Dates, Times, TimeIndexes and Time Series

Jeff Hallman

2016-11-18

Contents

Dates and Times in Base R

Dates

Early versions of R had a simple Date class, and code that handled somewhat more general time objects was in a separate `chron` package. Neither worked very well for doing various kinds of date arithmetic or as indexes into time seriesm and this led to a number of people writing their own time and date packages. But base R time and date code has improved a lot over the years.

The base `Date` class supports dates without times. Not including times simplifies things a lot, since you no longer have to worry about time zones and daylight savings times, or what to do if only a date was specified but not a time. Should the default time be noon? Midnight?

Here are some Date examples:

```
today <- Sys.Date(); today
```

```
OUTPUT>  [1] "2016-11-18"
```

```r
today - 1 ## yesterday
```

```
OUTPUT>  [1] "2016-11-17"
```

```r
today + 7
```

```
OUTPUT>  [1] "2016-11-25"
```

Internally, Dates are represented as the number of days since January 1, 1970. You can see this by `unclass`ing them:

```r
unclass(today)
```

```
OUTPUT>  [1] 17123
```

```r
baseDate <- as.Date("1970-01-01"); unclass(baseDate)
```

```
OUTPUT>  [1] 0
```

```r
as.integer(today - baseDate)
```

```
OUTPUT>  [1] 17123
```

Now you're wondering why the `as.integer` wrapper on that last one. That's because without it, you get a `difftime` object:

```r
myDifftime <- today - baseDate; myDifftime
```

```
OUTPUT>  Time difference of 17123 days
```

```r
unclass(myDifftime)
```

```
OUTPUT>  [1] 17123
OUTPUT>  attr(,"units")
OUTPUT>  [1] "days"
```

A difftime object represents a time interval. It's a number with a "units" attribute. You can also create a difftime object explicitly:

```r
twoWeeks <- as.difftime(2, units = "weeks")
today + twoWeeks
```

```
OUTPUT>  [1] "2016-12-02"
```

One problem with difftime's is that the units have to be "linear", i.e., weeks, days, hours, seconds, etc. Months, quarters and years don't work. If I tell you to add one month to a date, how many days do you add? It depends on what month the original date is in.

Times (w/Dates)

The POSIX standards specify a system for describing instants in time. R uses two timeDate classes, `POSIXct` and `POSIXlt` to implement this system. A POSIXct represents a time as the number of seconds since midnight on January 1, 1970, in the GMT time zone:

```r
aTime <- Sys.time(); aTime
```

```
OUTPUT>  [1] "2016-11-18 09:06:26 EST"
```

```r
class(aTime)
```

```
OUTPUT>  [1] "POSIXct" "POSIXt"
```

```r
unclass(aTime)
```

```
OUTPUT>  [1] 1479477987
```

```r
unclass(as.POSIXct("19700101:000001", tz= "GMT", format = "%Y%m%d:%H%M%S"))
```

```
OUTPUT>  [1] 1
OUTPUT>  attr(,"tzone")
OUTPUT>  [1] "GMT"
```

The `POSIXlt` represents instants in time using a list with named elements:

```r
unclass(as.POSIXlt(Sys.time()))
```

```
OUTPUT>  $sec
OUTPUT>  [1] 26.70608
OUTPUT>
OUTPUT>  $min
OUTPUT>  [1] 6
OUTPUT>
OUTPUT>  $hour
OUTPUT>  [1] 9
OUTPUT>
OUTPUT>  $mday
OUTPUT>  [1] 18
OUTPUT>
OUTPUT>  $mon
OUTPUT>  [1] 10
OUTPUT>
OUTPUT>  $year
OUTPUT>  [1] 116
OUTPUT>
OUTPUT>  $wday
OUTPUT>  [1] 5
OUTPUT>
OUTPUT>  $yday
OUTPUT>  [1] 322
OUTPUT>
OUTPUT>  $isdst
OUTPUT>  [1] 0
OUTPUT>
OUTPUT>  $zone
OUTPUT>  [1] "EST"
OUTPUT>
OUTPUT>  $gmtoff
OUTPUT>  [1] -18000
OUTPUT>
OUTPUT>  attr(,"tzone")
OUTPUT>  [1] ""    "EST" "EDT"
```

Date formats

You can convert most date and dateTime objects to strings and back using format specifiers. For example:

```r
format(Sys.time(), format = "%Y is year, %m is month, %d is day")
```

```
OUTPUT>  [1] "2016 is year, 11 is month, 18 is day"
```

```r
as.character(Sys.time(), format = "%b is abbreviated month name, %H is hour")
```

```
OUTPUT>  [1] "Nov is abbreviated month name, 09 is hour"
```

```r
format(Sys.time(), format = "There are also compound formats like %c")
```

```
OUTPUT>  [1] "There are also compound formats like Fri Nov 18 09:06:26 2016"
```

```r
as.POSIXct("2016-12-24 23:59", format = "%Y-%m-%d %H:%M") ## Santa arrival time
```

```
OUTPUT>  [1] "2016-12-24 23:59:00 EST"
```

Finally, some generic functions like `seq` have methods that work with both Dates and POSIXt times:

```r
seq(from = Sys.time(), by = "2 hours", length = 8)
```

```
OUTPUT>  [1] "2016-11-18 09:06:26 EST" "2016-11-18 11:06:26 EST"
OUTPUT>  [3] "2016-11-18 13:06:26 EST" "2016-11-18 15:06:26 EST"
OUTPUT>  [5] "2016-11-18 17:06:26 EST" "2016-11-18 19:06:26 EST"
OUTPUT>  [7] "2016-11-18 21:06:26 EST" "2016-11-18 23:06:26 EST"
```

TimeIndexes in the `tis` package

The `tis` package defines the `jul` (Julian date) and `ti` (Time Index) classes.

```r
#install.packages("tis")

library(tis)
```

Julian dates

are represented by instances of the `jul` class. Two Julian dates differ by the number of days between them. This is also true of `Date` objects in base R, but `jul` dates use a different base date. You can create a `jul` from a `ti` with the `jul()` function, which like `ymd()` has an optional `offset` argument. Like `ti's`, `jul` objects can be converted to and created from several other kinds of date objects.

```r
dayInYear <- jul(today()) - jul(20151231); dayInYear
```

```
OUTPUT>  [1] 323
```

ti objects

Time index objects are used for date calculations and as indexes for `tis` time series. A time index has two parts: a frequency and period. The period represents the number of periods elapsed since the base period for that frequency. Adding or subtracting an integer to a time index adds or subtracts that number to the period part, so that if `z` is a time index representing the week ending Monday, April 3 2006, `z + 1` is a time index representing the week ending April 10, `z + 3` represents the week ending April 24, and so on.

