Appendix B
Computations

We briefly describe two computer packages that can be used for all the computations carried out in the text. We recommend that students familiarize themselves with at least one of these. The description of R is quite complete, at least for the computations based on material in this text, whereas another reference is required to learn Minitab.

B.1 Using R

R is a free statistical software package that can be downloaded (http://cran.r-project.org/) and installed on your computer. A free manual is also available at this site.

Once you have R installed on your system, you can invoke it by clicking on the relevant icon (or, on Unix systems, simply typing "R"). You then see a window, called the *R Console* that contains some text and a prompt ‘ > ’ after which you type commands. Commands are separated by new lines or ‘ ; ’. Output from commands is also displayed in this window, unless it is purposefully directed elsewhere. To quit R, type `q()` after the prompt. To learn about anything in R, a convenient resource is to use Help on the menu bar available at the top of the R window. Alternatively, type `?name` after the prompt (and press enter) to display information about `name`, e.g., `?q` brings up a page with information about the terminate command `q`.

Basic Operations and Functions

A basic command evaluates an expression, such as

```
> 2+3
[1] 5
```

which adds 2 and 3 and produces the answer 5. Alternatively, we could *assign* the value of the expression to a variable such as

```
> a <- 2
```

where `<-` (less than followed by minus) assigns the value 2 to a variable called `a`. Alternatively `=` can be used for assignment as in `a = 2` but we will use `<-`. We can

then verify this assignment by simply typing `a` and hitting return, which causes the value of `a` to be printed.

```
> a
[1] 2
```

Note that R is case sensitive, so `A` would be a different variable than `a`. There are some restrictions in choosing names for variables and vectors, but you won't go wrong if you always start the name with a letter.

We can assign the values in a vector using the concatenate function `c()` such as

```
> b <- c(1,1,1,3,4,5)
> b
[1] 1 1 1 3 4 5
```

which creates a vector called `b` with six values in it. We can access the *i*-th entry in a vector `b` by referring to it as `b[i]`. For example,

```
> b[3]
[1] 1
```

prints the third entry in `b`, namely, `1`. Alternatively, we can use the `scan` command to input data. For example,

```
> b <- scan()
1:  1 1 1 3 4 5
7:
Read 6 items
> b
[1] 1 1 1 3 4 5
```

accomplishes the same assignment. Note that with the `scan` command, we simply type in the data and terminate data input by entering a blank line. We can also use `scan` to read data in from a file, and we refer the reader to `?scan` for this.

Sometimes we want vectors whose entries are in some pattern. We can often use the `rep` function for this. For example, `x <- rep(1,20)` creates a vector of 20 ones. More complicated patterns can be obtained, and we refer the reader to `?rep` for this.

Basic arithmetic can be carried out on variables and vectors using + (addition), - (subtraction), * (multiplication), / (division), and ^ (exponentiation). These operations are carried out componentwise. For example, we could multiply each component of `b` by itself via

```
> b*b
[1] 1 1 1 9 16 25
```

or multiply each element of `b` by 2 as in

```
> 2*b
[1] 2 2 2 6 8 10
```

which accomplishes this.

There are various functions available in R, such as `abs(x)` (calculates the absolute value of `x`), `log(x)` (calculates the natural logarithm of `x`), `exp(x)` (calculates *e* raised to the power `x`), `sin(x)`, `cos(x)`, `tan(x)` (which calculate the trigonometric functions), `sqrt(x)` (which calculates the square root of `x`), `ceiling(x)`, and `floor(x)` (calculate the ceiling and floor of `x`). When such a function is applied to a vector `x`, it returns a vector of the same length, with the function applied to each element of the original vector. There are numerous special functions available in R, but two important ones are `gamma(x)`, which returns the gamma function applied to `x`, and `lgamma(x)`, which returns the natural logarithm of the gamma function.

There are also functions that return a single value when applied to a vector. For example, `min(x)` and `max(x)` return respectively the smallest and largest elements in `x`; `length(x)` gives the number of elements in `x`; and `sum(x)` gives the sum of the values in `x`.

R also operates on logical quantities `TRUE` (or `T` for true) and `FALSE` (or `F` for false). Logical values are generated by conditions that are either true or false. For example,

```
> a <- c(-3,4,2,-1,-5)
> b <- a>0
> b
[1] FALSE TRUE TRUE FALSE FALSE
```

compares each element of the vector `a` with 0, returning `TRUE` when it is greater than 0 and `FALSE` otherwise, and these logical values are stored in the vector `b`. The following logical operators can be used: `<`, `<=`, `>=`, `>`, `==` (for equality), `!=` (for inequality) as well as `&` (for conjunction), `|` (for disjunction) and `!` (for negation). For example, if we create a logical vector `c` as follows:

```
> c <- c(T,T,T,T,T)
> b&c
[1] FALSE TRUE TRUE FALSE FALSE
> b|c
[1] TRUE TRUE TRUE TRUE TRUE
```

then an element of `b&c` is `TRUE` when both corresponding elements of `b` and `c` are TRUE, while an element of `b|c` is `TRUE` when at least one of the corresponding elements of `b` and `c` is TRUE.

Sometimes we may have variables that take character values. While it is always possible to code these values as numbers, there is no need to do this, as R can also handle character-valued variables. For example, the commands

```
> A <- c('a','b')
> A
[1] "a" "b"
```

create a character vector `A`, containing two values `a` and `b`, and then we print out this vector. Note that we included the character values in single quotes when doing the assignment.

