Working with Financial Time Series Data in R

Eric Zivot

Department of Economics, University of Washington

June 30, 2014

Preliminary and incomplete: Comments welcome

Introduction

In this tutorial, I provide a comprehensive summary of specifying, manipulating, and visualizing various kinds of financial time series data in R. Base R has limited functionality for handling general time series data. Fortunately, there are several R packages - **lubridate**, **quantmod**, **timeDate**, **timeSeries**, **zoo**, **xts**, **xtsExtra** - with functions for creating, manipulating and visualizing time date and time series objects. I will illustrate how to use the functions in these R packages for handling financial time series.

This tutorial is organized as follows.

1. Overview of time series objects in R
2. Overview of date and date-time objects in R
    a. Date class
    b. POSIXt classes
    c. Working with dates and times using the lubridate package
    d. timeDate class
3. The ts and mts classes for representing regularly spaced calendar time series
4. The zoo class for representing general time series
5. The xts class: an extension of zoo
6. The timeSeries class for representing general time series

Overview of Time Series Objects in R

The core data object for holding data in R is the `data.frame` object. A `date.frame` is a rectangular data object whose columns can be of different types (e.g., `numeric`, `character`, `logical`, `Date`, etc.). The `data.frame` object, however, is not designed to work efficiently with time series data. In particular, sub-setting and merging data based on a time index is cumbersome and transforming and aggregating data based on a time index is not at all straightforward. Furthermore, the default plotting methods in R are not designed for handling time series data. Hence, there is a need for a flexible time series class in R with a rich set of methods for manipulating and plotting time series data.

Base R has limited functionality for handling general time series data. For example, univariate and multivariate regularly spaced calendar time series data can be represented using the `ts` and `mts` classes, respectively. These classes have a limited set of method functions for manipulating and plotting time series data. However, these classes cannot adequately represent more general irregularly spaced non-calendar time series such intra-day transactions level financial price and quote data. Fortunately, there are several R packages that can be used to handle general time series data.

The table below lists the main time series objects that are available in R and their respective packages.

Time Series Object	Package	Description
`fts`	**fts**	An R interfact to tslib (a time series library in C++)
`its`	**its**	An S4 class for handling irregular time series
`irts`	**tseries**	irts objects are irregular time-series objects. These are scalar or vector valued time series indexed by a time-stamp of class "POSIXct".
`timeSeries`	**timeSeries**	**Rmetrics** package of time series tools and utilities. Similar to the Tibco S-PLUS `timeSeries` class
`ti`	**tis**	Functions and S3 classes for time indexes and time indexed series, which are compatible with FAME frequencies
`ts, mts`	**stats**	Regularly spaced time series objects
`zoo`	**zoo**	S3 class of indexed totally ordered observations which includes irregular time series.
`xts`	**xts**	Extension of the zoo class

The `ts` and `mts` classes in base R are suitable for representing regularly spaced calendar time series such as monthly sales or quarterly real GDP. In addition, several of the time series modeling functions in base R and in several R packages take `ts` and `mts` objects as data inputs. For handling more general irregularly spaced financial time series, by far the most used packages are **timeSeries**, **zoo** and **xts**. The **timeSeries** package is part of the suite of **Rmetrics** packages for financial data analysis and computational finance created by Diethelm Weurtz and his colleagues at ETZ Zurich (see www.Rmetrics.org). In these packages, `timeSeries` objects are the core data objects. However, outside of **Rmetrics**, `timeSeries` objects are not as frequently used as `zoo` and `xts` objects for representing time series data. Hence, in this tutorial I will focus mostly on using `zoo` and `xts` objects for handing general time series.[1]

Time series data represented by `timeSeries`, `zoo` and `xts` objects have a similar structure: the time index is stored as a vector in some (typically ordered) date-time object, and the data is stored in some rectangular data object. The resulting `timeSeries`, `zoo` or `xts` objects combine the time index and data into a single object. These objects can then be manipulated and visualized using various method functions.

Before discussing the time series objects in detail, I will give a comprehensive overview of the most useful date and date-time objects available in R. This knowledge is required to fully understand how to effectively work with time series objects in R.

Overview of Date and Date-Time Objects in R

There are several ways to represent a time index (sequence of dates or date-times) in R. Table 1 summarizes the main time index classes available in R.

Table 1 Date index classes in R

Class	Package	Description
`chron`	**chron**	Represent calendar dates and times within the day as the (signed) number of seconds since the beginning of 1970 as a numeric vector. Does not control for time zones.
`Date`	**base**	Represent calendar dates as the number of days since 1970-01-01
`yearmon`	**zoo**	Represent monthly data. Internally it holds the data as year plus 0 for January, 1/12 for February, 2/12 for March and so on in order that its internal representation is the same as `ts` class with `frequency = 12`.
`yearqtr`	**zoo**	Represent quarterly data. Internally it holds the data as year plus 0 for Quarter 1, 1/4 for Quarter 2 and so on in order that its internal representation is the same as `ts` class with `frequency = 4`.
`POSIXct`	**base**	Represent calendar dates and times within the day as the (signed) number of seconds since the beginning of 1970 as a numeric vector. Supports various time zone specifications (e.g. GMT, PST, EST etc.)
`POSIXlt`	**Base**	Represents local dates and times within the day as named list of vectors with date-time components.
`timeDate`	**timeDate**	The **Rmetrics** `timeDate` Sv4 class fulfils the conventions of the ISO

[1] A somewhat dated but still very useful survey of working with financial time series in R, especially with the functions in the **Rmetrics** suite of packages, is available in the free ebook "A Discussion of Time Series in R for Finance" by Diethelm Würtz, Yohan Chalabi and Andrew Ellis. This book can be downloaded from the Rmetrics website www.Rmetrics.org.

(Sv4)		8601 standard as well as of the ANSI C and POSIX standards. Beyond these standards **Rmetrics** has added the "Financial Center" concept which allows to handle data records collected in different time zones and mix them up to have always the proper time stamps with respect to your personal financial center, or alternatively to the GMT reference time. `timeDate` is almost compatible with the `timeDate` class in Tibco's S-PLUS.

The base R `Date` class handles dates (without times), and is the recommended class for representing financial data that are observed on discrete dates without regard to the time of day (e.g., daily closing prices). The base R `POSIXct` and `POSIXlt` classes allow for dates and times with control for time zones. This is the recommended class for representing dates associated with financial data observed at particular times within a day (e.g., prices or quotes observed during the trading hours of a day). The `chron` class is similar but is not used as often as the `POSIXt` classes.[2] The `yearmon` and `yearqtr` classes from the **zoo** package are convenient for representing regularly spaced monthly and quarterly data, respectively, when it is not necessary to specify exactly when during the month or quarter the data is observed. The **Rmetrics** `timeDate` class is an Sv4 class very similar to the S-PLUS `timeDate` class[3], is based on the POSIX standards, and is used throughout the **Rmetrics** suite of packages.

The Date Class (base R)

Use the `Date` class to represent a time index only involving dates but not times within a day. The `Date` class by default represents dates internally as the number of days since January 1, 1970. You create `Date` objects from a character string representing a date using the `as.Date()` function. The default format is "`YYYY/m/d`" or "`YYYY-m-d`"", where YYYY represents the four digit year, `m` represents the month digit and `d` represents the day digit. For example,

```
> my.date = as.Date("1970/1/1")
> my.date
[1] "1970-01-01"
> class(my.date)
[1] "Date"
> as.numeric(my.date)
[1] 0
> myDates = c("2013-12-19", "2003-12-20")
> as.Date(myDates)
[1] "2013-12-19" "2003-12-20"
```

Use the `format` argument to specify the input format of the date if it is not in the default format

```
> as.Date("1/1/1970", format="%m/%d/%Y")
[1] "1970-01-01"
> as.Date("January 1, 1970", format="%B %d, %Y")
[1] "1970-01-01"
> as.Date("01JAN70", format="%d%b%y")
```

[2] Spector (2004) gives an excellent overview of the `chron`, `Date`, and `POSIXt` classes in R.
[3] Some might say "ripped off" from.

```
[1] "1970-01-01"
```

Notice that the output format is always in the form "YYYY-m-d" regardless of the input format. To change the displayed output format of a date use the `format()` function

```
> format(my.date, "%b %d, %Y")
[1] "Jan 01, 1970"
```

Some date formats provide insufficient information to be unambiguously represented as a `Date` object. For example,

```
> as.Date("Jan 1970", format="%b %Y")
[1] NA
```

Table 2 below gives the standard date format codes.