A time index has class 'ti', and is usually created by the `ti` function. However, there is a `today()` function that returns a "daily" `ti`. If z is a ti, you can access its frequency code with `tif(z)` or its period with `period(z)`. You can also get the frequency name from `tifName(z)`. Note that you can have a vector `ti`, that is, a vector with class 'ti' whose individual elements each represent a different time index. Here are some examples:

```
march2007 <- ti(20070331, "monthly"); march2007
```

```
OUTPUT>  [1] 20070331
OUTPUT>  class: ti
```

```
today()
```

```
OUTPUT>  [1] 20161118
OUTPUT>  class: ti
```

```
thisWeek <- ti(latestMonday(), "wmonday"); thisWeek
```

```
OUTPUT>  [1] 20161114
OUTPUT>  class: ti
```

```
tif(today())
```

```
OUTPUT>  [1] 1001
```

```
tifName(today())
```

```
OUTPUT>  [1] "daily"
```

```
period(today())
```

```
OUTPUT>  [1] 42692
```

```
# How this is actually encoded is pretty simple:
tif(thisWeek)
```

```
OUTPUT>  [1] 1004
```

```
period(thisWeek)
```

```
OUTPUT>  [1] 6099
```

```
format(unclass(thisWeek), digits = 12)
```

```
OUTPUT>  [1] "10040000006099"
```

```
period(ti(latestMonday() + 28, "wmonday"))
```

```
OUTPUT>  [1] 6103
```

```
# yyyymmdd dates for the last day of the next 12 months:
ymd(currentMonth() + 0:11)
```

```
OUTPUT>   [1] 20161130 20161231 20170131 20170228 20170331 20170430 20170531
OUTPUT>   [8] 20170630 20170731 20170831 20170930 20171031
```

```
# the third Tuesday of the year for each of the next 5 years:
ymd(ti(firstDayOf(currentYear() + 1:5), "wtuesday") + 2)
```

```
OUTPUT>  [1] 20170117 20180116 20190115 20200121 20210119
```

If you want to convert a `ti` object to a yyyymmdd date, use the `ymd()` function:

```
dec57 <- ti(19571231, tif = "monthly")
# By default, ymd(aTi) gives you the last day of the period. To get the first day of the period, use th
ymd(jul(dec57 - 1) + 1)
```

```
OUTPUT>  [1] 19571201
```

The `ti` function needs to know what the frequency of the ti it's creating is supposed to be, and the first argument has to be a itself a ti object or something that specifies a point in time. You can supply additional arguments to help it figure out how to interpret the first argument, e.g.,

```r
ti("3/31/1957", tif = "daily", format = "%m/%d/%Y") + 1
```

```
OUTPUT>  [1] 19570401
OUTPUT>  class: ti
```

```r
ti("3/31/1957", freq = 12,      format = "%m/%d/%Y") + 1
```

```
OUTPUT>  [1] 19570430
OUTPUT>  class: ti
```

Finally, there are some convenience function like `holidays()` and `nextBusinessDay()` which knows about holidays.

Time Series

Base R contains a lot of code for representing and analyzing time series data. The fundamental class is `ts`, which can represent regularly spaced time series, such as annual, monthly and quarterly data. It doesn't work well with weekly data because not all years have the same number of weeks. It depends on what day of the year a particular year starts on and what day of the week the weeks you're looking at end on.

The tis package again

The inability of early versions of R to deal well with weekly, biweekly, daily and business-day data was the impetus behind early versions of the `tis` package. `tis` series have observation times tjat are represented by `ti` sequences. A `tis` is actually just a vector or matrix with class 'tis' and a `start` attribute that is the `ti` for the first observation. Then `start - 1 + k` is the time index for the k'th observation, and so on. If `x` is a `tis`, you can reference the fourth observation using either the number '4' or the `ti` given by `(start(x) + 3)` as an index into `x`. Here are some examples of creating a `tis` series and indexing into it:

```r
x <- tis(1:120, start = latestJanuary() - 108); x
```

```
OUTPUT>       Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec
OUTPUT>  2007   1   2   3   4   5   6   7   8   9  10  11  12
OUTPUT>  2008  13  14  15  16  17  18  19  20  21  22  23  24
OUTPUT>  2009  25  26  27  28  29  30  31  32  33  34  35  36
OUTPUT>  2010  37  38  39  40  41  42  43  44  45  46  47  48
OUTPUT>  2011  49  50  51  52  53  54  55  56  57  58  59  60
OUTPUT>  2012  61  62  63  64  65  66  67  68  69  70  71  72
OUTPUT>  2013  73  74  75  76  77  78  79  80  81  82  83  84
OUTPUT>  2014  85  86  87  88  89  90  91  92  93  94  95  96
OUTPUT>  2015  97  98  99 100 101 102 103 104 105 106 107 108
OUTPUT>  2016 109 110 111 112 113 114 115 116 117 118 119 120
OUTPUT>  class: tis
```

```r
k = 2; start(x) + k
```

```
OUTPUT>  [1] 20070331
OUTPUT>  class: ti
```

```
x[start(x) + 3]
```

```
OUTPUT>  [1] 4
```

```
x[end(x) + 1] <- x[end(x)]; x
```

```
OUTPUT>       Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec
OUTPUT>  2007   1   2   3   4   5   6   7   8   9  10  11  12
OUTPUT>  2008  13  14  15  16  17  18  19  20  21  22  23  24
OUTPUT>  2009  25  26  27  28  29  30  31  32  33  34  35  36
OUTPUT>  2010  37  38  39  40  41  42  43  44  45  46  47  48
OUTPUT>  2011  49  50  51  52  53  54  55  56  57  58  59  60
OUTPUT>  2012  61  62  63  64  65  66  67  68  69  70  71  72
OUTPUT>  2013  73  74  75  76  77  78  79  80  81  82  83  84
OUTPUT>  2014  85  86  87  88  89  90  91  92  93  94  95  96
OUTPUT>  2015  97  98  99 100 101 102 103 104 105 106 107 108
OUTPUT>  2016 109 110 111 112 113 114 115 116 117 118 119 120
OUTPUT>  2017 120
OUTPUT>  class: tis
```

```
x[end(x) + 3] <- 42; x
```

```
OUTPUT>       Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec
OUTPUT>  2007   1   2   3   4   5   6   7   8   9  10  11  12
OUTPUT>  2008  13  14  15  16  17  18  19  20  21  22  23  24
OUTPUT>  2009  25  26  27  28  29  30  31  32  33  34  35  36
OUTPUT>  2010  37  38  39  40  41  42  43  44  45  46  47  48
OUTPUT>  2011  49  50  51  52  53  54  55  56  57  58  59  60
OUTPUT>  2012  61  62  63  64  65  66  67  68  69  70  71  72
OUTPUT>  2013  73  74  75  76  77  78  79  80  81  82  83  84
OUTPUT>  2014  85  86  87  88  89  90  91  92  93  94  95  96
OUTPUT>  2015  97  98  99 100 101 102 103 104 105 106 107 108
OUTPUT>  2016 109 110 111 112 113 114 115 116 117 118 119 120
OUTPUT>  2017 120  NA  NA  42
OUTPUT>  class: tis
```

As you might expect, arithmetic operations (addition, subtraction, multiplication and division) can all be performed with combinations of scalars and series, and the 'tis' machinery will insure that everything is lined up and windowed, and that the return values will also be time indexed series. Furthermore, many of the standard R matrix and time series functions have 'tis' versions as well. For example, if `X` is a multivariate time indexed series with a column per component, then `rowSums(X)` is a univariate tis, as is `rowMeans(X)`. The `cbind()` function can be used to add columns to a tis, while `mergeSeries()` can do what it says. The `lag` and `diff` functions operate as you might expect, and you can `window` a series to cut off observations before and/or after particular time indexes.