Sometimes data values are missing and so are listed as `NA` (not available). Operations on missing values create missing values. Also, an impossible operation, such as 0/0, produces `NaN` (not a number).

Various objects can be created during an R session. To see those created so far in your session, use the command `ls()`. You can remove any objects in your workspace using the `rm` command. For example, `rm(x)` removes the vector `x`.

Probability Functions

R has a number of built-in functions for evaluation of the cdf, the inverse cdf, the density or probability function, and generating random samples for the common distributions we encounter in probability and statistics. These are distinguished by prefix and base distribution names. Some of the distribution names are given in the following table.

Distribution	R name and arguments	Distribution	R name and arguments
beta	beta(· ,a,b)	hypergeometric	hyper(· ,N,M,n)
binomial	binom(· ,n,p)	negative binomial	nbinom(· ,k,p)
chi-squared	chisq(· ,df)	normal	norm(· ,mu,sigma)
exponential	exp(· ,lambda)	Poisson	pois(· ,lambda)
F	f(· ,df1,df2)	t	t(· ,df)
gamma	gamma(· ,alpha,lambda)	uniform	unif(· ,min,max)
geometric	geom(· ,p)		

As usual, one has to be careful with the gamma distribution. The safest path is to include another argument with the distribution to indicate whether or not `lambda` is a rate parameter (density is $(\Gamma(\alpha))^{-1}\lambda^{\alpha}x^{\alpha-1}e^{-\lambda x}$) or a scale parameter (density is $(\Gamma(\alpha))^{-1}\lambda^{-\alpha}x^{\alpha-1}e^{-x/\lambda}$). So `gamma(·,alpha,rate=lambda)` indicates that `lambda` is a rate parameter, and `gamma(·,alpha,scale=lambda)` indicates that it is a scale parameter.

The argument given by · is specified according to what purpose the command using the distribution name has. To obtain the cdf of a distribution, precede the name by `p`, and then · is the value at which you want to evaluate the cdf. To obtain the inverse cdf of a distribution, precede the name by `q`, and then · is the value at which you want to evaluate the inverse cdf. To obtain the density or probability function, precede the name by `d`, and then · is the value at which you want to evaluate the density or probability function. To obtain random samples, precede the name by `r`, and then · is the size of the random sample you want to generate.

For example,

```
> x <- rnorm(4,1,2)
> x
[1] -0.2462307 2.7992913 4.7541085 3.3169241
```

generates a sample of 4 from the $N(1, 2^2)$ distribution and assigns this to the vector `x`. The command

```
> dnorm(3.2,2,.5)
[1] 0.04478906
```

evaluates the $N(2, .25)$ pdf at 3.2, while

```
> pnorm(3.2,2,.5)
[1] 0.9918025
```

evaluates the $N(2, 0.25)$ cdf at 3.2, and

```
> qnorm(.025,2,.5)
[1] 1.020018
```

gives the 0.025 quantile of the $N(2, 0.25)$ distribution.

If we have data stored in a vector `x`, then we can sample values from `x`, with or without replacement, using the `sample` function. For example, `sample(x,n,T)` will generate a sample of `n` from `x` with replacement, while `sample(x,n,F)` will generate a sample of `n` from `x` without replacement (note `n` must be no greater than `length(x)` in the latter case).

Sometimes it is convenient to be able to repeat a simulation so the same random values are generated. For this, you can use the `set.seed` command. For example, `set.seed(12345)` establishes the seed as `12345`.

Tabulating Data

The `table` command is available for tabulating data. For example, `table(x)` returns a table containing a list of the unique values found in `x` and their frequency of occurrence in `x`. This table can be assigned to a variable via

```
y <- table(x)
```

for further analysis (see The Chi-Squared Test section on the next page).

If `x` and `y` are vectors of the same length, then `table(x,y)` produces a cross-tabulation, i.e., counts the number of times each possible value of (x, y) is obtained, where x can be any of the values taken in `x` and y can be any of the values taken in `y`.

Plotting Data

R has a number of commands available for plotting data. For example, suppose we have a sample of size n stored in the vector `x`.

The command `hist(x)` will provide a frequency histogram of the data where the cutpoints are chosen automatically by R. We can add optional arguments to `hist`. The following are some of the arguments available.

`breaks` - A vector containing the cutpoints.

`freq` - A logical variable; when `freq=T` (the default), a frequency histogram is obtained, and when `freq=F`, a density histogram is obtained.

For example, `hist(x,breaks=c(-10,-5,-2,0,2,5,10),freq=F)` will plot a density histogram with cutpoints -10, -5, -2, 0, 2, 5, 10, where we have been careful to ensure that `min(x)` > -10 and `max(x)` < 10.

If y is another vector of the same length as x, then we can produce a scatter plot of y against x via the command `plot(x,y)`. The command `plot(x,y,type="l")` provides a scatter plot of y against x, but now the points are joined by lines. The command `plot(x)` plots the values in x against their index. The `plot(ecdf(x))` command plots the empirical cdf of the data in x.

A boxplot of the data in x is obtained via the `boxplot(x)` command. Side-by-side boxplots of the data in x, y, z, etc., can be obtained via `boxplot(x,y,z)`.

A normal probability plot of the values in x can be obtained using the command `qqnorm(x)`.

