Introduction to R

R is a general-purpose computer language. It is similar to S language, and much code written for S run unalterd under R. Its particular strength is in statistical, graphical capabilities. You can also do simple simulation with R. R is *free*: it's nothing to do with *free* as in beer. You get *free*dom to do whatever with R! So once you learn R, R will not disappear like a lot of other commercial products.

Some users think of R as a statistics system, but it is more of an environement within which statistics can be implemented. Capability of R can be easily extended by *packages*. There are many user-contributed packages which convers a wide range of **modern statistics**. These extra packages can be downloaded from The Comprehensive R Archive Network (cran): http://cran.us.r-project.org/ .

R runs on Linux, Mac, and Windows.

Home page:

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

Goto http://www.faculty.uaf.edu/ffnt/teaching/popgen/R-tutorial/R-tutorial.html for the online version of this document.

1 Installation

- For Mac OS-X, go http://cran.us.r-project.org/bin/macosx/ , and click R-x.x.x.dmg (x.x.x is a version number). Double click the downloaded disk image, and double click R.mpkg, and follow the instructions.
- For Windows, go http://cran.us.r-project.org/bin/windows/base/ , click "Downlad R x.x.x for Windows", and follow the direction of installer. I don't know much about Windows, but it was pretty simple.
- Linux users probably know how to install R (e.g., `yum install R` in Fedora/CentOS).

1.1 Packages

- R is modular, and you can enhance the functionality of R by using Contributed packages in CRAN (http://cran.us.r-project.org/web/packages/). To install these extra packages, you can see the Section 6 of "R Installation and Administration" (http://cran.us.r-project.org/doc/manuals/R-admin.html).
- One way of installing packages are using `install.packages()` inside of R.

  ```
  > install.packages("ape", dependencies = TRUE)
  ```

  This will automatically download and install the package "ape".
- To "use" the package,

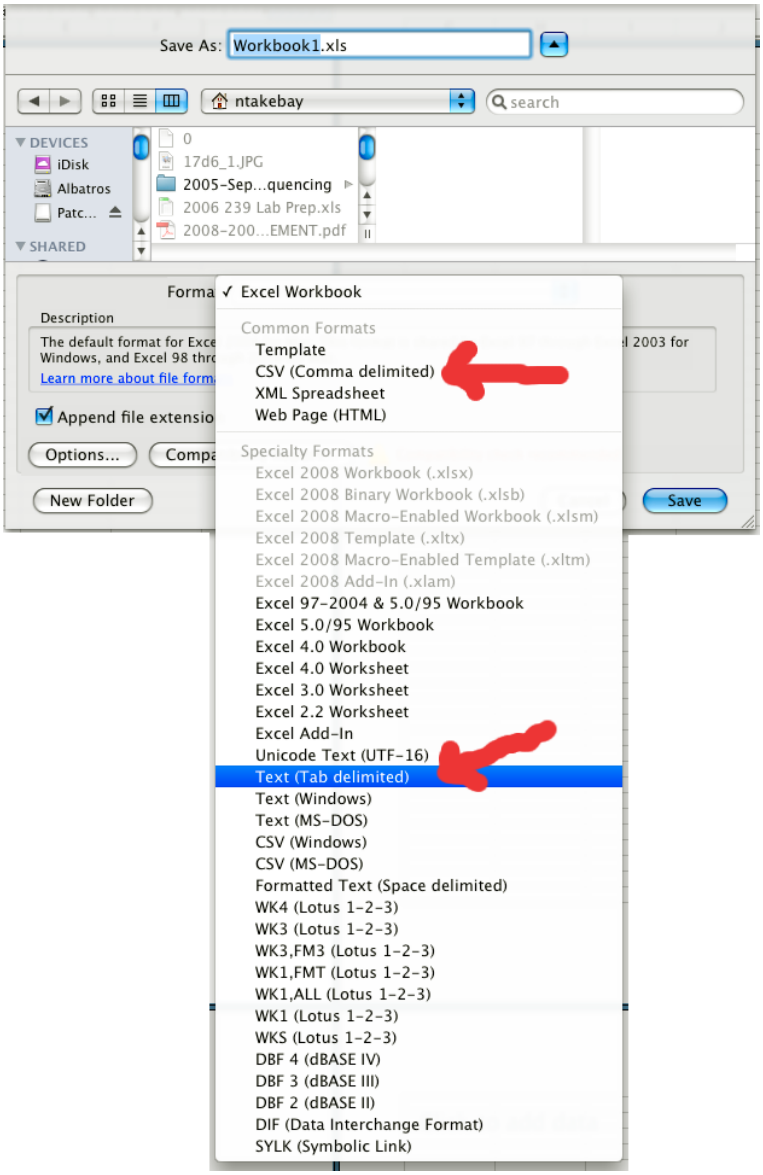
  ```
  > library("nlme")
  ```

  Note that package "nlme" is included in the default installation, so you did not need to install it before you use it.

