Appendix A: An Introduction to HB Modeling in R

In order to facilitate computation of the models in this book, we created a set of programs written in R. R is a general purpose programming and statistical analysis system. R is free and available on the web. We have made our suite of programs into what is called an R "package." Our package is named `bayesm`. This package is easy to download and install from within R and is thoroughly documented, including test examples.

This appendix provides an introduction to the R environment and `bayesm` using the example of a hierarchical binary logit model.

We start with the installation of the R statistical package and `bayesm`, provide a short introduction to the R language and programming, and conclude with a case study involving a heterogeneous binary logit model calibrated on conjoint data. We assume the user is working in a Windows environment. All R commands and objects are the same under Windows and LINUX but the install procedure and GUI are different.

1. Setting Up the R Environment

Obtaining R

Visit http://cran.r-project.org/ or google "R language." CRAN is a network of mirror sites that allow you to download precompiled binary versions of R or source.

Look under "Precompiled Binary Distributions" and click on **Windows**. Click on **base** on the next page and download the rwXXXX.exe file (XXXX=2001 at present) – right click and "save target as." Doubleclick the file name under Windows Explorer and R will install itself in the usual fashion for Windows software.

You may also obtain the optimized BLAS version for Pentium 4 chips by visiting the **contributed** link and click on the ATLAS directory. For Windows users the link is labeled **contrib** and is located directly below the **base** link. Simply download the RBLAS.dll file for your chip (invariably Pentium 4 found in the directory "P4/") and replace the RBLAS.dll file in the R/bin directory (this will be something like C:/Program Files/R/rwXXXX/bin). Our experience is that this will double the speed of many common matrix operations.

Customizing the R Shortcut

The standard R install will create a desktop shortcut to invoke R. This will start up R pointing to the rwXXXX directory . It is more useful to modify the shortcut to start up with R pointing to a directory which is "closer" to the directories you plan on working in. To modify the short-cut, right click on it and choose "properties", change the "Startin: " value to any valid directory on your machine. You can also copy the short-cut and make a short-cut for each of your major "projects."

If you have more than 500 MB of memory (we highly recommend min 1 GB memory), you may also want to add the option to increase the memory available on the short cut. This is

accomplished by adding a parameter to the command line which invokes R. Again, right click the short-cut and modify the command in the target line. For example:

"C:\Program Files\R\rwXXXX\bin\Rgui.exe" --max-mem-size=1900Mb

Note the closed-quote after Rgui.exe BEFORE the specification of the –max-mem-size parameter. You may also copy this shortcut to your "taskbar" – don't forget to "unlock" it first!

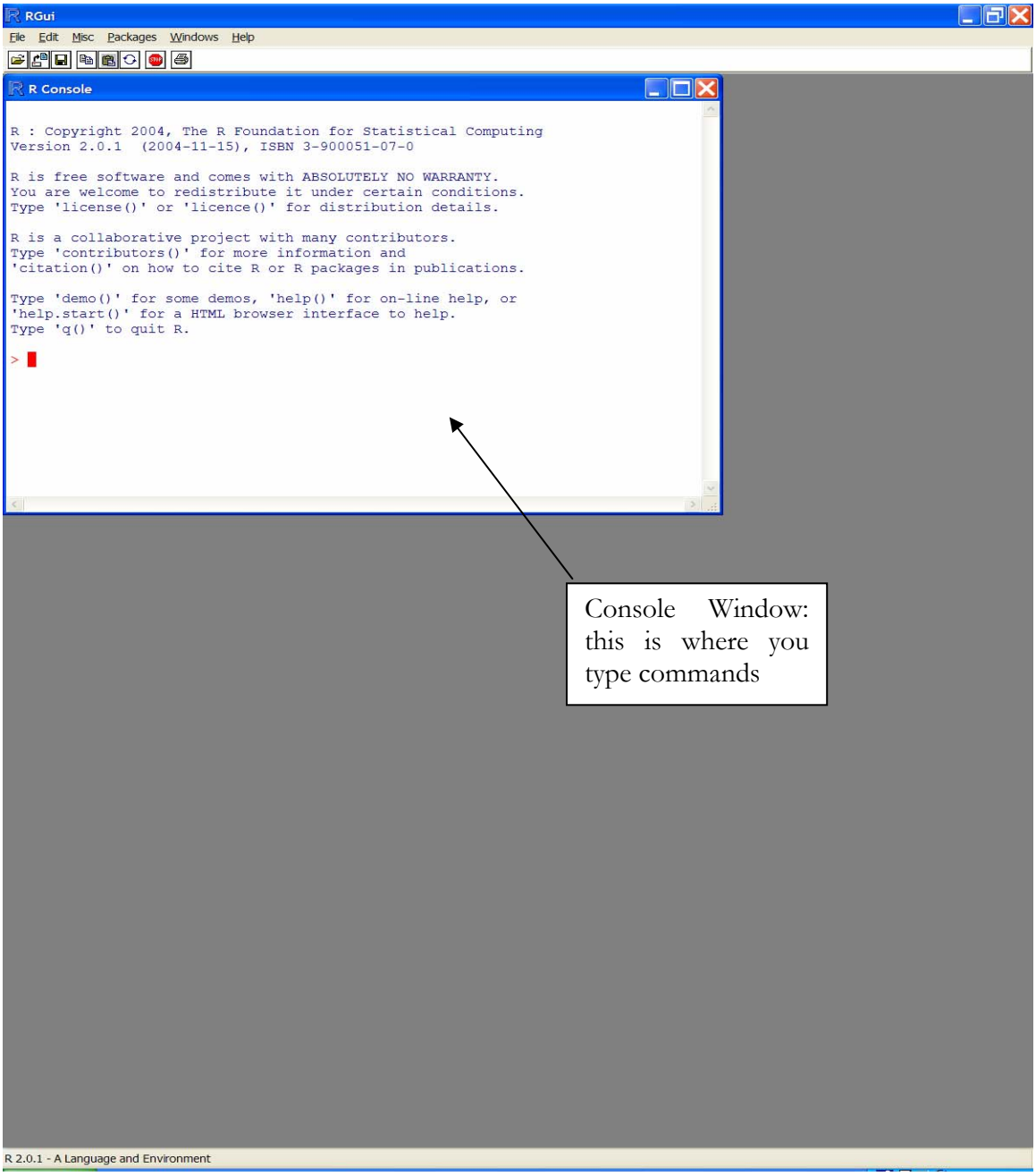
Invoking R and Using the R GUI

Click the short-cut to invoke R. Something like the screen shot below will appear.

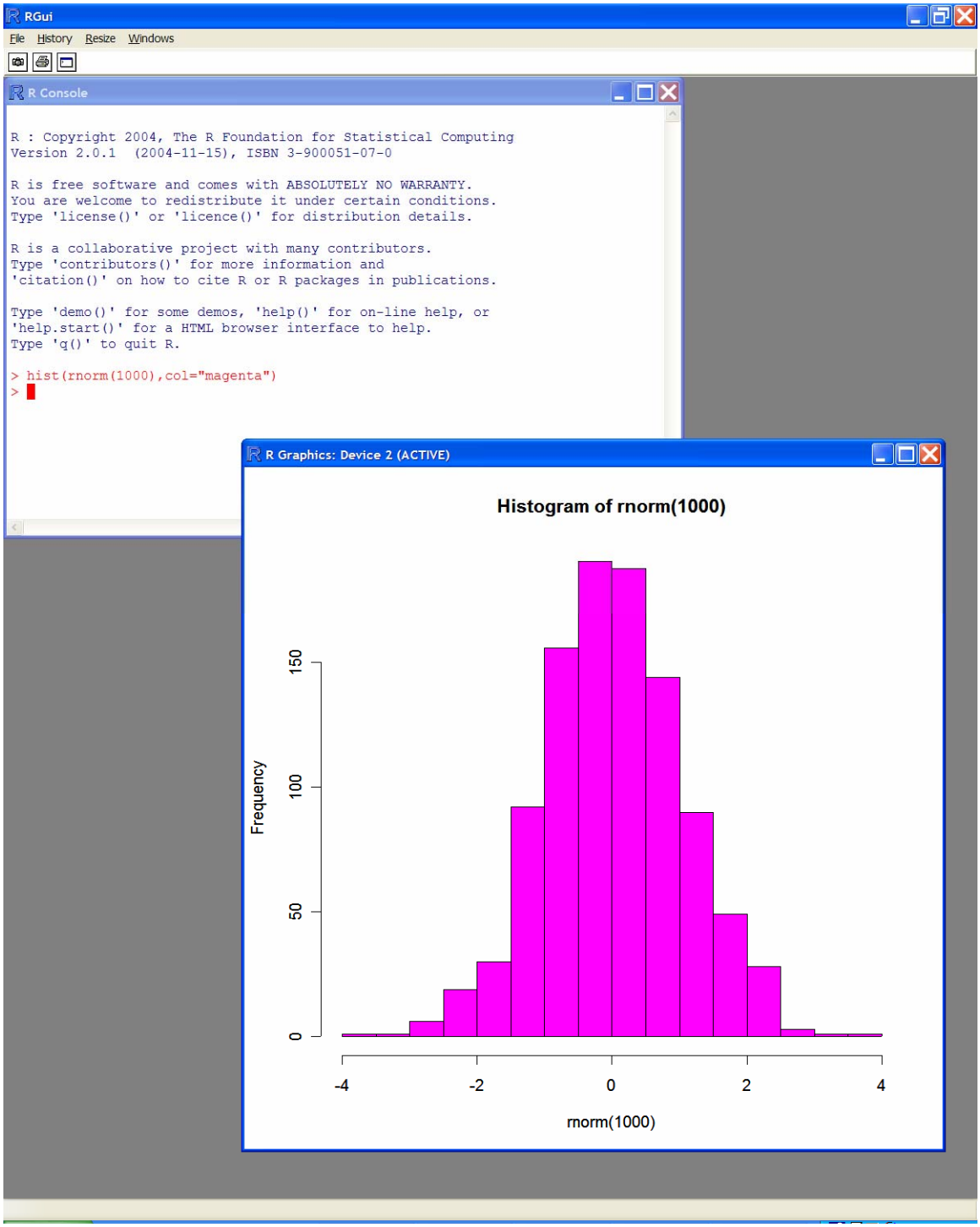
File Menu allows you to change directories, "source" or read in files of commands to execute.

Packages Menu allows you to select contributed packages to download and install automatically

Help Menu allows you access to all of the R manuals in PDF or HTML form.



If you create graphics, a separate graphics window will appear within the RGUI main window. For example, if we create a histogram of 1000, normal random numbers by using the `hist` command (more below in graphics section). A window will automatically open with the plot. (It is possible to create multiple graphics windows and "paint" different graphics in each – see commands `dev.cur(), dev.list(), dev.set()`)

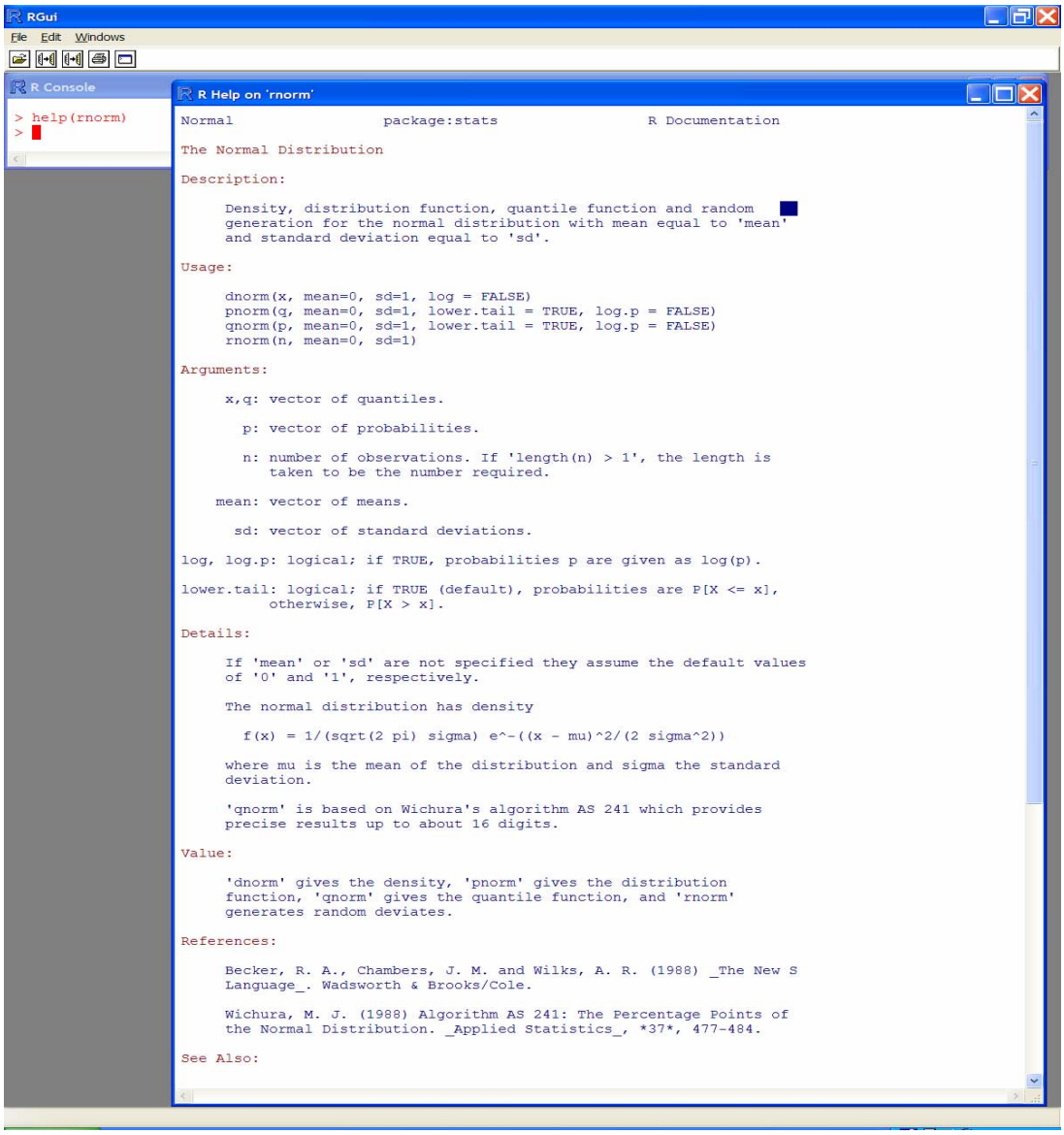


The contents of this graphics window can be "clipped" into the clipboard for pasting into your favorite application by selecting the graph window and using Ctl-W (not Ctl-C) and the paste or Ctl-V to paste into the application. Note: if you use Ctl-W you will copy the graph as a Windows metafile graphic which is device independent and will print at the highest resolution of the device. Ctl-C copies the graph as a bitmap which can compromise resolution.

