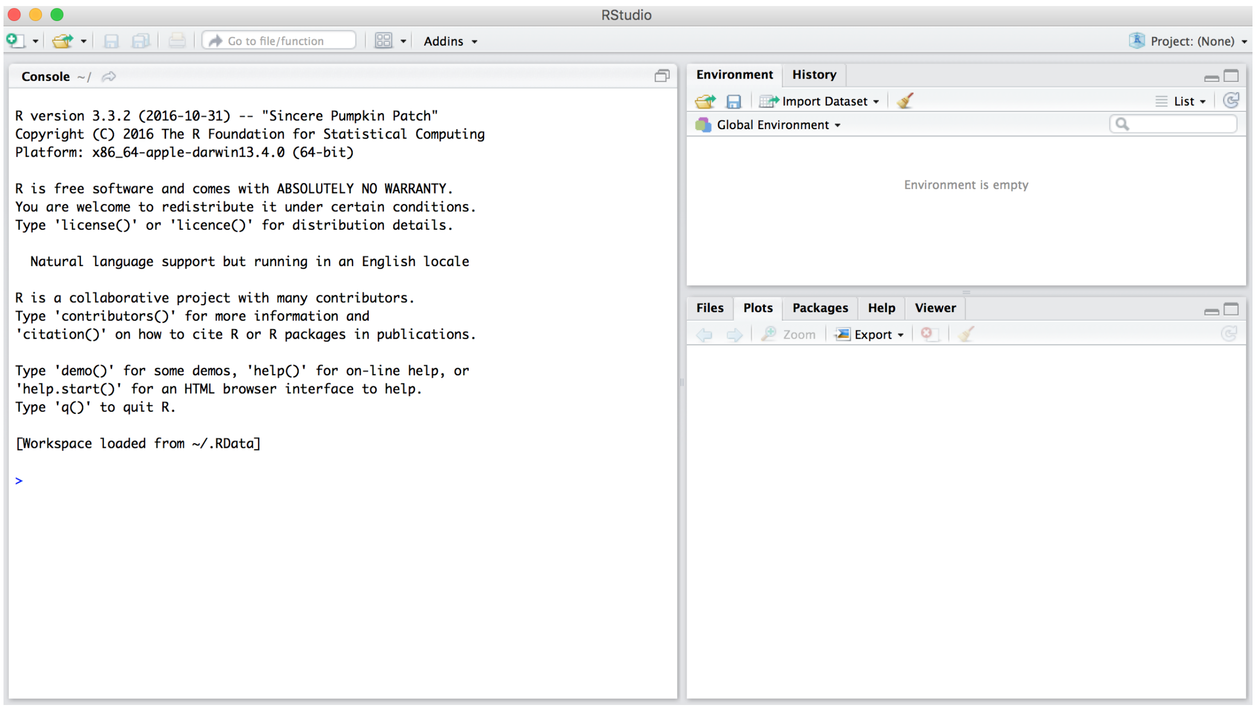
Intro to R

1 Introduction

This document provides a brief overview on R, a statistical programming language. In this document, we will learn how to use R using RStudio, which provides an interactive interface with R to make things (slightly) easier than using R in the command line with no graphics. To find instructions on how to install R and RStudio on your computer, you can follow the instructions at https://www.otexts.org/fpp/using-r (R and RStudio are already installed on all computers in the 4th floor labs of OEC). Once you have RStudio up and running, you should see something like this:



```
R version 3.3.2 (2016-10-31) -- "Sincere Pumpkin Patch"
Copyright (C) 2016 The R Foundation for Statistical Computing
Platform: x86_64-apple-darwin13.4.0 (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

  Natural language support but running in an English locale

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

[Workspace loaded from ~/.RData]

>
```

2 Using R as a Calculator

To start, let's learn how to use R in the simplest way possible, as a calculator. To do so, we can type mathematical formulas into the console, and R will compute the result. For example, suppose we wanted to add 2+2. The console would look like:

```
2+2

## [1] 4
```

We can also do more complicated formulas. Let's try:

$$2^3 \times \frac{8}{2}$$

In R, this looks like:

```
2^3*(8/2)

## [1] 32
```

In general, we use the following symbols:

Property	Symbol
Addition	a+b
Subtraction	a-b
Multiplication	a*b
Division	a/b
Exponentiation	a ˆ b
Natural Log	log(a)
Square Root	sqrt(a)

2.1 Examples

1. Compute $2 + 4$
2. Compute $\sum_{i=3}^{6} i$
3. Compute $\log(5)\sqrt{3}$
4. Compute $\left(\frac{10}{6}\right)^{log(4)}$

3 Creating and Using Variables

A more powerful feature of R (and most other statistical software packages) is that we can use it to store values for future reference or later use. In order to access a variable, we have to assign something (maybe a number, maybe some text) to it. If we try to call a variable that we have not yet assigned a value to, we get an error:

```
x

## Error in eval(expr, envir, enclos):  object 'x' not found
```

To assign a value to a variable, we use the symbol "$< -$" For example, let's assign the value of 16 to a variable, x (note that you can use any variable name you want, as long as it is continuous and begins with a letter...for example we could use x_1 or asdf32, but not 1_x or asdf 32):

```
x <- 16
```

Now when we type x into the console, it returns the value assigned to it:

```
x

## [1] 16
```

Once we have a value stored to a variable, we can perform functions on the variable. For example, suppose we wanted to compute the square root of 16. Since x has been assigned the value of 16, we could call:

```
sqrt(x)

## [1] 4
```

We can redefine a variable at any time. All we have to do is assign a new value to it. For example, suppose we now wanted x to be 49 instead of 16. We could write:

```
x<-49
```

Now when we call x in the console, it returns:

```
x

## [1] 49
```

4 Creating and Using Vectors

In addition to assigning singular values to variables, we can also create vectors. A vector is either a row or column that contains multiple values. One example of a vector that we will be working with frequently is a vector containing observations of a variable measured over time, such as the unemployment rate which is measured each month. We could use a vector to store all of these values.