Code	Value	Example
%d	Day of the month (decimal number)	23
%m	Month (decimal number)	11
%b	Month (abbreviated)	Jan
%B	Month (full name)	January
%y	Year (2 digit)	90
%Y	Year (4 digit)	1990

Table 2. Format codes for dates

Recall, dates are internally recorded as the (integer) number of days since 1970-01-01. As a result, you can also create a `Date` object from integer data. One way to convert an integer variable to a `Date` object is to use the `class()` function

```
> my.date = 0
> class(my.date) = "Date"
> my.date
[1] "1970-01-01"
```

Another way is to use the `as.Date()` function with optional argument `origin` if the origin date is different than the default 1970-01-01. For example, to determine the date that is 32500 days from 1900-01-01 use

```
> as.Date(32500, origin=as.Date("1900-01-01"))
[1] "1988-12-25"
```

Extracting Information from `Date` objects

Consider the `Date` object

```
> my.date
[1] "1970-01-01"
```

Suppose I want to extract the year component from this object as a character string or as an integer. I can do this using the `format()` function

```
> myYear = format(my.date, "%Y")
> myYear
[1] "1970"
> class(myYear)
[1] "character"
> as.numeric(myYear)
[1] 1970
> as.numeric(format(my.date, "%Y"))
[1] 1970
```

By specifying different format codes in the `format()` function, I can extract other components of the date such as the month or day.

Additionally, the `weekdays()`, `months()`, `quarters()` and `julian()` functions can be used to extract specific components of `Date` objects

```
> weekdays(my.date)
[1] "Thursday"
> months(my.date)
[1] "January"
> quarters(my.date)
[1] "Q1"
> julian(my.date, origin=as.Date("1900-01-01"))
[1] 25567
attr(,"origin")
[1] "1900-01-01"
```

Manipulating Date Objects

Having a numeric representation for dates allows for some simple date arithmetic. For example,

```
> my.date
[1] "1970-01-01"
> my.date + 1
[1] "1970-01-02"
> my.date - 1
[1] "1969-12-31"
> my.date + 31
[1] "1970-02-01"
```

Logical comparisons can also be made

```
> my.date
[1] "1970-01-01"
> my.date1 = as.Date("1980-01-01")
> my.date1 > my.date
[1] TRUE
```

Subtracting two `Date` objects creates a `difftime` object and shows the number of days between the two dates

```
> diff.date = my.date1 - my.date
> diff.date
```

```
Time difference of 3652 days
> class(diff.date)
[1] "difftime"
> as.numeric(diff.date)
[1] 3652
> my.date + diff.date
[1] "1980-01-01"
```

Creating `Date` Sequences

Very often sequences of dates are required in the construction of time series objects. The base R function `seq()` (with method function `seq.Date()` for objects of class `Date`) can create many types of date sequences. The arguments to `seq.Date()` are

```
> args(seq.Date)
function (from, to, by, length.out = NULL, along.with = NULL,
    ...)
```

where `from` specifies the starting date, `to` specifies the ending date and `by` specifies the increment of the sequence. The `by` increment is a character string, containing one of "`day`", "`week`", "`month`" or "`year`", and can be preceded by a (positive or negative) integer and a space, or followed by "`s`". For example, to create a bi-monthly sequence of `Date` objects starting 1993-03-01 and ending in 2003-03-01 use

```
> my.dates = seq(as.Date("1993/3/1"), as.Date("2003/3/1"), "2 months")
> head(my.dates)
[1] "1993-03-01" "1993-05-01" "1993-07-01" "1993-09-01" "1993-11-01"
[6] "1994-01-01"
> tail(my.dates)
[1] "2002-05-01" "2002-07-01" "2002-09-01" "2002-11-01" "2003-01-01"
[6] "2003-03-01"
```

Alternatively, use

```
> my.dates = seq(from=as.Date("1993/3/1"), by="2 months", length.out=61)
```

The `seq()` function can also be used to determine the date that is a specified number of days, weeks, months or years from a given date. For example, to find the date that is 5 months away from today's date use

```
> Sys.Date()
[1] "2014-01-10"
> seq(from=Sys.Date(), by="5 months", length.out=2)[2]
[1] "2014-06-10"
```

While the above is a clever solution, it is not very intuitive. The **lubridate** package, described later on, provides a much easier solution.

Plotting `Date` Objects

Given a data set of `Date` objects, it is possible to graphically summarize the distribution of dates using the `hist()` function (with method function `hist.Date()`). For example, the following code simulates 500 random dates between 2013-01-01 and 2014-01-01 and plots a histogram summarizing the number of dates within each month

```
> rint = round(runif(500)*365)
> startDate = as.Date("2013-01-01")
> myDates = startDate + rint
> head(myDates)
[1] "2013-10-05" "2013-10-23" "2013-11-20" "2013-05-27" "2013-07-11" "2013-
06-07"
> hist(myDates, breaks="months", freq=TRUE,
+      main="Distribution of Dates by Month",
+      col="slateblue1", xlab="",
+      format="%b %Y", las=2)
```

The resulting histogram is shown in Figure 1.

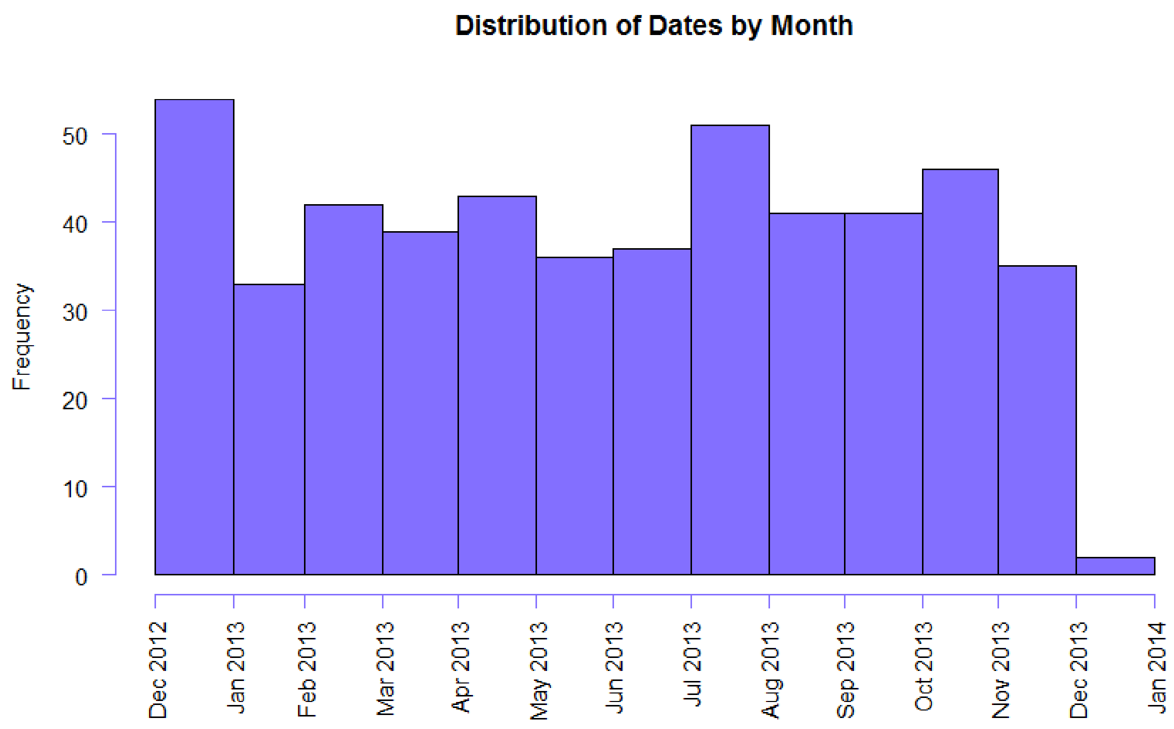


Figure 1 Histogram of Date Objects

The POSIXt classes (base R)

The POSIXt classes in R are derived from the POSIX system.