2 Basics: Importing data

2.1 From spread-sheet (e.g. OpenOffice.org and Excel)

- Easiest to transfer the data by converting the data into a simple text file. For example, spread-sheet can export the data to comma-delimited (CSV) or tab-delimited text files. tab-delimination is probably better because if some cells contains comma as a part of the data, column alignment may get screwed up with CSV.
- Steps in spread-sheet
  1. Select the sheet which contains the data
  2. Clean up the data
     - If your data contains clumn header, make the first row to be the column header.
     - Simplify the column header.
       * Use only **alphanumeric characters** (A-Z, 0-9). **Underscores** (_) and **periods** (.) are ok to use.
         Remove following characters: `#, comma (,), ;, :, ?`, etc.
       * Remove **spaces**.
         E.g. use "dryWeight" instead of "dry weight".
       * Each header should be unique.
         You can't have two columns with the same name of "weight". You should change them to "weight.1" and "weight.2".
       * R is **case-sensitive**.
         "dryWeight" and "dryweight" are different.
       * Actually, it probably works even if you don't follow the above rules. During the import, these spaces and bad characters automatically get converted to '.'.
     - Check the contents of data
       * Missing data can be left as empty cells or use 'NA' (not applicable).
         These empty cells automatically get converted to 'NA', which is the correct representation of missing data in R.
  3. Save as tab-delimite (or comma delimited) text file
     File -¿ Save As
     In "Fomat:", select "Text(Tab delimited)" (or "CSV")



2.2 To R environment

OK, we now have a data file with simple text format (either from some simulation or from Excel). After starting up R, type in the following commands in the R prompt ('>').

- Steps in R

  1. `read.table()`

     ```
     dat.in  <- read.table("data.txt", header=T, sep="\t")
     ```

- `read.table()` is a command (function) to read spread-sheet like text file.
  NOTE: All commands have this format of `commandName(arguments)`. Arguments are separated by comma's.
- Use "header=T" if the 1st row of the data in text is column names.
- Use "header=F" if not. This will create a data.frame.
- You may need to specify the full path:

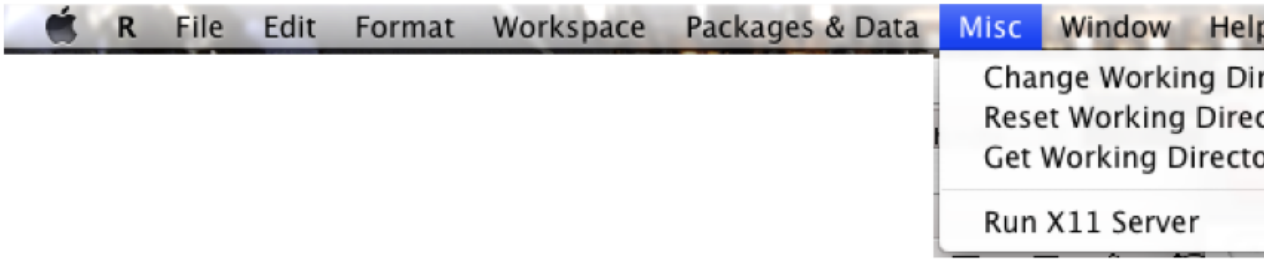
  ```
  > dat.in <- read.table("/Users/naoki/doc/analysis/data.txt", header=T)
  ```

  Or use `setwd()` to set the current working directory or the R process.

  ```
  > getwd()                          # print out the current working directory
  > setwd("/Users/naoki/doc/analysis/")
  > getwd()
  > dat.in <- read.table("data.txt", header=T)
  ```

  Or you can do the same thing if you are using GUI R.
  From the menu, select

  ```
  Misc -> Changing Working Directory ...
  ```



  And navigate the file system to find the directory which **CONTAINS** the data file.

2. Check if you have successfully imported the data.

   ```
   names(dat.in)
   ```

   This will show you the list of column names.

   ```
   dim(dat.in)
   ```

   This will show you the dimension (size) of data set: number of rows and columns.

