

# An Introduction to R

Luke Keele  
Nuffield College, Oxford University

February 24, 2006

## 1 Background

### 1.1 What is R?

R is a platform for the object-oriented statistical programming language S. It is widely used in statistics and has become quite popular in political science over the last decade. R is a shareware version of S-plus, and they are quite similar. Essentially R can be used as either a matrix-based programming language or as a standard statistical package that operates much like STATA or SPSS.

### 1.2 Where Can I get R?

The beauty of R is that it's shareware, so it's free to anyone. To obtain R for Windows (or Mac) go to The Comprehensive R Archive Network (CRAN) at <http://www.r-project.org>. Just download the executable file, and it will install itself.

## 2 Getting Started

Once you have installed R, there will be an icon on your desktop. Double click it and R will start up. You will notice the R does have a few pull-down menus, but mostly commands in R are entered on the command line:

>

Some preliminaries on entering commands.

- Expressions and commands in R are case-sensitive.
- Command lines do not need to be separated by any special character like a semicolon as in Limdep, SAS, or Gauss.
- Anything following the pound character (#) R ignores as a comment.
- An object name must start with an alphabetical character, but may contain numeric characters thereafter. A period may also form part of the name of an object. For example, x.1 is a valid name for an object in R.
- You can use the arrow keys on the keyboard to scroll back to previous commands.

**Saving Output.** Your output from a session in R can be saved using the `sink` command. To save your session to the file “Rintro.txt”:

```
> sink('a:\ Rintro.txt')
```

Now if you use the `print` command, your output will be saved to “Rintro.txt”. You can print strings of text like this:

```
> print('The mean of variable x is...')
```

and your “a:\ Rintro.txt” file will contain:

```
[1] 'The mean of variable x is...'
```

Another useful printing command is the `cat` command since it lets you mix object in R with text. For example, let's say we create the variable `x`:

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

Now we can print it as follows:

```
> cat("The mean of variable x is...", mean(x), '\n')
```

So now objects from R can be embedded into the statement you print. The character `\n` puts in a carriage return. You can also print any statistical output using the either `print` or `cat` commands. Remember, though, your output doesn't go to the log file unless you use one of the print commands. You can also copy and paste into Word or a text editor out of the R window. To turn off the sink command:

```
> sink()
```

**Objects.** R saves any object you create. To list the objects you have created in a session use either of the following commands:

```
> objects ()
```

```
> ls()
```

To remove all the objects in R type:

```
rm(list=ls(all=TRUE))
```

**Quitting.** To quit R type:

```
> q()
```

**Packages.** R has many useful add on components that are called packages. We will use a few packages in the practice session here. To load a package you simply type:

```
> library(packagename)
```

We will use this command shortly.

**Text Editing.** While one can type R commands one line at a time directly into the R console this is cumbersome and not at all efficient for writing programs. So instead most users type R commands into a text editor and then copy and paste them into R. Most simply this is done with the notepad. Type your R commands into the notepad and then cut and paste them into R. You can also use more advanced setups with text editors like Emacs or WinEdt.

**Reading in Data.** Getting data into R is quite easy. There are two primary ways to import data. The first is to read in a delimited text file with the `read.table` command. R will read in a variety of delimited files. (For all the options associated with this command type `?read.table` in R.) As an example read in the following dataset by Poe and Tate (1994) called `hmnrghts.txt`:

```
> hmnrghts <- read.table("e:\ hmnrghts.txt", header=TRUE, na="NA")
```

Remember you will have to specify the location of the file yourself depending on where you have it saved. The header option specifies whether the first line of the file contains the names of the variables, which here are: (1) **country**, (2) **democ**, a scale rating the level of democracy in a country, (3) **sdnew**, U.S. State Dept. scale of Political Terror, (4) **military**, a dummy variable for a military regime, (5) **gnpcats**, level of gnp in four categories, (6) **lpop**, log of population, (7) **civ.war**, a dummy variable for whether a country was involved in a civil war, and (7) **int.war**, a dummy variable for whether a country was involved in an international war. All variables are for 1993 only. Now type:

```
> hmnrghts
```

Our new dataset will print to the screen. This is not recommended with large datasets. To check the names of the variables in our dataset type:

```
> names(hmnrghts)
```

Now you can import data from another many other statistical packages. The foreign package in R makes it very easy to bring in data from other statistical packages, such as SAS, SPSS, and Stata. To bring in a dataset from Stata type:

```
> library(foreign)
```

```
> data.name <- read.dta(file.choose())
```

which lets you browse for a Stata dataset. Any data in Stata format that you select will be converted to R format. To read data in directly replace `file.choose()` with the path to the file and the file name like so:

```
> data.name <- read.dta('C:\mydata.dta')
```

One word of warning, the value labels from Stata do not always translate neatly. Often it is not a problem, but if you are having problems, try importing the data without value labels.