If we only have a few values in mind, we can create a vector by brute force, assigning all values in the vectors manually. For example, suppose that we wanted to create a vector containing the numbers 1, 2, 3, and 4. As discussed in the previous section, we could assign these to any variable name we want as long as it is continuous (no spaces) and begins with a letter. Let's use the variable v:

```
v <- c(1,2,3,4)
v

## [1] 1 2 3 4
```

To create the vector, we use the function c(), which stands for **c**ollect. To separate the items in a vector, we use a comma. When the elements in a vector contain a pattern, it is often more convenient to use alternate notation. We can create vectors that follow any pattern using the seq() function, which takes three arguments: the beginning number, the ending number, and what to increment by. This function is different than the ones you have seen so far, since it takes three arguments (instead of only one). For example:

```
v <- seq(1,4,1)
v

## [1] 1 2 3 4
```

or:

```
v <- seq(1,3,0.5)
v

## [1] 1.0 1.5 2.0 2.5 3.0
```

Finally, sometimes we want to create a vector in which every element is the same. To do this, we can use the rep() function. The rep() function takes two arguments: the number to repeat, and how many elements you want the vector to have. For example, suppose we wanted create a vector with 10 elements, with each element being 0. We would write:

```
v <- rep(0,10)
v

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

This will be particularly useful if we want to create a long empty vector that we will use to store the results of operations in.

Because I want to use it in the later examples, I am going to recreate the vector that we created just prior to that one.

```
v <- seq(1,3,0.5)
```

4.1 Vector Operations

Once we have a vector, we can perform operations on it. If we want to do the same operation to each element in the vector, we simply use functions or mathematical operators as we have used them before. For example, suppose we wanted to subtract 1 from each element of the vector v. We would write:

```
v-1

## [1] 0.0 0.5 1.0 1.5 2.0
```

We could also perform many of the same functions on a vector that we could perform on a variable that only contained a single number. For example, if we wanted to compute the square root of each element in v, we could write:

```
sqrt(v)

## [1] 1.000000 1.224745 1.414214 1.581139 1.732051
```

With vectors, we can now use additional functions. For example, we can compute summary statistics (such as mean, median, and some other percentiles) of any vector by using the summary() function. Let's use summary() on our vector v:

```
summary(v)

##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##     1.0     1.5     2.0     2.0     2.5     3.0
```

We can find the sum of all elements in v by using the sum() function:

```
sum(v)

## [1] 10
```

The mean of v by using the mean() function:

```
mean(v)

## [1] 2
```

The standard deviation of v by using the sd() function:

```
sd(v)

## [1] 0.7905694
```

The number of elements contained in v by using the length() function:

```
length(v)

## [1] 5
```

Any given percentile of v by using the quantile() function. Note that this function takes 2 arguments: the vector and the percentile that you would like, expressed as a decimal. For example, to find the 99th percentile of v, we would write:

```
quantile(v,0.99)

##  99%
## 2.98
```

4.2 Accessing a subset of a vector

Oftentimes, we would like to know what a particular element of a vector is. Other times, we would like to compute statistics on only a subset of our data. We can also access any element of our vector using the notation:

```
v[place]
```

where v is the vector of interest and "place" is the element (or elemnts) you would like to access. For example, if we wanted to access the first element of v, we would write:

```
v[1]

## [1] 1
```

To access more than one element, you could use a collection, a sequence, or a:b notation. For example, to return the first through third elements of our vector v, we could call:

```
v[c(1,2,3)]

## [1] 1.0 1.5 2.0
```

or

```
v[seq(1,3,1)]

## [1] 1.0 1.5 2.0
```

or

```
v[1:3]

## [1] 1.0 1.5 2.0
```

4.3 Replacing Elements of Vectors

Sometimes we might want to replace a single element of a vector. Let's take a look at v again:

```
v

## [1] 1.0 1.5 2.0 2.5 3.0
```

Suppose we wanted to replace an element of v with 1.27. We could write:

```
v[place] <- 1.27
```

where place is the element we want to replace. For example, to replace the first element of v, we would write:

```
v[1] <- 1.27
```

Now if we look at v, we will see the first element has been replaced:

```
v

## [1] 1.27 1.50 2.00 2.50 3.00
```

Now suppose that we wanted to replace multiple elements of v with the same number. We could do something very similar to what we did above. We could write:

```
v[places] <- 1.27
```

where places is the list of elements that we wanted to replace. For example, suppose we wanted to replace the first three elements of v with 5.12. We could write:

```
v[1:3] <- 5.12
v

## [1] 5.12 5.12 5.12 2.50 3.00
```

Finally, we might want to replace multiple elements with different numbers. For example, maybe we want to "fix" v by restoring the first three elements to what they used to be before we started changing things. We need to be careful, because we need to have the correct number of elements in our replacement vector. We could write:

```
v[1:3] <- c(1.0,1.5,2.0)
```

or

```
v[1:3] <- seq(1.0,2.0,0.5)
```

If we look at v, we will see that the first three elements are what they used to be before we started chaging elements in this subsection.