There are two POSIXt sub-classes available in R: POSIXct and POSIXlt. The POSIXct class represents date-time values as the signed number of seconds (which includes fractional seconds) since midnight GMT (UTC – universal time, coordinated) 1970-01-01. This is analogous to the Date class with addition of times during the day. The POSIXlt class represents date-time values as a named list with elements for the second (sec), minute (min), hour (hour), day of the month (mday), month (mon), year (year), day of the week (wday), day of the year (yday), and daylight savings time flag (isdst), respectively.

Creating POSIXct Objects

You can create POSIXct objects from a character string representation of a date-time using the as.POSIXct() function. The default format of the date-time is "YYYY-mm-dd hh:mm:ss" or "YYYY/mm/dd hh:mm:ss" with the hour, minute and second information being optional.

```
> myDateTimeStr = "2013-12-19 10:17:07"
> myPOSIXct = as.POSIXct(myDateTimeStr)
> myPOSIXct
[1] "2013-12-19 10:17:07 PST"
> class(myPOSIXct)
[1] "POSIXct" "POSIXt"
> as.numeric(myPOSIXct)
[1] 1.387e+09
```

If no time zone specification is given in the optional argument tz, then the default value tz="" specifies the local system specific time zone as given by the Sys.timezone() function

```
> Sys.timezone()
[1] "PST"
```

The time zone specification is an attribute of the POSIXct object

```
> attributes(myPOSIXct)
$class
[1] "POSIXct" "POSIXt"

$tzone
[1] ""
```

Use the optional format argument if the date-time string is not in the default format

```
> myDateTimeStr1 = "19-12-2003 10:17:07"
> myPOSIXct1 = as.POSIXct(myDateTimeStr1, format="%d-%m-%Y %H:%M:%S")
> myPOSIXct1
[1] "2003-12-19 10:17:07 PST"
```

The most common set of format codes for representing character dates under the POSIX standard are listed in Table xxx. These codes, and others, are explained in the help file for the function `strptime()`.

Code	Description	Example	Code	Description	Example
%a	Abbreviated weekday	Mon	%A	Full weekday	Monday
%b	Abbreviated month	Jan	%B	Full month	January
%c	Locale specific date and time		%d	Decimal day of month	01
%H	Decimal hours (24)	16	%I	Decimal hours (12)	08
%j	Decimal day of year	234	%m	Decimal month	07
%M	Decimal minute	12	%p	AM/PM indicator	
%S	Decimal second	35	%U	Decimal week of year (starting on Sunday)	
%w	Decimal weekday	1	%W	Decimal week of year (starting on Monday)	
%x	Locale specific date		%X	Locale specific time	
%y	2-digit year	91	%Y	4-digit year	1991
%z	Full time zone name		%Z	Abbreviated Time-zone name	PST

Because `POSIXct` objects have an internal representation as the number of seconds from some origin date-time, you can also create them from numeric data

```
> numDate = 0
> myPOSIXct2 = as.POSIXct(0, origin="1970-01-01")
> myPOSIXct2
[1] "1969-12-31 16:00:00 PST"
> as.numeric(myPOSIXct2)
[1] 0
```

Because PST (Pacific Standard Time) is 8 hours earlier than GMT/UTC, the date-time is displayed as 1969-12-31 16:00:00 PST and not 1970-01-01 UTC. Although the numeric representation is still 0 (because `POSIXct` objects are defined as the number of seconds from 1970-01-01 UTC), the time zone specification affects how the date-time is displayed and how numeric calculations with `POSIXct` objects are evaluated. For example, consider what happens if I add 8 hours to `myPOSIXct2`

```
> myPOSIXct3 = myPOSIXct2 + 8*60*60
> myPOSIXct3
[1] "1970-01-01 PST"
> as.numeric(myPOSIXct3)
[1] 28800
```

In many situations it is best to define date-times in GMT (UTC) to avoid time zone complications when manipulating date-times

```
> myPOSIXct4 = as.POSIXct(0, origin="1970-01-01", tz="UTC")
> myPOSIXct4
[1] "1970-01-01 UTC"
> as.numeric(myPOSIXct4)
[1] 0
```

You can use `Sys.setenv(TZ="UTC")` to set the system time zone to GMT (UTC) so that it becomes the default time zone when calling `as.POSIXct()`.

You can also create a `POSIXct` object directly from numeric data giving the individual components of the date-time and a character time zone specification using the `ISOdatetime()` function[4]

```
> myPOSIXct5 = ISOdatetime(year=2013, month=12, day=19,
+                          hour = 10, min = 17, sec = 7,
+                          tz = "")
> class(myPOSIXct5)
[1] "POSIXct" "POSIXt"
> myPOSIXct5
[1] "2013-12-19 10:17:07 PST"
```

Changing the Output Format and Extracting Date-Time Components

You can change the output format of `POSIXct` objects using `format()`together with the format codes given in Table x.

```
> myPOSIXct
[1] "2013-12-19 10:17:07 PST"
> format(myPOSIXct, format="%b %d, %Y")
[1] "Dec 19, 2013"
```

This provides a handy way of extracting any component of a `POSIXct` object. For example, to extract the full month name, time zone abbreviation, numeric year value, and numeric second value, use

```
> format(myPOSIXct, format="%B")
[1] "December"
> format(myPOSIXct, format="%Z")
[1] "PST"
> as.numeric(format(myPOSIXct, format="%Y"))
[1] 2013
> as.numeric(format(myPOSIXct, format="%S"))
[1] 7
```

As with `Date` objects, you can also use the `weekdays()`, `months()`, `quarters()` and `Julian()` functions on `POSIXct` objects. As explained in the next sub-section, another way to extract components from a `POSIXct` object is to convert it to a `POSIXlt` object and then extract the desired list component.

[4] You can also use the related `ISOdate()` function, which sets `hour=12`, `min=0`, `sec=0`, and `tz="GMT"` by default.

The `format()` function also allows you to see date-times in different time zones

```
> myPOSIXct4
[1] "1970-01-01 UTC"
> format(myPOSIXct4, tz="")
[1] "1969-12-31 16:00:00"
> format(myPOSIXct4, tz="EST")
[1] "1969-12-31 19:00:00"
```

Creating `POSIXlt` Objects

You can create `POSIXlt` objects using the `as.POSIXlt()` or `strptime()` functions (the `strptime()` function is a C level function)

```
> myDateTimeStr
[1] "2013-12-19 10:17:07"
> myPOSIXlt = as.POSIXlt(myDateTimeStr)
> myPOSIXlt
[1] "2013-12-19 10:17:07"
> class(myPOSIXlt)
[1] "POSIXlt" "POSIXt"
```

If the input date-time string is not in the default format, use the optional `format` argument together with the appropriate format codes from Table 2

```
> myDateTimeStr1 = "19-12-2003 10:17:07"
> myPOSIXlt1 = as.POSIXlt(myDateTimeStr1, format="%d-%m-%Y %H:%M:%S")
```

Although `POSIXlt` objects are lists with named components, the component names are annoyingly hidden.

```
> names(myPOSIXlt)
NULL
```