**Missing Data.** R designates missing values with `NA`. It translates missing values from other statistics packages into the `NA` missing format. To create a new data set with all the missing values deleted through listwise deletion, type:

```
>hmnrghts <- na.omit(hmnrghts)
```

**Arrays, Vectors, Data Frames, and Matrices.** R distinguishes between vectors, data frames and matrices. While vectors are indexed by length and matrices are indexed by rows and columns. Data frames are a matrix that R designates as a data set. With a data frame, the columns of the matrix can be referred to as variables. After reading in a data set, R will treat your data as a data frame, which means that you can refer to any variable within a data frame by adding `$VARIABLENAME` to the name of the data frame. For example, to print out the variable `country` to see which countries are in the dataset type:

```
> hmnrghts$country
```

Or you can use the `attach` command so you can use variable names individually. For example, after

```
> attach(hmnrghts)
```

you can now refer to `country` as an individual vector, without having to refer to the name of the data frame where it is located.

**Assignment and Arithmetic.** The way to assign values to a vector or matrix is to use the `<-` command. So to create a vector `a` with the specific values of 3, 4, and 5:

```
>a <- c(3,4,5)
```

Or to create a 10 by 1 vector of 1's:

```
> rep(1, 10)
```

To create a 4 by 4 matrix `b` with values of 3:

```
> b <- matrix(3, ncol=4,nrow=4)
```

To change specific values, you can index a vector `a` by the  $n$ th element, use `a[n]`, to index a  $n$  by  $k$  matrix `b` for the value  $x_{ij}$ , where  $i$  represents the row and  $j$  represents the column, use the following syntax `b[i, j]`. If you want to select all values of the  $j$ th column of `b` you can use `b[,j]`. For example, to return the 3rd column of matrix `b`, type:

```
> b[,3]
```

To get the 2nd row of `b`:

```
> b[2,]
```

The assignment command is also used when performing basic vector and matrix operations. For example, to assign vector **a** the sum of two vectors **b** and **c**, we type:

```
> a <- b + c
```

All other arithmetic operations, such as  $+$ ,  $-$ ,  $*$ ,  $/$ ,  $'$  and  $\log$  work similarly. Note that simple multiplication by  $*$  performs multiplication element by element, while  $\%*\%$  performs matrix multiplication. Some other useful matrix functions include:

To take the transpose of **b**:

```
> t(b)
```

To take the inverse of **b**:

```
> solve(b)
```

To obtain the length of a vector **a**:

```
> length(a)
```

To obtain the dimension of a matrix:

```
dim(b)
```

Can you find the dimensions of `hmrrights`? The dimension should be the number of variables by the number of cases.

To take the sum of a vector:

```
> sum(a)
```

To take the mean of a vector:

```
> mean(a)
```

To take the variance of a vector:

```
> var(a)
```

Or take the standard deviation with the `sd` command.

**Sequences.** To create a vector that contains values counting from 1 to 10, type:

```
> c(1:10)
```

A more general command is the `seq` command, which allows you to define the intervals of a sequence, as well as starting and ending values. For example to create a sequence from -2 to 1 in increments of .25:

```
> seq(-2,1, by=0.25)
```

```
[1] -2.00 -1.75 -1.50 -1.25 -1.00 -0.75 -0.50 -0.25 0.00 0.25 0.50 0.75 [13]  
1.00
```

**Drawing Random Numbers.** R allows you to draw random numbers from a wide variety of distributions. Let's say you want to draw a scalar from the standard normal distribution: to draw  $\mathbf{a} \sim N(0, 1)$ , use the command `rnorm`:

```
> a <- rnorm(1)
```

To draw a 10 by 10 matrix  $\mathbf{b}$  from  $N(10, 4)$ :

```
> b <- matrix(rnorm(100, mean=10, sd=2), nrow=10, ncol=10)
```

To draw a vector with 10 values from a  $\chi^2$  distribution with four degrees of freedom:

```
> c <- rchisq(10,4)
```

For each distribution, R can also provide the PDF, CDF, and quantile function, for the normal distribution these are, respectively: `pnorm`, `dnorm` and `qnorm`.