Obtaining Help in R

There are a number of different ways to obtain help in R. The "Help" menu allows you to access the manuals in PDF or HTML form as well as to search for keywords. Note: the help menu does not appear in the R GUI unless the console window is "active." To make any window active, click on the "button bar" or the blue bar on the top of the window.

You can also use the `help` and `help.search` commands in the R console windows. For example, we just used two commands, `hist` and `rnorm`. `help(rnorm)` produces a window with the following content (a short cut is the command `?rnorm`):



Each R help window has the same sections:

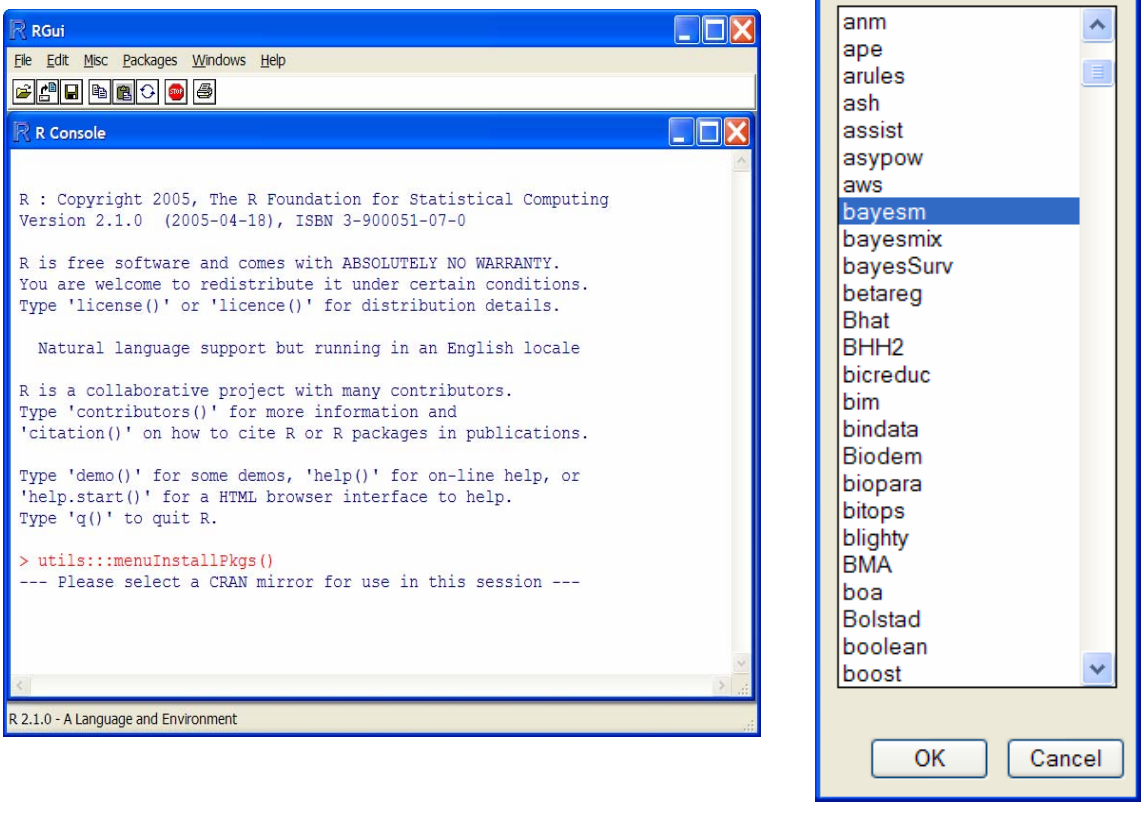
Description
Usage
Arguments
Details
Value
References
See Also
Examples (note this is "cut-off" in the screen shot above)

Usage/Arguments/Examples are the most useful.

If you are not sure what command you need, `help.search("key word")` can be very useful.

Installing *bayesm*

bayesm can be installed from a CRAN mirror site by selecting the "Packages" tab on the toolbar in the R software package. Users must first set the CRAN mirror, then install ***bayesm*** directly from CRAN by simply clicking on "bayesm" in the pop-up window displayed. This is displayed in the screen capture below.



Program Editing and R

For all but the simplest tasks, it is useful to edit a file with R commands in it. R syntax is sufficiently complex that it is difficult to write directly into the command window without making numerous syntax errors.

Open a text editor (VIM – improved vi is recommended; this is shareware, http://www.vim.org/); type in R commands and save the file. You can either cut the commands from the editor window and paste into the R console window to run or use the R command, `source` – also available from the File menu in the Windows GUI.

2. The R Language

R is a functionally-oriented language. All commands are functions which act upon objects of various types. All commands produce objects as well. The basic R command is of the form: object=function(object). Functions can be composed to produce powerful (but sometimes hard-to-read) expressions. Users can define their own functions. Writing these user functions constitutes R-programming.

Let's start by reading in some data. Suppose we have a file in a spreadsheet that with some regression data on several different units. The file has a UNIT variable to identify which unit the data comes from, a dependent variable Y, and two independent variables X1, X2.

UNIT	Y	X1	X2
A	1	0.23815	0.4373
A	2	0.55508	0.47938
A	3	3.03399	-2.17571
A	4	-1.49488	1.66929
B	10	-1.74019	0.35368
B	9	1.40533	-1.2612
B	8	0.15628	-0.27751
B	7	-0.93869	-0.0441
B	6	-3.06566	0.14486

We write this data out of Excel by saving it as a text (tab-delimited file), data.txt (use the **save as** option on the **file** menu and choose text file in the file type box). Note: there is no simple, direct way to read .XLS files in R.

We can read this file into R using the `READ.TABLE` command.

```
> df=read.table("data.txt",header=TRUE)
> df
  UNIT  Y       X1       X2
1    A  1  0.23815  0.43730
2    A  2  0.55508  0.47938
3    A  3  3.03399 -2.17571
4    A  4 -1.49488  1.66929
5    B 10 -1.74019  0.35368
6    B  9  1.40533 -1.26120
```

```
7    B  8  0.15628 -0.27751
8    B  7 -0.93869 -0.04410
9    B  6 -3.06566  0.14486
```

The read.table function has two arguments: the name of the file, and the argument "header." There are many other arguments but they are optional and often have defaults. The default for the header argument is the value FALSE. Using the argument, `header=TRUE`, tells the read.table function to expect that the first line of the file will contain (delimited by spaces or tabs) the names of each variable. TRUE and FALSE are examples of reserved values in R indicating a logical switch for true or false. Another useful reserved value is NULL which is often used to create an object with nothing in it.

The command `df=read.table(…)` assigns the output of the read.table function to the R object named "df." The object "df" is a member of a class or type of object called a data frame. A "data frame" is preferred by R as the format for datasets. A data frame contains a set of observations on variables with the same number of observations in each variable. In this example, each of the variables, Y, X1, and X2, is of type numeric (R does not distinguish between integers and floating point numbers), while the variable UNIT is character.

There are two reasons to store your data as a data frame: 1. most R statistical functions require a data frame and 2. the data frame object allows the user to access the data either via the variables names or by viewing the dataframe as a two-dimensional array.

```
> df$Y
[1]  1  2  3  4 10  9  8  7  6
> mode(df$Y)
[1] "numeric"
> df[,2]
[1]  1  2  3  4 10  9  8  7  6
```

We can refer to the Y variable in df by using the df$XXX notation (where XXX is the name of the variable). The "mode" command confirms that this variable is, indeed, numeric. We can also access the Y variable by using notation in R for subsetting a portion of an array. The notation `df[,2]` means the values of the 2nd column of df. Below we will explore the many ways we can subset an array or matrix.

Using Built-In Functions: Running a regression

Let's now use the built-in linear model function in R to run a regression of Y on X1 and X2, pooled across both units A and B.

```
> lmout=lm(Y ~ X1 + X2, data=df)
> names(lmout)
 [1] "coefficients"  "residuals"     "effects"       "rank"
 [5] "fitted.values" "assign"        "qr"            "df.residual"
 [9] "xlevels"       "call"          "terms"         "model"
> print(lmout)

Call:
lm(formula = Y ~ X1 + X2, data = df)
```

```
Coefficients:
(Intercept)           X1           X2
      5.084       -1.485       -2.221
> summary(lmout)

Call:
lm(formula = Y ~ X1 + X2, data = df)

Residuals:
    Min      1Q  Median      3Q     Max
-3.3149 -2.4101  0.4034  2.5319  3.2022

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)   5.0839     1.0194   4.987  0.00248 **
X1           -1.4851     0.8328  -1.783  0.12481
X2           -2.2209     1.3820  -1.607  0.15919
---
Signif. codes:  0 `***' 0.001 `**' 0.01 `*' 0.05 `.' 0.1 ` ' 1

Residual standard error: 2.96 on 6 degrees of freedom
Multiple R-Squared: 0.3607,     Adjusted R-squared: 0.1476
F-statistic: 1.693 on 2 and 6 DF,  p-value: 0.2612
```

`lm` is the function in the package (stats) which fits linear models. Note that the regression is specified via a "formula" that tells `lm` which is the dependent and independent variables. We assign the output from the `lm` function to the object, `lmout`. `lmout` is a special type of object called a "list." A list is simply an ordered collection of objects of any type. The names command will list the names of the elements of the list. We can access any element of the list by using the $ notation.

```
> lmout$coef
(Intercept)           X1           X2
   5.083871   -1.485084   -2.220859
```

Note that we only need to specify enough of the name of the list component to uniquely identify it, e.g. `lmout$coef` is the same as `lmout$coefficients`.

We can "print" the object `lmout` and get a brief summary of its contents. Print is a generic command which uses a different "print method" for each type of object. Print recognizes that `lmout` is a list of type lm and uses a specific routine to printout the contents of the list. A more useful summary of contents of `lmout` can be obtained with the `summary` command.

Inspecting objects and the R workspace

When you start up R, R looks for a file .Rdata in the directory in which R is started from (you can also double-click the file to start R). This file contains a copy of the R "workspace" which is a list of R objects created by the user. For example we just created two R objects in the example above: `df` (the data frame) and `lmout,` the `lm` output object.

To list all objects in the current workspace, use the command `ls()`.

```
> ls()
[1] "df"    "lmout"
```

This doesn't tell us too much about the objects. If you just type the object name at the command prompt and return, then you will invoke the default print method for this type of object as we saw above in the data frame example.

As useful command is the `structure` (`str` for short) command.

```
> str(df)
`data.frame':   9 obs. of  4 variables:
 $ UNIT: Factor w/ 2 levels "A","B": 1 1 1 1 2 2 2 2 2
 $ Y   : int  1 2 3 4 10 9 8 7 6
 $ X1  : num   0.238  0.555  3.034 -1.495 -1.740 ...
 $ X2  : num   0.437  0.479 -2.176  1.669  0.354 ...
```

Note that the str command tells us a bit about the variables in the data frame. The UNIT variable is of type "factor" with two levels. Type "factor" is used by many of the built-in R functions and is way to store qualitative variables.

The R workspace exists only in memory. You must either save the workspace when you exist (you will be prompted for this) or you must recreate the objects again.

Vectors, Matrices and Lists

From our point of view, the power of R comes from statistical programming at a relatively high level. To do so, we will need to organize data as vectors, arrays and lists. Vectors are ordered collections of the same type of object. If we access one variable from our data frame above, it will be a vector.

```
> df$X1
[1]    0.23815    0.55508    3.03399  -1.49488  -1.74019    1.40533
0.15628 -0.93869 -3.06566
> length(df$X1)
[1] 9
> is.vector(df$X1)
[1] TRUE
```

The function `is.vector` returns a logical flag as to whether or not the input argument is a vector. We can also create a vector with the `c()` command.

```
> vec=c(1,2,3,4,5,6)
> vec
[1] 1 2 3 4 5 6
> is.vector(vec)
[1] TRUE
```

A matrix is a two dimensional array. Let's create a matrix from a vector.

```
> mat=matrix(c(1,2,3,4,5,6),ncol=2)
```

```
> mat
     [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6
```

`matrix()` is a command to create a matrix from a vector. The option `"ncol"` is used to create the matrix with a specified number of columns (see also `nrow`). Note that the matrix is created column by column from the input vector (first subscripts varies the fastest). We can create a matrix row by row with the following command:

```
> mat=matrix(c(1,2,3,4,5,6),byrow=TRUE,ncol=2)
> mat
     [,1] [,2]
[1,]    1    2
[2,]    3    4
[3,]    5    6
```

We can also convert a data frame into a matrix.

```
> dfmat=as.matrix(df)
> dfmat
  UNIT Y    X1         X2
1 "A"  " 1" " 0.23815" " 0.43730"
2 "A"  " 2" " 0.55508" " 0.47938"
3 "A"  " 3" " 3.03399" "-2.17571"
4 "A"  " 4" "-1.49488" " 1.66929"
5 "B"  "10" "-1.74019" " 0.35368"
6 "B"  " 9" " 1.40533" "-1.26120"
7 "B"  " 8" " 0.15628" "-0.27751"
8 "B"  " 7" "-0.93869" "-0.04410"
9 "B"  " 6" "-3.06566" " 0.14486"
> dim(dfmat)
[1] 9 4
```

Note that all of the values of the results matrix are character as one of the variables in the data frame (UNIT) is character-valued. Finally, matrices can be created from other matrices and vectors using the `cbind` (column bind) and `rbind` (row bind) commands.

```
> mat1
     [,1] [,2]
[1,]    1    2
[2,]    3    4
[3,]    5    6
> mat2
     [,1] [,2]
[1,]    7   10
[2,]    8   11
[3,]    9   12
> cbind(mat1,mat2)
     [,1] [,2] [,3] [,4]
[1,]    1    2    7   10
[2,]    3    4    8   11
```

```
[3,]    5    6    9   12
> rbind(mat1,mat2)
     [,1] [,2]
[1,]    1    2
[2,]    3    4
[3,]    5    6
[4,]    7   10
[5,]    8   11
[6,]    9   12
> rbind(mat1,c(99,99))
     [,1] [,2]
[1,]    1    2
[2,]    3    4
[3,]    5    6
[4,]   99   99
```

R supports multi-dimensional arrays as well. Below is an example of creating a three dimensional array from a vector.

```
> ar=array(c(1,2,3,4,5,6),dim=c(3,2,2))
> ar
, , 1

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

, , 2

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

Again, the array is created by using vector for the first dimension, then the second, and then third. A 3 x 2 x 2 array as 12 elements not the six provided as an argument. R will repeat the input vector as necessary until the required number of elements are obtained.

A list is an ordered collection of objects of any type. It is the most flexible object in R that can be indexed. As we have seen in the `lm` function output, lists can also have names.