Plotting and calculating with an actual series

Download Starbux monthly return date into a tis series

Let's start by downloading the monthly return data from sbuxPrices.csv, and by using tisFromCsv() to read a tis series from it.

```
# Download the CSV file and look at the first few lines:
url <- "http://assets.datacamp.com/course/compfin/sbuxPrices.csv"
```

```
download.file(url, dest = "sbux.csv")
flines <- readLines("sbux.csv"); flines[1:5]
```

```
OUTPUT>  [1] "Date,Adj Close" "3/31/1993,1.13" "4/1/1993,1.15"  "5/3/1993,1.43"
OUTPUT>  [5] "6/1/1993,1.46"
```

There is a problem with the dates. All of the dates except the first one are the first weekday of a month. Let's make the heroic assumption that the first date was supposed to be the first of March 1993 rather than the 31'st. Here's a quick fix in R:

```
flines[2] <- gsub("3/31", "3/1", flines[2])
cat(flines, file = "sbux.csv", sep = "\n")
flines <- readLines("sbux.csv"); flines[1:5]
```

```
OUTPUT>  [1] "Date,Adj Close" "3/1/1993,1.13"  "4/1/1993,1.15"  "5/3/1993,1.43"
OUTPUT>  [5] "6/1/1993,1.46"
```

Now that looks OK. We'll use the `tisFromCsv` function to read the data into a `tis` time series. Note we set the stringsAsFactors global option to FALSE to avoid a PITA

```
options(stringsAsFactors = FALSE)
sbux <- tisFromCsv("sbux.csv", dateCol = "Date", dateFormat = "%m/%d/%Y")
str(sbux)
```

```
OUTPUT>  List of 1
OUTPUT>   $ Adj.Close:Class 'tis'  atomic [1:181] 1.13 1.15 1.43 1.46 1.41 1.44 1.63 1.59 1.32 1.32 ...
OUTPUT>    .. ..- attr(*, "start")=Class 'ti'  num 1.03e+13
```

```
# sbux is a list of length one, because tisFromCsv returns a list of the series
# it reads in. Let's just make it be the series itself:
sbux <- sbux[[1]]
# Info about sbux
class(sbux)
```

```
OUTPUT>  [1] "tis"
```

```
start(sbux)
```

```
OUTPUT>  [1] 19930331
OUTPUT>  class: ti
```

```
frequency(sbux)
```

```
OUTPUT>  [1] 12
```

```
tifName(sbux)
```

```
OUTPUT>  [1] "monthly"
```

```
dateRange(sbux)
```

```
OUTPUT>  [1] 19930331 20080331
OUTPUT>  class: ti
```

```
head(ti(sbux))
```

```
OUTPUT>  [1] 19930331 19930430 19930531 19930630 19930731 19930831
OUTPUT>  class: ti
```

```
head(time(sbux))
```

```
OUTPUT>  [1] 1993.244 1993.326 1993.411 1993.493 1993.578 1993.663
```

Notice that `time(x)` returns the times of the observations in years, with the first day of the year counting as day 0 of that year. So `time(jul(20010101))` is 2001, while `time(jul(20001231))` is 2000.997.

To get the subseries that starts and ends on particular dates, use the `window()` function:

```
sbuxShort <- window(sbux, start = 19940301, end = 19950301)
sbuxShort
```

```
OUTPUT>        Jan  Feb  Mar  Apr  May  Jun  Jul  Aug  Sep  Oct  Nov  Dec
OUTPUT>  1994           1.45 1.77 1.69 1.50 1.72 1.68 1.37 1.61 1.59 1.63
OUTPUT>  1995 1.43 1.42 1.43
OUTPUT>  class: tis
```

What happened here is that the `window` method for `tis` objects helpfully first converted the `start` and `end` parameters to `ti` objects with the same `tif` as `sbux`, and then cut off the series before the start ti and after the end ti.

Subsetting directly

If you just want the observation for July 1996, you can do this:

```
sbux[ti(19960701, tif = tif(sbux))]
```

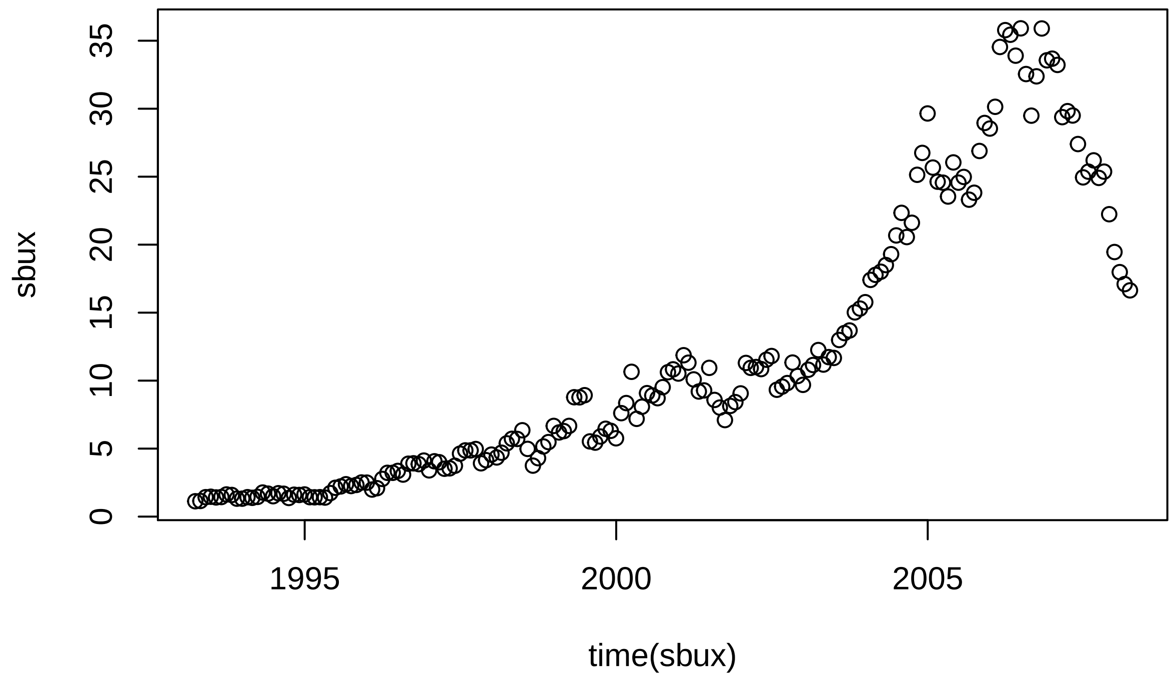
```
OUTPUT>  [1] 3.09
```

```
# Verify this (note the c(year, period) form for start and end
window(sbux, start = c(1996, 1), end = c(1996,12))
```