- A little bit of explanation and summary:

  What we have done above is that you used the command (function) `read.table()`, and the results of this command (data read in the way R can understand) is stored in a container called `dat.in`.

  `<-` is called an assignment (it looks like an arrow). The results of `read.table()` is assigned to `dat.in`.

  You can use whatever name for this storage container (I chose `dat.in`). Also, R can import multiple files. For example if you have two files to read in, you can store one set of data in `simDat` and the othe set in `obsDat`.

  This storage container of spread-sheet like data is called **data frame**, which we will talk more in the next section.

2.3 Exercise: importing data

Try importing some Excel files.

3 Quick tour of R

In this section, we'll look what kind of things R can do. I'll save the detail for the later section.

Let's say that we had grew 10 plants in shady and sunny environments, and measured the leaf lengths and width. You can use the data imported in the previous Exercise.

- Read in a tab delimited file, and create a data frame.

  ```
  dat.in <- read.table("data.txt", header=T, sep="\t")
  ```

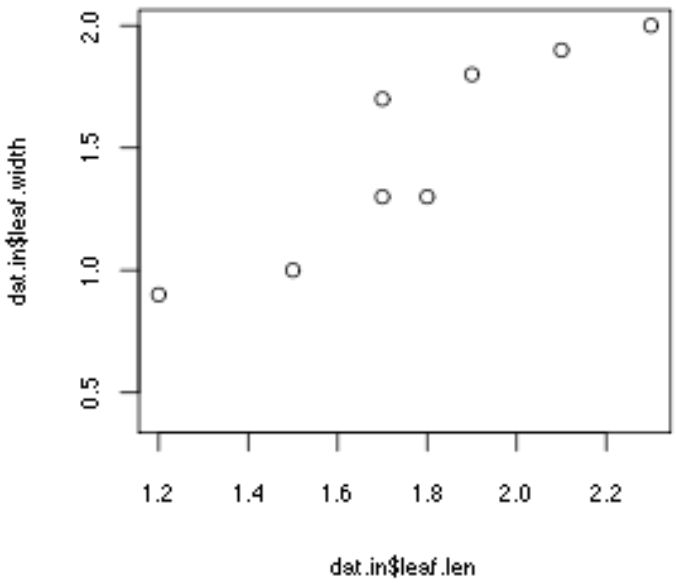
- Check the contents.

  ```
  names(dat.in)
  dim(dat.in)
  dat.in
  ```

- Some data exploration

  1. Scatter plot

     ```
     plot(dat.in$leaf.len, dat.in$leaf.width)
     ```

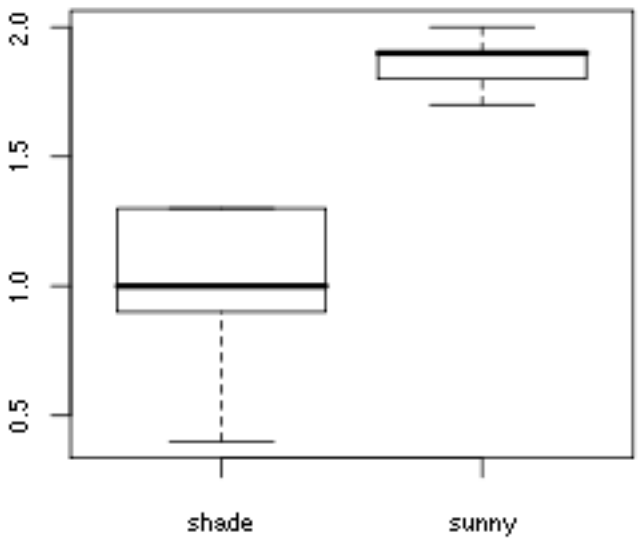
     This shows the leaf length and width seems to be correlated.

  2. Calculating the correlation, "use=" option specify how to deal with NA's.

     ```
     cor(dat.in$leaf.width, dat.in$leaf.len, use="complete.obs")
     ```

3. box plot

```
boxplot(leaf.width ~ treatment, data=dat.in)
```



This shows the plants in sunny environment have bigger leaves.