```
> l=list(1,"a",c(4,4),list(FALSE,2))
> l
[[1]]
[1] 1

[[2]]
[1] "a"

[[3]]
[1] 4 4
```

```
[[4]]
[[4]][[1]]
[1] FALSE

[[4]][[2]]
[1] 2

> l=list(num=1,char="a",vec=c(4,4),list=list(FALSE,2))
> l$num
[1] 1
> l$list
[[1]]
[1] FALSE

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

In the example, we created a list of a numeric value, character, vector and another list. We also can name each component and access them with the $ notation.

Accessing Elements and Subsetting Vectors, Arrays, and Lists

To access an element of a vector, simply enclose index of that element in square brackets.

```
> vec=c(1,2,3,2,5)
> vec[3]
[1] 3
```

To access a sub-set of elements, there are two approaches: 1. specify a vector of integers of the required indices, 2. specify a logical variable which is TRUE for the desired indices.

```
> index=c(3:5)
> index
[1] 3 4 5
> vec[index]
[1] 3 2 5
> index=vec==2
> index
[1] FALSE  TRUE FALSE  TRUE FALSE
> vec[index]
[1] 2 2
> vec[vec!=2]
[1] 1 3 5
```

`c(3:5)` creates a vector from the "pattern" or sequence from 3 to 5. The `seq` command can create a wide variety of different patterns.

To properly understand the example of the logical index, it should be noted that "=" is an assignment operator while "==" is a comparison operator. Vec==2 creates a logical vector with flags for if the elements of vec are 2. The last example uses the "not equal" comparison operator !=. We can also access the elements not in a specified index vector.

```
> vec[-c(3:5)]
```

```
[1] 1 2
```

To access elements of arrays, we can use the same ideas for vectors but we must specify a set of row and column indices. If no indices are specified, we get all of the elements on that dimension. For example, earlier we used the notation `df[,2]` to access the second column of the data frame `df`.

We can pull off the observations corresponding to unit "A" from the matrix version of dfmat using the commands:

```
> dfmat
  UNIT Y    X1         X2
1 "A"  " 1" " 0.23815" " 0.43730"
2 "A"  " 2" " 0.55508" " 0.47938"
3 "A"  " 3" " 3.03399" "-2.17571"
4 "A"  " 4" "-1.49488" " 1.66929"
5 "B"  "10" "-1.74019" " 0.35368"
6 "B"  " 9" " 1.40533" "-1.26120"
7 "B"  " 8" " 0.15628" "-0.27751"
8 "B"  " 7" "-0.93869" "-0.04410"
9 "B"  " 6" "-3.06566" " 0.14486"
> dfmat[dfmat[,1]=="A",2:4]
  Y    X1         X2
1 " 1" " 0.23815" " 0.43730"
2 " 2" " 0.55508" " 0.47938"
3 " 3" " 3.03399" "-2.17571"
4 " 4" "-1.49488" " 1.66929"
```

The result is a 4 x 3 matrix. Note that we are using the values of the dfmat to index into itself. This means that R evaluates the expression `dfmat[,1] == "A"` and passes the result into the matrix subsetting operator `[ ]` which is a function that processes `dfmat`.

To access elements of lists, we can use the $ notation if the element has a name or we can use a special operator `[[ ]]`. To see how this works, let's make a list with two elements, each corresponding to the observations for unit A and B. Note that the size of the matrices corresponding to each unit is different – unit A has four obs and unit B has five! This means that we can't use a three dimensional array to store this data (we would need a "ragged" array).

```
>
ldata=list(A=dfmat[dfmat[,1]=="A",2:4],B=dfmat[dfmat[,1]=="B",2:4
])
> ldata
$A
  Y    X1         X2
1 " 1" " 0.23815" " 0.43730"
2 " 2" " 0.55508" " 0.47938"
3 " 3" " 3.03399" "-2.17571"
4 " 4" "-1.49488" " 1.66929"

$B
```

```
  Y    X1         X2
5 "10" "-1.74019" " 0.35368"
6 " 9" " 1.40533" "-1.26120"
7 " 8" " 0.15628" "-0.27751"
8 " 7" "-0.93869" "-0.04410"
9 " 6" "-3.06566" " 0.14486"

> ldata[1]
$A
  Y    X1         X2
1 " 1" " 0.23815" " 0.43730"
2 " 2" " 0.55508" " 0.47938"
3 " 3" " 3.03399" "-2.17571"
4 " 4" "-1.49488" " 1.66929"

> is.matrix(ldata[1])
[1] FALSE
> is.list(ldata[1])
[1] TRUE
> ldata$A
  Y    X1         X2
1 " 1" " 0.23815" " 0.43730"
2 " 2" " 0.55508" " 0.47938"
3 " 3" " 3.03399" "-2.17571"
4 " 4" "-1.49488" " 1.66929"
> is.matrix(ldata$A)
[1] TRUE
```

If we specify `ldata[1]`, we don't get the contents of the list element (which is a matrix) but we get a list! If we specify `ldata$A`, we obtain the matrix. If we have a long list or we don't wish to name each element, we can use the `[[ ]]` operator to access elements in the list.

```
> is.matrix(ldata[[1]])
[1] TRUE
> ldata[[1]]
  Y    X1         X2
1 " 1" " 0.23815" " 0.43730"
2 " 2" " 0.55508" " 0.47938"
3 " 3" " 3.03399" "-2.17571"
4 " 4" "-1.49488" " 1.66929"
```

Loops

As with all interpreted languages, loops in R are slow. That is, they typically take more time than if implemented in a compiled language. On the other hand, matrix/vector operations are typically faster in R than in compiled language such as C and Fortran unless the optimized BLAS is called. Thus, wherever possible, "vectorization" or writing expressions as only involving matrix/vector arithmetic is desirable. This is more of an art than a science, however.

If a computation is fundamentally iterative (such as maximization or MCMC simulation), a loop will be required. A simple loop can be accomplished with the for structure. The syntax is of the form

```
for (var in range) { }
```

var is a numeric loop index. Range is a range of values of var. Enclosed in the braces is any valid R expression. There can be more than one R statement in the R expression. The simplest example is a loop over a set of i values from 1 to N.

```
x=0
for (i in 1:10)
    {
     x=x+1
    }
```

Let's loop over both units and create a list of lists of the regression output from each.

```
> ldatadf=list(A=df[df[,1]=="A",2:4],B=df[df[,1]=="B",2:4])
> lmout=NULL
> for (i in 1:2) {
+    lmout[[i]]=lm(Y ~ X1+X2,data=ldatadf[[i]])
+    print(lmout[[i]])
+    }

Call:
lm(formula = Y ~ X1 + X2, data = ldatadf[[i]])

Coefficients:
(Intercept)           X1           X2
      4.494       -2.860       -3.180


Call:
lm(formula = Y ~ X1 + X2, data = ldatadf[[i]])

Coefficients:
(Intercept)           X1           X2
      9.309        1.051        1.981
```

Here we subset the data frame directly rather than the matrix created from the data frame to avoid the extra-step of converting character to numeric values and so that we can use the lm function which requires data frame input. We can see that the same sub-setting command that works on arrays will also work on data frames.

Implicit Loops

In many contexts, a loop is used to compute the results of applying a function to either the row or column dimensions of an array. For example, if we wish to find the mean of each variable in a data frame, we want to apply the function "mean" to each column. This can be done with the `apply()` function.

```
> apply(df[,2:4],2,mean)
         Y         X1         X2
 5.5555556 -0.2056211 -0.0748900
```

The first argument specifies the array, the second the dimension (1=row, 2=col), and the third the function to be applied. In R, the apply function is simply an elegant loop so don't expect to speed things up with this. Of course, we could write this as a matrix operation which would be much faster.

Matrix Operations

One of the primary advantages of R is that we can write matrix/vector expressions directly in R code. Let's review some of these operators by computing a pooled regression using matrix statements.

The basic functions needed are:

`%*%`	matrix multiplication e.g. (X %*% Y) note: X or Y or both can be vectors
`chol(X)`	compute "square" or Cholesky root of square, pd matrix X=U'U where `U=chol(X);` U is upper triangular
`chol2inv(chol(X))`	compute inverse of square pd matrix using its Cholesky root
`crossprod(X,Y)`	t(X) %*% Y -- very efficient
`diag`	extract diagonal of matrix or create diagonal matrix from a vector

Less frequently used are:

`%x%`	Kronecker product (to be used carefully as Kronecker products can create very large arrays)
`backsolve()`	used to compute inverse of a triangular array

The R statements are:

```
y=as.numeric(dfmat[,2])
X=matrix(as.numeric(dfmat[,3:4]),ncol=2)
X=cbind(rep(1,nrow(X)),X)
XpXinv=chol2inv(chol(crossprod(X)))
bhat=XpXinv%*%crossprod(X,y)
res=as.vector(y-X%*%bhat)
ssq=as.numeric(res%*%res/(nrow(X)-ncol(X)))
se=sqrt(diag(ssq*XpXinv))
```

The first two statements create y and X. Then we add a column of ones using the `rep()` function for the intercept and compute the regression using Cholesky roots. Note that we must convert res to a vector to use the statement res %*% res. We also must convert ssq to a scalar from a 1 x 1 matrix to compute the standard errors in the last statement. We note that the method above is very stable numerically, but some users would prefer the QR decomposition. This would be simpler, but our experience has shown that the method above is actually faster in R.

Other Useful Built-In R Functions

R has thousands of built-in function and thousands more than can be added from contributed packages. Some functions used in the book include:

`rnorm`	draw univariate normal random variates
`runif`	draw uniform random variates
`rchisq`	draw chi-sq random variates
`mean`	compute mean of a vector
`var`	compute Covariance matrix given matrix input
`quantile`	computes quantiles of a vector
`optim`	general purpose optimizer
`sort`	sort a vector
`if`	standard if statement (includes else clause)
`while`	while loop
`scan`	read from a file to a vector
`write`	write a matrix to a file
`sqrt`	square root
`log`	natural log
`%%`	modulo (e.g. 100%%10 = 0)
`round`	round to a specified number of sign digits
`floor`	greatest integer < argument
`.C`	interface to C and C++ code (more later)

User-defined Functions

The regression example above is a situation for which a user-defined function would be useful. To create a function object in R, simply enclose the R statements in braces as assign this to a function variable.

```
myreg=function(y,X){
#
# purpose: compute lsq regression
#
# arguments:
#    y -- vector of dep var
#    X -- array of indep vars
#
# output:
#    list containing lsq coef and std errors
#
XpXinv=chol2inv(chol(crossprod(X)))
bhat=XpXinv%*%crossprod(X,y)
res=as.vector(y-X%*%bhat)
```

```
ssq=as.numeric(res%*%res/(nrow(X)-ncol(X)))
se=sqrt(diag(ssq*XpXinv))
list(b=bhat,std_errors=se)
}
```

The code above should be executed either by cutting and pasting into R or by sourcing a file containing this code. This will define an object called "myreg"

```
ls()
 [1] "ar"           "bhat"         "df"           "dfmat"        "i"            "index"
 [7] "l"            "last.warning" "ldata"        "ldatadf"      "ldataidf"     "lmout"
[13] "mat"          "mat1"         "mat2"         "myreg"        "names"        "res"
[19] "se"           "ssq"          "vec"          "X"            "XpXinv"       "y"
```

To execute the function, we simply type it in with arguments at the command prompt or in another source file.

```
> myreg(X=X,y=y)
$b
          [,1]
[1,]  5.083871
[2,] -1.485084
[3,] -2.220859

$std_errors
[1] 1.0193862 0.8327965 1.3820287
```

`myreg` returns a list with b and the standard errors.

Objects are passed by copy in R rather than by reference. This means that if we give the command `myreg(Z,d),` a copy of Z will be assigned to the "local" variable X in the function myreg and a copy of d to y. In addition, variables created in the function (e.g. XpXinv and res in myreg) are created only during the execution of the function and then erased when the function returns to the calling environment.

The arguments are passed and copied in the order supplied at the time of the call so that you must be careful. The statement `myreg(d,Z)` will not execute properly. However, if we explicitly name the arguments as in `myreg(X=Z,y=d)` then the arguments can be given in any order.

Many functions have default arguments and R has what is called "lazy" function evaluation which means that if an argument is not needed it is not checked. See Introduction to R for a more discussion on default and other types of arguments. If a local variable cannot be found while executing a function, R will look in the environment or workspace that the function was called from. This can be convenient but it can also be dangerous!

Many functions are dependent on other functions. If a function called within a function is only used by that calling function and has no other use, it can be useful to define these utility functions in the calling function definition. This means that they will not be visible to the user of the function.

Example:

```
Myfun= function(X,y) {
#
# define utilty function needed
#
Util=function(X) { … }
#
# main body of myfun
#
…
}
```

Debugging Functions

It is a good practice to define your functions in a file and "source" them into R. This will allow you to recreate your set of function objects for a given project without having to save the workspace.