```
v

## [1] 1.0 1.5 2.0 2.5 3.0
```

4.4 Exercises

1. Create a vector, v, that includes all integers from 10 to 20.
2. Create a vector, v, that goes from 0.0 to 1.5, incrementing by 0.01
   (a) What is the 29th element of this vector?
   (b) Find the 29th through 42nd elements of this vector.
   (c) Starting from the first element, find every other element of this vector (i.e. $1, 3, 5, \cdots T$)
3. Consider the following discrete distribution:

y_i	p_i
25	0.01
26	0.04
27	0.05
28	0.05
29	0.10
30	0.20
31	0.20
32	0.15
33	0.10
34	0.10

   (a) Create a vector, Y, that contains all the elements listed under y_i.
   (b) Create a vector, P, that conains all the probabilities listed under p_i.
   (c) Using the vectors you have created, along with the function sum() and vector multiplication, compute the expected value of this discrete distribution. (Hint: if we have two vectors, v and p, each the same length, we can multiply their respective elements by writing $v * p$)

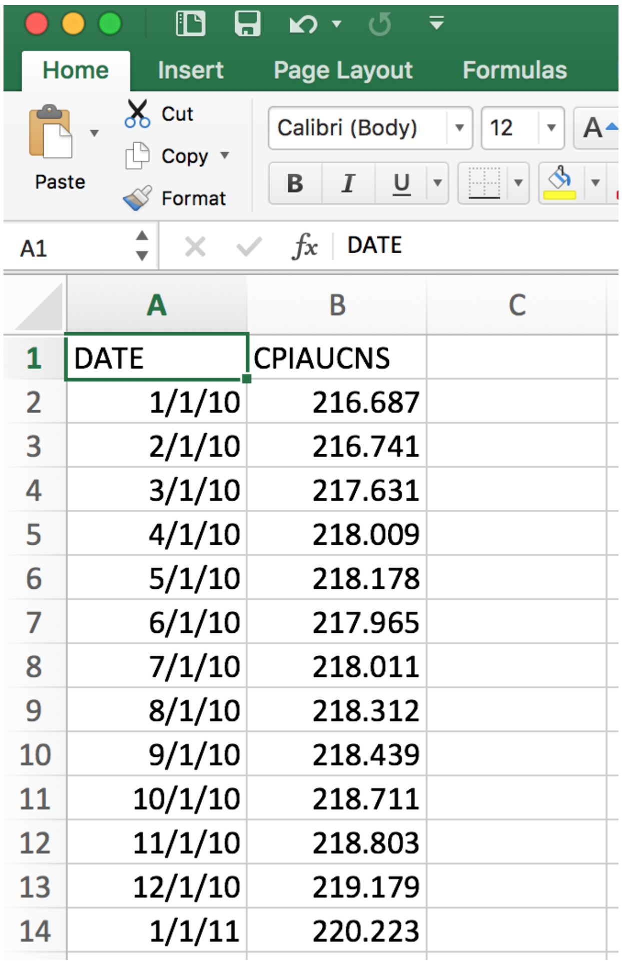
5 Reading in Data

Instead of creating a vector from scratch after we collect data, it is often easier to read data in from another file. Usually our data will come in a .csv, .xls, or .xlsx file type. While many of you may be more familiar with excel files (.xls), csv files are also very common when working with data, and can be obtained from many different data sources.

5.1 Reading in CSV Files

To read in a .csv file, we can use the function read.csv(). This function has several arguments, and many of them are optional. To get a more detailed description of what this function does, you can type help(read.csv) into RStudio.

Suppose that we downloaded a .csv file. As an example, I will use price level data downloaded from the website FRED. If we were to open the .csv file in Microsoft Excel, it would look like this:



	A	B	C
1	DATE	CPIAUCNS	
2	1/1/10	216.687	
3	2/1/10	216.741	
4	3/1/10	217.631	
5	4/1/10	218.009	
6	5/1/10	218.178	
7	6/1/10	217.965	
8	7/1/10	218.011	
9	8/1/10	218.312	
10	9/1/10	218.439	
11	10/1/10	218.711	
12	11/1/10	218.803	
13	12/1/10	219.179	
14	1/1/11	220.223	

We can see that there are two columns, and that the first line of each column contains a header. We will use the read.csv() function and assign the data to a variable called CPI_data:

```
CPI_data <- read.csv("/Users/chec9200/Downloads/CPI_2000.csv")
```

Note that when the .csv file has a header, we don't have to specify any options. In other words, the default is that read.csv assumes that the first row of your data file contains headers. If it does not contain headers, then you could use the following option:

```
CPI_data <- read.csv("/Users/acheck/Downloads/CPI_2000.csv",header=FALSE)
```

To look at the data that you have just loaded, you can use the View() function. Note that functions and variables are capitalization sensitive (i.e. if you type "view" instead of "View", it won't work).

5.2 Creating a Time-Series Vector

Some objects in R have special properties. After we read in our data using the read.csv() function, we have our data in a standard variable. However, the CPI data we read in is a time-series, since it contains repeated observations of CPI measured over time. In R, there is a time-series object that can take advantage of this information to make things like creating plots easier.

To tell R that our data is a time-series, we need to use the ts() function. The ts() function takes three arguments: the data, the start date, and the frequency. If we View(CPI), we will see that it begins in January, 1947 (1947-01-01). The next date is February, 1947 (1947-02-01). Therefore, we can see two things. The frequency of the data is monthly, and the start date is the first month of 1947. To declare our time series, we write:

```
library(fpp2) # Load fpp2 package..you need to do this at the beginning of every session.
 CPI <- ts(CPI_data[,2], start = c(2000,1), frequency=12)
```

The first argument of the function is the data we would like to declare as a time-series. The notation CPI_data[,2] gives us all rows of CPI_data, and the second column (that's what the 2 does). We do this since the first column contains only dates, and we only want to declare the actual CPI data as a time-series. The next argument is the start date. In this context, the notation c(1947,1) means the 1st period of 1947. Finally, the last argument is where we declare the frequency (how many times per year the data is collected). For monthly data, we use 12. For quarterly data we would use 4. For weekly data, 52, etc.

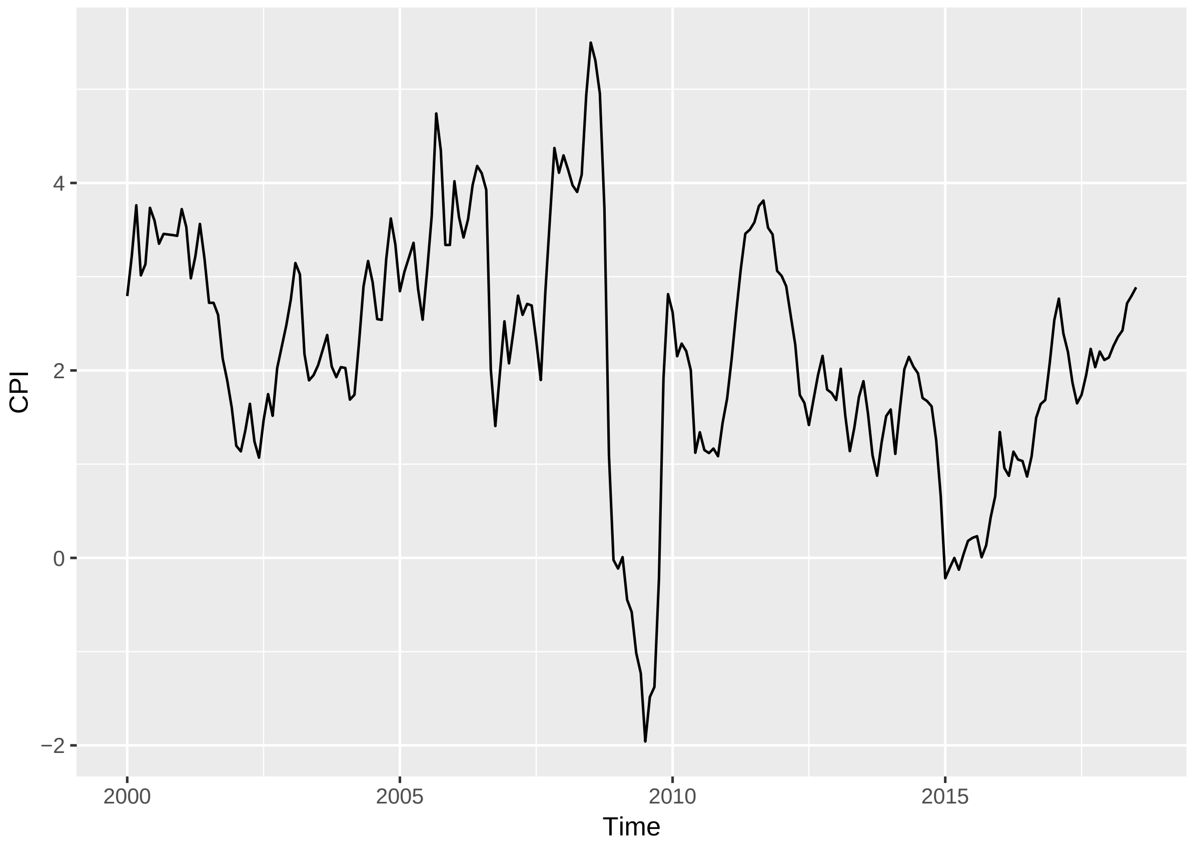
6 Creating Plots

Another powerful feature of R is creating plots. We will consider five different types of plots.

6.1 Time Plots

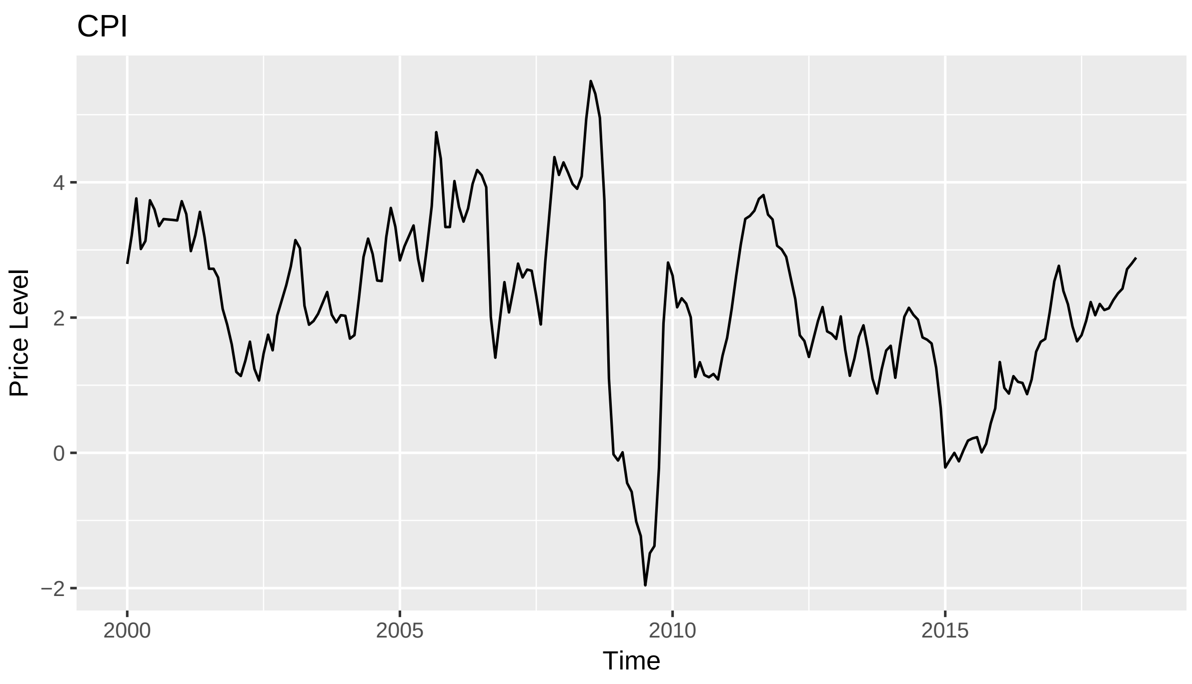
After declaring a variable as a time series variable (see section 5.2), it is relatively easy to create a time plot. Suppose we had a variable called CPI. To create a time plot of CPI, we would type:

```
autoplot(CPI)
```



However, it is often useful to specify some options. We will use the option "main" to define the main title of the plot, "xlab" to define the label on the x-axis, and "ylab" to define the label on the y-axis:

```
autoplot(CPI) + ggtitle("CPI") + xlab("Time") + ylab("Price Level")
```



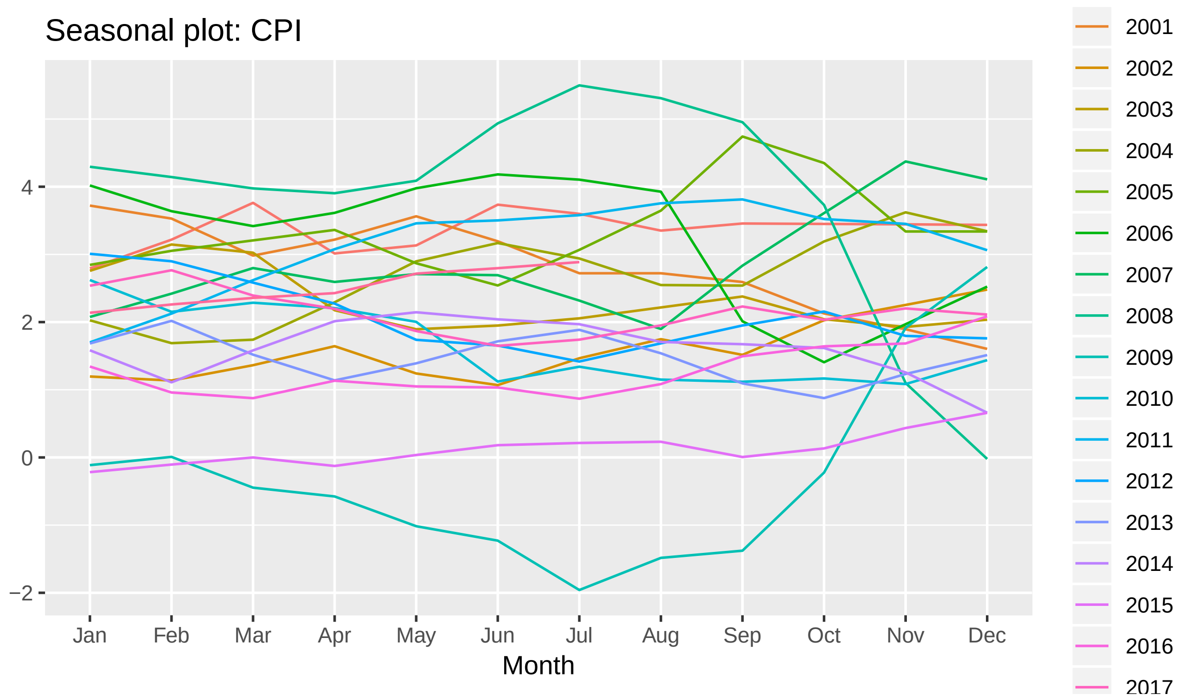
Note that with the options, the order doesn't matter. For example, we could also write:

```r
autoplot(CPI) + ylab("Price Level") + xlab("Time") + ggtitle("CPI")
```

6.2 Seasonal Plots

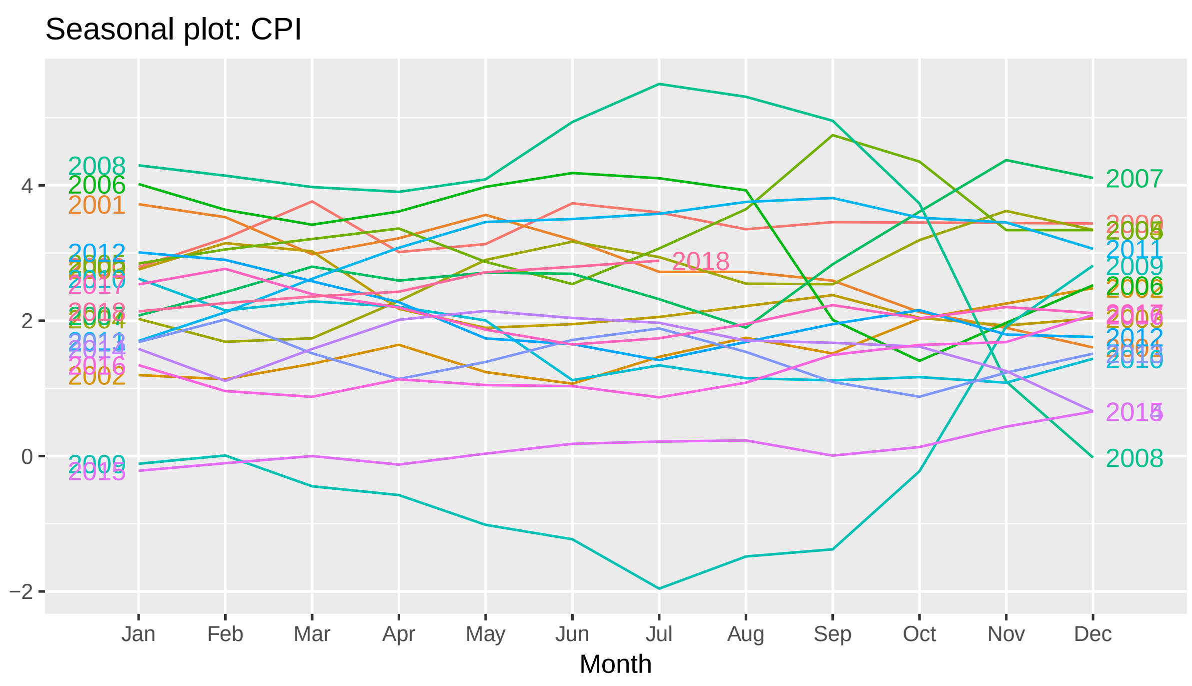
Another type of plot we might be interested in is a seasonal plot. This plot will help us see seasonal patterns more clearly, if they are present. To plot a seasonal plot, we would write:

```r
ggseasonplot(CPI)
```



We could use all of the options mentioned in the previous subsection to name a main plot title, and x and y axis labels. To add year labels and to turn each line a different color, we could write:

```
ggseasonplot(CPI,year.labels=TRUE, year.labels.left = TRUE)
```

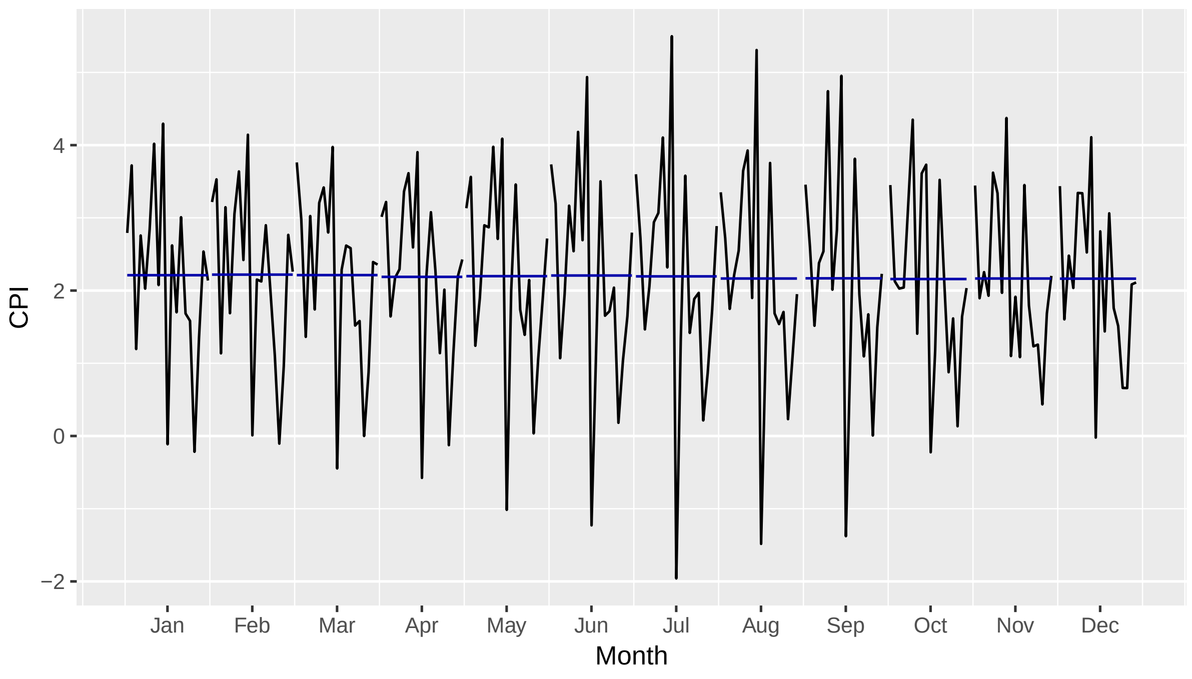


The option year.labels=TRUE puts year labels on the plot, while year.labels.left = TRUE eunsures that the labels will be on both the right and left of the lines.

6.3 Seasonal Subseries Plots

Another type of plot we might be interested in is a seasonal subseries plot. Similarly to a seasonal plot, this plot will help us see seasonal patterns more clearly, if they are present. To plot a seasonal subseries plot, we would write:

```
ggsubseriesplot(CPI)
```

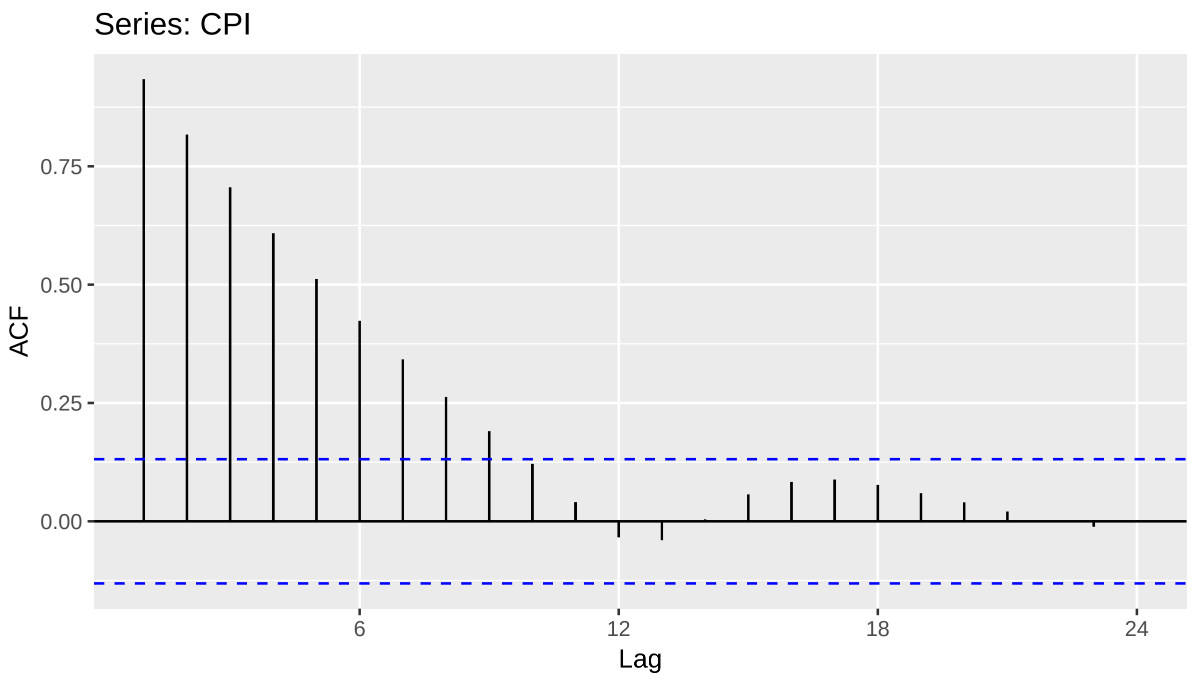


We could use the options mentioned above to name a main plot title, and x and y axis labels.

6.4 ACF Plot

Another type of plot we might be interested in is an autocorrelation function (Acf) plot. This plot can help identify features of the data (or features of the forecasting errors, if we are plotting those) that can help us determine an appropriate forecasting model.