**Sampling.** If you want to sample from a given object use the command `sample`. Suppose we want a sample of 10 ten numbers from  $\mathbf{b}$  our 10 by 10 matrix of random numbers:

```
> sample(b,10)
```

This gives you a vector of ten random elements from  $\mathbf{b}$ . However this is not the best way to perform a bootstrap. If you're interested in bootstrapping (and who isn't?), install the `boot` package as below and run some examples:

```
> library(boot)
```

```
> example(boot)
```

**Concatenation.** You can concatenate by column or by row with `cbind` and `rbind` commands. Suppose  $\mathbf{a}$  and  $\mathbf{b}$  are both  $n \times 1$  vectors:

```
> a <-c(3,4,5)
```

```
> b <-c(10,11,12)
```

```
> cbind(a,b)
```

**Apply.** The `apply` command is often the most efficient way to do vectorized calculations. For example, to calculate the means for all the variables in the humanrights data set:

```
> apply(as.matrix(hmnrghs[,3:7]),2,mean)
```

```

      a  b
[1,] 3 10
[2,] 4 11
[3,] 5 12
> rbind(a,b)

      [1,] [2,] [3,]
a      3    4    5
b     10   11   12

```

In the command here, we refer to columns 3 to 7 of the `hmnrights` data frame, which are variables 3 to 7. The `2` tells `R` to apply a mathematical function along the columns of the matrix, and `mean` is the function we want to apply to each column. If we used a `1` instead of a `2`, `R` would apply the calculation to the rows of the data frame. Any function defined in `R` can be used with the `apply` command.

Can you calculate the standard error of these variables using `apply`? One thing to note. If the data you import from `STATA` has value labels in it, the `apply` function will not work. It only works on variables that `R` has designated as a numeric object, which is why here I use the `as.numeric` command to temporarily coerce the `hmnrights` data frame into a matrix.

### 3 Logical Statements

Logical statements in `R` are evaluated as to whether they are `TRUE` or `FALSE`. Table 1 summarizes the different logical operators in `R`.

Table 1: Logical Operators in `R`

Operator	Means
<code>&lt;</code>	Less Than
<code>&lt;=</code>	Less Than or Equal To
<code>&gt;</code>	Greater Than
<code>&gt;=</code>	Greater Than or Equal To
<code>==</code>	Equal To
<code>=</code>	Not Equal To
<code>&amp;</code>	And
<code> </code>	Or

For example, suppose we wanted to know which countries have had a civil war and have above average GNP:

```
> civ.war>1 & gnpcats>2.6
```

returns a vector of `TRUE` and `FALSE` for every observation. In this case, we see that there are no countries with above average GNP that have been involved in a civil war (in 1992 anyway).

## 4 Recoding

Often we need to recode variables, and there are a variety of ways to do this in R. The basic syntax for creating mathematical transformations of variables follows the form of the example below. Suppose we want the actual population of each country instead of its logarithm:

```
> pop <- exp(lpop)
```

Any type of mathematical operator could be substituted in for `exp`. Another standard type of recoding we might want to do is the creation of a dummy variable that is coded as 1 if the observation meets certain conditions and 0 otherwise. For example, let's say instead of having categories of GNP, we just want to compare the highest category of GNP to all the others. We can do this in a couple of different ways:

```
gnp.dummy <- as.numeric(gnpcats>3)
```

Here we use a logical statement and modify it with the `as.numeric` statement which turns each `TRUE` into a 1 and each `FALSE` into a 0. This could be accomplished in another ways as well, but to demonstrate a second technique I'll use different type of example. Let's say instead of having the population of each country we wanted a 3 category ordinal level variable.

```
> pop.3 <- lpop
> pop.3[lpop <15.18] <- 1
> pop.3[lpop >= 15.18 & lpop < 17.11] <- 2
> pop.3[lpop >= 17.11] <- 3
```

In commands above, I have created a new variable from 1 to 3, with countries with a log population below 15 coded as one and so on. Of course, this syntax could of course be used to create a dummy variable or for other recoding.

There is one quirk in R when it comes to recoding variables or creating new ones. When you recode or create new variables, those changes are made to the global version of the data frame and not to the specific data frame that you think you are changing. To make changes to a specific data set detach it before making the changes and then re-attach it. To make the change above then you would type: `detach(hmnrghs)` before making the changes and then typing `attach(hmnrghs)` after the changes are completed.