To debug a function, you can use the brute force method of placing print statements in the function. `cat()` can be useful here. For example, we can define a "debugging" version of myreg which prints out the value of se in the function. The `cat` command prints out a statement reminding us of where the "print" output comes from (note the use of fill=TRUE which insures that a new line will be generated on the console).

```
myreg=function(y,X){
#
# purpose: compute lsq regression
#
# arguments:
#    y -- vector of dep var
#    X -- array of indep vars
#
# output:
#    list containing lsq coef and std errors
#
XpXinv=chol2inv(chol(crossprod(X)))
bhat=XpXinv%*%crossprod(X,y)
res=as.vector(y-X%*%bhat)
ssq=as.numeric(res%*%res/(nrow(X)-ncol(X)))
se=sqrt(diag(ssq*XpXinv))
cat("in myreg, se = ",fill=TRUE)
print(se)
list(b=bhat,std_errors=se)
}
```

When run, this new function will produce the output:

```
> myregout=myreg(y,X)
in myreg, se =
[1] 1.0193862 0.8327965 1.3820287
```

R also features a simple debugger. If you "debug" a function, you can step through the function and inspect the contents of local variables. One can also modify their contents.

```
> debug(myreg)
> myreg(X,y)
debugging in: myreg(X, y)
debug: {
    XpXinv = chol2inv(chol(crossprod(X)))
    bhat = XpXinv %*% crossprod(X, y)
    res = as.vector(y - X %*% bhat)
    ssq = as.numeric(res %*% res/(nrow(X) - ncol(X)))
    se = sqrt(diag(ssq * XpXinv))
    cat("in myreg, se = ", fill = TRUE)
    print(se)
    list(b = bhat, std_errors = se)
}
Browse[1]>
debug: XpXinv = chol2inv(chol(crossprod(X)))
Browse[1]> X
[1]  1  2  3  4 10  9  8  7  6
Browse[1]> #OOPS!
debug: bhat = XpXinv %*% crossprod(X, y)
Browse[1]> XpXinv
             [,1]
[1,] 0.002777778
Browse[1]> Q
> undebug(myreg)
```

If there are loops in the function, the debugging command "c" can be used to allow the loop to finish. "Q" quits the debugger. You must turn off the debugger with the undebug command! If you want to debug other functions called by `myreg`, you must `debug()` 'em first!

Elementary Graphics

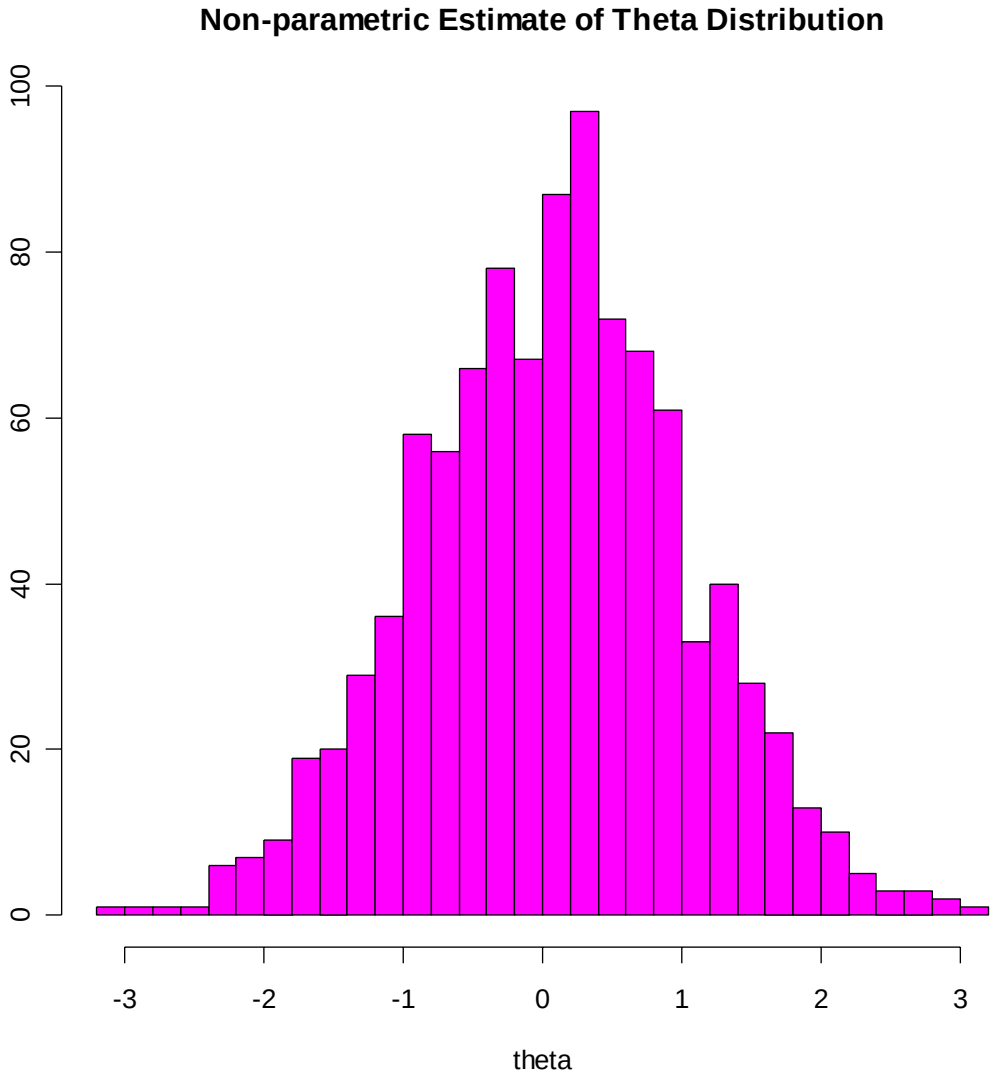
Graphics in R can be quite involved as the graphics capabilities are very extensive. For some examples of what is possible issue the commands `demo(graphics)`, `demo(image)` and `demo(persp)`. Let's return to our first example – a histogram of a distribution.

```
hist(rnorm(1000),breaks=50,col="magenta")
```

This creates a histogram with 50 bars and with each bar filled in the color "magenta" (type `colors()` to see the list of available colors). This plot can be improved by inclusion of plot parameters to change the x and y axis labels and as well as the "title" of the plot.

```
hist(rnorm(1000),breaks=30,col="magenta",xlab="theta",ylab="",mai
n="Non-parametric Estimate of Theta Distribution")
```

produces



Three other basic plots are useful:

`plot(x,y)`	scatterplot of x vs y
`plot(x)`	sequence plot of x
`matplot(X)`	sequence plots of columns of X
`acf(x)`	acf of time series in x

The col, xlab, ylab, and main parameters work on all of these plots.

In addition, the parameters

`type="l"`	connects scatterplot points with a lines
`lwd=x`	specifies the width of lines (1 is default, > 1 is thicker)
`lty=x`	specifies type of line (e.g. solid vs dashed)
`xlim/ylim=c(z,w)`	specifies x/y axis runs from z to w

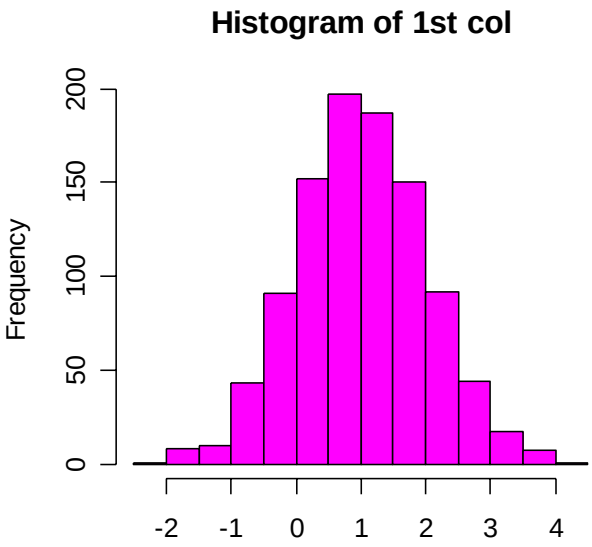
are useful. `?par` displays all of the graphic parameters available.

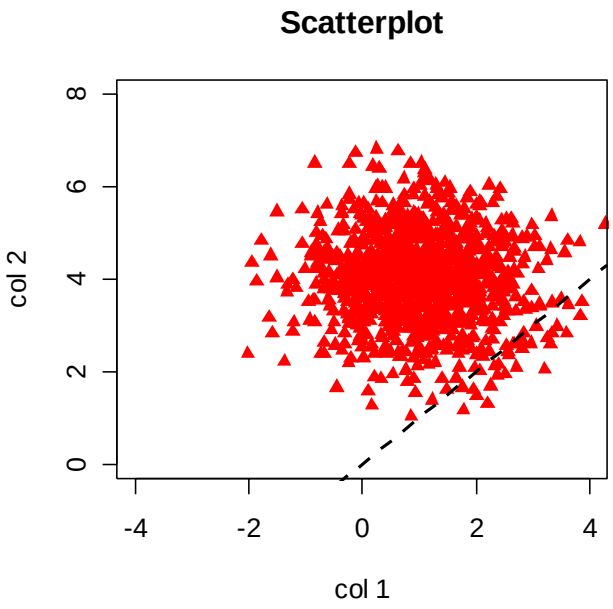
It is often useful to display more than one plot per page. To do this, we must change the global graphic parameters with the command, `par(mfrow=c(x,y))`. This specifies an array of plots x by y plotted row by row.

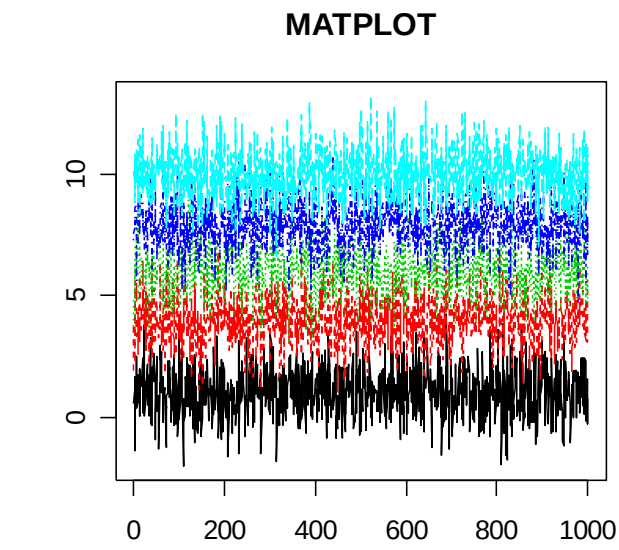
```
par(mfrow=c(2,2))
X=matrix(rnorm(5000),ncol=5)
X=t(t(X)+c(1,4,6,8,10))

hist(X[,1],main="Histogram of 1st col",col="magenta",xlab="")
plot(X[,1],X[,2],xlab="col 1", ylab="col 2",pch=17,col="red",
      xlim=c(-4,4),ylim=c(0,8))
title("Scatterplot")
abline(c(0,1),lwd=2,lty=2)
matplot(X,type="l",ylab="",main="MATPLOT")
acf(X[,5],ylab="",main="ACF of 5th Col")
```

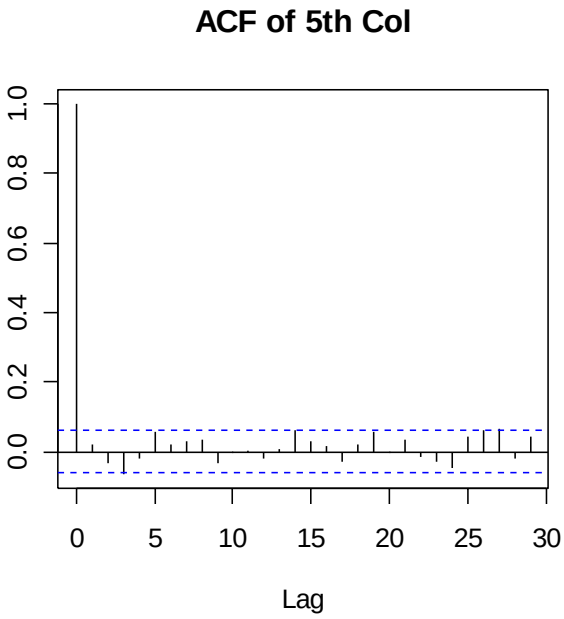
`title()` and `abline()` are examples of commands which modify the current "active" plot. Other userful functions are `points()` and `lines()` to add points and points connected by lines to the current plot.

The commands above will produce


Histogram of 1st col
Frequency
0
50
100
150
200
-2
-1
0
1
2
3
4


Scatterplot
col 2
0
2
4
6
8
-4
-2
0
2
4
col 1


MATPLOT
0
5
10
0
200
400
600
800
1000


ACF of 5th Col
0.0
0.2
0.4
0.6
0.8
1.0
0
5
10
15
20
25
30
Lag

System Information

The following commands provide information about the operating system:

`memory.limit()`	current memory limit
`memory.size()`	current memory size
`system.time(R expression)`	times execution of R expression
`proc.time()[3]`	current R session cpu usage in seconds
`getwd()`	obtain current working directory
`setwd()`	set current working directory
`Rprof(file="filename")`	turns on profiling and writes to filename
`Rprof("")`	turns off profiling
`summaryRprof(file="filename")`	summarizes output in profile file

Examples of usage are given below.

```
> memory.size()
[1] 191135504
> getwd()
[1] "C:/userdata/per/class/37904"
> x=matrix(rnorm(1e07),ncol=1000)
> memory.size()
[1] 332070456
> memory.limit()
[1] 1992294400
> begin=proc.time()[3]
> z=crossprod(x)
> end=proc.time()[3]
> print(end-begin)
[1] 6.59
>test=function(n){x=matrix(rnorm(n),ncol=1000);z=crossprod(x);
cz=chol(z)}
> Rprof("test.out")
> test(1e07)
> Rprof()
> summaryRprof("test.out")
$by.self
          self.time self.pct total.time total.pct
rnorm          4.40     48.9       4.40      48.9
crossprod      4.16     46.2       4.16      46.2
matrix         0.22      2.4       4.72      52.4
.Call          0.12      1.3       0.12       1.3
as.vector      0.10      1.1       4.50      50.0
chol           0.00      0.0       0.12       1.3
test           0.00      0.0       9.00     100.0

$by.total
          total.time total.pct self.time self.pct
test            9.00     100.0      0.00      0.0
matrix          4.72      52.4      0.22      2.4
as.vector       4.50      50.0      0.10      1.1
rnorm           4.40      48.9      4.40     48.9
crossprod       4.16      46.2      4.16     46.2
.Call           0.12       1.3      0.12      1.3
```

```
chol            0.12       1.3      0.00      0.0

$sampling.time
[1] 9
```

The profile shows that virtually all of the time in the test function was in the generation of normal random numbers and in computing cross-products. The Cholesky root of a 1000 x 1000 matrix is essentially free! `crossprod` is undertaking 5 billion floating point multiplies (1/2 of 10,000 x 1,000*1,000).

More Lessons Learned from Timing

If you are going to fill up an array with results, preallocate space in the array. Do not append to an existing array.

```
> n=1e04
> x=NULL
> zero=c(rep(0,5))
> begin=proc.time()[3]
> for (i in 1:n) {x=rbind(x,zero) }
> end=proc.time()[3]
> print(end-begin)
[1] 6.62
> x=NULL
> begin=proc.time()[3]
> x=matrix(double(5*n),ncol=5)
> end=proc.time()[3]
> print(end-begin)
[1] 0.07
```

3. Hierarchcial Bayes Modeling – An Example

You must install and load our R package, ***bayesm,*** in order to access the data and programs discussed in this section. See appendix B for further details.