4. Mean leaf width for each treatment group: 0.98 for shade and 1.86 for sunny.

```
mean(dat.in[dat.in$treatment == 'shade', "leaf.width"])
mean(dat.in[dat.in$treatment == 'sunny', "leaf.width"])
```

5. linear model (ANOVA)

```
anova.fit <- lm (leaf.width ~ treatment, data=dat.in)
anova(anova.fit)
```

Created anova table:

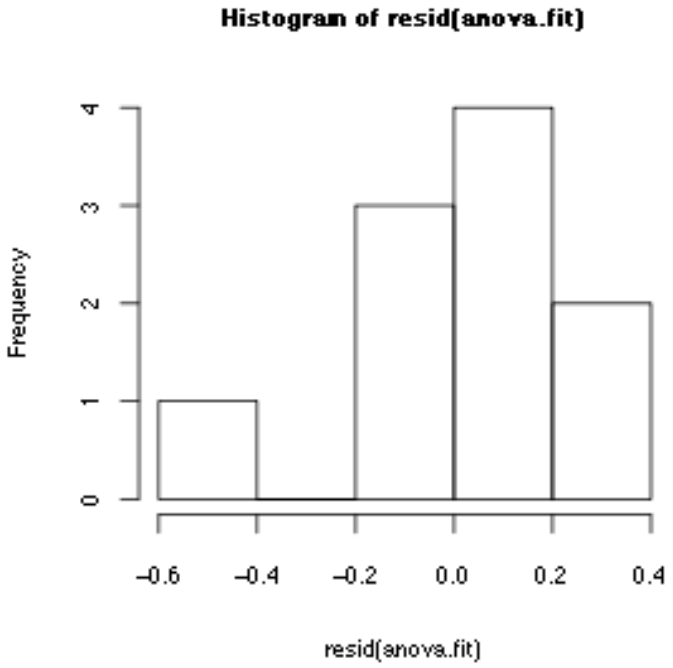
```
Analysis of Variance Table

Response: leaf.width
          Df Sum Sq Mean Sq F value    Pr(>F)
treatment  1  1.936   1.936  25.813 0.0009523 ***
Residuals  8  0.600   0.075
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

Slightly differnt wayt to look at the fit of linear model.

```
summary(anova.fit)
```

Looking at the distribution of residuals:

```
hist(resid(anova.fit))
```



This doesn't make a nice gaussian distribution because there are only 10 observations.

3.1 Exercise: repeat the analysis with leaf.len (start from producing the box plot).

4 Basics: data manipulation/massaging

- misc
  - case sensitive
  - You can use alphanumeric characters, underscore (_) and period (.).
  - "#" is used to add comments, so everything after "#" is ignored by R.
  - Each command can be terminated by semi-colon (;) OR new-line.
  - Number of "spaces" doesn't matter.
- Assignment of **variables (objects)** and expression
  Basic commands can be *assignment* or *expression*.

```
> d <- 6.2    # Assignment
> d           # Expression: display the value assigned to varidable d
> 6.2   ->  d # another way of assignment
```

Once you assign values to variables, R will remember the values in the memory. But I bet that your memory isn't as good as R's. When you forget what variables you have defined, type:

```
> ls()                # list
```

If you want R to forget the definition of variables, type:

```
> rm(d)                # remove d
```

- Getting help

  R comes with great online documentations. Since R is extremely feature rich, nobody can memorize all details, so it's essential to know how to find the documentation.

```
> help("rm")
> help.search("bootstrap")  # keyword search
> help.start()
```

- Vectors
  "c" means concatenate.

```
> d <- c(2.5, 2, 4, 5)   # a vector of 4 elements
> d
> e <- c(1,1,1)          # a vector of 3 elements
> e
> f <- c(d, e)           # f formed by concatenating d and e, 7 elements
> f
> f[4]                   # extract the 4-th element
> g <- f[3:6]            # create a new vector with 3rd to 6-th elements
```

Note well that () and [] are not interchangeable

  - () is used to indicate *functions* or arithmetic groupings.
  - [] is used to indicate the *indices* of a vector.

  Other ways to create vectors

```
> a <- rep(2, 10)
> a
> a <- seq(3, 9)
> a
> 1:10
> seq(3, 4, 0.1)
```

- Arithmetic operations

```
> 5.1 / 3 + 2 * ((3 + 4.1)^2 - 5)
> d2 - 3
```

Operations on vectors — vectorized operations is one of the quirks/strengths of R.

Operations are performed element by element.

```
> g <- c(1,2,3)
> g + e
> e / g
> g + 2
> g * (e + 1)
> 1:9 / g
```

Note that the two vectors can have different lengths. The shorter vector is *recycled* as often as needed.

- Math functions
  These functions are applied to *each* element.