To see them use the `unclass()` function

```
> names(unclass(myPOSIXlt))
[1] "sec"   "min"   "hour"  "mday"  "mon"   "year"  "wday"  "yday"
[9] "isdst"
```

You can extract any of the above list components

```
> myPOSIXlt$sec
[1] 7
> myPOSIXlt$hour
[1] 10
> myPOSIXlt$mday
[1] 19
> myPOSIXlt$mon
[1] 11
> myPOSIXlt$year
[1] 113
> myPOSIXlt$wday
[1] 4
```

```
> myPOSIXlt$yday
[1] 352
> myPOSIXlt$isdst
[1] 0
```

Converting `POSIXct` Objects to `POSIXlt` Objects and Vice-Versa

You can convert a `POSIXct` object to a `POSIXlt` objects and vice-versa using the `as.POSIXct()` and `as.POSIXlt()` functions, respectively

```
> myPOSIXct
[1] "2013-12-19 10:17:07 PST"
> class(myPOSIXct)
[1] "POSIXct" "POSIXt"
> myPOSIXlt = as.POSIXlt(myPOSIXct)
> class(myPOSIXlt)
[1] "POSIXlt" "POSIXt"
```

Once reason for converting a `POSIXct` object to a `POSIXlt` object is to extract certain components of the date-time. For example, to get the numeric value for the seconds of `myPOSIXct` use

```
> as.POSIXlt(myPOSIXct)$sec
[1] 7
```

Converting `POSIXt` Objects to `Date` Objects and Vice-Versa

- Coercing to Date removes within day time information as well as time zone information
- Coercing a Date to POSIXt imposes a time zone

You can convert a `POSIXt` object to a `Date` object using the `as.Date()` function

```
> myPOSIXct
[1] "2013-12-19 10:17:07 PST"
> myDate = as.Date(myPOSIXct)
> myDate
[1] "2013-12-19"
> class(myDate)
[1] "Date"
```

Doing so removes the within day time and time zone information.

Similarly, you can convert a `Date` object to a `POSIXt` object using the `as.POSIXct()` or `as.POSIXlt()` functions

```
> myPOSIXct = as.POSIXct(myDate)
> myPOSIXct
[1] "2013-12-18 16:00:00 PST"
> class(myPOSIXct)
[1] "POSIXct" "POSIXt"
```

To set specific time zones, you must first convert the `Date` object to a `POSIXlt` object then to a `POSIXct` object

```
> myPOSIXct = as.POSIXct(myDate, tz="GMT")
> myPOSIXct
[1] "2013-12-18 16:00:00 PST"
> myPOSIXlt = as.POSIXlt(myDate, tz="GMT")
> myPOSIXlt
[1] "2013-12-19 UTC"
> myPOSIXct = as.POSIXct(myPOSIXlt)
> myPOSIXct
[1] "2013-12-19 UTC"
```

`POSIXt` Objects and Ultra High Frequency Data

`POSIXt` objects can represent intra-day date-times with times less than a second using the fact that fractions of a second are allowed. For example,

```
> HfDateTimeStr = "2013-12-19 10:17:07.125"
> HfPOSIXct = as.POSIXct(HfDateTimeStr)
> HfPOSIXct
[1] "2013-12-19 10:17:07 PST"
```

Here, the intra-day time is specified to 7 seconds and 125 milliseconds. To see the fractional seconds, use

```
> options(digits.secs = 3)
> HfPOSIXct
[1] "2013-12-19 10:17:07.125 PST"
```

Manipulating `POSIXt` Objects

Because `POSIXt` objects have internal numeric representations, you can add and subtract `POSIXt` objects and perform logical operations on them. If you have a vector of `POSIXt` objects, you can use the `min()`, `max()` and `range()` functions.

Differencing two `POSIXt` objects creates a `difftime` object

```
> dt1 = as.POSIXct("2013-12-23 00:00:00")
> dt2 = as.POSIXct("2013-12-23 05:00:00")
> diffDateTime = dt2 - dt1
> class(diffDateTime)
[1] "difftime"
> diffDateTime
Time difference of 5 hours
> units(diffDateTime)
[1] "hours"
> as.numeric(diffDateTime)
[1] 5
```

The units of a `difftime` object can be changed with the `units()` function

```
> units(diffDateTime) = "secs"
> diffDateTime
Time difference of 18000 secs
> as.numeric(diffDateTime)
```

```
[1] 18000
```

Creating regularly spaced sequences of `POSIXct` objects is easy using the `seq()` function (using the method function `seq.POSIXct()`). For example, to create an intra-day sequence every 5 seconds from 9:30 am to 4 pm use

```
> startDate = as.POSIXct("2013-12-23 9:30:00")
> endDate =  as.POSIXct("2013-12-23 16:00:00")
> dateSeq5sec = seq(from=startDate, to=endDate, by="5 sec")
> head(dateSeq5sec)
[1] "2013-12-23 09:30:00 PST" "2013-12-23 09:30:05 PST"
[3] "2013-12-23 09:30:10 PST" "2013-12-23 09:30:15 PST"
[5] "2013-12-23 09:30:20 PST" "2013-12-23 09:30:25 PST"
> tail(dateSeq5sec)
[1] "2013-12-23 15:59:35 PST" "2013-12-23 15:59:40 PST"
[3] "2013-12-23 15:59:45 PST" "2013-12-23 15:59:50 PST"
[5] "2013-12-23 15:59:55 PST" "2013-12-23 16:00:00 PST"
> length(dateSeq5sec)
[1] 4681
```

The yearmon class (Package zoo)

Use the `yearmon` class to represent regularly spaced monthly dates. This class is particularly useful for representing date information associated with monthly economic and financial time series.

The `yearqtr` class (Package zoo)

Use the `yearqtr` class to represent regularly spaced quarterly dates. This class is useful for representing date information associated with quarterly economic time series.

Working with Dates and Times Using the lubridate Package

The functions in the **lubridate** package (available on CRAN), created by Garrett Grolemund and Hadley Wickham, make working with dates and times in R a little easier.[5] The functions in lubridate help users (1) identify and parse date-time data; (2) extract and modify components of a date-time; (3) perform accurate calculations with date-times and timespans; (4) handle time zones and daylight savings time.

To load the lubridate package and see the available functions use

```
> library(lubridate)
> library(help=lubridate)
```

Parsing Dates and Times

Instead of using `as.POSIXct()` to create `POSIXct` objects from character strings, you can use the smart parsing **lubridate** functions whose names are based on the ordering of the date-time information in the character strings

```
> ymd("20131219")
[1] "2013-12-19 UTC"
> ymd("2013 Dec 19")
```

[5] See Grolemund, G. , and Wickham, H. (2011). "Dates and Times Made Easy with lubridate". *Journal of Statistical Software*, Volume 40, Issue 3.

```
[1] "2013-12-19 UTC"
> ymd_hms("20131219101707")
[1] "2013-12-19 10:17:07 UTC"
> ymd_hms("2013 Dec 19 10:17:07")
[1] "2013-12-19 10:17:07 UTC"
> mdy("Dec 19, 2013")
[1] "2013-12-19 UTC"
> mdy_hms("December 19, 2013 10:17:07")
[1] "2013-12-19 10:17:07 UTC"
> dmy_hms("19-Dec, 2013 10:17:07")
[1] "2013-12-19 10:17:07 UTC"
```

Notice how the **lubridate** functions do not require a format string. They implement a smart parsing algorithm to automatically figure out the date-time information.