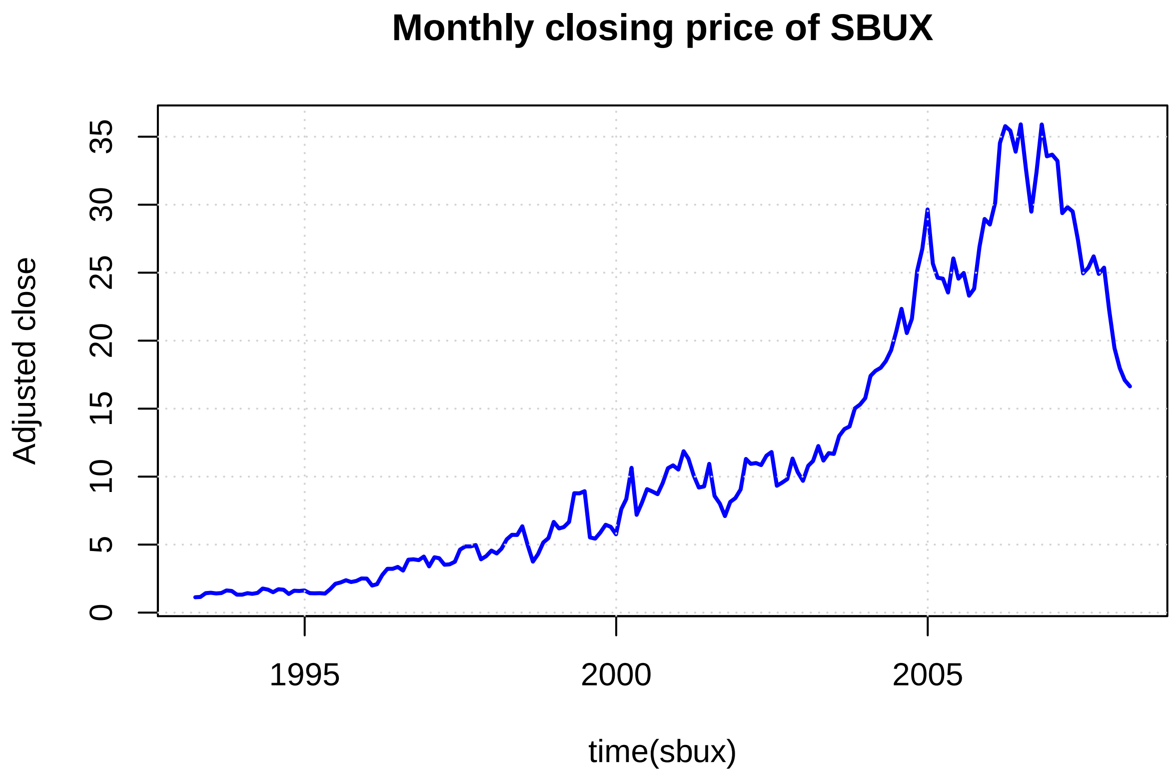
```
OUTPUT>        Jan  Feb  Mar  Apr  May  Jun  Jul  Aug  Sep  Oct  Nov  Dec
OUTPUT>  1996 1.99 2.09 2.77 3.22 3.22 3.36 3.09 3.89 3.92 3.86 4.12 3.40
OUTPUT>  class: tis
```

If we want to plot the data, we need some X-axis coordinates. One way to do this is with the `time()` function we saw above:

```
plot(time(sbux), sbux)
```

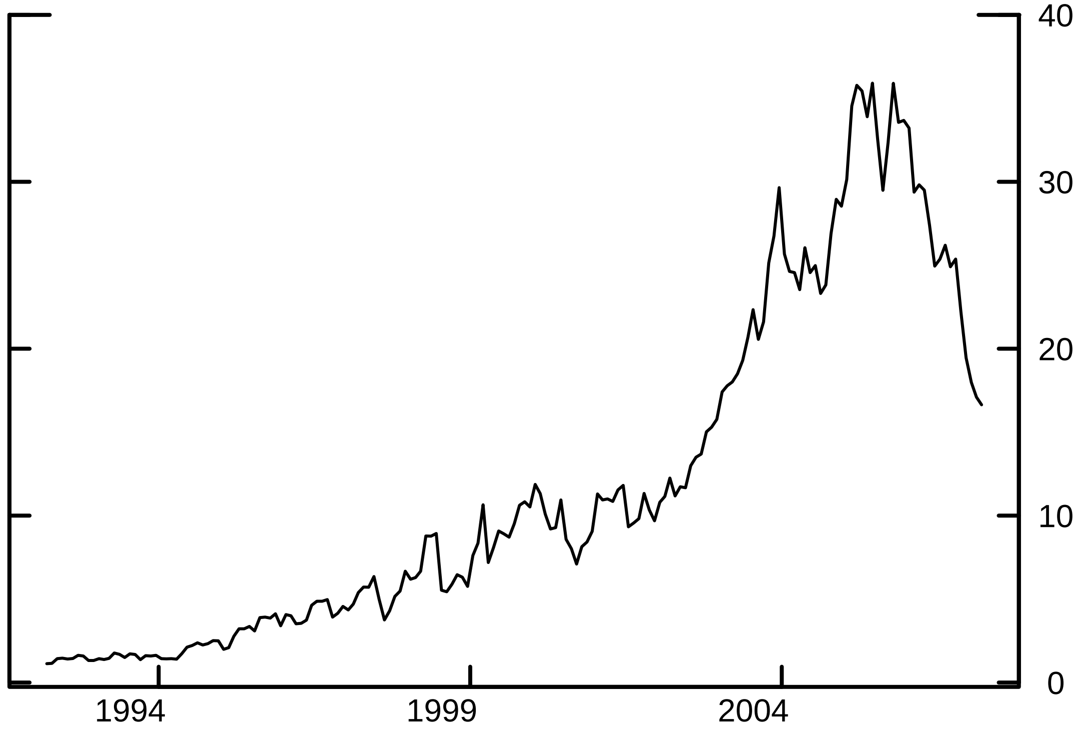



```r
# or dress it up a little
plot(time(sbux), sbux, type = "l", col = "blue",
     lwd = 2, ylab = "Adjusted close",
     main = "Monthly closing price of SBUX")
grid()
```