```
> a <- c(1, 2, 3, 4, 5, 6)
> sqrt(a)
> exp(a)
> log(a)
> a^2
> sum(a)
> prod(a)
> mean(a)
```

- Types of vectors

  Numeric vectors/variables: Examples: c(1.0, 2.1, -0.3), 2:10

  Character vectors/variables: Each element is a character string

```
> pets <- c(2,11,4)
> names(pets) <- c("cat","fish","shrimp")
> num.fur.balls <- pets["cat"]
```

  Logical vectors/variables: Contains TRUE or FALSE

  - R allows manipulation of logical quantities: TRUE or FALSE.
  - Additionally, logical vector can take NA (not available) for missing data.
  - Generally, logical vectors are created by *conditions*
  - Logical operators: `<, <=, >, >=, ==, !=`

```
> a <- 1:5
> t.or.f <- c(T, F, F, F, T)
> gt3 <- a > 3
> even <- a %% 2 == 0
> a[t.of.f]
> b <- c(0.5, 0.2, NA, 0.1)
> b <- b[! is.na(b)]         # eliminate the missing data
```

- Data frames

  - Data frames contain rows and columns, similar to spread-sheets.

```
> x <- c(1, 2, 3, 4)
> y <- c(5, 6, 7, 8)
> z <- c(9, 10, 11, 12)
> dat <- data.frame(x, y, z)       # creates 4 rows, 3 columns data frame
> dat
> names(dat)                       # each column has a name
> named(dat) <- c("c1", "c2", "z") # changing the column names
```

  - Extraction of elements

```
> dat[2,3]         # element of row 2, column 3
> dat[2, "z"]      # same thing, but using the column name
> dat[1,]          # first row
> dat[,2]          # 2nd column
> dat[2:4,c(1,3)]  # subset, 2-4 rows and 1 & 3 columns
> dat$c1           # extracting the c1 column by name
```

  - Attach/detach
    Frequently, you need to access the columns of dataframes.
    `attach()` will make the column names visible temporarily.

```
> attach(dat)
> newVect <- c1 + c2   # exactly same as newVect <- dat$c1 + dat$c2
> z <- c1 * c2         # Note dat$z is not changed
> dat$z <- c1 * c2     # This changes dat$z.
> dat$modded <- z + c2 # This will add a new column with name "modded"
> detach(dat)          # Stop the attach
> c1
```

  - Manipulating data frames
    `cbind()`: column bind, combine data frames or vectors by columns.
    `rbind()`: row bind

```
> dim(dat)              # shows the number of columns and rows
> a <- c(13,14,15,16)
> dat <- cbind(dat, a)  # add 4-th column
> dat
> rbind(dat, dat[3:4,]) # extract 3-4-th rows and attach it at the end
> dat[dat[,1] > 2,]     # select the rows, whose 1st column > 2
```

    For the comparison, you can use `>`, `<`, `>=`, `<=`, `==`, `!=`.
    Also you can use & (and), | (or), ! (not) to make logical conditions.

  - Getting information about variables/objects

```
> dim(dat)    # dimenion of the object
> ncol(dat)   # number of columns
> nrow(dat)   # number of rows
> length(a)   # length of a vector
```

- Importing data
  If you have data stored in some spread sheet format, export the data in tab delimited text format (or any other decent text formats, such as comma separated text, works).

  ```
  > dat.in  <- read.table("data.txt", header=T, sep="\t")
  ```

  - Use "header=T" if the 1st row of the data in text is column names.
  - Use "header=F" if not. This will create a data.frame.
  - You may need to specify the full path:

    ```
    > dat.in <- read.table("/Users/naoki/doc/analysis/data.txt", header=T)
    ```

    Or use `setwd()` to set the current working directory or the R process.

    ```
    > getwd()                           # print out the current working directory
    > setwd("/Users/naoki/doc/analysis/")
    > getwd()
    > dat.in <- read.table("data.txt", header=T)
    ```

- Other types of objects

  - **matrices** or more generally **arrays**: multi-dimensional generalizations of vectors.
  - **factors**: handles categorical data (e.g. sex: female, male, or hermaphrodite).
  - **lists**: a general form of vector in which the various elements need not be of the same type.

- More advanced, but useful functions for data manipulations

  - `apply(), lapply(), sapply(), tapply()`
  - `is.na(), any(), all()`

5 Functions

R functions are used to provide users with programmed procedures. There are many convenient functions which come with R. For example, you have used c(), rbind(), dim(), read.table(). You can learn how to use functions by help("functionName"). If you don't know the function name, use help.search("keywords").

```
> help("read.table")
> help.search("ridge regression")
```

- How to write your functions
  Here is a simple example.