The default time zone for the **lubridate** functions is GMT/UTC. Different time zones can be set with the optional argument `tz`

```
> ymd_hms("2013 Dec 19 10:17:07", tz="")
[1] "2013-12-19 10:17:07 PST"
```

The above functions also work with numeric inputs. For example,

```
> ymd(20131219)
[1] "2013-12-19 UTC"
```

The current date-time can be captured with `now()`, and the current date with `today()`

Setting and Extracting Information

Table xx lists the **lubridate** functions for extracting and setting information from a date-time object (`Date` or `POSIXt` object)

Date Component	Extractor Function
Year	`year()`
Month	`month()`
Week	`week()`
Day of year	`yday()`
Day of month	`mday()`
Day of week	`wday()`
Hour	`hour()`
Minute	`minute()`
Second	`second()`
Time zone	`tz()`

For example,

```
> myDateTime = ymd_hms("2013 Dec 19 10:17:07")
> myDateTime
```

```
[1] "2013-12-19 10:17:07 UTC"
> year(myDateTime)
[1] 2013
> month(myDateTime)
[1] 12
> week(myDateTime)
[1] 51
> yday(myDateTime)
[1] 353
> mday(myDateTime)
[1] 19
> wday(myDateTime)
[1] 5
> hour(myDateTime)
[1] 10
> minute(myDateTime)
[1] 17
> second(myDateTime)
[1] 7
> tz(myDateTime)
[1] "UTC"
> wday(myDateTime, label=TRUE)
[1] Thurs
Levels: Sun < Mon < Tues < Wed < Thurs < Fri < Sat
> month(myDateTime, label=TRUE)
[1] Dec
12 Levels: Jan < Feb < Mar < Apr < May < Jun < Jul < Aug < ... < Dec
```

The extractor functions can also be used to set elements of a date-time to particular values

```
> mday(myDateTime) = 20
> myDateTime
[1] "2013-12-20 10:17:07 UTC"
```

You can modify multiple components of a date-time object using the `update()` function

```
> update(myDateTime, year=2014, month=1,
+        day=1, hour=5, min=0, sec=0)
[1] "2014-01-01 05:00:00 UTC"
```

Performing Calculations with Date-Times and Timespans

Handling Time Zones and Daylight Savings Time

The `timeDate` class (Packages SplusTimeDate and timeDate)

To be completed.

Time Series Objects in R

Representing Regularly Spaced Data as `ts`/`mts` Objects

Regularly spaced time series data, data that are separated by a fixed interval of time, can be represented as objects of class `ts`. Such data are typically observed monthly, quarterly or annually. `ts` objects are created using the `ts()` constructor function (base R). For example,

```
> sbux.ts = ts(data=sbux.df$Adj.Close, frequency = 12,
             start=c(1993,3), end=c(2008,3))
> class(sbux.ts)
[1] "ts"

> msft.ts = ts(data=msft.df$Adj.Close, frequency = 12,
             start=c(1993,3), end=c(2008,3))
```

The argument `frequency = 12` specifies that that prices are sampled monthly. The starting and ending months are specified as a two element vector with the first element giving the year and the second element giving the month. When printed, `ts` objects show the dates associated with the observations.

```
> sbux.ts
       Jan    Feb    Mar    Apr    May    Jun    Jul    Aug    Sep    Oct    Nov
1993                1.19   1.21   1.50   1.53   1.48   1.52   1.71   1.67   1.39
…
```

The functions `start()` and `end()` show the first and last dates associated with the data

```
> start(sbux.ts)
[1] 1993    3
> end(sbux.ts)
[1] 2008    3
```

The `time()` function extracts the time index as a `ts` object

```
> time(sbux.ts)
          Jan       Feb       Mar       Apr       May       Jun   …
1993                         1993.167 1993.250 1993.333 1993.417  …
```

The frequency per period and time interval between observations of a `ts` object may be extracted using

```
> frequency(sbux.ts)
```

```
[1] 12

> deltat(sbux.ts)
[1] 0.08333333
```

However, subsetting a `ts` object produces a numeric object

```
> tmp = sbux.ts[1:5]
> class(tmp)
[1] "numeric"

> tmp
[1] 1.19 1.21 1.50 1.53 1.48
```

To subset a `ts` object and preserve the date information use the `window()` function

```
> tmp = window(sbux.ts, start=c(1993, 3), end=c(1993,8))
> class(tmp)
[1] "ts"

> tmp
      Mar  Apr  May  Jun  Jul  Aug
1993 1.19 1.21 1.50 1.53 1.48 1.52
```

The arguments `start=c(1993, 3)` and `end=c(1993,8)` specify the beginning and ending dates of the window.

Merging `ts` objects

- ts.intersect()

To combine the `ts` objects `sbux.ts` and `msft.ts` into a single object use the `cbind()` function

```
> sbuxmsft.ts = cbind(sbux.ts, msft.ts)
> class(sbuxmsft.ts)
[1] "mts" "ts"
```

Since `sbuxmsft.ts` contains two `ts` objects, it is assigned the additional class `mts` (multiple time series). The first five rows are

```
> window(sbuxmsft.ts, start=c(1993, 3), end=c(1993,7))
         sbux.ts msft.ts
Mar 1993    1.19    2.43
Apr 1993    1.21    2.25
May 1993    1.50    2.44
```

```
Jun 1993    1.53    2.32
Jul 1993    1.48    1.95
```

Plotting `ts` objects

`ts` objects have their own plot method (`plot.ts`)

```
> plot(sbux.ts, col="blue", lwd=2, ylab="Adjusted close",
+      main="Monthly closing price of SBUX")
```

which produces the plot in Figure 1. To plot a subset of the data use the `window()` function inside of `plot()`

```
> plot(window(sbux.ts, start=c(2000,3), end=c(2008,3)),
+      ylab="Adjusted close",col="blue", lwd=2,
+     main="Monthly closing price of SBUX")
```

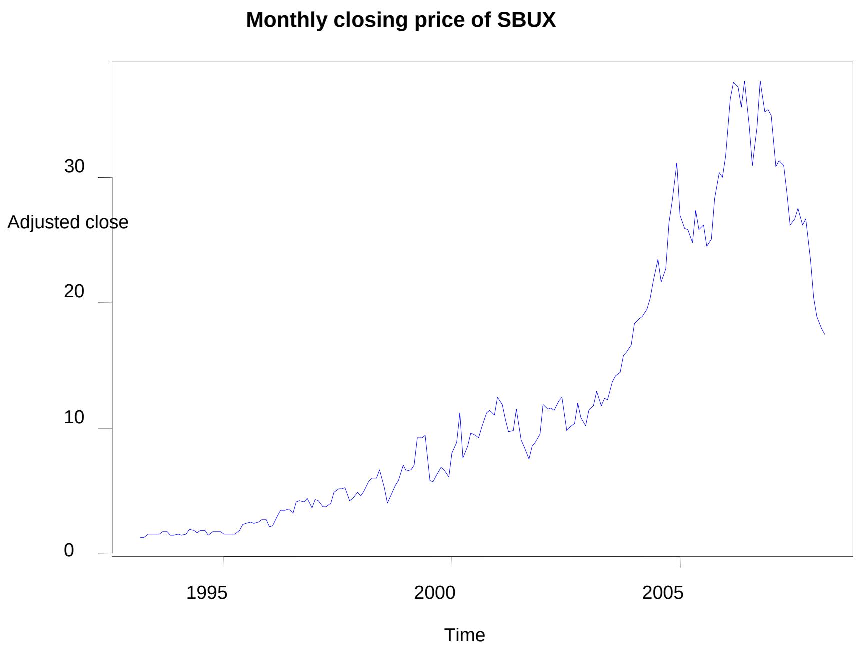


Figure 2 Plot created with `plot.ts()`

For `ts` objects with multiple columns (`mts` objects), two types of plots can be created. The first type, illustrated in Figure 2, puts each series in a separate panel

```
> plot(sbuxmsft.ts)
```

