```

```
> table(z)
```

```
z
```

```
CA NE OR
```

```
4  2  3
```

```
> z <- factor(c("CA", "NE", "OR", "CA", "CA", "OR", "OR",  
"CA", "NE"), levels=c("OR", "CA", "NE", "WA"))
```

```
> z      #Categorical variable with 4 levels, first is OR
```

```
[1] CA NE OR CA CA OR OR CA NE
```

```
Levels: OR CA NE WA
```

```
> table(z)
```

```
z
```

```
OR CA NE WA
```

```
3  4  2  0
```