Data from the article by Allenby and Ginter (1995) is used to illustrate hierarchical Bayes analysis within the R environment. The context of the study was a bank offering a credit card to out-of-state customers, where the penalty associated with being an "unknown" brand, relative to other card characteristics, was to be assessed. Estimation of the dis-utility of having an unknown brand name would allow management to devise incentive programs that would make the credit card attractive to consumers. The attribute and attribute-levels under study are presented in table 1.

Preferences were obtained from a tradeoff study in which respondents were asked to choose between two credit cards that were identical in every respect except for two attributes. For example, respondents were asked to state their preference between the following offerings:

> The first card has a medium fixed annual interest rate and a medium annual fee, and
> The second card has a high fixed annual interest rate and low annual fee.

Each respondent provided responses to between 13 to 17 paired-comparisons involving a fraction of the attributes. A respondent trading-off interest rates and annual fees, for example, did not choose between rebates and credit lines. As a result it was not possible to obtain fixed-effect estimates of the entire vector of part-worths for any specific respondent. Moreover, even if all attribute levels were included for each respondent, constraints on the length of the questionnaire preclude collecting a sufficient number of trade-offs for accurate estimation of individual respondent part-worths. We will demonstrate that this data limitation is less important in random-effect models that pool information across respondents. In all, a total of 14,799 paired-comparisons from 946 respondents were available for analysis. Demographic information (age, income and gender) of the respondents were also available for analysis.

Choice probabilities for the binary logit model can be expressed in two ways:

$$\text{(A.1)} \qquad \Pr(y_{h,i} = 1) = \frac{\exp[x_{h,i,1}'\beta_h]}{\exp[x_{h,i,1}'\beta_h] + \exp[x_{h,i,2}'\beta_h]} = \frac{\exp[(x_{h,i,1} - x_{h,i,2})'\beta_h]}{1 + \exp[(x_{h,i,1} - x_{h,i,2})'\beta_h]}$$

where "h" denotes the respondent, "i" denotes the paired-comparison, "$\Pr(y_{h,i}=1)$" denotes the probability of selecting the first card, "$\Pr(y_{h,i}=2)$" is the choice probability for the second card, "x" is an attribute vector of dimension 14 with elements taking on values of 0 or 1 denote absence or presence of the attribute-level, and "β" is a vector of utility, or part-worth coefficients to be estimated. Because of identification restrictions, the first level of each attribute in table 1 was set to zero, and the remaining attributes are dummy-coded to reflect the marginal utility associated with changes in the attribute-levels from the first level.

The choice data file contains 14,799 rows and 16 columns. The first column is respondent identifier (h), the second column is the choice indicator (y) and the remaining 14 columns contain the difference in the attribute vectors ($x_{h,i,1}$-$x_{h,i,2}$). The first 20 lines of the choice data file are displayed in table 2. Values of the choice indicator take on values of 1 and 0, with 1 indicating that the first card was selected by the respondent, and 0 indicating that the second card was selected. The elements of the differenced attribute vector take on values of 1, 0 and -1. Sample observations from the demographic data file are presented in table 3. Income is recorded in thousands of dollars ($000) and the gender variable is coded "1" for female.

Both the choice/attribute and demographic data are available in the ***bayesm*** dataset, `bank`. To access, this data use `data(bank)` (after installing and loading ***bayesm***, see appendix B). To list information about this dataset, use `help(bank)`.

Reading and Organizing Data for Analysis

The choice data are read into the R statistical package and organized for analysis with the hierarchical binary logit model using the commands in the file "run rhierBinLogit.R":

```
data(bank)
choiceAtt=bank$choiceAtt
Z=bank$demo
```

```
## center demo data so that mean of random-effects
## distribution can be interpretted as the average respondents

Z[,1]=rep(1,nrow(Z))
Z[,2]=Z[,2]-mean(Z[,2])
Z[,3]=Z[,3]-mean(Z[,3])
Z[,4]=Z[,4]-mean(Z[,4])
Z=as.matrix(Z)

hh=levels(factor(choiceAtt$id))
nhh=length(hh)
lgtdata=NULL
for (i in 1:nhh) {
      y=choiceAtt[choiceAtt[,1]==hh[i],2]
      nobs=length(y)
      X=as.matrix(choiceAtt[choiceAtt[,1]==hh[i],c(3:16)])
      lgtdata[[i]]=list(y=y,X=X)
            }

cat("Finished Reading data",fill=TRUE)
fsh()


Data=list(lgtdata=lgtdata,Z=Z)
Mcmc=list(R=20000,sbeta=0.2,keep=20)
out=rhierBinLogit(Data=Data,Mcmc=Mcmc)
```

The first and second lines of the file "`run rhierBinLogit.R`" reads the data into the R package, and the last line of the file calls a subroutine to execute the Markov chain Monte Carlo computations and return the results. The remaining lines of the file organize the data and set parameters for analysis.

Commands in "run rhierBinLogit.R" can be copied and pasted into the R package from a text editor (e.g., Vim), or can be invoked in batch mode by sourcing the entire file. We begin our tutorial using the cut and paste option to control the program. Use of the "source" command requires that the R package have access to all the subroutines needed in the analysis, which have not yet been explained.

The R response to the first two commands is to return the prompt ">":

```
>data(bank)
>choiceAtt=bank$choiceAtt
>Z=bank$demo
>
```

indicating the command has been successfully executed. The objects choiceAtt and Z are now available for use. Both are data frames. R stores the data as an array whose dimension can be determined from the "dim" command:

```
> dim(choiceAtt)
[1] 14799    16
> dim(Z)
[1] 946   4
```

```
>
```

indicating that choiceAtt is comprised of 14,799 rows and 16 columns, and d has 946 rows (one for each respondent) and four columns. More information about these variables can be obtained with the command "str" for "structure":

```
> str(choiceAtt)
`data.frame':   14799 obs. of  16 variables:
 $ id    : int  1 1 1 1 1 1 1 1 1 1 ...
 $ choice: int  1 1 1 1 1 1 1 1 0 1 ...
 $ d1    : int  1 1 1 0 0 0 0 0 0 0 ...
 $ d2    : int  0 0 0 0 0 0 0 0 0 0 ...
 $ d3    : int  0 0 0 0 0 0 0 0 0 0 ...
 $ d4    : int  -1 1 0 0 0 0 0 1 -1 0 ...
 $ d5    : int  0 -1 1 0 0 0 0 0 1 -1 ...
 $ d6    : int  0 0 -1 0 0 0 0 0 0 1 ...
 $ d7    : int  0 0 0 1 1 -1 -1 0 0 0 ...
 $ d8    : int  0 0 0 0 0 1 1 0 0 0 ...
 $ d9    : int  0 0 0 -1 1 -1 1 0 0 0 ...
 $ d10   : int  0 0 0 0 -1 0 -1 0 0 0 ...
 $ d11   : int  0 0 0 0 0 0 0 0 0 0 ...
 $ d12   : int  0 0 0 0 0 0 0 0 0 0 ...
 $ d13   : int  0 0 0 0 0 0 0 -1 -1 -1 ...
 $ d14   : int  0 0 0 0 0 0 0 0 0 0 ...
> str(Z)
`data.frame':   946 obs. of  4 variables:
 $ id    : int  1 2 3 4 6 7 8 9 10 11 ...
 $ age   : int  60 40 75 40 30 30 50 50 50 40 ...
 $ income: int  20 40 30 40 30 60 50 100 50 40 ...
 $ gender: int  1 1 0 0 0 0 1 0 0 0 ...
>
```

The data array, Z, contains demographic variables. We must mean center these variables so that the mean of the random-effects distribution can be interpreted as the average respondent's part-worths. To do this, we replace the first column of Z with the intercept vector and mean center age, income, gender.

```
> Z[,1]=rep(1,nrow(Z))
> Z[,2]=Z[,2]-mean(Z[,2])
> Z[,3]=Z[,3]-mean(Z[,3])
> Z[,4]=Z[,4]-mean(Z[,4])
> Z=as.matrix(Z)
```

The data array, choiceAtt, contains a respondent identifier in column1, the choice indicator in column 2, and a set of explanatory variables in columns 3-16. Each respondent provided between 13 to 17 responses to product descriptions, so it is not possible to format the data as a high dimensional array. We instead form a data list by first using the commands:

```
> hh=levels(factor(choiceAtt$id))
> nhh=length(hh)
>
```

to identify the unique respondent indicators (hh) and to determine the number of respondents in the study. The output from the first of these two commands is a variable "hh" that denotes "households" containing a list of unique id codes:

```
> str(hh)
 chr [1:946] "1" "2" "3" "4" "6" "7" "8" "9" "10" "11" "12" ...
> nhh
[1] 946
>
```

The quotes (" ") around the numbers indicate that the levels of the respondent id's are stored as a vector of characters, in contrast to the elements of z that are stored as integers. The variable "nhh" is a scalar with value equal to 946, the number of respondents in the study.

The variables "hh" and "nhh" are used to create a data structure that allows easy access to each respondent's data. This is accomplished by creating a data list for each respondent that contains their choice vector (y_h) and matrix of attribute descriptors (X_h) whose lengths correspond to the number of responses. The data structure is named "Data", and is created by iterating a set of commands using a loop:

```
> hh=levels(factor(choiceAtt$id))
> nhh=length(hh)
>
> lgtdata=NULL
>
> for (i in 1:nhh) {
+ y=choiceAtt[choiceAtt[,1]==hh[i],2]
+ nobs=length(y)
+ X=as.matrix(choiceAtt[choiceAtt[,1]==hh[i],c(3:16)])
+ lgtdata[[i]]=list(y=y,X=X)
+ }
> Data=list(lgtdata=lgtdata,Z=Z)
>
```

The NULL command initializes the variable Data as a "list" with zero length, and the loop beginning with the "for" command expands this list to contain respondents' data. Upon completion of the loop, the variable Data contains 946 elements, with each element containing the respondent's data. The first element of Data is comprised of two variables, y and X, lgtdata[[1]]$y and lgtdata[[1]]$X as follows:

```
Data$lgtdata[[1]]
$y
 [1] 1 1 1 1 1 1 1 1 0 1 1 1 0 1 1 0

$X
   Med_FInt  Low_FInt  Med_VInt  Rewrd_2  Rewrd_3  Rewrd_4  Med_Fee  Low_Fee  Bank_B
Out_State
1          1         0         0       -1        0        0        0        0       0
0
2          1         0         0        1       -1        0        0        0       0
0
3          1         0         0        0        1       -1        0        0       0
0
4          0         0         0        0        0        0        1        0      -1
0
5          0         0         0        0        0        0        1        0       1
-1
6          0         0         0        0        0        0       -1        1      -1
0
7          0         0         0        0        0        0       -1        1       1
-1
8          0         0         0        1        0        0        0        0       0
0
9          0         0         0       -1        1        0        0        0       0
0
10         0         0         0        0       -1        1        0        0       0
0
11         0         0         0       -1        0        0        1       -1       0
0
12         0         0         0        1       -1        0       -1        0       0
0
13         0         0         0       -1        0        0        0        0       0
0
14         0         0         0       -1        0        0        0        0       0
0
15         0         0         0        0        0        0        0        0      -1
0
16         0         0         0        0        0        0        0        0       1
0
   Med_Rebate High_Rebate High_CredLine Long_Grace
1           0           0             0          0
2           0           0             0          0
3           0           0             0          0
4           0           0             0          0
5           0           0             0          0
6           0           0             0          0
7           0           0             0          0
8           0           0            -1          0
9           0           0            -1          0
10          0           0            -1          0
11          0           0             0          0
12          0           0             0          0
13         -1           0             0          0
14          1          -1             0          0
15          0           0            -1          0
16          0           0            -1          0
```

which corresponds to the data for the first respondent. Moreover, choices for the respondents are stored as vectors, and the (differenced) attribute levels (X) are stored as a matrix so that matrix operations (e.g., matrix multiplication) can be performed:

```
> is.vector(Data$lgtdata[[1]]$y)
[1] TRUE
> is.matrix(Data$lgtdata[[1]]$X)
[1] TRUE
```

The remaining lines of code are used to i) write comments to the screen so that the user can monitor progress of the program and ii) create other lists that are used within the subroutine that is called.

The list "Mcmc" is a list of parameters that control the Markov chain:

```
> Mcmc=list(R=20000,sbeta=0.2,keep=20)
>
```

where "R" is the number of iterations of the chain, "sbeta" is used to control the step size of the Metropolis-Hastings random-walk chain, and "keep" indicates the proportion of draw retained for analysis. If keep=1, every draw is kept. For keep=20, every 20^{th} draw is keep, and the other 19 draws are discarded from analysis. The "keep" variable is useful for reducing the memory requirements of the program. Contents of "Mcmc" can be checked by entering this variable name (Mcmc) at the command prompt:

```
> Mcmc
$R
[1] 20000

$sbeta
[1] 0.2

$keep
[1] 20

>
```

Subroutines in R

Subroutines are used in R and other computing environments to organize computations that are repetitively performed. Examples include the evaluation of the likelihood of the data associated with equation (A.1) and computations associated with generating draws from the inverted Wishart distribution for covariance matrices. Subroutines in R are comprised of three basic components: i) the calling arguments; ii) commands for computations; and iii) a list of values that are returned. The subroutine then computes the log of the likelihood for equation (A.1) is:

```
loglike = function(y,X,beta) {
prob = exp(X%*%beta)/(1+exp(X%*%beta))
prob = prob*y + (1-prob)*(1-y)
sum(log(prob))
}
```

The variables passed to the function are y, the vector of choices, X, the matrix of differenced attributes, and beta, a vector of coefficients. The second line of the subroutine computes the probability according to equation **Error! Reference source not found.**. The command "%*%" is interpreted as "matrix multiplication" in R, and the expression "X%*%beta" yields a vector of elements, each of which are exponentiated, and divided by the expression in the

denominator on an element by element basis. Results are stored in a variable "prob" which is a vector of choice probabilities for the observations.

The third line of the subroutine computes the probabilities of the observed choices. If the respondent selected the first choice alternative, then the choice probability need not be changed. However, if the second choice alternative was selected, the associated probability in the likelihood becomes (1-Pr(y=1)). The computations on the third line on an element by element basis using standard notation for multiplication "*", addition "+" and subtraction "-".