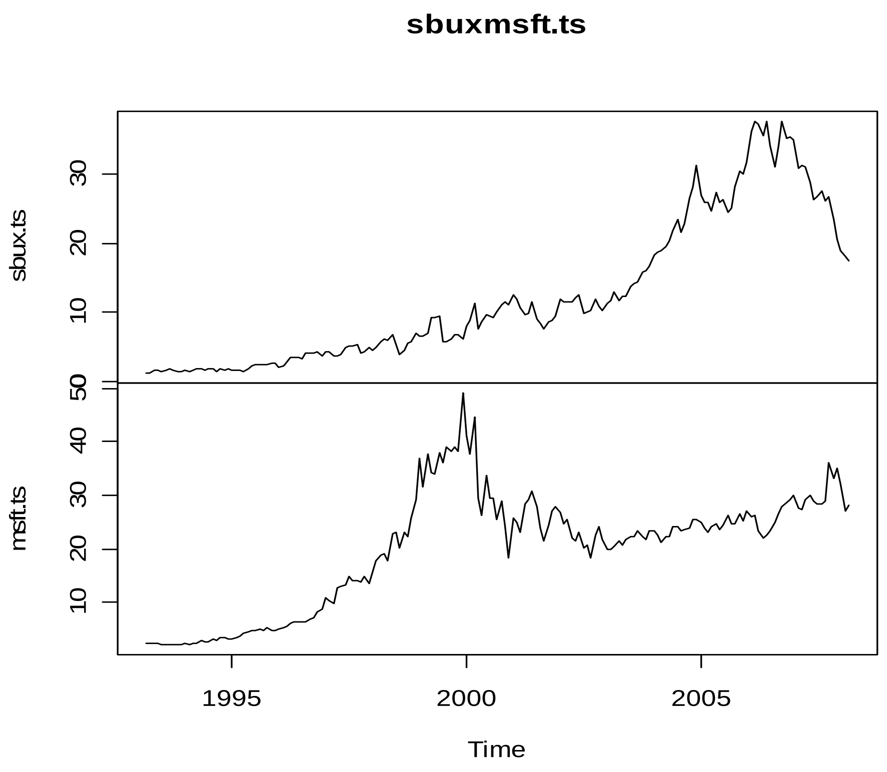


Figure 3 Multiple time series plot

The second type, shown in Figure 3, puts all series on the same plot

```
> plot(sbuxmsft.ts, plot.type="single",
+      main="Monthly closing prices on SBUX and MSFT",
+      ylab="Adjusted close price",
+      col=c("blue", "red"), lty=1:2)
> legend(1995, 45, legend=c("SBUX","MSFT"), col=c("blue", "red"),
+        lty=1:2)
```

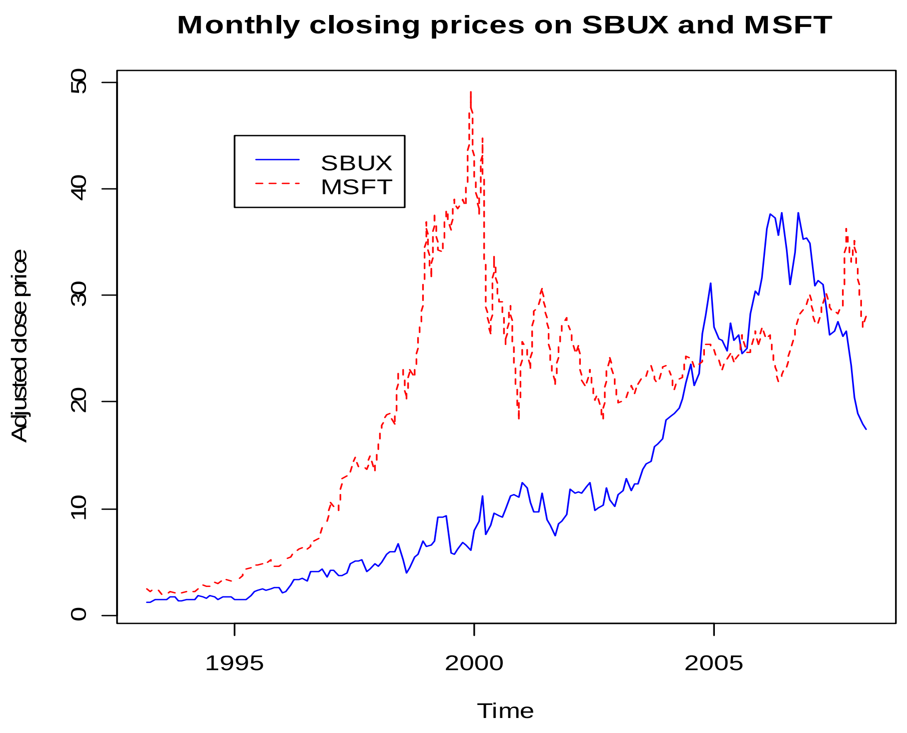


Figure 4 Multiple time series plot

Manipulating `ts` objects and computing returns

Some common manipulations of time series data involve lags and differences using the functions `lag()` and `diff()`.

Creating lagged data

For example, to lag the price data in `sbux.ts` by one time period use

```
>  lag(sbux.ts)
```

To lag the price data by 12 periods use

```
>  lag(sbux.ts, k=12)
```

Notice what happens when you combine a `ts` object with its lag

```
> cbind(sbux.ts, lag(sbux.ts))
         sbux.ts lag(sbux.ts)
Feb 1993      NA         1.19
Mar 1993    1.19         1.21
Apr 1993    1.21         1.50
May 1993    1.50         1.53
```

```
Jun 1993    1.53          1.48
```

The `lag()` function shifts the time index back by an amount k. To shift the time index forward set k to a negative number

```
> lag(sbux.ts, k=-1)
> lag(sbux.ts, k=-12)
> cbind(sbux.ts, lag(sbux.ts, k=-1))
         sbux.ts lag(sbux.ts, k = -1)
Mar 1993    1.19                   NA
Apr 1993    1.21                 1.19
May 1993    1.50                 1.21
Jun 1993    1.53                 1.50
Jul 1993    1.48                 1.53
```

Creating differences

To compute the first difference in prices, $p_t - p_{t-1}$, use

```
> diff(sbux.ts)
```

Notice that application of `diff()` is equivalent to

```
> sbux.ts – lag(sbux.ts, k=-1)
```

To compute a 12 lag difference (annual difference for monthly data) use

```
> diff(sbux.ts, lag=12)
```

which is equivalent to using

```
> sbux.ts – lag(sbux.ts, k=-12)
```

Notice what happens when you combine a `ts` object with its first difference

```
> cbind(sbux.ts, diff(sbux.ts))
         sbux.ts diff(sbux.ts)
Mar 1993    1.19            NA
Apr 1993    1.21          0.02
May 1993    1.50          0.29
Jun 1993    1.53          0.03
Jul 1993    1.48         -0.05
```

You can use the `diff()` and `lag()` functions together to compute the simple one period return

```
> sbuxRetSimple.ts = diff(sbux.ts)/lag(sbux.ts, k=-1)
> msftRetSimple.ts = diff(msft.ts)/lag(msft.ts, k=-1)
> window(cbind(sbuxRetSimple.ts, msftRetSimple.ts),
+        start=c(1993,4), end=c(1993,7))
         sbuxRetSimple.ts msftRetSimple.ts
```

```
Apr 1993       0.01680672      -0.07407407
May 1993       0.23966942       0.08444444
Jun 1993       0.02000000      -0.04918033
Jul 1993      -0.03267974      -0.15948276
```

Similarly, to compute the 12-period simple return use

```
> diff(sbux.ts, lag=12)/lag(sbux.ts, k=-12)
```

You can use the `log()` and `diff()` functions together to compute continuously compounded returns

```
> sbuxRet.ts = diff(log(sbux.ts))

> msftRet.ts = diff(log(msft.ts))

> window(cbind(sbuxRet.ts, msftRet.ts), start=c(1993,4),
+         end=c(1993,7))
          sbuxRet.ts  msftRet.ts
Apr 1993  0.01666705 -0.07696104
May 1993  0.21484475  0.08106782
Jun 1993  0.01980263 -0.05043085
Jul 1993 -0.03322565 -0.17373781
```

To compute the 12-period continuously compounded return use

```
> diff(log(sbux.ts), lag=12)
```

R Modeling Functions that Work with ts/mts Objects

To be completed

- arima
- forecast package
- strucchange

Representing General Time Series Data as zoo Objects

The `ts` class is rather limited, especially for representing financial data that is not regularly spaced. For example, the `ts` class cannot be used to represent daily financial data because such data are only observed on business days. That is, a business day time clock generally runs from Monday to Friday skipping the weekends. So data are equally spaced in time within the week but the spacing between Friday and Monday is different. This type of irregular spacing cannot be represented using the `ts` class. Similarly, intra-day transactions-level financial price and quote data are irregularly spaced because real world transactions typically do not occur on a regular time clock (e.g. every second, or every minute) during the day, and many financial instruments are only traded during certain trading hours during the day.