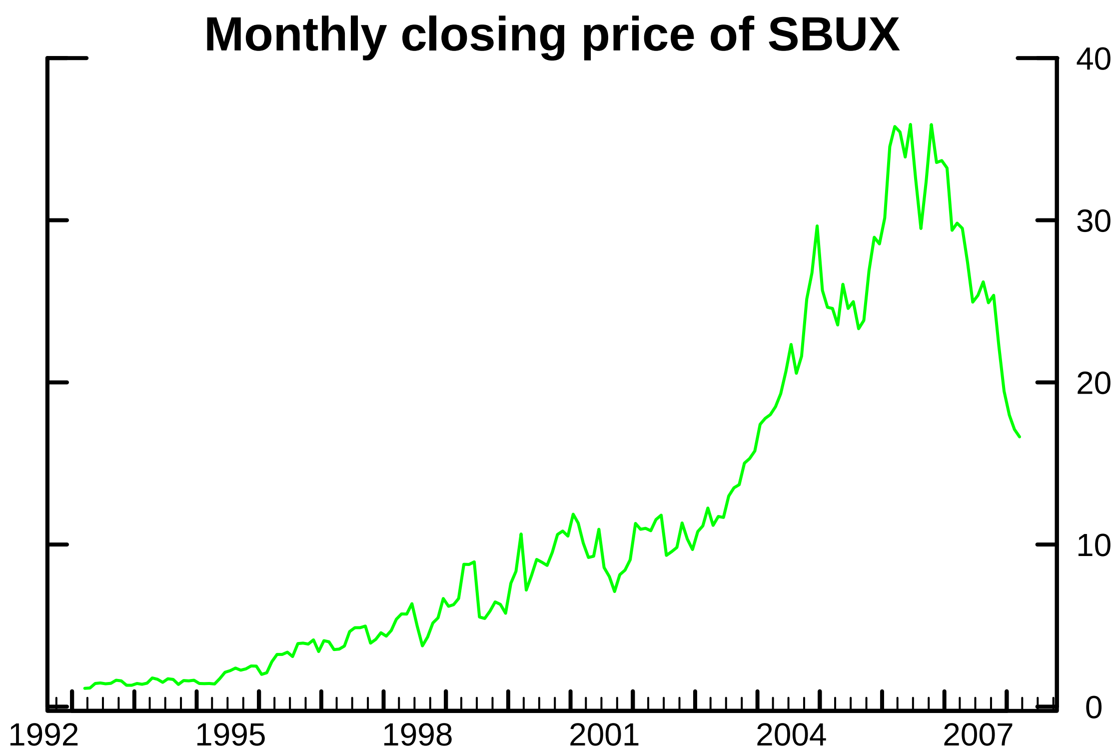


or we can use a plotting function that knows something about tis series:

```
tisPlot(sbux)
```




```
# and dress this one up a bit
tisPlot(sbux, color = "green",
        xTickFreq = "annual", xTickSkip = 2,
        xUnlabeledTickFreq = "annual",
        xMinorTickFreq = "quarterly",
        head = "Monthly closing price of SBUX")
```



Calculate simple returns

If you denote by the stock price at the end of month t, the simple return is given by:

- R[t] = (P[t] - P[t-1]) / P[t-1] - the percentage price difference.

Task

Our task in this exercise is to compute the simple returns for every time point n.

We *could* do this the hard way by creating a vector **a** that contains all but the first observation of `sbux` and a second vector **b** that contains all but the last observation, then our simple returns would just be `(a - b)/b`, but since we have our data in the form of a time series, we can just do this:

```
simpReturn <- diff(sbux)/lag(sbux, -1); head(round(simpReturn, 4))
```

```
OUTPUT>           Apr      May      Jun      Jul      Aug      Sep
OUTPUT>  1993  0.0177  0.2435  0.0210 -0.0342  0.0213  0.1319
OUTPUT>  class: tis
```

In economics, the usual meaning of `lag` is the previous period's value of a series, but that is exactly the opposite of how the `stats` package in R defines `lag()`. So the `tis` package also defines the `Lag()` function which works the way economists usually expect it to, making our simple return code a bit nicer:

```
simpReturn <- diff(sbux)/Lag(sbux); head(round(simpReturn, 4))
```

```
OUTPUT>           Apr      May      Jun      Jul      Aug      Sep
OUTPUT>  1993  0.0177  0.2435  0.0210 -0.0342  0.0213  0.1319
```

```
OUTPUT>  class: tis
```

These kinds of calculations are done so often that `tis` has a function for doing this and stating the return at an annual rate:

```
annRateReturn <- growth.rate(sbux); head(round(annRateReturn, 4))
```

```
OUTPUT>            Apr      May      Jun      Jul      Aug      Sep
OUTPUT>  1993  21.2389 292.1739  25.1748 -41.0959  25.5319 158.3333
OUTPUT>  class: tis
```

If you don't want the annual percentage rate, just the simple return, divide by 100 times the frequency of sbux:

```
simpReturn <- growth.rate(sbux)/1200; head(round(simpReturn, 4))
```

```
OUTPUT>           Apr     May     Jun     Jul     Aug     Sep
OUTPUT>  1993  0.0177  0.2435  0.0210 -0.0342  0.0213  0.1319
OUTPUT>  class: tis
```

Compute continuously compounded 1-month returns

As you might remember, the relation between single-period and multi-period returns is multiplicative for single returns. That can be inconvenient. The yearly return, for example, is the geometric mean of the monthly returns.

So in practice we often prefer to work with continuously compounded returns. These returns have an additive relationship between single and multi-period returns and are defined as:

- r[t] = ln(1 + R[t])

with R[t] the simple return and r[t] the continuously compounded return at moment t.

Continuously compounded returns can be computed easily in R by realizing that

In R, the `log()` function computes natural logs, so

```
compReturn <- diff(log(sbux)); head(round(compReturn, 4))
```

```
OUTPUT>           Apr     May     Jun     Jul     Aug     Sep
OUTPUT>  1993  0.0175  0.2179  0.0208 -0.0348  0.0211  0.1239
OUTPUT>  class: tis
```