  ```
  standardize <- function(x)
  {
  # Inputs: a vector x
  # Outputs: the standardized version of x
          m<-mean(x)
          std<-sqrt(var(x))
          result<-(x - m)/std
          return(result)
  }
  ```

You can use this function in the same way you use the build-in functions.

```
> a <- c(1,4,6,10, 12)
> astd <- standardize(a)
```

So functions make it easier to do the same analysis many times.

To see the commands which make up the function, just type the name of functions WITHOUT parentheses.

```
> standardize
```

- Storing your custom functions (programs)

  Typing in the functions each time is tedious. Instead, you can type the useful custom functions in a plain text file. You keep accumulating your small useful programs in this file and analysis become quicker. You can read in all of your functions next time you need it by source(). In this way, you can automate all of the analysis, and distribute your analysis programs (functions) for other people to do the same analysis.

  ```
  > source("myCmd.r")    # read in the text file, and execute the commands
  ```

- Iteration

  The R commands, for() or while(), are used for iteration. Here is an example of using for() inside a function: Watterson's coefficient is $a_m = \sum_{i=1}^{m-1} \frac{1}{i}$.

  ```
  wattersonCoef <- function(m) {
    tempSum <- 0
    for (i in 1:(m-1)) {
      tempSum <- tempSum + 1 / i
    }
    return(tempSum)
  }

  ### same thing using while()
  wattersonCoefAlt <- function (m) {
    tempSum <- 0
    cntr <- 1

    while (cntr <= m-1) {
      tempSum <- tempSum + 1 / cntr
      cntr <- cntr + 1
    }
    return(tempSum)

  }
  ```

- Conditional

```
absVal <- function (val) {
  if (length(val) > 1) {
    # Note that val should be a single value, not a vector
    return(NA)
  } else if (val < 0) {
    return (- val)
  } else {
    return (val)
  }
}
```

- Random Number Generator

```
# returns 100 rand numbers following binominal distn of 40 trials w/ p = 0.2
> binom <- rbinom(n = 100, size = 40, prob = 0.2)
> hist(binom)
> mean(binom)
> var(binom)    # 40 * 0.2 * (1-0.2)
## normal distribution
> norm <- rnorm(500, mean=0, sd=1)
## poisson distribution
> poi <- rpois(n=1000, lambda=4)
> hist(poi)
> mean(poi)
> var(poi)     # mean = var = lambda
## geometric distribution
> geo <- rgeom(n=1000, prob=0.1)
```

6 Plot

Flexible graphics capability is the strengths of R.

- A histogram can be made by typing:

```
> x <- rnorm(1000)
> hist(x)
```

- Scatter plot can be made by

```
> x <- 0:10
> y <- log(x)
> plot(x,y)
```

You can add a straight line of y = a + b x by abline(a, b).

```
> abline(0.5, 0.2)    # y-intercept = 0.5,  slope = 0.2
```

You can add more points by:

```
> points (x, x * 0.2, pch=4)  # pch is points "character", i.e., symbol to use
```

You can connect the points by lines:

```
> lines (x, y)
> lines (x, x * 0.2, lty = 2) # lty is line type
```

You can see the pre-set points characters:

```
> example(points)
```

- To save this plot to a postscript file called, graph1.ps, type

```
> dev.print(postscript,file="graph1.ps")
> dev.print(pdf,file="graph1.pdf")   # to make pdf
```

The file, graph1.ps, is saved in the directory in which R is invoked. Other graphical devices (i.e. pdf, jpeg, xfig, etc.) can be specified. To see other graphical devices, see `help(Devices)`.

- A matrix of scatterplots

`pairs(X)` produces a pairwise scatterplot matrix of the variables defined by the columns of X. Every column of X is plotted against every other column of X.

```
> iris[1:5,]   # Anderson's iris data set
> pairs(iris[1:4], main = "Anderson's Iris Data -- 3 species",
+       pch = 21, bg = c("red", "green3", "blue")[unclass(iris$Species)])
> help(iris)
> help(pairs)  # Try the examples.
```

- multiple plots per page

```
> par(mfrow = c(2,2))
> for (i in 1:4) { hist(iris[,i], xlab=names(iris)[i], main=i)}
> par(mfrow=c(1,1))     # setting the parameter back to the default
```

`par(mfcol=c(2,2))` is similar, but figures are filled by *column*.

- Bar plots:

```
> VADeaths                          # death rates per 1000 in Virginian in 1940
> tVADeaths <- t(VADeaths)[,5:1]  # transpose, and change order of columns
> mp <- barplot(tVADeaths, beside=T, legend=colnames(VADeaths), ylim=c(0,130))
> fakeSE <- 2 * sqrt (1000 * tVADeaths/100)  # making a fake error bar
> mp                                # contains x-coordinates
> segments (mp, tVADeaths, mp, tVADeaths+fakeSE, lwd=1.5)
```

You can easily make a customized "wrapper" function to do all of this if you repeatedly use the same plotting procedure.

- Advanced graphics:
  See R Graph Gallery: http://addictedtor.free.fr/graphiques/ for more advanced graphics. The web page contains the example codes to produce the fancy figures, so you can learn through immitation.

7 Statistics

R is equipped with a cutting edge statistical tools. If your tool is not found, you can easily make your custom function for the new statistics, and people from all over the world will appreciate your effort (Open Source rules!).

- t-test

```
> x1 <- c(3,2,5,1,4)
> x2 <- c(6,4,5,7,2)
> t.test(x2, mu=10, alternative="less")  # 1-way 1 sample t-test
> t.test(x1, x2)                        # 2 sample t-test
> help(t.test)
> wilcox.test(x1,x2, paired=T)          # matched-pairs Wilcoxon test
```

- Linear Regression

```
> plot(iris$Petal.Width, iris$Petal.Length)
> fit.i1 <- lm(Petal.Length ~ Petal.Width, data=iris)
> coef(fit.i1)
> summary(fit.i1)
> abline(coef(fit.i1))
> pred1 <- predict(fit.i1, interval="confidence") # narrow confidence band
> pred2 <- predict(fit.i1, interval="prediction") # wide prediction band
> petW <- iris$Petal.Width
> matlines(petW, pred1, col="red")
> matlines(petW, pred2, col="blue")
> plot(fitted(fit.i1), resid(fit.i1))
```

- ANOVA

```
> av <- lm(Petal.Length ~ Species, data=iris)  # 1-way Anova
> anova(av)
> summary(av)
> boxplot(Petal.Length ~ Species, data=iris)
> with(iris, stripchart(Petal.Length ~ Species, "jitter", vert=T))
```

  Make sure the independent variable is a **factor**. If it is numerical (e.g. 1, 2, 3), convert the column to factors:

```
> dat$x <- factor(dat$x)
```

  Without the conversion, you are doing linear regression.

  The traditional 1-way ANOVA assumes the equal variance for all treatment groups. The following test (Welch 1951) relaxes this assumption:

```
oneway.test(Petal.Length ~ Species, data=iris)
```

- Model formulae and linear models

  - Notation:

    - y, x1, and x2 are numeric variables.
    - A, B, and C are factors.
  - `y ~ x` or `y ~ x + 1`
    Simple linear regression of *y* on *x* with intercept: $y = a_1 \cdot x + a_0$.
  - `y ~ x - 1`
    Simple linear regression without intercept; i.e., force the line to go through the origin: $y = a_1 \cdot x$
  - `log(y) ~ x1 + x2`
    Multiple regression with log-transformed *y*. Implicitly, the intercept is included.
  - `y ~ x + I(x^2)`
    Polynomial regresion: $y = a_0 + a_1 \cdot x + a_2 \cdot x^2$
  - `y ~ A + B`
    Two-way ANOVA without interaction term
  - `y ~ A + B + A:B` or `y ~ A*B`
    Two-way ANOVA with interaction term
  - `y ~ A*B*C` or `y ~ A + B + C + A:B + B:C + A:C + A:B:C`
    Three-way ANOVA with all possible interactions
  - `y ~ A*B*C - A:B:C`
    Three-way ANOVA without the 3-way interaction (A:B:C) term.
  - `y ~ A + x`
    Single classification ANCOVA with class determined by A and with covariate *x*
  - `y ~ A * x`
    ANCOVA with class determined by A. Slopes are obtained for each class. *x*

  ANCOVA example

```
> fit.cov1 <- lm(Petal.Length ~ Sepal.Length + Species, data=iris)
> anova(fit.cov1)
> fit.cov2 <- lm(Petal.Length ~ Sepal.Length * Species, data=iris)
> anova(fit.cov2)
> summary(fit.cov2)
```

- Other common statistical functions

  glm Generalized linear model, includes logistic regression

  lme Linear Mixed-effects Models. It is a part of package **nlme** "non-linear mixed effects models". You need to load the package by:

```
> library(nlme)
> help(lme)
```

  Surv Survival analysis. Need to load survival package

8 Examples

8.1 coalescence

Note that state *i* is the period when there are *i* lineages in the genealogy. Number of generations until the end of the state *i* is geometric distribution with $p = \binom{i}{2}\frac{1}{2N}$.