A barplot can be obtained using the `barplot` command. For example,

```
> h <- c(1,2,3)
> barplot(h)
```

produces a barplot with 3 bars of heights 1, 2, and 3.

There are many other aspects to plotting in R that allow the user considerable control over the look of plots. We refer the reader to the manual for more discussion of these.

Statistical Inference

R has a powerful approach to fitting and making inference about models. Models are specified by the symbol ~. We do not discuss this fully here but only indicate how to use this to handle the simple and multiple linear regression models (where the response and the predictors are all quantitative), the one and two-factor models (where the response is quantitative but the predictors are categorical), and the logistic regression model (where the response is categorical but the predictors are quantitative). Suppose, then, that we have a vector y containing the response values.

Basic Statistics

The function `mean(y)` returns the mean of the values in y, `var(y)` returns the sample variance of the values in y, and `sd(y)` gives the sample standard deviation. The command `median(y)`returns the median of y, while `quantile(y,p)` returns the sample quantiles as specified in the vector of probabilities p. For example, `quantile(y,c(.25,.5,.75))` returns the median and the first and third quantiles. The function `sort(y)` returns a vector with the values in y sorted from smallest to largest, and `rank(y)` gives the ranks of the values in y.

The t-Test

For the data in y, we can use the command

```
> t.test(y,mu=1,alternative="two.sided",conf.level=.95)
```

to carry out a t-test. This computes the P-value for testing $H_0 : \mu = 1$ and forms a 0.95-confidence interval for μ.

The Chi-Squared Test

Suppose y contains a vector of counts for k cells and prob contains hypothesized probabilities for these cells. Then the command

```
> chisq.test(y,p=prob)
```

carries out the chi-squared test to assess this hypothesis. Note that `y` could also correspond to a one-dimensional table.

If `x` and `y` are two vectors of the same length, then `chisq.test(x,y)` carries out a chi-squared test for independence on the table formed by cross-tabulating the entries in `x` and `y`. If we first create this cross-tabulation in the table `t` using the `table` function, then `chisq.test(t)` carries out this test.

Simple Linear Regression

Suppose we have a single predictor with values in the vector `x`. The simple linear regression model $E(y \mid x) = \beta_1 + \beta_2 x$ is then specified in R by `y~x`. We refer to `y~x` as a model formula, and read this as "`y` is modelled as a linear model involving `x`." To carry out the fitting (which we have done here for a specific set of data), we use the fitting linear models command `lm`, as follows. The command

```
> regexamp <- lm(y~x)
```

carries out the computations for fitting and inference about this model and assigns the result to a structure called `regexamp`. Any other valid name could have been used for this structure. We can now use various R functions to pick off various items of interest. For example,

```
> summary(regexamp)
Call:
lm(formula = y~x)
Residuals:
  Min      1Q    Median 3Q     Max
-4.2211 -2.1163 0.3248 1.7255 4.3323
Coefficients:
 Estimate          Std.  Error  t value Pr(>|t|)
(Intercept) 6.5228   1.2176     5.357   4.31e-05 ***
    x       1.7531   0.1016    17.248   1.22e-12 ***
--
Signif.  codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.'
0.1 ' ' 1
Residual standard error:  2.621 on 18 degrees of freedom
Multiple R-squared:  0.9429, Adjusted R-squared:  0.9398

F-statistic:  297.5 on 1 and 18 DF, p-value:  1.219e-12
```

uses the `summary` function to give us all the information we need. For example, the fitted line is given by $6.5228 + 1.7531x$. The test of $H_0 : \beta_2 = 0$ has a P-value of 1.22×10^{-12}, so we have strong evidence against H_0. Furthermore, the R^2 is given by 94.29%. Individual items can be accessed via various R functions and we refer the reader to `?lm` for this.

Multiple Linear Regression

If we have two quantitative predictors in the vectors `x1` and `x2`, then we can proceed just as with simple linear regression to fit the linear regression model $E(y \mid x_1, x_2) = \beta_1 + \beta_2 x_1 + \beta_2 x_2$. For example, the commands

```
> regex <- lm(y~x1+x2)
> summary(regex)
```

fit the above linear model, assign the results of this to the structure `regex`, and then the `summary` function prints out (suppressed here) all the relevant quantities. We read `y~x1+x2` as, "`y` is modelled as a linear model involving `x1` *and* `x2`." In particular, the F-statistic, and its associated P-value, is obtained for testing $H_0 : \beta_2 = \beta_3 = 0$.

This generalizes immediately to linear regression models with k quantitative predictors $x_1, ..., x_k$. Furthermore, suppose we want to test that the model only involves $x_1, ..., x_l$ for $l < k$. We use `lm` to fit the model for all k predictors, assign this to `regex`, and also use `lm` to fit the model that only involves l predictors and assign this to `regex1`. Then the command `anova(regex,regex1)` will output the F-statistics, and its P-value, for testing $H_0 : \beta_{l+1} = \cdots = \beta_k = 0$.

One- and Two-Factor ANOVA

Suppose now that A denotes a categorical predictor taking two levels a_1 and a_2. Note that the values of A may be character in value rather than numeric, e.g., `x` is a character vector containing the values `a1` and `a2`, used to denote at which level the corresponding value of `y` was observed. In either case, we need to make this into a factor `A`, via the command

```
> A <- factor(x)
```

so that `A` can be used in the analysis. Then the command

```
> aov(y~A)
```

produces the one-way ANOVA table. Of course, `aov` also handles factors with more than two levels. To produce the cell means, use the command `tapply(y,A,mean)`.