Compare simple and continuously compounded returns

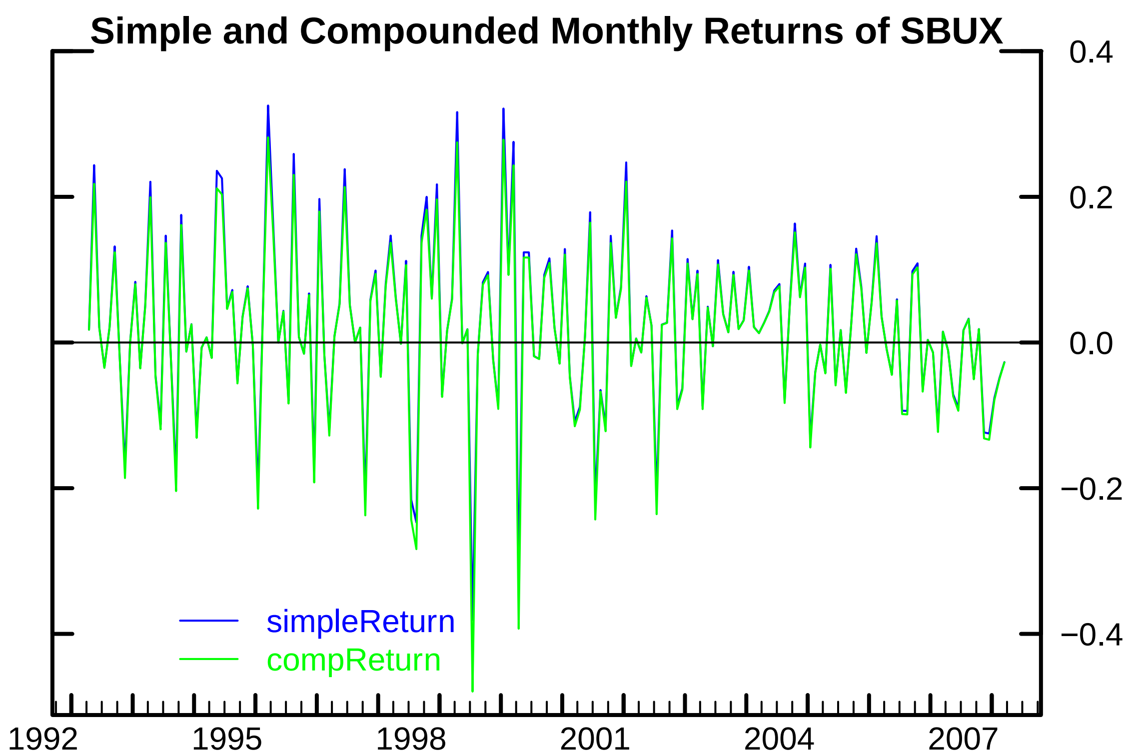
We could do this by plotting them both on the same plot, or by putting them in a matrix together. Let's do both:

```
both <- cbind(simpleReturn = diff(sbux)/Lag(sbux), compReturn = diff(log(sbux)))
window(round(both, 4), end = start(both) + 11)
```

```
OUTPUT>           simpleReturn compReturn
OUTPUT>  19930430       0.0177     0.0175
OUTPUT>  19930531       0.2435     0.2179
OUTPUT>  19930630       0.0210     0.0208
OUTPUT>  19930731      -0.0342    -0.0348
OUTPUT>  19930831       0.0213     0.0211
OUTPUT>  19930930       0.1319     0.1239
OUTPUT>  19931031      -0.0245    -0.0248
```

```
OUTPUT>  19931130      -0.1698    -0.1861
OUTPUT>  19931231       0.0000     0.0000
OUTPUT>  19940131       0.0833     0.0800
OUTPUT>  19940228      -0.0350    -0.0356
OUTPUT>  19940331       0.0507     0.0495
OUTPUT>  class: tis
```

```
tisPlot(both, color = c("blue", "green"),
        lineType = "solid",  ## otherwise the second line will be dashed
        lineWidth = 1,       ## a bit thinner than the default 1.5
        xTickFreq = "annual", xTickSkip = 2,
        xUnlabeledTickFreq = "annual",
        xMinorTickFreq = "quarterly",
        head = "Simple and Compounded Monthly Returns of SBUX", headCex = 1.2)
## add a horizontal line at zero
abline(h = 0)
## put a legend on there just because we're showing off
tisLegend(yrel = 0.8)
```



The returns are so close that the blue line was mostly overwritten by the green one.

Aggregation and disaggregation

The `tis` package has a powerful `convert` function that can aggregate a tis series to a lower frequency, or disaggregate it to a higher frequency. This is sometimes useful when you have to analyz the relationship between a quarterly series and a monthly series. To fit a regression model, for example, between quarterly

GDP and monthly M2, you'd start by constructing quarterly M2. It is a bad idea to go the other way, turning a quarterly series into a monthly one, because while the monthly series thus created has more observations than the quarterly one had, it does not have any more actual information in it. The statistics computed from the manufactured monthly data will not have the standard distributions, making inferences from them highly questionable.

Much the same is true of missing data. Missing data is just that – missing. Imputing observations or interpolating between existing observations does not add any new information, so again your statistics will be biased.

A quick regression model:

```
load("gdpAndMoney.rds")
ls()  ## should show 'gdp' and 'm2'
```

```
OUTPUT>   [1] "annRateReturn" "aTime"         "baseDate"      "both"
OUTPUT>   [5] "compReturn"    "dayInYear"     "dec57"         "flines"
OUTPUT>   [9] "gdp"           "k"             "latestPlot"    "m2"
OUTPUT>  [13] "march2007"     "myDifftime"    "sbux"          "sbuxShort"
OUTPUT>  [17] "simpReturn"    "thisWeek"      "today"         "twoWeeks"
OUTPUT>  [21] "url"           "x"
```

```
dateRange(gdp)
```

```
OUTPUT>  [1] 19590331 20160630
OUTPUT>  class: ti
```

```
dateRange(m2)
```

```
OUTPUT>  [1] 19590131 20160630
OUTPUT>  class: ti
```

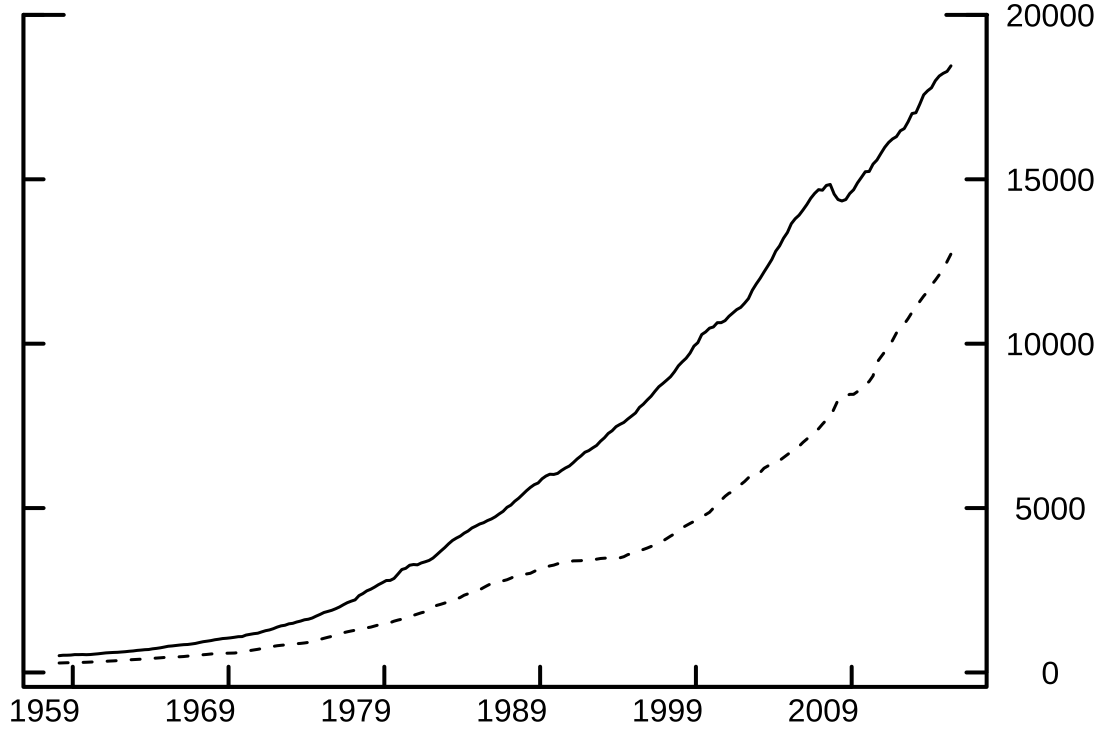
```
tifName(gdp)
```

```
OUTPUT>  [1] "quarterlydecember"
```