```
ggAcf(CPI)
```

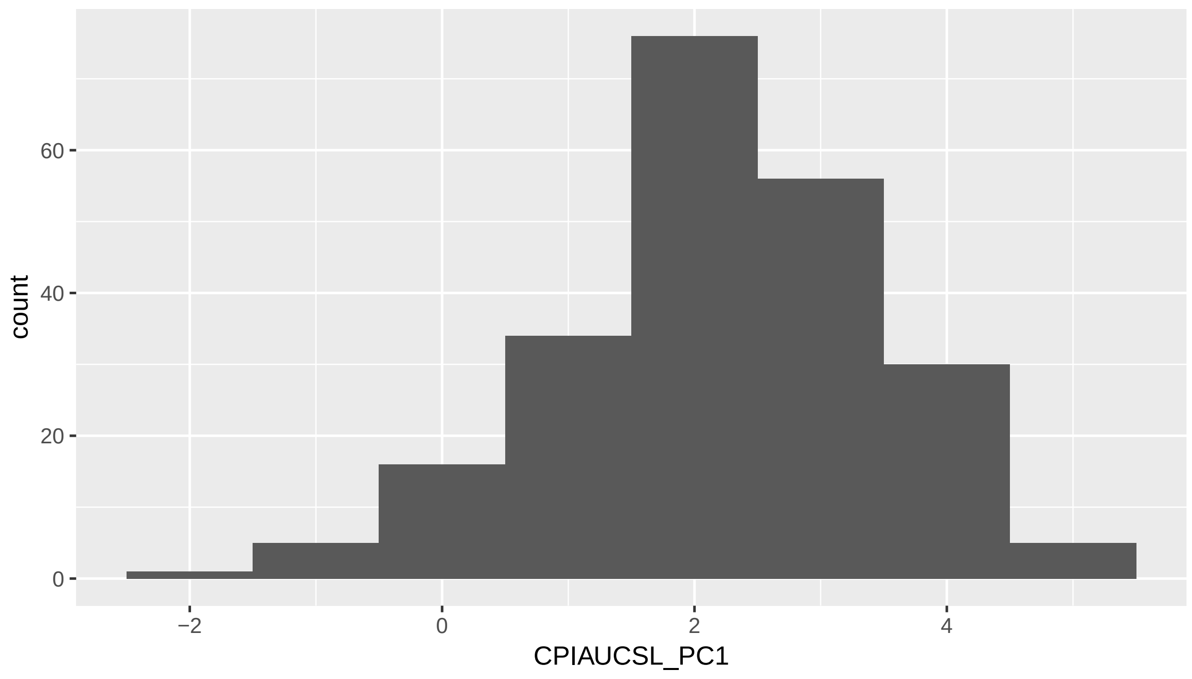


We could use the options mentioned above to name a main plot title, and x and y axis labels.

6.5 Histogram

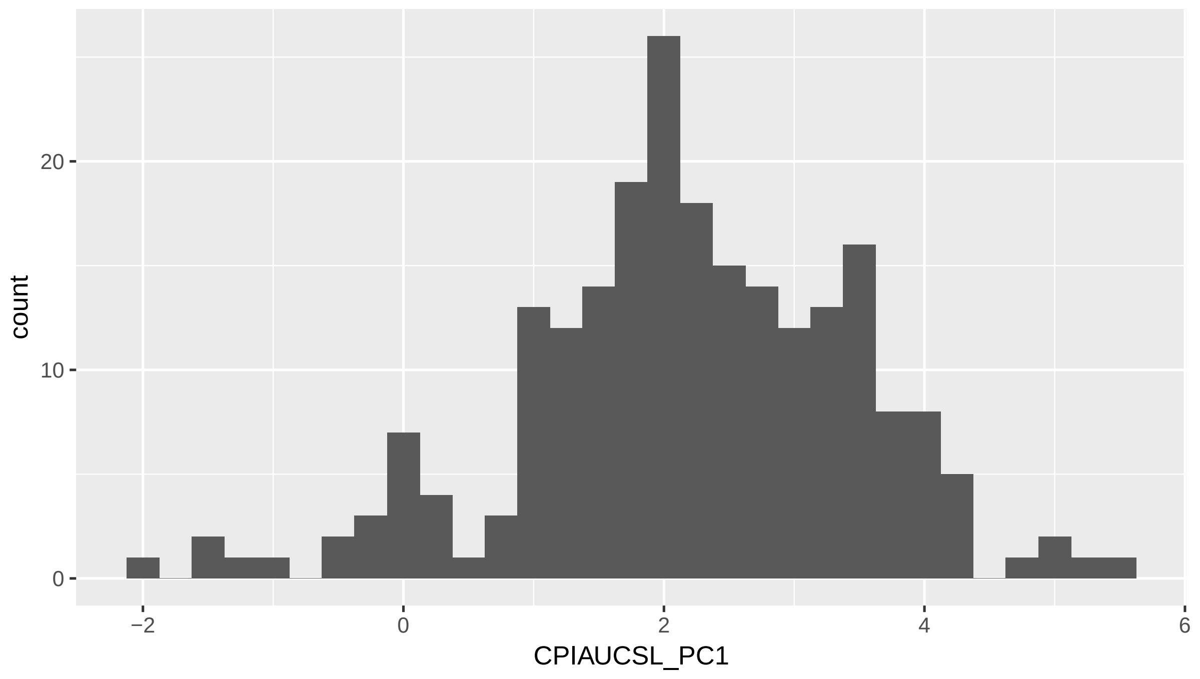
Another type of plot we might be interested in is an histogram. This plot can help us to identify if the data (or the forecasting errors) are Normally distributed, or if they have a larger spread or skew. To plot a histogram, we can type:

```
ggplot(CPI_data, aes(x=CPIAUCSL_PC1)) + geom_histogram(binwidth=1)
```



where the argument binwidth specifies the width of the bar. If instead of plotting the data grouped into 1 percentage point intervals, we plotted the data grouped into 0.25 percentage point intervals, we would have:

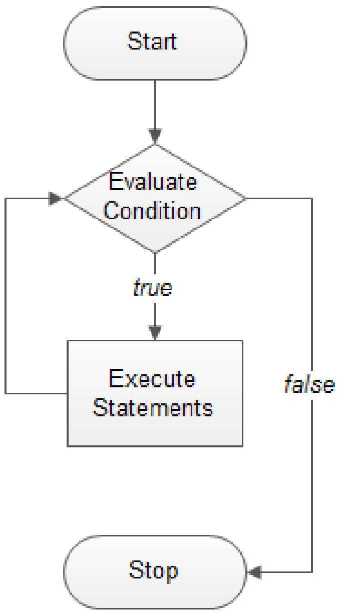
```
ggplot(CPI_data, aes(x=CPIAUCSL_PC1)) + geom_histogram(binwidth=0.25)
```



You always need to be careful when using histograms, because the binwidth can make a big difference in the appearance of the data. In addition, with time series data, if there is a trend in the data then a histogram would not be appropriate.

7 For Loops (Advanced)

Suppose we want to repeat an operation many times. Sometimes, the best way to do this is to use what is called a "for loop". The logic underlying the "for loop" can be summarized with the flowchart below:



In R, a for loop is written with the following syntax:

```
for (i in v) {


}
```

where v is a vector and i takes the value of each of the elements in the vector, one at a time.

Suppose we wanted to add all of the numbers between 50 and 100. We could do this several different ways. First, let's define a vector, Y, that will hold all of the values between 50 and 100.

```
Y <- seq(50,100,1)
```

The first way we could add these numbers is to use the built in sum() function:

```
sum(Y)

## [1] 3825
```

Let's use a loop instead. We will define a variable called *count* which will eventually contain the sum of all numbers from 50 to 100. Let's give *count* a starting value of zero:

```r
count <- 0
```

Now, we will run a loop that will produce the sum that we want. To do this, we will cycle through all the values of Y, adding each to the previous:

```r
for (i in Y){
  count <- count + i
}
```

The loop above starts with the value of the first element of Y, 50. It adds 50 to the previous value of *count*, which was zero. So now $count = 50$. Next, the loop goes back to the top, and now i will take the value of the second element of Y, 51. It adds 51 to the previous value of *count*, which was 50. So now $count = 101$. This process repeats until we reach the final element of Y. Let's put it all together and see what the value is:

```r
count <-0
for (i in Y){
  count <- count + i
}
count

## [1] 3825
```

We see that our answer, 3,825, is the same as the answer produced by the sum() function.

Another way to do the same thing is to manually pull each element from Y. To do this, we need to know how many elements are in Y. We will store this value in a variable, T:

```r
T <- length(Y)
```

Now, we will have i go from 1 to T, and we will pull the i^{th} element from the vector Y:

```r
T <- length(Y)
count <- 0
for (i in 1:T){
  count <- count + Y[i]
}
count

## [1] 3825
```

On the first run through the loop, $i = 1$. We add $Y[1]$, the first element of Y to count. Next, $i = 2$, so we add $Y[2]$, the second element of Y, to count. This process continues until $i = T$. We can see that this gave us the exact same answer as the previous loop.

We can also do slightly more complicated things. For example, perhaps we wanted to keep a running mean of Y. A running mean takes the mean from the first element through the i^{th}

element. For example, the running mean of Y as defined above at $i = 1$ is 50, while the running mean at $i = 2$ is $\frac{50+51}{2} = 50.5$.

To do this, we first need to create an empty storage vector that will eventually contain the running mean at each element. Let's call this *running_mean*.

```r
Y <- seq(50,100,1)
T <- length(Y)
running_mean <- rep(0,T)
```

The rep() function creates a vector of repeated elements. Here, I have set each element to 0, and the vector contains T elements. Next, we will use a for loop to compute the running mean at each element, and save that running mean to the running_mean vector. Then I will print the result.

```r
for (i in 1:T){
  running_mean[i] <- mean(Y[1:i])
}
running_mean

##  [1] 50.0 50.5 51.0 51.5 52.0 52.5 53.0 53.5 54.0 54.5 55.0 55.5 56.0 56.5
## [15] 57.0 57.5 58.0 58.5 59.0 59.5 60.0 60.5 61.0 61.5 62.0 62.5 63.0 63.5
## [29] 64.0 64.5 65.0 65.5 66.0 66.5 67.0 67.5 68.0 68.5 69.0 69.5 70.0 70.5
## [43] 71.0 71.5 72.0 72.5 73.0 73.5 74.0 74.5 75.0
```

We can see that the first two elements were exactly what we expected (50 and 50.5). If you are ever unsure as to whether or not your loop is working correctly, it is a good idea to try and do a few of the calculations "by hand" and make sure you get the same answer. If you don't, there is either a mistake in the loop or in your "by hand" calculation. In general, since loops are usually harder, the mistake is probably in your loop.

7.1 Exercises

1. Use a for loop to add all of the numbers from 1 to 100.

2. Repeat part (c) of exercise 4.3.3 using a for loop instead of the sum() function.

3. The function rnorm(n,μ,σ) takes three arguments: the number of numbers to produce (n), the mean of a normal distribution (μ), and the standard deviation of a normal distribution (σ). This function returns a vector of length n that contains random draws from $\mathcal{N}(\mu, \sigma)$. For example, rnorm(1,0,2) will return 1 draw from a nomral distribution with mean 0 and standard deviation 2.0. Use this function and a for loop to take 100 draws from the following model:

$$y_t = y_{t-1} + \varepsilon_t$$
$$\varepsilon_t \sim \mathcal{N}(0, 0.5)$$
$$y_0 = 0.0$$