Suppose there is a second factor B taking 5 levels $b_1, \ldots, b_5$. If this is the factor `B` in R, then the command

```
> aov(y~A+B+A:B)
```

produces the two-way ANOVA for testing for interactions between factors `A` and `B`. To produce the cell means, use the command `tapply(y,list(A,B),mean)`. The command `aov(y~A+B)` produces the ANOVA table, assuming that there are no interactions.

Logistic Regression

Suppose we have binary data stored in the vector `y`, and `x` contains the corresponding values of a quantitative predictor. Then we can use the generalized linear model command `glm` to fit the logistic regression model $P(Y = 1 \mid x) = \exp\{\beta_1 + \beta_2 x\}/(1 + \exp\{\beta_1 + \beta_2 x\})$. The commands

```
> logreg <- glm(y~x,family=binomial)
> summary(logreg)
```

fit the logistic regression model, assign the results to `logreg` and then the `summary` command outputs this material. This gives us the estimates of the β_i, their standard errors, and P-values for testing that the $\beta_i = 0$.

Control Statements and R Programs

A basic control statement is of the form `if (expr1) expr2 else expr3`, where `expr1` takes a logical value, `expr2` is executed if `expr1` is `T`, and `expr3` is executed if `expr1` is `F`. For example, if `x` is a variable taking value -2, then

```
> if (x<0) {y <- -1} else {y <- 1}
```

results in `y` being assigned the value -1. Note that the `else` part of the statement can be dropped.

The command `for (name in expr) expr2` executes `expr2` for each value of `name` in `expr1`. For example,

```
> for (i in 1:10) print(i)
```

prints the value of the variable `i` as `i` is sequentially assigned values in $\{1, 2, \ldots, 10\}$. Note that `m:n` is a shorthand for the sequence $(m, m+1, \ldots, n)$ in R. As another example,

```
> for (i in 1:20) y[i] <- 2^i
```

creates a vector `y` with 20 entries, where the i-th element of `y` equals 2^i.

The `break` terminates a loop, perhaps based on some condition holding, while `next` halts the processing of the current iteration and advances the looping index. Both `break` and `next` apply only to the innermost of nested loops.

Commands in R can be grouped by placing them within braces `{expr1; expr2; ...}`. The commands within the braces are executed as a unit. For example,

```
> for (i in 1:20) {print(i); y[i] <- 2^i}; print(y[i])}
```

causes `i` to be printed, `y[i]` to be assigned, and `y[i]` to be printed, all within a `for` loop.

Often when a computation is complicated, such as one that involves looping, it is better to put all the R commands in a single file and then execute the file in batch mode. For example, suppose you have a file `prog.R` containing R code. Then the command `source("pathname/prog.R")` causes all the commands in the file to be executed.

It is often convenient to put comments in R programs to explain what the lines of code are doing. A comment line is preceded by # and of course it is not executed.

User-Defined Functions

R also allows user-defined functions. The syntax of a function definition is as follows.

function name <- **function**(*arguments*) {
    function body;
    return(*return value*);
}

For example, the following function computes the sample coefficient of variation of the data `x`.

```
coef_var <- function(x) {
    result <- sd(x)/mean(x);
    return(result);
}
```

Then if we want to subsequently compute the coefficient of variation of data `y` we simply type `coef_var(y)`.

Arrays and Lists

A vector of length m can also be thought of as one-dimensional array of length m. R can handle multidimensional *arrays*, e.g., $m \times n$, $m \times n \times p$ arrays, etc. If `a` is a three-dimensional array, then `a[i,j,k]` refers to the entry in the (i, j, k)-th position of the array. There are various operations that can be carried out on arrays and we refer the reader to the manual for these. Later in this manual, we will discuss the special case of two-dimensional arrays, which are also known as matrices. For now, we just think of arrays as an object in which we store data.

A very general data structure in R is given by a *list*. A list is similar to an array with several important differences.

1. Any entry in an array is referred to by its index. But any entry in a list may be referred to by a character name. For example, the fitted regression coefficients are referred to by `regex$coefficients` after fitting the linear model `regex <- lm(y ~ x1 + x2)`. The dollar mark ($) is the entry reference operator, that is, `varname$entname` indicates the "entname" entry in the list "varname."

2. While an array stores only the same type of data, a list can store any R objects. For example, the `coefficients` entry in a linear regression object is a numeric vector, and the `model` entry is a list.

3. The reference operators are different: `arr[i]` refers to the i-th entry in the array `arr`, and `lst[[i]]` refers to the i-th entry in the list `lst`. Note that i can be the entry name, i.e., `lst$entname` and `lst[['entname']]` refer to the same data.

Examples

We now consider some examples relevant to particular sections or examples in the main text. To run any of these codes, you first have to define the functions. To do this, load

the code using the `source` command. Arguments to the functions then need to be specified. Note that lines in the listings may be broken unnaturally and continue on the following line.