### Factors

Categorical variables in R can be given a special designation as factors. If you designate a categorical variable as a factor, R will treat it as such in statistical operation and create dummy variables for each level when it is used in a regression. If you import a variable with no numeric coding, R will automatically treat the variable as a factor. For example, the variable `country` in the data set is automatically treated as a factor. Once R thinks a variable is a factor it has a series of special commands that can be used. For example, try:

```
levels(country)
```

This lists all the levels in the factor with their names. The levels of a factor need not be named. To change the levels of a factor to character strings, it exactly the same as recoding

the variable in the second example, but use quotes around the character string that you want the level of a factor to take on. To change which level is the first level (i.e. to change which category R will use as the reference category in a regression) use the `relevel` command. For example:

```
> country <- relevel(country, 'united states')
```

The first level in the factor and the reference category in a dummy variable regression.

## 5 Basic Models

### 5.1 Ordinary Least Squares

Estimating an OLS model in R is done with the `lm` command. Let's predict human rights violations as function of the variables `democ` and `gnpcats`:

```
> hmnrghts.model <- lm(sdnew ~ democ + gnpcats, data=hmnrghs)
```

So the dependent variable comes first and all independent variables, must be separated with the `+`. You also need to specify the data frame which R should use when it estimates the model. One thing to note. After typing the above command, R will not automatically print any output. To see the output you have to type the following command:

```
> summary(hmnrghts.model)
```

Which will give you the following output:

```
Call:
lm(formula = sdnew ~ democ + gnpcats)

Residuals:
Min       1Q   Median       3Q      Max
-2.4881   -0.5839   -0.1624    0.6421    3.0252

Coefficients:
              Estimate Std. Error t value Pr(> |t|)
(Intercept)   3.90950    0.17403   22.464 <2e-16 ***
democ         -0.09577    0.02370   -4.040 9.37e-05 ***
gnpcats       -0.32564    0.05879   -5.539 1.78e-07 ***
---
Signif codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 1 on 122 degrees of freedom
Multiple R-squared:  0.4105, Adjusted R-squared:  0.4009
F-statistic: 42.48 on 2 and 122 DF, p-value: 9.969e-15
```

The `hmnrghs.model` object also includes several other regression outputs including the `coefficients`, `residuals`, `cov.unscaled` (the variance-covariance matrix) and `fitted.values`. To see these outputs take the model name and use the `$` to append the object. For example,

to see only the coefficients from the model type:

```
> hmnrights.model$coefficients

(Intercept)   democ   gnpcats
3.90950      -0.09577  -0.32564
```

The `lm` command is also designed to allow you to do interactions on the fly. For example, to test for an interaction between democracy and the level of GNP we would use the following command:

```
> hmnrights.model2 <- lm(sdnew ~ democ + gnpcats + democ*gnpcats)
```

Type this command and then look at the results. You will see that R creates an interaction and includes it in the model.

## 5.2 General Linear Models

The `glm` command allows you to run a variety of models. To use the `glm` command you must specify your formula, the family, and the link function. For example, let's say we wanted to predict whether a country had a military regime as a function of GNP and population using a logit model:

```
> mil.model <- glm(military ~ lpop + gnpcats, family=binomial(link=logit))
> summary(mil.model)
```

Which will give you the following output:

```
Call:
glm(military ~ lpop + gnpcats, family=binomial(link=logit))

Deviance Residuals:
    Min       1Q   Median       3Q      Max
-0.9886   -0.7021  -0.2784  -0.1170   3.0956

Coefficients:
            Estimate Std. Error t value Pr(> |t|)
(Intercept)   2.7284     3.1318   0.871   0.3836
democ         -0.1676     0.1907  -0.879   0.3794
gnpcats       -0.9873     0.3509  -2.813   0.0049 **
---
Signif codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Dispersion parameter for binomial family taken to be 1)
Null deviance: 103.040 on 124 degrees of freedom
Residual deviance: 84.893 on 122 degrees of freedom
AIC: 90.893
```

Number of Fisher Scoring Iterations: 6

If you wanted to estimate a probit, you would use the exact same command, but substitute `in` for `probit` in the specification of the link function.

## 6 Graphics

Most graphing in R is done using the `plot` command which has the basic structure `plot(x,y)`. For example, let's say we wanted to plot human rights violations by the `lpop` variable. To do that we would use the following command:

```
> plot(lpop, sdnew)
```

This gives us Figure 1.