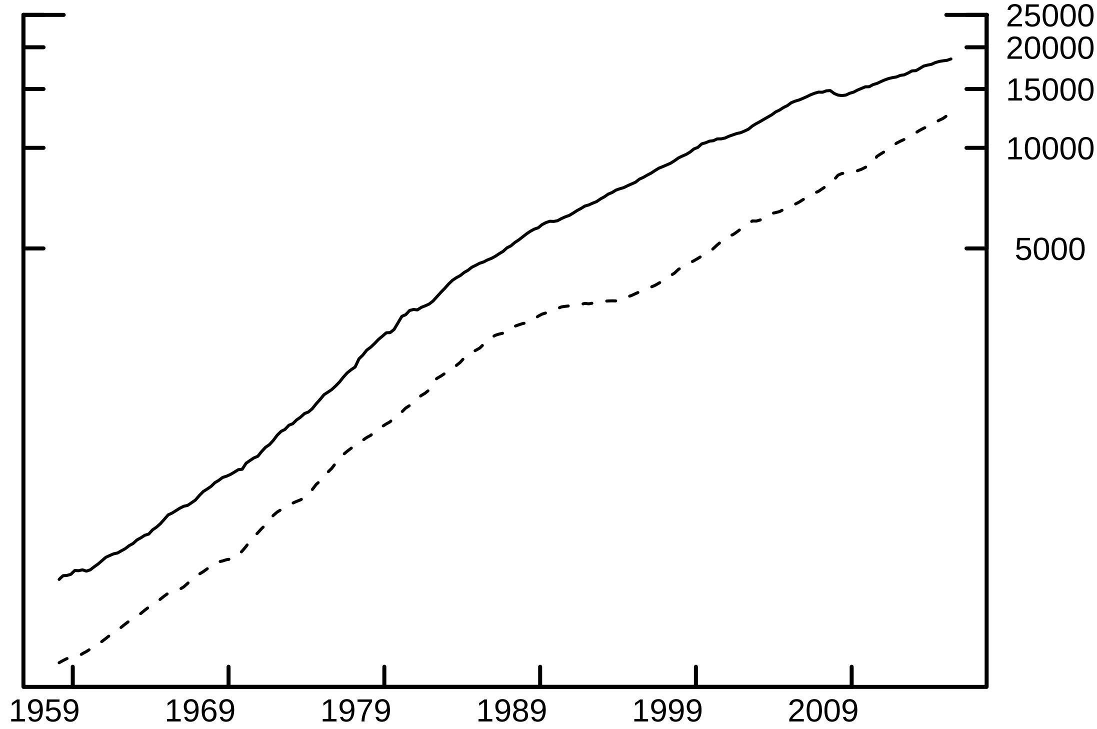
```
tifName(m2)
```

```
OUTPUT>  [1] "monthly"
```

```
qm2 <- convert(m2, tif = "quarterly", observed = "averaged")
tisPlot(gdp, qm2)
```




```r
tisPlot(gdp, qm2, log = T)
```



It is pretty clear from the two plots that we should be working in logs. Next thing we could do is fit a model and look at it:

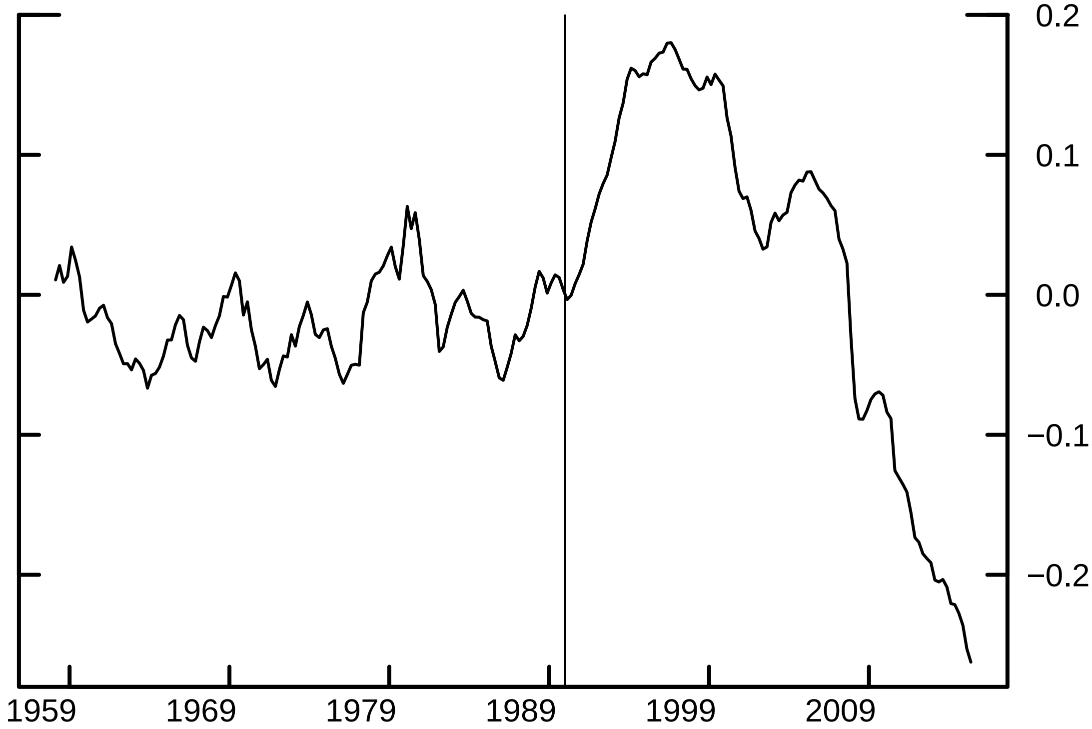
```
lgdp <- log(gdp)
lm2  <- log(qm2)
mod <- lm(lgdp ~ lm2)
summary(mod)
```

```
OUTPUT>
OUTPUT>  Call:
OUTPUT>  lm(formula = lgdp ~ lm2)
OUTPUT>
OUTPUT>  Residuals:
OUTPUT>       Min       1Q   Median       3Q      Max
OUTPUT>  -0.26238 -0.04529 -0.00720  0.05270  0.18018
OUTPUT>
OUTPUT>  Coefficients:
OUTPUT>              Estimate Std. Error t value Pr(>|t|)
OUTPUT>  (Intercept) 0.458843   0.041792   10.98   <2e-16 ***
OUTPUT>  lm2         1.018492   0.005387  189.07   <2e-16 ***
OUTPUT>  ---
OUTPUT>  Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
OUTPUT>
OUTPUT>  Residual standard error: 0.09034 on 228 degrees of freedom
OUTPUT>  Multiple R-squared:  0.9937,   Adjusted R-squared:  0.9936
OUTPUT>  F-statistic: 3.575e+04 on 1 and 228 DF,  p-value: < 2.2e-16
```

```r
tisPlot(lgdp, mod$fitted.values)
```