```
durationOfStateI <- function(popSize, state) {
  prob <- choose(state, 2) / (2 * popSize)   # prob. of coalescence
  generations <- rgeom(1, prob) + 1
  return(generations)
}

makeGenealogy <- function (popSize, sampleSize) {
  result <- numeric(0)
  for (i in 2:sampleSize) {
    ti <- durationOfStateI(popSize, i)
    result <- c(result, ti)
  }
  return(result)
}

### The following is a better way of doing the same thing
makeGenealogy <- function (popSize, sampleSize) {
  return(sapply(2:sampleSize, durationOfStateI, popSize=popSize))
}

totalTreeLength <- function(popSize, sampleSize) {
  genealogy <- makeGenealogy(popSize, sampleSize)
  brlen <- sum(genealogy * 2:sampleSize)
  return(brlen)
}

numSegSites <- function(popSize, sampleSize, mu) {
  totBrLen <- totalTreeLength(popSize, sampleSize)
  numMut <- rpois(1, totBrLen * mu)  # poisson distribution
  return(numMut)
}

simCoal <- function(popSize, sampleSize, mu, trials){
  result <- numeric(0)
  for (i in 1:trials) {
    mut <- numSegSites(popSize, sampleSize, mu)
    result <- c(result, mut)
  }
  return(result)
}

> simDat <- simCoal(popSize=1000, sampleSize=10, mu=0.001, trials=2000)
> 4 * 1000 * 0.0001 * wattersonCoef(10)
```

Remember $S = \theta a_m$, where S is number of segregating sites, $\theta = 4N\mu$, and a_m is Watterson's coefficient?

8.2 Selection at 1 locus

```
### freq(A1) = p, freq(A2) = 1-p
```

```
### w11 = 1, w12 = 1-h * s, w22 = 1 - s

## population mean fitness
wbar <- function (p,s,h) {
  return(1 - 2 * p * (1-p) * h * s - (1-p)^2 * s)
}

## Changes in allele frequency after 1 generation
delta.p <- function (p, s, h) {
  q <- 1-p
  mean.w <- wbar(p, s, h)
  result <- p * q * s * (p * h + q * (1-h))/mean.w
  return  (result)
}

recursiveAlleleFreq <- function (init.p, s, h, generations) {
  result <- c(init.p)
  curFreq <-init.p
  for (i in 1:generations) {
    nextFreq <- curFreq + delta.p(curFreq, s, h)
    result <- c(result, nextFreq)
    curFreq <- nextFreq
  }
  return(result)
}

> inip <- 0.01
> sel <- 0.1
> num.gen <- 1200
>
> domi <- recursiveAlleleFreq(inip, sel, 0, num.gen)   # A1 domi
> rec <- recursiveAlleleFreq(inip, sel, 1, num.gen)    # A1 recessive
> add <- recursiveAlleleFreq(inip, sel, 0.5, num.gen)  # Additive
>
> plot(0:num.gen, domi, type="l", xlab="generations", ylab="p",
     cex.lab=1.4, cex.axis=1.3, main="s=0.1, init(p) = 0.01")
> lines(0:num.gen, rec)
```

8.3 Genetic Drift

You can download the source code from http://www.faculty.uaf.edu/ffnt/teaching/popgen/R-tutorial/gen.drift.r .

```
> source("gen.drift.r")
>  plot.gen.drift(popSize=10, initFreq=0.2, num.generations=100, num.trials=10)
```

This will start the simulation with N=10 and initial allele freq=0.2. It will repeat the simulation for 10 times (num.trials). In addition to the plot, it will print out number of trials that the allele went fixation and extinction, and calculate the fixation probability. Remember fixation probability = p (the initial allele frequency). Confirm

this with various setting of popSize and initFreq. Notice that population become homozygous immediately with N=10, but it takes longer time when N is larger. So you may need to increase the number of generations (e.g. num.generations=1000).

9 Books

- *An Introduction to R* by Venbles, et al. It can be downloaded from CRAN.
- *Introductory Statistics with R* by Peter Dalgaard
- *Modern Applied Statistics with S* by Venables and Ripley
- *Mixed-Effects Models in S and S-Plus* by Pinheiro and Bates
- *Linear Mixed Models for Longitudinal Data* by Verbeke and Molenberghs
- *S Programming* by Venables and Ripley
- *Statistical Models in S* by Chambers and Hastie