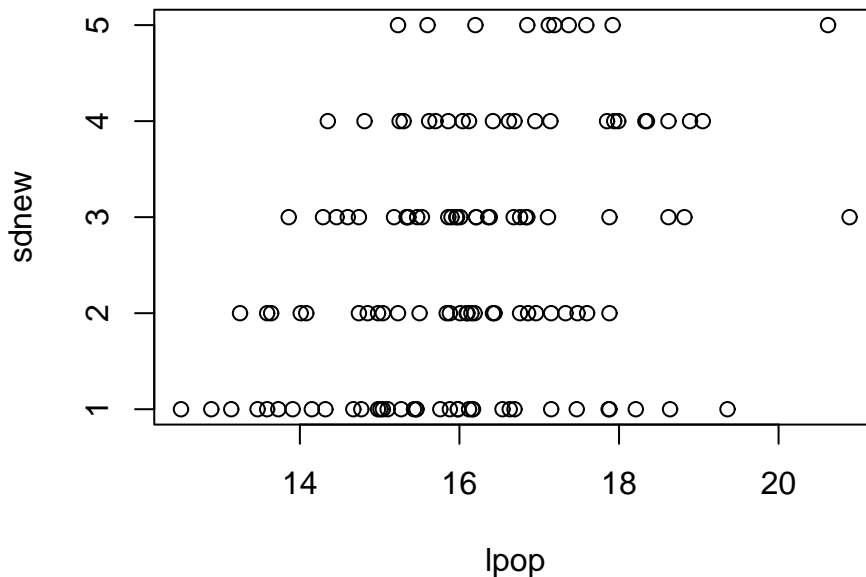


Figure 1: A Sample R Graphic

Given that `sdnew` is an ordinal variable it's harder to see the relationship between the two variables. Often it helps to add some random perturbations to the plot. This is easily done in R with the `jitter` command:

```
> plot(lpop, jitter(sdnew))
```

The graph in Figure 2 is much nicer. R also has a number of subcommands for the `plot` function to add options to your graphics:

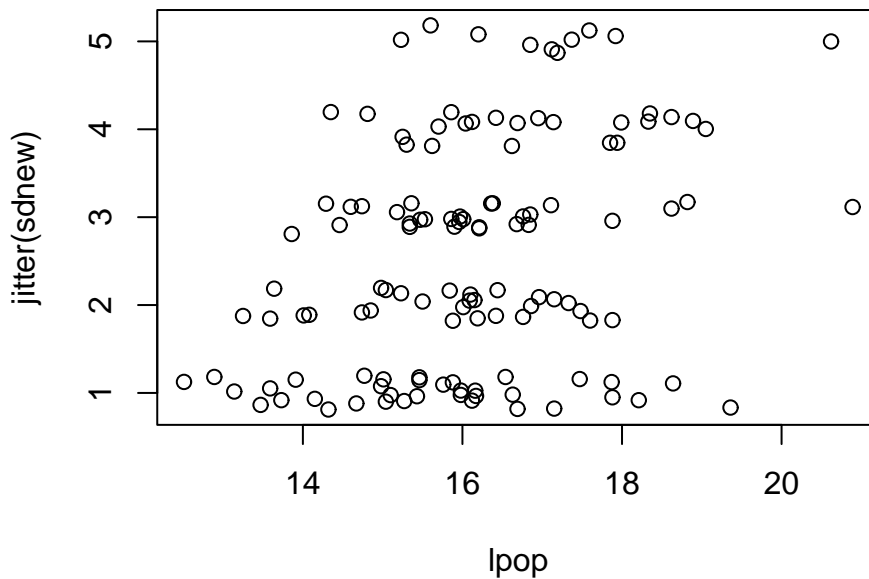


Figure 2: A Graphic with the Jitter Command

```
> plot(lpop,jitter(sdnew), xlab='Log of Population',
       ylab='Human Rights Violations',main='Human Rights Violations by Population')
```

This adds labels for the x and y axis as well as a main title. A subtitle can be added as well. R has a number of other graphic definitions that can be set to customize your graphs. Use `?plot` to see them all.

We can also easily plot elements from the models that we estimated. Let's plot the fitted values from the regression model we estimated against the one of the regressors, the measure of population.

```
> plot(hmnrghs.model$fit, lpop, xlab='Fitted Human Rights Violations',
       ylab='Population', main='Residual Plot')
```

Here we can see the fitted values of our model against one of the regressors, a standard test for heteroskedasticity in linear models.