A very flexible time series class is zoo (Zeileis' ordered observations) created by Achim Zeileis and Gabor Grothendieck[6] and available in the package zoo on CRAN. The zoo class was designed to handle time series data with an arbitrary ordered time index. This index could be a regularly spaced sequence of dates, an irregularly spaced sequence of dates, or a numeric index. A zoo object essentially attaches date information stored in a vector with data information stored in a matrix.[7]

Install and load the package zoo into R before completing the examples in the next sections.

```
> library(zoo)
```

To create a zoo object one needs an ordered time index and data. The time index must have the same number of rows as the data object and can be any vector containing ordered observations. Typically, the time index is an object of class Date, POSIXct, yearmon, yearqtr or timeDate.

Consider creating zoo objects from the monthly information in the data.frame objects sbux.df and msft.df. First, create a time index of class Date starting in March, 1993 and ending in March, 2003. This can be done directly using the seq() function

```
> td = seq(as.Date("1993/3/1"), as.Date("2003/3/1"), "months")
> class(td)
[1] "Date"

> head(td)
[1] "1993-03-01" "1993-04-01" "1993-05-01" "1993-06-01" "1993-07-01"
[6] "1993-08-01"
```

Notice that the created dates are beginning of the month dates.

Alternatively, a time index can be created by coercing the character date strings in the Date column of sbux.df to objects of class Date

```
> td2 = as.Date(sbux.df$Date, format="%m/%d/%Y")
> head(td2)
[1] "1993-03-31" "1993-04-01" "1993-05-03" "1993-06-01" "1993-07-01"
[6] "1993-08-02"
```

Now that we have a time index, we can create the zoo object by combining the time index with numeric data using the zoo() constructor function

```
> sbux.z = zoo(x=sbux.df$Adj.Close, order.by=td)
> msft.z = zoo(x=msft.df$Adj.Close, order.by=td)

> class(sbux.z)
[1] "zoo"

> str(sbux.z)
'zoo' series from 1993-03-01 to 2003-03-01
```

[6] This is not his real name.

[7] Data stored in a data.frame object is not supported by zoo. A data.frame will be converted to a matrix prior to creating the zoo object.

```
  Data: num [1:121] 1.19 1.21 1.5 1.53 1.48 1.52 1.71 1.67 1.39 1.39
...
  Index: Class 'Date'  num [1:121] 8460 8491 8521 8552 8582 ...

> head(sbux.z)
1993-03-01 1993-04-01 1993-05-01 1993-06-01 1993-07-01 1993-08-01
      1.19       1.21       1.50       1.53       1.48       1.52
```

The time index and data can be extracted using the `index()` and `coredata()` functions

```
> index(sbux.z)
  [1] "1993-03-01" "1993-04-01" "1993-05-01" "1993-06-01" "1993-07-01"
…
> coredata(sbux.z)
  [1]  1.19  1.21  1.50  1.53  1.48  1.52  1.71  1.67  1.39  1.39
…
```

The `start()` and `end()` functions also work for `zoo` objects

```
> start(sbux.z)
[1] "1993-03-01"

> end(sbux.z)
[1] "2003-03-01"
```

An advantage of `zoo` objects is that subsetting can be done with the time index. For example, to extract the data for March 1993 and March 2004 use

```
> sbux.z[as.Date(c("2003/3/1", "2004/3/1"))]
2003-03-01 2004-03-01
     12.88      18.93
```

The `window()` function also works with `zoo` objects

```
> window(sbux.z, start=as.Date("2003/3/1"), end=as.Date("2004/3/1"))

2003-03-01 2003-04-01 2003-05-01 2003-06-01 2003-07-01 2003-08-01
…
2003-10-01 2003-11-01 2003-12-01 2004-01-01 2004-02-01 2004-03-01
     15.80      16.08      16.58      18.31      18.70      18.93
```

Subsetting with dates is a little clunky for `zoo` objects. More convenient subsetting using date information is available for `xts` objects, which are extensions of `zoo` objects.

Creating lags and differences works the same way for `zoo` objects as it does for `ts` objects.

Merging zoo objects

To combine the zoo objects `sbux.z` and `msft.z` into a single object use either the `cbind()` or the `merge()` functions

```
> sbuxmsft.z = cbind(sbux.z, msft.z)
> class(sbuxmsft.z)
[1] "zoo"
```

```
> head(sbuxmsft.z)
           sbux.z msft.z
1993-03-01   1.19   2.43
1993-04-01   1.21   2.25
1993-05-01   1.50   2.44
1993-06-01   1.53   2.32
1993-07-01   1.48   1.95
1993-08-01   1.52   1.98
```

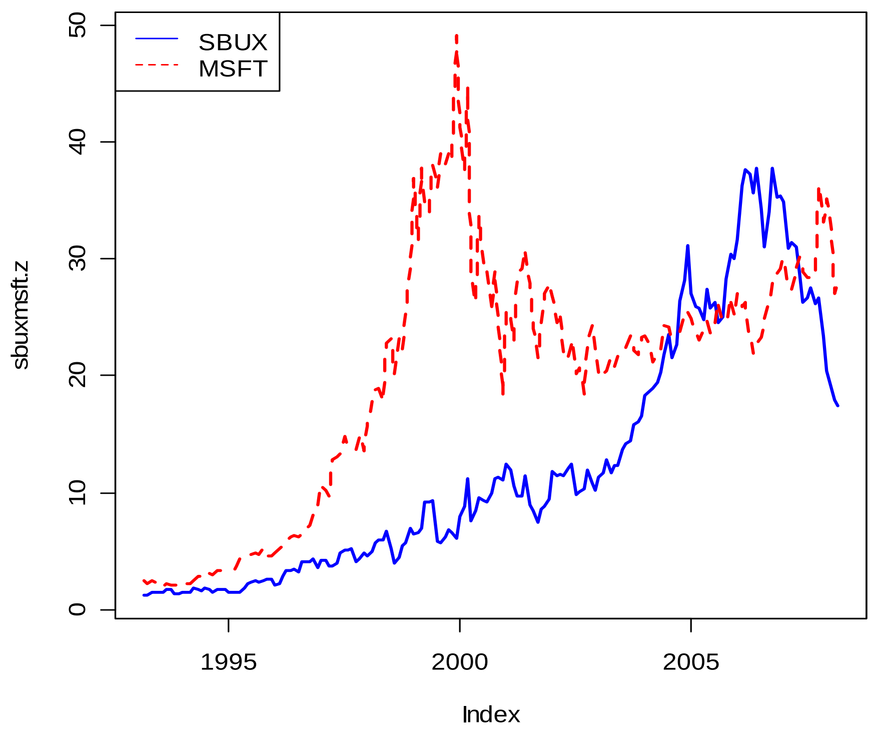
Use `cbind()` when combining `zoo` objects that have the same time index, and use `merge()` when the objects have different time indices. Note, you can only combine `zoo` objects for which the time index is of a common class (e.g. all time indices are `Date` objects).