EXAMPLE B.1.1 *Bootstrapping in Example 6.4.2*
The following R code generates bootstrap samples and calculates the median of each of these samples. To run this code, type `y <- bootstrap_median(m,x)`, where `m` is the number of bootstrap samples, `x` contains the original sample, and the medians of the resamples are stored in `y`. The statistic to be bootstrapped can be changed by substituting for `median` in the code.

```
#   Example B.2.1
#   function name: bootstrap_median
#   parameters:
#       m   resample size
#       x   original data
#   return value:
#       a vector of resampled medians
#   description: resamples and stores its median
  bootstrap_median <- function(m,x) {
      n <- length(x);
      result <- rep(0,m);
      for(i in 1:m) result[i] <- median(sample(x,n,T));
   return(result);
  }
```

■

EXAMPLE B.1.2 *Sampling from the Posterior in Example 7.3.1*
The following R code generates a sample of from the joint posterior in Example 7.3.1. To run a simulation, type

```
  post <- post_normal(m,x,alpha0,beta0,mu0,tau0square)
```

where `m` is the Monte Carlo sample size and the remaining arguments are the hyperparameters of the prior. The result is a list called (in this case) `post`, where `post$mu` and `post$sigmasq` contain the generated values of μ and σ^2, respectively. For example,

```
> x <- c(11.6714, 1.8957, 2.1228, 2.1286, 1.0751, 8.1631,
1.8236, 4.0362, 6.8513, 7.6461, 1.9020, 7.4899, 4.9233,
8.3223, 7.9486);
> post <- post_normal(10**4,x,2,1,4,2)
> z <- sqrt(post$sigmasq)/post$mu
```

runs a simulation as in Example 7.3.1, with $N = 10^4$.

```
#   Example B.2.2
#   function name: post_normal
#   parameters:
#       m       sample size
```

```
#       x       data
#       alpha0  shape parameter for 1/sigma^2
#       beta0   rate parameter for 1/sigma^2
#       mu0     location parameter for mu
#       tau0square  variance ratio parameter for mu
#   returned values:
#       mu      sampled mu
#       sigmasq sampled sigmasquare
#   description: samples from the posterior distribution
#   in Example 7.3.1
#
post_normal<-function(m,x,alpha0,beta0,mu0,tau0square){
# set the length of the data
n <- length(x);
# the shape and rate parameters of the posterior dist.
# alpha_x = first parameter of the gamma dist.
#         = (alpha0 + n/2)
alpha_x <- alpha0 + n/2
# beta_x  = the rate parameter of the gamma dist.
beta_x  <- beta0 + (n-1)/2 * var(x) + n*(mean(x)-mu0)**2/
           2/(1+n*tau0square);
# mu_x    = the mean parameter of the normal dist.
mu_x    <- (mu0/tau0square+n*mean(x))/(n+1/tau0square);
# tausq_x = the variance ratio parameter of the normal
#           distribution
tausq_x <- 1/(n+1/tau0square);
# initialize the result
result  <- list();
result$sigmasq <- 1/rgamma(m,alpha_x,rate=beta_x);
result$mu      <- rnorm(m,mu_x,sqrt(tausq_x *
                        result$sigmasq));
return(result);
}
```

EXAMPLE B.1.3 *Calculating the Estimates and Standard Errors in Example 7.3.1*
Once we have a sample of values from the posterior distribution of ψ stored in `psi`, we can calculate the interval given by the mean value of psi plus or minus 3 standard deviations as a measure of the accuracy of the estimation.

```
# Example B.2.3
# set the data
x <- c(11.6714, 1.8957, 2.1228, 2.1286, 1.0751, 8.1631,
   1.8236, 4.0362, 6.8513, 7.6461, 1.9020, 7.4899,
   4.9233, 8.3223, 7.9486);
post <- post_normal(10**4,x,2,1,4,2);
# compute the coefficient of variation
```

```
psi <- sqrt(post$sigmasq)/post$mu;
psi_hat <- mean(psi <= .5);
psi_se  <- sqrt(psi_hat * (1-psi_hat))/sqrt(length(psi));
# the interval
cq <- 3
cat("The three times s.e. interval is ",
   "[",psi_hat-cq*psi_se, ", ", psi_hat+cq*psi_se,"]\n");
```

EXAMPLE B.1.4 *Using the Gibbs Sampler in Example 7.3.2*
To run this function, type

```
post<-gibbs_normal(m,x,alpha0,beta0,lambda,mu0,
                   tau0sq,burnin=0)
```