Let's say we wanted multiple graphs on a single page in order to make comparisons. For example, let's say we wanted to look at the residual plots for both population and GNP from the regression model we estimated earlier. To do that we have to use the `par` command. The `par` command is a lower level graphing command which allows you to make a variety of adjustments to graphs. Type `?par` to see all that it controls. Figure 5 is an example of a graph done with the `par` command:

```
par(mfrow=c(2,1))
plot(lpop, hmnrghs.model$resid, ylab="OLS Residuals",
```

## Human Rights Violations by Population

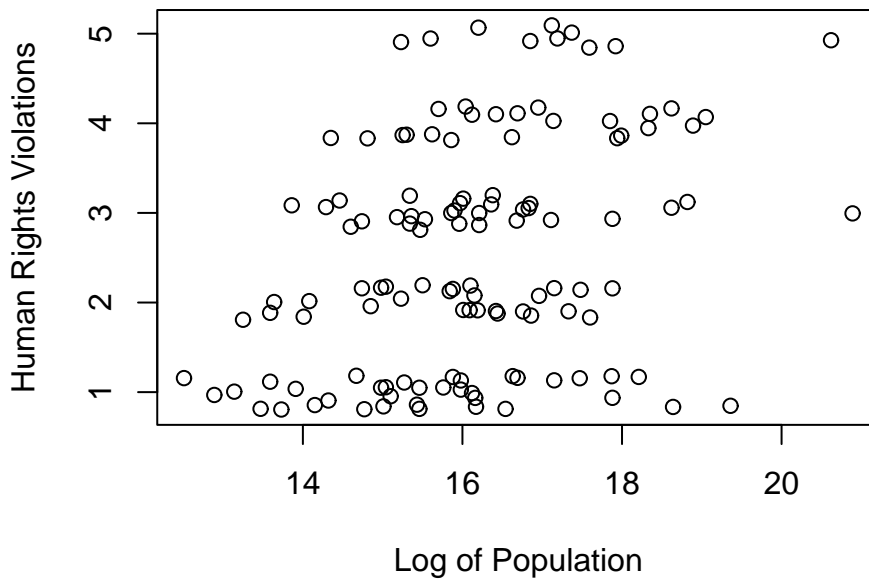


Figure 3: More R Graphics Commands

```
xlab="Population", main="Residual Plot")
plot(jitter(as.numeric(gnpcats)), hmrngts.model$resid, ylab="OLS Residuals",
xlab="GNP", main="")
```

Here the `par` command tells R to create a 2 by 1 set of graphs. You then need to supply two graphs. You can do up to eight graphs on a single page. By the looks of things here it would appear that there is some linear relationship between GNP and the fitted values in our model, indicating heteroskedasticity.

**Graphic Output.** The easiest way to get graphics out of R is to simply right-click on the figure itself and then choose to save the figure as either a metafile (for use in Word) or as a postscript file for use in LaTeX. The default size is for a full page graph. To resize the graph just use the mouse to resize the plot window. For LaTeX users who want to set the bounding box size use the `postscript` command. Before you plot the graph, type:

```
> postscript('FILENAME.eps', horizontal=FALSE width=#, height=#).
```

This will set the width and height of the graphic in inches. You may have to experiment with this at first to get the size you want. The graphics in this document have a bounding box that is 5 inches wide and 4 inches high. The `horizontal` command changes the orientation of the graphic from landscape to portrait orientation on the page. Change it to `TRUE` to have the graphic adopt a landscape orientation.

## Plot of Fitted Values

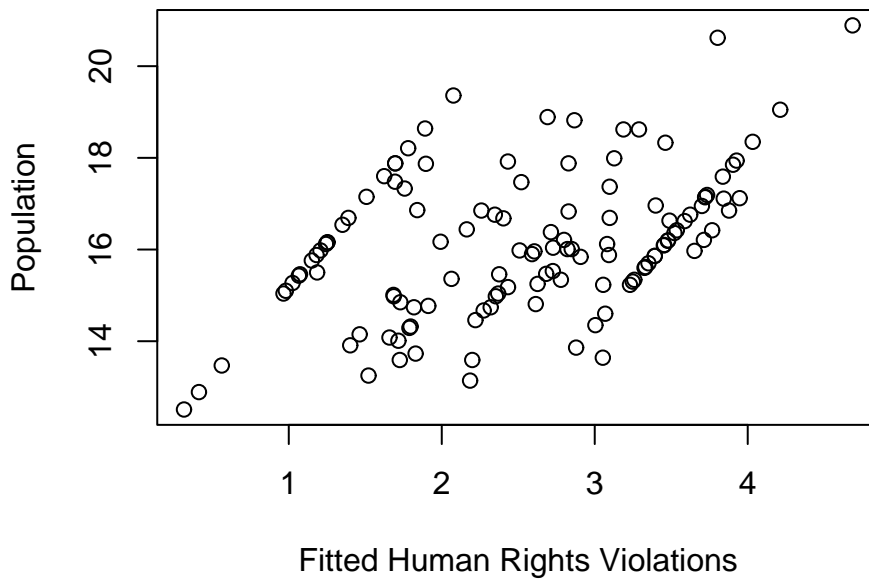


Figure 4: An OLS Fitted Values Plot

## 7 Loops

Loops are easy to write in R and can be used to repeat calculations. The basic structure for a loop is:

```
> for (i in 1:10) {COMMANDS}
```