Not too bad, but something doesn't look right about that plot. Let's focus in on the residuals.

```r
tisPlot(mod$residuals)
## Not good! Looks like the model broke down at the end of 1990
abline(v = 1991)
```

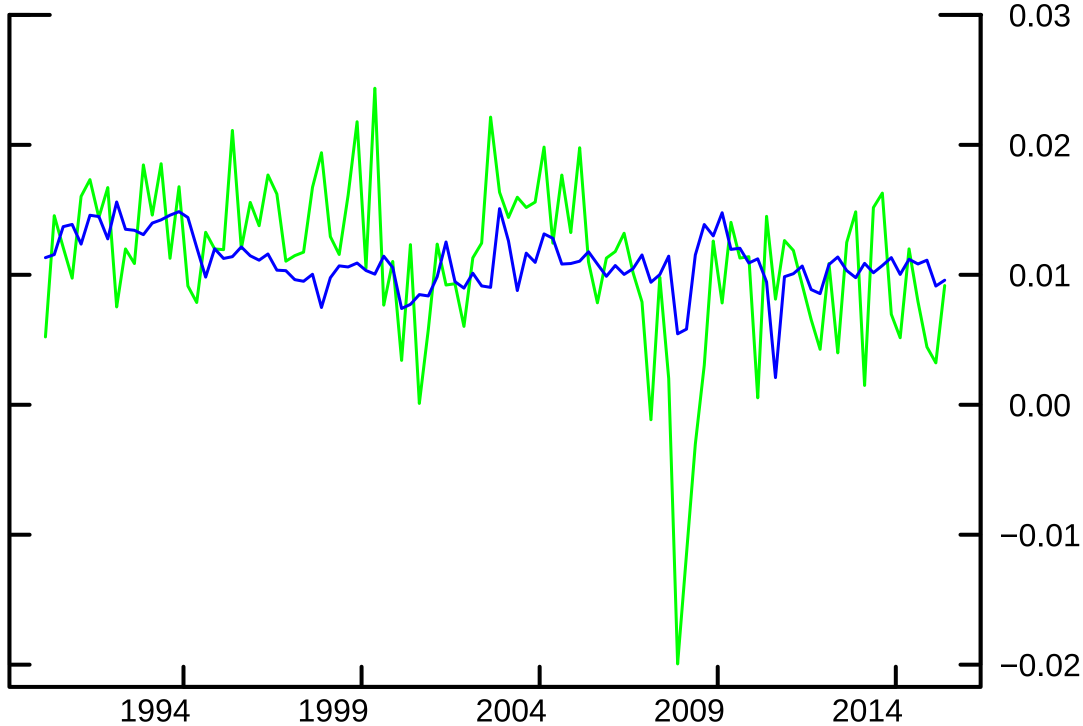


It is obvious from this plot that the relationship between gdp and m2 changed in the early 1990's. A reasonable approach here is to fit our first model for data prior to 1991, and something else after that. It looks to your instructor like maybe post-1991 should be modeled in log first differences, i.e., growth rates. Let's see:

```
ggdp <- window(diff(log(gdp)), start = c(1991, 1))
gm2  <- window(diff(log(qm2)), start = c(1991, 1))
gmod <- lm(ggdp ~ gm2)
summary(gmod)
```

```
OUTPUT>
OUTPUT>  Call:
OUTPUT>  lm(formula = ggdp ~ gm2)
OUTPUT>
OUTPUT>  Residuals:
OUTPUT>        Min         1Q     Median         3Q        Max
OUTPUT>  -0.0253994 -0.0032283  0.0006464  0.0034979  0.0143037
OUTPUT>
OUTPUT>  Coefficients:
OUTPUT>               Estimate Std. Error t value Pr(>|t|)
OUTPUT>  (Intercept)  0.014772   0.001253  11.788  < 2e-16 ***
OUTPUT>  gm2         -0.284671   0.082107  -3.467 0.000777 ***
OUTPUT>  ---
OUTPUT>  Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
OUTPUT>
OUTPUT>  Residual standard error: 0.006152 on 100 degrees of freedom
OUTPUT>  Multiple R-squared:  0.1073,   Adjusted R-squared:  0.09838
OUTPUT>  F-statistic: 12.02 on 1 and 100 DF,  p-value: 0.000777
```

```
tisPlot(ggdp, gmod$fitted.values, lineType = "solid", color = c("green", "blue"))
```



Actually, that doesn't look all that great either. Now you know why the Fed hasn't been replaced by a four-line R program yet.

Homework exercises 1 through 5

Use the sbux dataset in your workspace to answe the following question. Submit a copy of your R code justifying your answer

Question 1: Compute one simple Starbucks return

1: What is the simple monthly return between the end of December 2004 and the end of January 2005?

Possible answers

- A: 13.55%
- B: -12.82%
- C: -14.39%
- D: -13.41%
- E: 15.48%

Hint

- Remember that you can access the first element of the sbux vector with sbux[1].

- The simple return is the difference between the first price and the second Starbucks price, divided by the first price.

Question 2: Compute one continuously compounded Starbucks return

2: What is the continuously compounded monthly return between December 2004 and January 2005?

Possible answers

- A: 15.48%
- B: -13.41%
- C: -12.82%
- D: -14.39%
- E: 13.55%

Hint * Do you still remember how you calculated the simple return in the previous exercise?

- The continuously compounded return is just the natural logarithm of the simple return plus one.

Question 3: Monthly compounding

3: Assume that all twelve months have the same return as the simple monthly return between the end of December 2004 and the end of January 2005. What would be the annual return with monthly compounding in that case?

Possible answers

- A: 172.73%
- B: -160.92%
- C: -82.22%
- D: -80.72%
- E: -84.50%

Hint * In the first exercise you calculated the simple return between December 2004 and January 2005.

- Have a look a the wikipedia article on compound interest and think about how that applies to this situation.

Question 4: Simple annual Starbucks return

4: Use the data in sbux and compute the actual simple annual return between December 2004 and December 2005.

Your workspace still contains the vector sbux with the adjusted closing price data for Starbucks stock over the period December 2004 through December 2005.

Possible answers

- A: -2.15%
- B: -8.44%
- C: -12.34%
- D: -2.17%
- E: -6.20%

Hint * Use sbux[1] to extract the first price and sbux[length(sbux)] to extract the last price.

*To get the simple annual return, calculate the price difference and divide by the initial price.

Question 5: Annual continuously compounded return

5: Use the data sbux and compute the actual annual continuously compounded return between December 2004 and December 2005.

Possible answers

- A: 6.20%
- B: -2.17%
- C: -12.34%
- D: -2.15%
- E: 8.44%

Hint * Do you still remember how you calculated the annual Starbucks return in the previous exercise? * Well, the continuously compounded annual return is just the natural logarithm of that return plus one.