as this creates a list called `post`, where `post$mu` and `post$sigmasq` contain the generated values of μ and σ^2, respectively. Note that the `burnin` argument is set to a nonnegative integer and indicates that we wish to discard the first `burnin` values of μ and σ^2 and retain the last `m`. The default value is `burnin=0`.

```
# Example B.2.4
#   function name: gibbs_normal
#   parameters
#     m        the size of posterior sample
#     x        data
#     alpha0   shape parameter for 1/sigma^2
#     beta0    rate parameter for 1/sigma^2
#     lambda   degree of freedom of Student's t-dist.
#     mu0      location parameter for mu
#     tau0sq   scale parameter for mu
#     burnin   size of burn in. the default value is 0.
#
#   returnrd values
#       mu       sampled mu's
#       sigmasq  sampled sigma^2's
#   description: samples from the posterior in Ex. 7.3.2
#
gibbs_normal <- function(m,x,alpha0,beta0,lambda,mu0,
                         tau0sq,burnin=0) {
  # initialize the result
  result <- list();
  result$sigmasq <- result$mu <- rep(0,m);
  # set the initial parameter
  mu      <- mean(x);
  sigmasq <- var(x);
  n       <- length(x);

  # set parameters
```

```
  alpha_x <- n/2 + alpha0 + 1/2;
  # loop
  for(i in (1-burnin):m) {
   # update v_i's
   v      <- rgamma(n,(lambda+1)/2,rate=((x-mu)**2/
                          sigmasq/lambda+1)/2);
   # update sigma-square
   beta_x <-(sum(v*(x-mu)**2)/lambda+(mu-mu0)**2/
                                  tau0sq)/2+beta0;
   sigmasq<- 1/rgamma(1,alpha_x,rate=beta_x);
   # update mu
   r      <- 1/(sum(v)/lambda+1/tau0sq);
   mu     <- rnorm(1,r*(sum(v*x)/lambda+mu0/tau0sq),
                        sqrt(r*sigmasq));
   # burnin check
   if(i < 1) next;
   result$mu[i]      <- mu;
   result$sigmasq[i] <- sigmasq;
  }
  result$psi <- sqrt(result$sigmasq)/result$mu;
  return(result);
}
```

EXAMPLE B.1.5 *Batching in Example 7.3.2*

The following R code divides a series of data into batches and calculates the batch means. To run the code, type `y<-batching(k,x)` to place the consecutive batch means of size `k`, of the data in the vector `x`, in the vector `y`.

```
#   Example B.2.5
#   function name: batching
#   parameters:
#       k   size of each batch
#       x   data
#   return value:
#       an array of the averages of each batch
#   description: this function separates the data x into
#   floor(length(x)/k) batches and returns the array of
#   the averages of each batch
batching <- function(k,x) {
    m <- floor(length(x)/k);
    result <- rep(0,m);
    for(i in 1:m) result[i] <- mean(x[(i-1)*k+(1:k)]);
    return(result);
}
```

EXAMPLE B.1.6 *Simulating a Sample from the Distribution of the Discrepancy Statistic in Example 9.1.2*
The following R code generates a sample from the discrepancy statistic specified in Example 9.1.2. To generate the sample, type `y<-discrepancy(m,n)` to place a sample of size `m` in `y`, where `n` is the size of the original data set. This code can be easily modified to generate samples from other discrepancy statistics.

```
#   Example B.2.6
#   function name: discrepancy
#   parameters:
#       m   resample size
#       n   size of data
#   return value:
#       an array of m discrepancies
#   description: this function generates m discrepancies
#   when the data size is n
discrepancy <- function(m,n) {
    result <- rep(0,m);
    for(i in 1:m) {
        x    <- rnorm(n);
        xbar <- mean(x);
        r    <- (x-xbar)/sqrt((sum((x-xbar)**2)));
        result[i] <- -sum(log(r**2));
    }
    return(result/n);
}
```

■

EXAMPLE B.1.7 *Generating from a Dirichlet Distribution in Example 10.2.3*
The following R code generates a sample from a Dirichlet($\alpha_1, \alpha_2, \alpha_3, \alpha_4$) distribution. To generate from this distribution, first assign values to the vector `alpha`, and then type `ddirichlet(n,alpha)` where `n` is the sample size.

```
#   Example B.2.7
#   function name: ddirichlet
#   parameters:
#       n      sample size
#       alpha  vector(alpha1,...,alphak)
#   return value:
#       a (n x k) matrix. rows are i.i.d. samples
#   description: this function generates n random samples
#   from Dirichlet(alpha1,...,alphak) distribution
ddirichlet <- function(n,alpha) {
    k <- length(alpha);
    result <- matrix(0,n,k);
    for(i in 1:k) result[,i] <- rgamma(n,alpha[i]);
    for(i in 1:n) result[i,] <- result[i,] /
                                        sum(result[i,]);
```

```
    return(result);
}
```

Matrices

A matrix can be thought of as a collection of data values with two subscripts or as a rectangular array of data. So if `a` is a matrix, then `a[i,j]` is the (i, j)-th element in `a`. Note that `a[i,]` refers to the i-th row of `a` and `a[,j]` refers to the j-th column of `a`. If a matrix has m rows and n columns, then it is an $m \times n$ matrix, and m and n are referred to as the dimensions of the matrix.

Perhaps the simplest way to create matrices is with `cbind` and `rbind` commands. For example,

```
> x<-c(1,2,3)
> y<-c(4,5,6)
> a<-cbind(x,y)
> a
 x y
[1,] 1 4
[2,] 2 5
[3,] 3 6
```

creates the vectors `x` and `y`, and the `cbind` command takes `x` as the first column and `y` as the second column of the newly created 3×2 matrix `a`. Note that in the printout of `a`, the columns are still labelled `x` and `y`, although we can still refer to these as `a[,1]` and `a[,2]`. We can remove these column names via the command `colnames(a)<-NULL`. Similarly, the `rbind` command will treat vector arguments as the rows of a matrix. To determine the number of rows and columns of a matrix `a`, we can use the `nrow(a)` and `ncol(a)` commands. We can also create a diagonal matrix using the `diag` command. If `x` is an n-dimensional vector, then `diag(x)` is an $n \times n$ matrix with the entries in `x` along the diagonal and 0's elsewhere. If `a` is an $m \times n$ matrix, then `diag(a)` is the vector with entries taken from the main diagonal of `a`. To create an $n \times n$ identity matrix, use `diag(n)`.