Finally, the fourth line of the subroutine computes the log of the likelihood of beta given the data. The last line of the subroutine is what is passed back, or returned, to the calling statement. In this example, it is simply the value of the log likelihood. In other instances, a list will be returned. Note that the statements of the likelihood are contained within the curved brackets "{" and "}". In addition, it should be noted that the subroutine is written to avoid looping over the observations, significantly improving the speed of execution.

The subroutine is called within an R program with a command such as:

```
lognew = loglike(Data[[1]]$y,Data[[1]]$X,betan)
```

which passes the first respondent's data (y, X) and a parameter vector "betan" to the subroutine. The log likelihood is returned and stored in the variable "lognew."

Subroutines are entered into the R statistical package using the "Source R code …" command located under the "File" tab on the toolbar at the top of the R window. The code for subroutines is saved in a text file (using Vim, notepad, or other text editor), and, once "sourced" into the R system, can be used as functions in other programs. When you install our package, ***bayesm***, you will have access to a large number of pre-defined functions the function `rhierBinLogit` used in this example.

The last line of the file "run rhierBinLogit.R":

```
out=rhierBinLogit(Data,Mcmc)
```

calls a function named `rhierBinLogit` with arguments "Data" and "Mcmc" as defined above. The result is stored in the variable "out." If the function `rhierBinLogit` returns a scalar value on the last line, as with the function `loglike`, then `out` will contain that scalar value. If the last line of the subroutine is a matrix, then `out` will store the matrix. However, if the last line is a list (which it is in this subroutine), then "out" will contain a list of variables that can be accessed using the "$" convention, i.e., `out$varname`.

Hierarchical Bayes Binary Logit R Code

The complete specification of the HB binary logit model augments the likelihood in equation (A.1) with a random-effects distribution of heterogeneity for β_h:

(A.2) $$\beta_h \sim \text{Normal}(\Delta' z_h, V_\beta)$$

or

(A.3) $$B = Z\Delta + U \qquad u_i \sim \text{Normal}(0, V_\beta)$$

where u_i and β_i are the i^{th} rows of B and U.

The mean of the random-effects distribution is dependent on the values of the demographic variables, z_h, and the estimated matrix of coefficients Δ. The prior distribution for Δ is:

(A.4) $$\text{vec}(\Delta \mid V_\beta) \sim \text{Normal}(\text{vec}(\bar{\Delta}), A^{-1} \otimes V_\beta)$$

and the prior on the covariance matrix is:

(A.5) $$V_\beta \sim \text{IW}(\nu, V_0)$$

The function `rhierBinLogit` extracts the relevant information from the calling arguments "Data" and "Mcmc," initializes storage space for the draws, and then executes the computations associated with the Markov chain. Information contained in the calling arguments are copied into local variables.

To display the code in `rhierBinLogit`, type `rhierhBinLogit` at the command prompt in R. This will display the code in the R console window.

The first part of `rhierBinLogit` checks for valid arguments and implements defaults. Here we are using the defaults for all Prior settings.

The heart of the function starts with allocation of storage for the draws.

```
Vbetadraw=matrix(double(floor(R/keep)*nvar*nvar),ncol=nvar*nvar)
betadraw=array(double(floor(R/keep)*nlgt*nvar),dim=c(nlgt,nvar,fl
oor(R/keep)))
Deltadraw=matrix(double(floor(R/keep)*nvar*nz),ncol=nvar*nz)
oldbetas=matrix(double(nlgt*nvar),ncol=nvar)
oldVbeta=diag(nvar)
oldVbetai=diag(nvar)
oldDelta=matrix(double(nvar*nz),ncol=nvar)

betad = array(0,dim=c(nvar))
betan = array(0,dim=c(nvar))
reject = array(0,dim=c(R/keep))
llike=array(0,dim=c(R/keep))
```

where `Vbetadraw` are the saved draws of the random-effects covariance matrix (V_β), `betadraw` are the saved draws of the respondent-level coefficients (β_h), Deltadraw contains the draws of regression coefficients in the random-effects distribution (Δ). A total of

R/keep iterations are saved, and the draw of V_β is stored as a row vector of dimension nxvar^2.

Current draws of the variables are referred to as "old" variables, and take on the names "oldbetas," "oldVbeta," "oldDelta," and "oldVbetai," the later of which is equal to V_β^{-1}. The variables "betad" and "betan" are the "default" and "new" draws of beta used in the Metropolis-Hastings step of the algorithm. The variables "reject" and "llike" are summary measures of the performance of the chain and will be discussed later.

The function `init.rmultiregfp` is then called to set up parameters for the multivariate regression:

```
#
# set up fixed parm for the draw of Vbeta, Delta=Delta
#
Fparm=init.rmultiregfp(Z,ADelta,Deltabar,nu,V0)
```

Variables that monitor the speed of the chain, and used to write out time-to-completion information to the screen are initialized prior to the start of the first iteration:

```
itime=proc.time()[3]
cat("MCMC Iteration (est time to end - min)",fill=TRUE)
flush.console()
```

and are found at the end of the iteration:

```
if((j%%1000)==0)     {
ctime=proc.time()[3]
timetoend=((ctime-itime)/j)*(R-j)
cat(" ",j," (",round(timetoend/60,1),")",fill=TRUE)
flush.console() }
```

The "%%" is a "mod" operator in R, so time-to-completion information will be written to the R window every 500th iteration.

The algorithm for estimating the model proceeds in two major steps for each of R iterations. The first step involves generating draws of β_h for each respondent, h=1,…,nhh:

```
for (j in 1:R) {
      rej = 0
      logl = 0
      sV = sbeta*oldVbeta
      root=t(chol(sV))

#     Draw B-h|B-bar, V

      for (i in 1:nlgt) {

            betad = oldbetas[i,]
            betan = betad + root%*%rnorm(nvar)
```

```
# data
            lognew = loglike(Data[[i]]$y,Data[[i]]$X,betan)
            logold = loglike(Data[[i]]$y,Data[[i]]$X,betad)
# heterogeneity
logknew  =  -.5*(t(betan)-Demo[i,]%*%oldDelta)  %*%  oldVbetai  %*%
(betan-t(Demo[i,]%*%oldDelta))

logkold  =  -.5*(t(betad)-Demo[i,]%*%oldDelta)  %*%  oldVbetai  %*%
(betad-t(Demo[i,]%*%oldDelta))

# MH step
            alpha = exp(lognew + logknew - logold - logkold)
            if(alpha=="NaN") alpha=-1
            u = runif(n=1,min=0, max=1)
            if(u < alpha) {
                  oldbetas[i,] = betan
                  logl = logl + lognew } else {
                  logl = logl + logold
                  rej = rej+1  }
            }
```

A random-walk Metropolis-Hastings algorithm is used here, where the candidate vector of coefficients, β_h^n, is obtained by perturbing the existing vector of coefficients, β_h^d by a normal draw with mean zero and covariance proportional to the current draw of the covariance matrix, sbeta * V_β. The likelihood of the data is evaluated for both old and new coefficient vector, is multiplied by the density contribution from the distribution of heterogeneity, and used to form alpha (α) the acceptance probability for the Metropolis-Hastings algorithm. A uniform draw (u) is generated, and if u is less than alpha, the new vector of coefficients is accepted, otherwise it is rejected. The variable "logl" is used to accumulate a estimate of the log likelihood of the data evaluated at the posterior draws, and the variable "rej" monitors the proportion of draws that are accepted by the algorithm. This set of computations is computed for each respondent on each iteration of the Markov chain.

The second major step generates the draws of the matrix of regression coefficients Δ and the covariance matrix V_β:

```
#     Draw B-bar and V as a multivariate regression
out=rmultiregfp(oldbetas,Z,Fparm)
oldDelta=out$B
oldVbeta=out$Sigma
oldVbetai=chol2inv(chol(oldVbeta))
```

which employs the function `rmultiregfp` which is part of our package, ***bayesm***. It is therefore important to "source" this file prior to estimating the HB logit model so that the subroutine is available for use.

The remaining statements in the `rhierBinLogit` function are used to save the draws into the appropriate storage locations. The last line of the subroutine, which specifies the information passed back to the calling program, is a list:

```
list(betadraw=betadraw,Vbetadraw=Vbetadraw,Deltadraw=Deltadraw,ll
ike=llike,reject=reject)
```

which means that the saved draws are retrievable as "out$betadraw", "out$Vbetadraw", etc.

Obtaining Parameter Estimates

The Markov chain program is initiated by sourcing the file "`run rhierBinLogit.R`". The call to rhierBinLogit will print out the problem and then give a trace of every 100th iteration along with the estimated time to completion.

A total of 20,000 iterations of the Markov chain were completed in 130.69 minutes on Pentium 4 chip running at 1.78 GHz. This translates to about 150 iterations per minute for the 946 respondents and 14,799 observations. The structure of the variable "out" is:

```
> str(out)
List of 5
 $ betadraw : num [1:946, 1:14, 1:1000] 0.01251 -0.36642 ...
 $ Vbetadraw: num [1:1000, 1:196] 0.0691 0.0963 0.1058 ...
 $ Deltadraw: num [1:1000, 1:56] -0.0147 -0.0141 -0.0302 ...
 $ llike    : num [, 1:1000] -9722 -9520 -9323 -9164 -9007 ...
 $ reject   : num [, 1:1000] 0.621 0.614 0.607 0.605 0.608 ...
>
```

Figure 1 diplays a plot of the draws of the mean of the random-effects distribution, Δ. The plot is of the "intercept" elements of Δ:

```
> index=4*c(0:13)+1
> matplot(out$Deltadraw[,index],type="l",xlab="Iterations/20",ylab="
",main="Average Respondent Part-Worths")
>
```

Similarly, figure 2 displays a plot of the diagonal elements of the covariance matrix:

```
> index=c(0:13)*15+1
> index
 [1]   1  16  31  46  61  76  91 106 121 136 151 166 181 196
> matplot(out$Vbetadraw[,index],type="l",xlab="Iterations/20",ylab="
",main="V-beta Draws")
>
```

Figure 3 displays a plot of the log likelihood values that are useful for assessing model fit:

```
> plot(out$llike,type="l",xlab="Iterations/20",ylab=" ",
main="Posterior Log Likelihood")
>
```

Figure 4 displays a plot of the rejection rate of the Metropolis-Hastings algorithm:

```
> plot(out$reject,type="l",xlab="Iterations/20",ylab=" ",
main="Rejection Rate of Metropolis-Hastings Algorithm")
>
```

It is important to remember that the draws converge in distribution to the posterior distribution of the model parameters, in contrast to other forms of estimation (e.g., maximum likelihood and method of moments) where convergence is to a point. Upon convergence, the draws can be used to construct summary measures of the distribution, such as histograms, confidence intervals and point estimates. Figure 5 displays the distribution of heterogeneity for selected part-worths using draws of the individuals respondent part-worths, $\{\beta_h, h=1,\ldots,H\}$:

```
> par(mfrow=c(3,2),oma=c(2,0,3,0))
> plot(density(out$betadraw[,1,500:1000]),main="Medium Fixed
Interest",xlab=" ",xlim=c(-15,15),ylim=c(0,.25))
> plot(density(out$betadraw[,2,500:1000]),main="Low Fixed
Interest",xlab=" ",xlim=c(-15,15),ylim=c(0,.25))
> plot(density(out$betadraw[,8,500:1000]),main="Low Annual
Fee",xlab=" ",xlim=c(-15,15),ylim=c(0,.25))
> plot(density(out$betadraw[,10,500:1000]),main="Out-of-
State",xlab=" ",xlim=c(-15,15),ylim=c(0,.25))
> plot(density(out$betadraw[,12,500:1000]),main="High Rebate",xlab="
",xlim=c(-15,15),ylim=c(0,.25))
> plot(density(out$betadraw[,14,500:1000]),main="Long Grace
Period",xlab=" ",xlim=c(-15,15),ylim=c(0,.25))
> mtext("Distribution of Heterogeneity for Selected Part-
Worths",side=3,outer=TRUE,cex=1.3)
>
```

and Figure 6 displays estimates of the individual-level posterior distributions for one respondent (#250) using $\{\beta_{250}\}$:

```
> par(mfrow=c(3,2),oma=c(2,0,3,0))
> plot(density(out$betadraw[250,1,500:1000]),main="Medium Fixed
Interest",xlab=" ",xlim=c(-15,15),ylim=c(0,.35))
> plot(density(out$betadraw[250,2,500:1000]),main="Low Fixed
Interest",xlab=" ",xlim=c(-15,15),ylim=c(0,.35))
> plot(density(out$betadraw[250,8,500:1000]),main="Low Annual
Fee",xlab=" ",xlim=c(-15,15),ylim=c(0,.35))
> plot(density(out$betadraw[250,10,500:1000]),main="Out-of-
State",xlab=" ",xlim=c(-15,15),ylim=c(0,.35))
> plot(density(out$betadraw[250,12,500:1000]),main="High
Rebate",xlab=" ",xlim=c(-15,15),ylim=c(0,.35))
> plot(density(out$betadraw[250,14,500:1000]),main="Long Grace
Period",xlab=" ",xlim=c(-15,15),ylim=c(0,.35))
> mtext("Part-Worth Distributions for Respondent
250",side=3,outer=TRUE,cex=1.3)
>
```

Point estimates of the model parameters can be easily generated from the saved draws using the "apply" command. The mean and standard deviations of the coefficient matrix Δ are:

```
> t(matrix(apply(out$Deltadraw[500:1000,],2,mean),ncol=14))
             [,1]          [,2]          [,3]        [,4]
 [1,]  2.51252961 -0.0130934197  0.0109217374  0.10601398
 [2,]  4.88334510 -0.0253067253  0.0205866845  0.32386912
```

```
 [3,]  3.12238931  0.0017723979  0.0254455424 -0.35367756
 [4,]  0.06106729  0.0047622654  0.0005271013 -0.24818139
 [5,] -0.39090470  0.0216154219  0.0140768695 -0.22417571
 [6,] -0.29720158  0.0192823095  0.0159921716 -0.24252345
 [7,]  2.14173758 -0.0038188098  0.0024588301  0.66838042
 [8,]  4.15796844 -0.0095154517  0.0039886913  1.30239133
 [9,] -0.39701125  0.0007499859  0.0029704428  0.12386156
[10,] -3.75767707 -0.0029890846  0.0129988106 -0.05409753
[11,]  1.42600619 -0.0079048966  0.0027301444  0.23053783
[12,]  2.45602178 -0.0142630568  0.0208663278  0.37872870
[13,]  1.11570025 -0.0095763934 -0.0029792910  0.36773564
[14,]  3.39939098 -0.0201861790  0.0192602035  0.29644011

> t(matrix(apply(out$Deltadraw[500:1000,],2,sd),ncol=14))
            [,1]        [,2]        [,3]      [,4]
 [1,] 0.11498124 0.008107402 0.004534418 0.1943663
 [2,] 0.17897434 0.012265078 0.007142939 0.3418811
 [3,] 0.20341931 0.013359792 0.007444453 0.4252213
 [4,] 0.09873792 0.006967416 0.003527673 0.1590828
 [5,] 0.12626135 0.009705119 0.005814927 0.2495430
 [6,] 0.17520820 0.012854631 0.008517397 0.3767015
 [7,] 0.12045459 0.008761116 0.005717579 0.2454403
 [8,] 0.19746405 0.013435594 0.008607683 0.3741644
 [9,] 0.09928150 0.007453370 0.004176626 0.2140552
[10,] 0.23588829 0.018475045 0.009567594 0.4383213
[11,] 0.09206959 0.007181072 0.004703831 0.2367981
[12,] 0.16244095 0.011654455 0.008262366 0.3555747
[13,] 0.10998549 0.008830440 0.005254209 0.2575200
[14,] 0.13678767 0.009390693 0.005692094 0.2537742
>
```

Similar statistics are readily computed for the elements of V_β and any of the individual-level parameters $\{\beta_h\}$. Estimates of the posterior mean of Δ and V_β are summarized in tables 4 and 5.