To demonstrate how loops work, let's do a demonstration of the law of large numbers:

```
# First create a storage matrix
store <- matrix(NA,1000,1)
# Start the Loop for (i in 1:1000){
a <- rnorm(i)
store[i] <- mean(a)
}
plot(store, type='o')
```

In this simple program with a loop, I demonstrate the Law of Large Numbers. In each pass of the program, R samples from a  $N(0,1)$  distribution and then takes the mean of that sample. The sample size increases with each pass of the loop. As such, our plot should show the mean of the sample converging to the true mean of zero as the sample size increases.

In Figure 6 you can see the results for this simple program. Loops are necessary for many types of programs particularly if you want to do a Monte Carlo analysis. However, they should be avoided whenever possible, since they will slow down your program considerably. If possible try and use the `apply` command. R also supports while loops which follow a similar command

**Residual Plot**

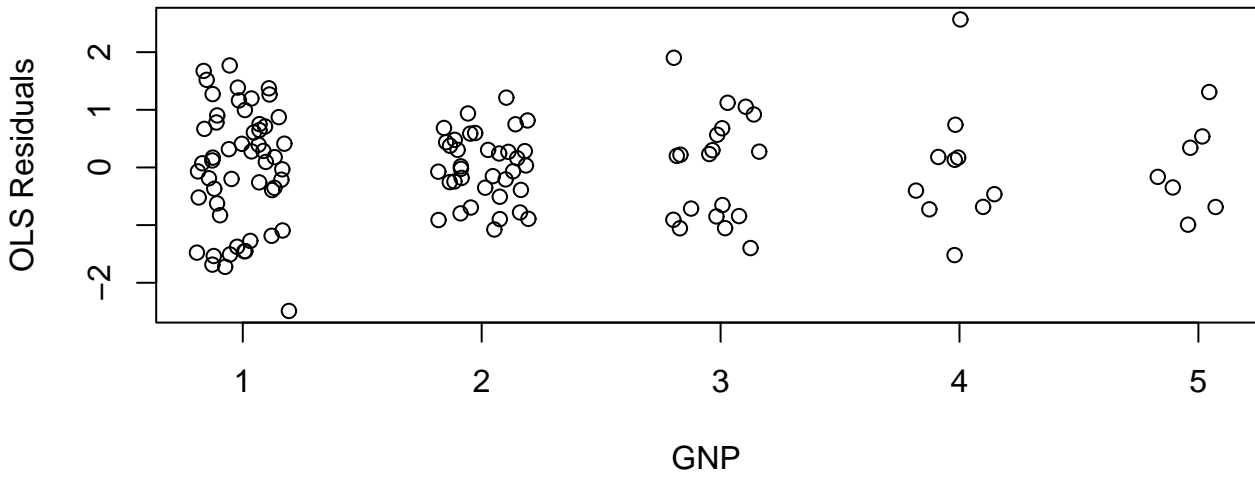
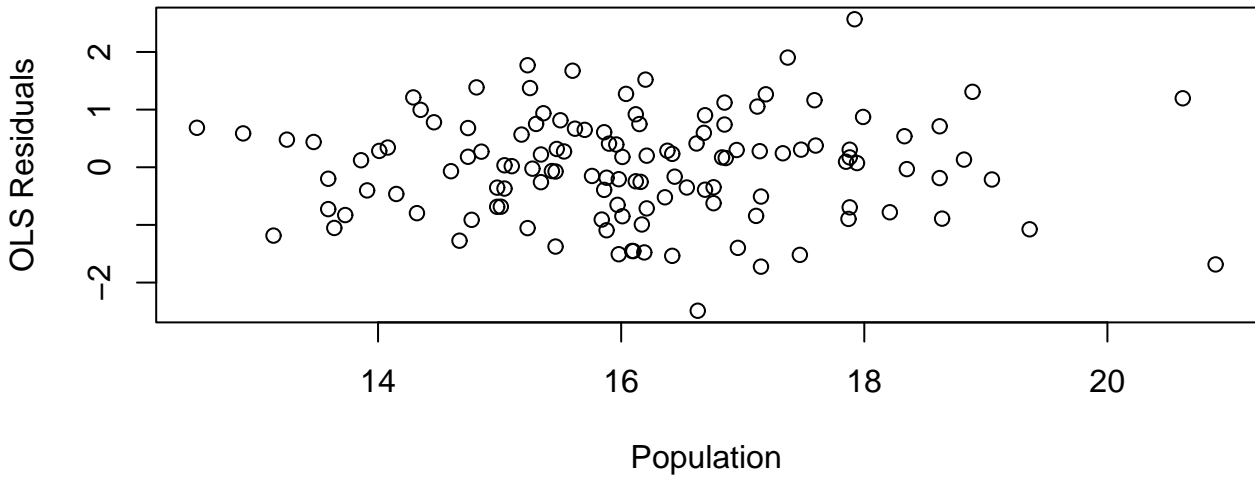