There are a number of operations that can be carried out on matrices. If matrices `a` and `b` are $m \times n$, then `a+b` is the $m \times n$ matrix formed by adding the matrices componentwise. The transpose of `a` is the $n \times m$ matrix `t(a)`, with i-th row equal to the i-th column of `a`. If `c` is a number, then `c*a` is the $m \times n$ matrix formed by multiplying each element of `a` by `c`. If `a` is $m \times n$ and `b` is $n \times p$, then `a%*%b` is the $m \times p$ matrix product (Appendix A.4) of `a` and `b`. A numeric vector is treated as a column vector in matrix multiplication. Note that `a*b` is also defined when `a` and `b` are of the same dimension but this is the componentwise product of the two matrices, which is quite different from the matrix product.

If `a` is an $m \times m$ matrix, then the inverse of `a` is obtained as `solve(a)`. The `solve` command will return an error if the matrix does not have an inverse. If `a` is a square matrix, then `det(a)` computes the determinant of `a`.

We now consider an important application.

EXAMPLE B.1.8 *Fitting Regression Models*
Suppose the n-dimensional vector `y` corresponds to the response vector and the $n \times k$ matrix `V` corresponds to the design matrix when we are fitting a linear regression model given by $E(y \mid V) = V\beta$. The least-squares estimate of β is given by `b` as computed in

```
b<-solve(t(V)%*%V)%*%t(V)%*%y
```

with the vector of predicted values `p` and residuals `r` given by

```
> p<-V%*%b
> r<-y-p
```

with squared lengths

```
> slp<-t(p)%*%p
> slr<-t(r)%*%r
```

where `slp` is the squared length of `p` and `slr` is the squared length of `r`. Note that the matrix `solve(t(V)%*%V)` is used for forming confidence intervals and tests for the individual β_i. Virtually all the computations involved in fitting and inference for the linear regression matrix can be carried out using matrix computations in R like the ones we have illustrated. ∎

Packages

There are many packages that have been written to extend the capability of basic R. It is very likely that if you have a data analysis need that cannot be met with R, then you can find a freely available package to add. We refer the reader to `?install.packages` and `?library` for more on this.

B.2 Using Minitab

All the computations found in this text were carried out using Minitab. This statistical software package is very easy to learn and use. Other packages such as SAS or R (see section B.1) could also be used for this purpose.

Most of the computations were performed using Minitab like a calculator, i.e., data were entered and then a number of Minitab commands were accessed to obtain the quantities desired. No programming is required for these computations.

There were a few computations, however, that did involve a bit of programming. Typically, this was a computation in which numerous operations to be performed many times, and so looping was desirable. In each such case, we have recorded here the Minitab code that we used for these computations. As the following examples show, these programs were never very involved.

Students can use these programs as templates for writing their own Minitab programs. Actually, the language is so simple that we feel that anyone using another language for programming can read these programs and use them as templates in the same way. Simply think of the symbols `c1`, `c2`, etc. as arrays where we address the i-th element in the array `c1` by `c1(i)`. Furthermore, there are constants `k1`, `k2`, etc.

A Minitab program is called a *macro* and must start with the statement `gmacro` and end with the statement `endmacro`. The first statement after `gmacro` gives a name to the program. Comments in a program, put there for explanatory purposes, start with `note`.

If the file containing the program is called `prog.txt` and this is stored in the root directory of a disk drive called `c`, then the Minitab command

```
MTB> %c:/prog.txt
```

will run the program. Any output will either be printed in the Session window (if you have used a `print` command) or stored in the Minitab worksheet.

More details on Minitab can be found by using `Help` in the program. We provide some examples of Minitab macros used in the text.

EXAMPLE B.2.1 *Bootstrapping in Example 6.4.2*
The following Minitab code generates 1000 bootstrap samples from the data in `c1`, calculates the median of each of these samples, and then calculates the sample variance of these medians.

```
gmacro
bootstrapping
base 34256734
note - original sample is stored in c1
note - bootstrap sample is placed in c2 with each one
note   overwritten
note - medians of bootstrap samples are stored in c3
note - k1 = size of data set (and bootstrap samples)
let k1=15
do k2=1:1000
sample 15 c1 c2;
replace.
let c3(k2)=median(c2)
enddo
note - k3 equals (6.4.5)
let k3=(stdev(c3))**2
print k3
endmacro
```

■

EXAMPLE B.2.2 *Sampling from the Posterior in Example 7.3.1*
The following Minitab code generates a sample of 10^4 from the joint posterior in Example 7.3.1. Note that in Minitab software, the Gamma(α, β) density takes the form $(\beta^{-\alpha}/\Gamma(\alpha))x^{\alpha-1}e^{-x/\beta}$. So to generate from a Gamma(α, β) distribution, as defined in this book, we must put the second shape parameter equal to $1/\beta$ in Minitab.

```
gmacro
normalpost
note - the base command sets the seed for the random
note   numbers
```

```
base 34256734
note - the parameters of the posterior
note - k1 = first parameter of the gamma distribution
note      = (alpha_0 + n/2)
let k1=9.5
note - k2 = 1/beta
let k2=1/77.578
note - k3 = posterior mean of mu
let k3=5.161
note - k4 = (n + 1/(tau_0 squared) )^(-1)
let k4=1/15.5
note - main loop
note - c3 contains generated value of sigma**2
note - c4 contains generated value of mu
note - c5 contains generated value of coefficient of
        variation
do k5=1:10000
random 1 c1;
gamma k1 k2.
let c3(k5)=1/c1(1)
let k6=sqrt(k4/c1(1))
random 1 c2;
normal k3 k6.
let c4(k5)=c2(1)
let c5(k5)=sqrt(c3(k5))/c4(k5)
enddo
endmacro
```