Plotting zoo objects

The `plot()` function can be used to plot `zoo` objects, and follows a syntax similar to the `plot()` function used for plotting `ts` objects. The following commands produce the plot illustrated in Figure 4

```
> # plot one series at a time and add a legend
> plot(sbux.z, col="blue", lty=1, lwd=2, ylim=c(0,50))
> lines(msft.z, col="red", lty=2, lwd=2)
> legend(x="topleft", legend=c("SBUX","MSFT"), col=c("blue","red"),
+         lty=1:2)
```

```
> # plot multiple series at once
> plot(sbuxmsft.z, plot.type="single", col=c("blue","red"), lty=1:2,
+       lwd=2)
> legend(x="topleft", legend=c("SBUX","MSFT"), col=c("blue","red"),
+         lty=1:2)
```



Manipulating zoo objects

To be completed

There are several useful functions for manipulating zoo objects

Converting a `ts` object to a `zoo` object

To be completed.

Importing data into a `zoo` object

The function `read.zoo()` can read data from a text file stored on disk and create a zoo object. This function is based on the Base R function `read.table()` and has a similar syntax. For example, to read the date and price information in the text file `sbux.csv` and create the zoo object `sbux.z` with a time index of class yearmon use

```
> sbux.z2 = read.zoo("C:/classes/econ424/fall2008/sbuxPrices.csv",
+                     format="%m/%d/%Y", sep=",", header=T)

> # convert index to yearmon
> index(sbux.z2) = as.yearmon(index(sbux.z2))

> head(sbux.z2)
Mar 1993 Apr 1993 May 1993 Jun 1993 Jul 1993 Aug 1993
    1.19     1.21     1.50     1.53     1.48     1.52
```

Representing General Time Series with xts Objects

Importing Data Directly from Yahoo!

The function `get.hist.quote()` in the package tseries can be used to directly import data on a single ticker symbol from `finance.yahoo.com` into a `zoo` object (multiple symbols are not supported).

To download daily adjusted closing price data on SBUX over the period March 1, 1993 through March 1, 2008 use (make sure the tseries package has been installed)

```
> library(tseries)
> SBUX.z = get.hist.quote(instrument="sbux", start="1993-03-01",
+                         end="2008-03-01", quote="AdjClose",
+                         provider="yahoo", origin="1970-01-01",
+                         compression="d", retclass="zoo")
trying URL
'http://chart.yahoo.com/table.csv?s=sbux&a=2&b=01&c=1993&d=2&e=01&f=20
08&g=d&q=q&y=0&z=sbux&x=.csv'
Content type 'text/csv' length unknown
opened URL
downloaded 179 Kb

time series ends   2008-02-29
```

The optional argument `origin=”1970-01-01”` sets the origin date for the internal numeric representation of the date index, and the argument `compression=”d”` indicates that daily data should be downloaded. The object SBUX`.z` is of class `zoo` and the date index is of class `Date`

```
> class(SBUX.z)
[1] "zoo"

> class(index(SBUX.z))
[1] "Date"

> start(SBUX.z)
[1] "1993-03-01"

> end(SBUX.z)
[1] "2008-02-29"
```

Importing Comma Separated Value (.csv) Data into R

You can freely download a wide variety of asset price data from finance.yahoo.com. By default, it gets saved in a comma separated value (`.csv`) file. This is a text file where each value is separated (delimited) by a comma ",". This type of file is easily read into both Excel and R. Excel opens `.csv` files directly. The easiest way import data in .csv files into R is to use the R function `read.csv()`.

To illustrate, I have downloaded from Yahoo! daily price data on Starbucks stock over the period xxx – xxx and stored the data in the .csv file sbux.csv. This file is available on my website http://faculty.washington.edu/ezivot/MFTSR/sbux.csv.

To illustrate, consider the monthly adjusted closing price data on Starbucks (SBUX) and Microsoft (MSFT) in the files `sbuxPrices.csv` and `msftPrices.csv`. These file are available on the class homework page. The first 5 rows of the `sbuxPrices.csv` file are

```
Date,Adj Close
3/31/1993,1.19
4/1/1993,1.21
5/3/1993,1.5
6/1/1993,1.53
```

Notice that the first row contains the names of the columns, the date information is in the first column with the format m/d/YYYY, and the adjusted closing price (close price adjusted for stock splits and dividends) is in the second column. Assume that this file is located in the directory `C:\classes\econ424\fall2008`. To read the data into R use

```
>  sbux.df = read.csv("C:/classes/econ424/fall2008/sbuxPrices.csv",
+                      header = TRUE, stringsAsFactors = FALSE)
```

Now do the same for the Microsoft data.

Remarks:

1. Note how the directory structure is specified using forward slashes "/". Alternatively, you can use double back slashes "\\" instead of a single forward slash "/". The reason is that the back slash character "\" is a special control character in R that signifies the beginning of a command.
2. The argument `header = TRUE` indicates that the column names are in the first row of the file.
3. The argument `stringsAsFactors = FALSE` tells the function to treat the date information as character data and not to convert it to a factor variable.

The SBUX data is imported into `sbux.df`, which is an object of class `data.frame`

```
> class(sbux.df)

[1] "data.frame"
```

A `data.frame` object is a rectangular data object (i.e., organized in rows and columns) with the data in columns. The column names are

```
> colnames(sbux.df)

[1] "Date"      "Adj.Close"
```

And the first 6 rows are

```
> head(sbux.df)

       Date Adj.Close
1 3/31/1993      1.19
2  4/1/1993      1.21
3  5/3/1993      1.50
4  6/1/1993      1.53
5  7/1/1993      1.48
6  8/2/1993      1.52
```

The data in the columns can be of different types. The `Date` column contains the date information as character data, and the `Adj.Close` column contains the adjusted price data as numeric data. Notice that the dates are not all monthly closing dates. However, the adjusted closing prices are for the last trading day of the month.

```
> class(sbux.df$Date)

[1] "character"

> class(sbux.df$Adj.Close)

[1] "numeric"
```

Representing time series data in a `data.frame` object has the disadvantage that the date index information is not efficiently used. In particular, you cannot subset observations based on the date index. You must subset by observation number. For example, to extract the prices between March, 1994 and March, 1995 you must use something like the following:

```
> which(sbux.df$Date == "3/1/1994")
[1] 13

> which(sbux.df$Date == "3/1/1995")
[1] 25

> sbux.df[13:25,]
        Date Adj.Close
13  3/1/1994      1.52
14  4/4/1994      1.86
…
25  3/1/1995      1.50
```

In addition, the default plot method for `data.frame` objects do not utilize the date information for the x-axis. For example, the following call to `plot()` creates a line plot but with a dummy time index for the x-axis:

```
> plot(sbux.df$Adj.Close, type="l")
```

Trying to create a xy-plot with the dates on the x-axis creates an error:

```
> plot(sbux.df$Date, sbux.df$Adj.Close, type="l")
```

Adding Dates as rownames to a data.frame object

To be completed

1. Create data.frame with all numeric data and with character dates as rownames
2. Certain advantages
    a. Allows easy conversion to zoo and xts objects
    b. Can subset on character dates
    c. Plotting time series data does become a bit easier

Importing Excel Data into R

Core R does not have functions for reading data directly from Excel spreadsheets. Two packages, RODBC and xlsReadWrite, have functions that can be used to read data directly from Excel files. The xlsReadWrite package is easier to use but, unfortunately, it has been removed from CRAN due to GPL licensing issues[8].

xlsReadWrite

The package xlsReadWrite contains the function `read.xls()` for reading data from Excel spreadsheets, and the function `write.xls()` for writing data to Excel spreadsheets.

RODBC

The package RODBC contains functions for communicating with ODBC databases, and Excel can be treated as a database.

[8] I have posted the xlsReadWrite package on the class R Hints page.