Figure 5: Including Multiple Graphs on the Same Page.

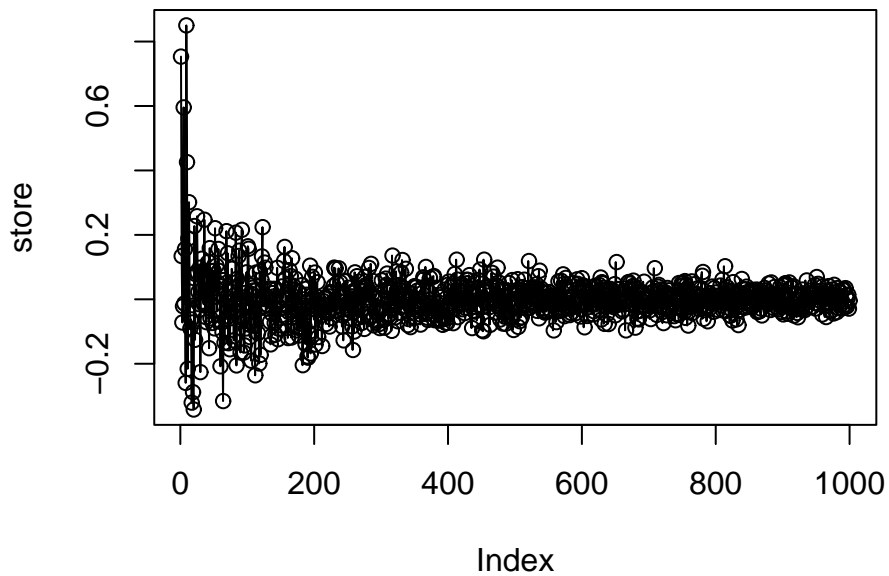


Figure 6: The Law of Large Numbers and Loops in Action

structure:

```

j <- 1
while(j < 10) {COMMANDS
j <- j + 1
}

```

## 8 Functions

Another way to program in R is to create your own functions. Surprisingly enough this is done with the `function` command. The `function` command uses the basic following syntax:

```
> function.name <- function{INPUTS}{YOUR ARGUMENTS}
```

Let's do a simple example. Let's say you wanted to make your own jitter function:

```
> jitter <- function(x){x + rnorm(length)(x)}
```

Now if we type:

```
> jitter(lpop)
```

Normal random errors will be added to each value of the `lpop` variable.

## 9 Where To Go From Here

The best place to go from here is straight to R. The more you use it for actual analysis the better you will be at R. When you get stuck the best place to look is the “Introduction to R” which is under the Help pull down menu under Manuals. The next place to look is Fox’s “Introduction to Regression in R and S-plus.” And for a more in-depth treatment see Venables and Ripley’s Modern Applied Statistics in S.

## 10 Exercises

1. Import the Human Rights data set into R (If you haven’t done so already and makes sure all missing values have been removed.).
2. Recode the `lpop` variable into a new variable of quartiles from 1-4.
3. Turn this new variable into a factor and label the levels of the factor with character strings.
4. Estimate an OLS regression with `sdnew` as the dependent variable. Use all of the other variables in the Human Rights data set as independent variables except `lpop` instead use the factor version of this variable.
5. Relevel the population factor variable and re-run the regression. What is the difference between these two regressions?
6. Plot the residuals against the fitted values for this model (Put the residuals on the y axis).
7. Finally re-estimate the logit model from section 5.2 except estimate this model as a probit model.

Turn in all R commands and results. Make your homework as neat as possible.