EXAMPLE B.2.3 *Calculating the Estimates and Standard Errors in Example 7.3.1*
We have a sample of 10^4 values from the posterior distribution of ψ stored in C5. The following computations use this sample to calculate an estimate of the posterior probability that $\psi \leq 0.5$ (k1), as well as to calculate the standard error of this estimate (k2), the estimate minus three times its standard error (k3), and the estimate plus three times its standard error (k4).

```
let c6=c5 le .5
let k1=mean(c6)
let k2=sqrt(k1*(1-k1))/sqrt(10000)
let k3=k1-3*k2
let k4=k1+3*k2
print k1 k2 k3 k4
```

EXAMPLE B.2.4 *Using the Gibbs Sampler in Example 7.3.2*
The following Minitab code generates a chain of length 10^4 values using the Gibbs sampler described in Example 7.3.2.

```
gmacro
gibbs
base 34256734
note - data sample is stored in c1
note - starting value for mu.
let k1=mean(c1)
note - starting value for sigma**2
let k2=stdev(c1)
let k2=k2**2
note - lambda
let k3=3
note - sample size
let k4=15
note - n/2 + alpha_0 + 1/2
let k5=k4/2 +2+.5
note - mu_0
let k6=4
note - tau_0**2
let k7=2
note - beta_0
let k8=1
let k9=(k3/2+.5)
note - main loop
do k100=1:10000
note - generate the nu_i in c10
do k111=1:15
let k10=.5*(((c1(k111)-k1)**2)/(k2*k3) +1)
let k10=1/k10
random 1 c2;
gamma k9 k10.
let c10(k111)=c2(1)
enddo
note - generate sigma**2 in c20
let c11=c10*((c1-k1)**2)
let k11=.5*sum(c11)/k3+.5*((k1-k6)**2)/k7 +k8
let k11=1/k11
random 1 c2;
gamma k5 k11.
let c20(k100)=1/c2(1)
let k2=1/c2(1)
note - generate mu in c21
let k13=1/(sum(c10)/k3 +1/k7)
```

```
let c11=c1*c10/k3
let k14=sum(c11)+k6/k7
let k14=k13*k14
let k13=sqrt(k13*k2)
random 1 c2;
normal k14 k13.
let c21(k100)=c2(1)
let k1=c2(1)
enddo
endmacro
```

EXAMPLE B.2.5 *Batching in Example 7.3.2*
The following Minitab code divides the generated sample, obtained via the Gibbs sampling code for Example 7.3.2, into batches, and calculates the batch means.

```
gmacro
batching
note - k2= batch size
let k2=40
note - k4 holds the batch sums
note - c1 contains the data to be batched (10000 data values)
note - c2 will contain the batch means (250 batch means)
do k10=1:10000/40
let k4=0
do k20=0:39
let k3=c1(k10+k20)
let k4=k4+k3
enddo
let k11=floor(k10/k2) +1
let c2(k11)=k4/k2
enddo
endmacro
```

EXAMPLE B.2.6 *Simulating a Sample from the Distribution of the Discrepancy Statistic in Example 9.1.2*
The following code generates a sample from the discrepancy statistic specified in Example 9.1.2.

```
gmacro
goodnessoffit
base 34256734
note - generated sample is stored in c1
note - residuals are placed in c2
note - value of D(r) are placed in c3
note - k1 = size of data set
let k1=5
```

```
do k2=1:10000
random k1 c1
let k3=mean(c1)
let k4=sqrt(k1-1)*stdev(c1)
let c2=((c1-k3)/k4)**2
let c2=loge(c2)
let k5=-sum(c2)/k1
let c3(k2)=k5
enddo
endmacro
```

EXAMPLE B.2.7 *Generating from a Dirichlet Distribution in Example 10.2.3*
The following code generates a sample from a Dirichlet($\alpha_1, \alpha_2, \alpha_3, \alpha_4$) distribution, where $\alpha_1 = 2, \alpha_2 = 3, \alpha_3 = 1, \alpha_4 = 1.5$.

```
gmacro
dirichlet
note - the base command sets the seed for the random
note   number generator (so you can repeat a simulation).
base 34256734
note - here we provide the algorithm for generating from
note   a Dirichlet(k1,k2,k3,k4) distribution.
note - assign the values of the parameters.
let k1=2
let k2=3
let k3=1
let k4=1.5
let k5=K2+k3+k4
let k6=k3+k4
note - generate the sample with i-th sample in i-th row
note   of c2, c3, c4, c5, ....
do k10=1:5
random 1 c1;
beta k1 k5.
let c2(k10)=c1(1)
random 1 c1;
beta k2 k6.
let c3(k10)=(1-c2(k10))*c1(1)
random 1 c1;
beta k3 k4.
let c4(k10)=(1-c2(k10)-c3(k10))*c1(1)
let c5(k10)= 1-c2(k10)-c3(k10)-c4(k10)
enddo
endmacro
```