Table 1
Description of the Data

Sample Size	946 Respondents
	14,799 Observations
Attributes and Attribute-Levels	
1. Interest Rate	High, Medium, Low fixed Medium variable
2. Rewards	The reward programs consisted of annual fee waivers or interest rebate reductions for specific levels of card usage and/or checking account balance. Four reward programs were considered.
3. Annual Fee	High, Medium, Low
4. Bank	Bank A, Bank B, Out-of-State Bank
5. Rebate	Low, Medium, High
6. Credit line	Low, High
7. Grace period	Short, Long
Demographic Variables	Age (years) Annual Income ($000) Gender (=1 if female, =0 if male)

Table 2
Sample Choice Observations

id	choice	d1	d2	d3	d4	d5	d6	d7	d8	d9	d10	d11	d12	d13	d14
1	1	1	0	0	-1	0	0	0	0	0	0	0	0	0	0
1	1	1	0	0	1	-1	0	0	0	0	0	0	0	0	0
1	1	1	0	0	0	1	-1	0	0	0	0	0	0	0	0
1	1	0	0	0	0	0	0	1	0	-1	0	0	0	0	0
1	1	0	0	0	0	0	0	1	0	1	-1	0	0	0	0
1	1	0	0	0	0	0	0	-1	1	-1	0	0	0	0	0
1	1	0	0	0	0	0	0	-1	1	1	-1	0	0	0	0
1	1	0	0	0	1	0	0	0	0	0	0	0	0	-1	0
1	0	0	0	0	-1	1	0	0	0	0	0	0	0	-1	0
1	1	0	0	0	0	-1	1	0	0	0	0	0	0	-1	0
1	1	0	0	0	-1	0	0	1	-1	0	0	0	0	0	0
1	1	0	0	0	1	-1	0	-1	0	0	0	0	0	0	0
1	0	0	0	0	-1	0	0	0	0	0	0	-1	0	0	0
1	1	0	0	0	-1	0	0	0	0	0	0	1	-1	0	0
1	1	0	0	0	0	0	0	0	0	-1	0	0	0	-1	0
1	0	0	0	0	0	0	0	0	0	1	0	0	0	-1	0
2	1	1	0	0	-1	0	0	0	0	0	0	0	0	0	0
2	1	1	0	0	1	-1	0	0	0	0	0	0	0	0	0
2	1	1	0	0	0	1	-1	0	0	0	0	0	0	0	0
2	1	0	0	0	0	0	0	1	0	-1	0	0	0	0	0

Table 3
Sample Demographic Observations

id	age	income	gender
1	60	20	1
2	40	40	1
3	75	30	0
4	40	40	0
6	30	30	0
7	30	60	0
8	50	50	1
9	50	100	0
10	50	50	0
11	40	40	0
12	30	30	0
13	60	70	0
14	75	50	0
15	40	70	0
16	50	50	0
17	40	30	0
18	50	40	0
19	50	30	1
20	40	60	0
21	50	100	0

Table 4
Posterior Mean of Γ

Attribute-Levels	Intercept	Age	Income	Gender
Medium Fixed Interest	2.513	-0.013	0.011	0.106
Low Fixed Interest	4.883	-0.025	0.021	0.324
Medium Variable Interest	3.122	0.002	0.025	-0.354
Reward Program 2	0.061	0.005	0.001	-0.248
Reward Program 3	-0.391	0.022	0.014	-0.224
Reward Program 4	-0.297	0.019	0.016	-0.243
Medium Annual Fee	2.142	-0.004	0.002	0.668
Low Annual Fee	4.158	-0.010	0.004	1.302
Bank B	-0.397	0.001	0.003	0.124
Out-of-State Bank	-3.758	-0.003	0.013	-0.054
Medium Rebate	1.426	-0.008	0.003	0.231
High Rebate	2.456	-0.014	0.021	0.379
High Credit Line	1.116	-0.010	-0.003	0.368
Long Grace Period	3.399	-0.020	0.019	0.296

Table 5
Posterior Mean of V_β

Attribute-Levels														
Medium Fixed Interest	2.8	4.5	4.3	0.0	0.2	0.5	1.1	2.0	0.3	1.2	0.7	1.4	0.9	1.4
Low Fixed Interest	4.5	8.8	8.2	0.0	0.4	0.9	1.8	3.4	0.5	2.1	1.2	2.5	1.2	2.4
Medium Variable Interest	4.3	8.2	10.2	-0.1	0.5	1.2	1.9	3.7	0.1	1.5	1.5	2.9	2.1	3.0
Reward Program 2	0.0	0.0	-0.1	1.2	0.2	-0.2	-0.3	-0.5	0.0	0.5	-0.3	-0.5	-0.4	-0.5
Reward Program 3	0.2	0.4	0.5	0.2	2.0	1.8	-0.2	-0.7	0.5	1.4	-0.4	-0.9	0.4	-1.0
Reward Program 4	0.5	0.9	1.2	-0.2	1.8	3.0	-0.3	-0.8	0.2	0.3	-0.6	-1.2	0.3	-1.1
Medium Annual Fee	1.1	1.8	1.9	-0.3	-0.2	-0.3	4.1	6.9	1.4	4.4	1.0	2.2	2.3	2.6
Low Annual Fee	2.0	3.4	3.7	-0.5	-0.7	-0.8	6.9	13.5	2.7	8.1	2.1	4.7	4.3	5.1
Bank B	0.3	0.5	0.1	0.0	0.5	0.2	1.4	2.7	3.5	5.6	1.2	2.5	2.0	0.9
Out-of-State Bank	1.2	2.1	1.5	0.5	1.4	0.3	4.4	8.1	5.6	15.9	2.3	4.8	3.9	2.0
Medium Rebate	0.7	1.2	1.5	-0.3	-0.4	-0.6	1.0	2.1	1.2	2.3	2.4	4.0	2.0	2.1
High Rebate	1.4	2.5	2.9	-0.5	-0.9	-1.2	2.2	4.7	2.5	4.8	4.0	8.5	3.6	4.5
High Credit Line	0.9	1.2	2.1	-0.4	0.4	0.3	2.3	4.3	2.0	3.9	2.0	3.6	6.3	2.6
Long Grace Period	1.4	2.4	3.0	-0.5	-1.0	-1.1	2.6	5.1	0.9	2.0	2.1	4.5	2.6	5.0

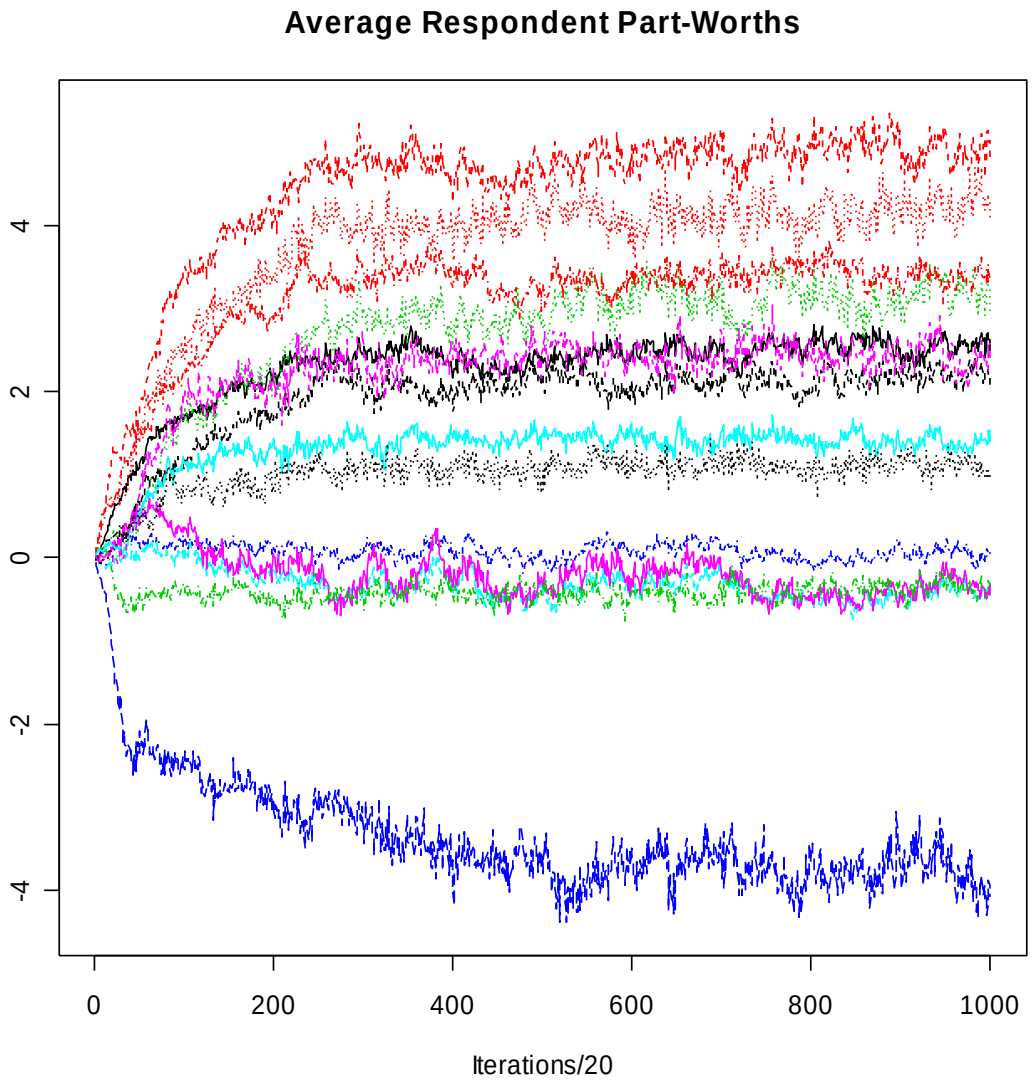


Figure 1. Draws of the mean of the random-effects distribution. Every 20th draw was retained for analysis.

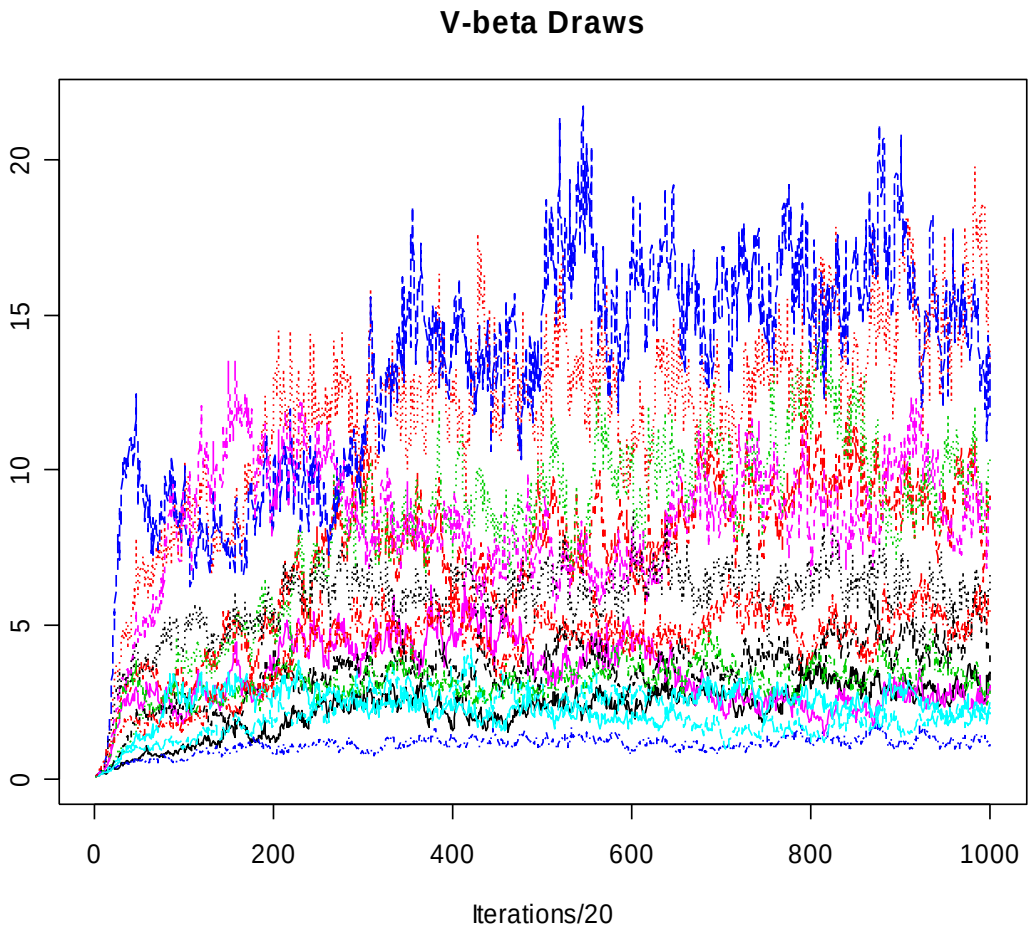


Figure 2. Draws of the diagonal elements of the covariance matrix of random-effects. Every 20th draw retained for analysis.

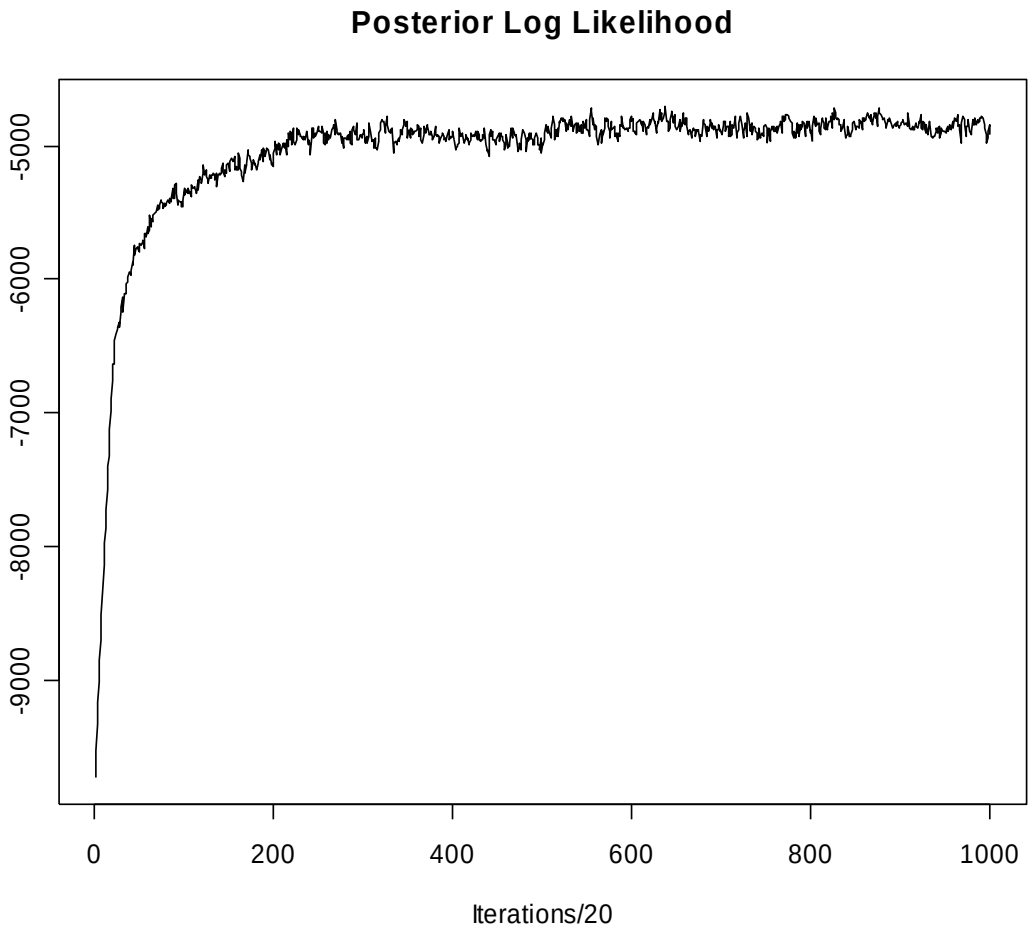


Figure 3. Values of the log-likelihood of the data evaluated at posterior draws of individual-level part-worth estimates (β_h). Every 20th draw retained for analysis.

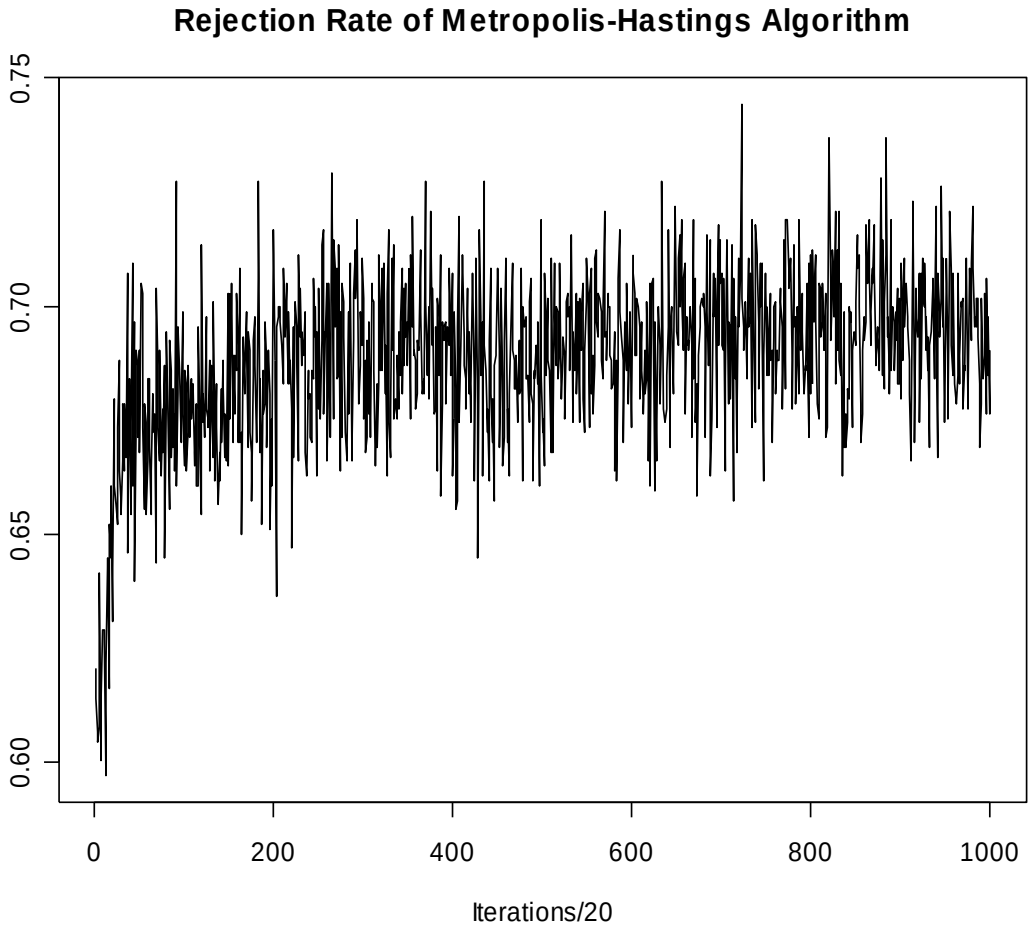


Figure 4. Rejection rate of Metropolis-Hastings Algorithm. Every 20th draw retained for analysis.

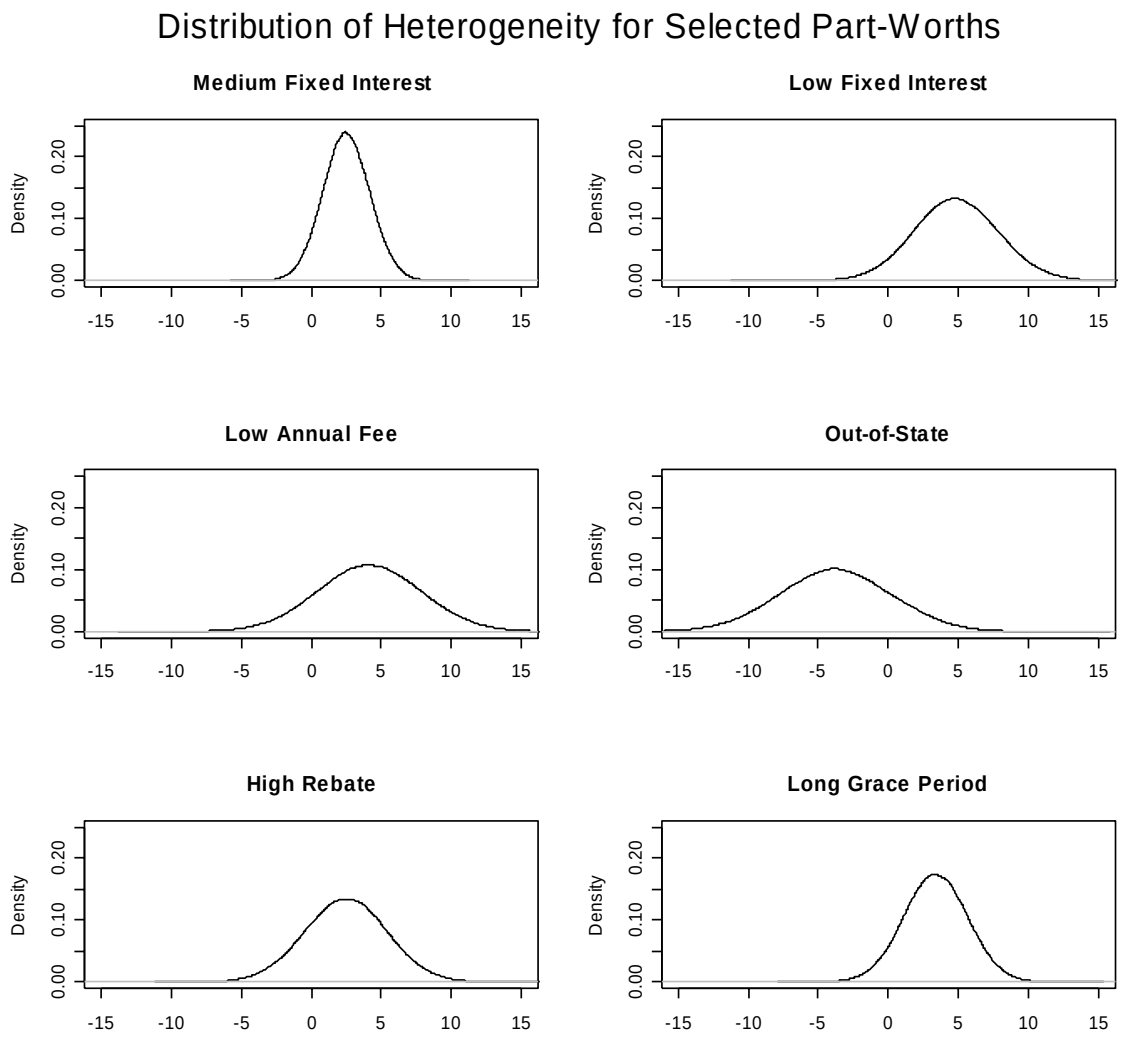


Figure 5. Posterior distribution of heterogeneity.

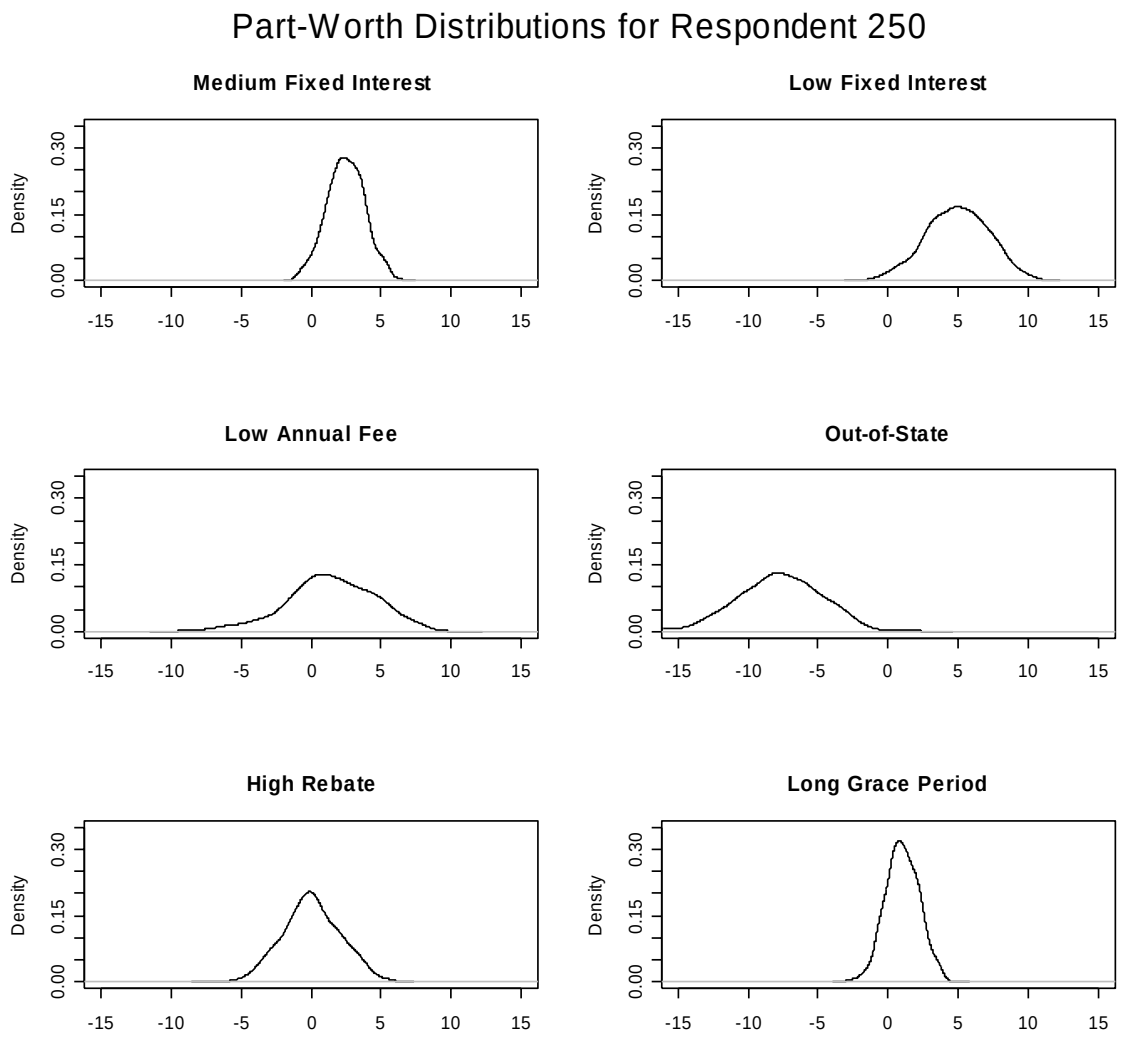


Figure 6. Posterior distribution of part-worths for respondent #250.