

2

Time Series Specification, Manipulation, and Visualization in **S-PLUS**

2.1 Introduction

Time series data may be stored, manipulated and visualized in a variety of ways in **S-PLUS**¹. This chapter discusses the basics of working with financial time series data in the form of **S-PLUS** “**timeSeries**” objects. It begins with a discussion of the specification of “**timeSeries**” and “**timeDate**” objects in **S-PLUS** and gives examples of how to specify common “**timeDate**” sequences for financial time series. Basic manipulations of financial time series are discussed and illustrated. These manipulations include aggregating and disaggregating time series, handling of missing values, creations of lags and differences and asset return calculations. The chapter ends with an overview of time series visualization tools and techniques, including the **S-PLUS** plotting functions for “**timeSeries**” as well as specialized plotting functions in **S+FinMetrics**.

2.2 The Specification of “**timeSeries**” Objects in **S-PLUS**

Financial time series data may be represented and analyzed in **S-PLUS** in a variety of ways. By far the most flexible way to analyze, manipulate

¹Chapters 25-27 in the *S-PLUS Guide to Statistic (Vol. II)* discusses the analysis of time series in **S-PLUS**.

and visualize time series data is through the use of S-PLUS calendar-based “timeSeries” objects. A calendar-based “timeSeries” object, hereafter referred to as simply a “timeSeries” is an S version 4 (sv4) object that stores time and date information from a “timeDate” object in a `positions` slot and time series data from any rectangular data object (vector, matrix or data frame) in a `data` slot. Additionally, summary information about the time series may be stored in the `title`, `documentation`, `units` and `attributes` slots.

To illustrate a typical “timeSeries” object, consider the S+FinMetrics “timeSeries” object `singleIndex.dat` which contains monthly closing price data on Microsoft and the S&P 500 index over the period January 1990 through January 2001:

```
> class(singleIndex.dat)
[1] "timeSeries"

> slotNames(singleIndex.dat)
[1] "data"           "positions"       "start.position"
[4] "end.position"   "future.positions" "units"
[7] "title"          "documentation"   "attributes"
[10] "fiscal.year.start" "type"

> singleIndex.dat@title
[1] "Monthly prices on Microsoft and S&P 500 Index"

> singleIndex.dat@documentation
[1] "Monthly closing prices over the period January 1900"
[2] "through January 2001 adjusted for dividends and stock"
[3] "splits."

> singleIndex.dat@units
[1] "Monthly price"

> singleIndex.dat[1:5,]
Positions  MSFT  SP500
Jan 1990   1.2847 329.08
Feb 1990   1.3715 331.89
Mar 1990   1.5382 339.94
Apr 1990   1.6111 330.80
May 1990   2.0278 361.23
```

The date information in the `positions` slot may be extracted directly or by using the `positions` extractor function:

```
> singleIndex.dat@positions[1:5]
[1] Jan 1990 Feb 1990 Mar 1990 Apr 1990 May 1990
```

```
> positions(singleIndex.dat)[1:5]
[1] Jan 1990 Feb 1990 Mar 1990 Apr 1990 May 1990
```

The generic `start` and `end` functions may be used to extract the start and end dates of a “timeSeries” object:

```
> start(singleIndex.dat)
[1] Jan 1990
> end(singleIndex.dat)
[1] Jan 2001
```

The date information in the `positions` slot is an object of class “timeDate”

```
> class(positions(singleIndex.dat))
[1] "timeDate"
```

Details on “timeDate” objects are given later on in this chapter.

The time series data in the `data` slot may be accessed directly or through the `seriesData` extractor function:

```
> singleIndex.dat@data[1:5,]
      MSFT  SP500
1 1.2847 329.08
2 1.3715 331.89
3 1.5382 339.94
4 1.6111 330.80
5 2.0278 361.23

> seriesData(singleIndex.dat)[1:5,]
      MSFT  SP500
1 1.2847 329.08
2 1.3715 331.89
3 1.5382 339.94
4 1.6111 330.80
5 2.0278 361.23
```

In general, the time series data in the `data` slot is a “rectangular” data object and is usually a data frame or a matrix. For example,

```
> class(seriesData(singleIndex.dat))
[1] "data.frame"
```

In fact, “timeSeries” objects themselves are “rectangular” data objects and so the functions `numRows`, `numCols`, `colIds` and `rowIds` may be used to extract useful information:

```
> is.rectangular(singleIndex.dat)
[1] T
> numRows(singleIndex.dat)
[1] 133
```

```

> numCols(singleIndex.dat)
[1] 2
> colIds(singleIndex.dat)
[1] "MSFT" "SP500"
> rowIds(singleIndex.dat)[1:5]
[1] Jan 1990 Feb 1990 Mar 1990 Apr 1990 May 1990

```

2.2.1 Basic Manipulations

Basic manipulation of “timeSeries” objects may be done in the same way as other S-PLUS objects. Mathematical operations may be applied to “timeSeries” objects in the usual way and the result will be a “timeSeries” object. Subscripting a “timeSeries” works in the same way as subscripting a data frame or matrix. For example, a “timeSeries” with the prices on Microsoft may be extracted from `singleIndex.dat` using

```

> msft.p = singleIndex.dat[, "MSFT"]
> msft.p = singleIndex.dat[, 1]
> msft.p@title = "Monthly closing price on Microsoft"
> msft.p@documentation =
+ c("Monthly closing price adjusted for stock",
+ "splits and dividends.")
> msft.p@units = "US dollar price"
> class(msft.p)
[1] "timeSeries"

```

Subsamples from a “timeSeries” may be extracted by creating an index of logical values that are true for the times and dates of interest. For example, consider creating a subsample from the “timeSeries” `singleIndex.dat` over the period March 1992 through January 1993.

```

> smpl = (positions(singleIndex.dat) >= timeDate("3/1/1992") &
+ positions(singleIndex.dat) <= timeDate("1/31/1993"))
> singleIndex.dat[smpl,]

```

	Positions	MSFT	SP500
Mar 1992		4.938	403.7
Apr 1992		4.594	414.9
May 1992		5.042	415.4
Jun 1992		4.375	408.1
Jul 1992		4.547	424.2
Aug 1992		4.656	414.0
Sep 1992		5.031	417.8
Oct 1992		5.547	418.7
Nov 1992		5.820	431.4
Dec 1992		5.336	435.7
Jan 1993		5.406	438.8

S-PLUS 7 supports subscripting a “timeSeries” object directly with dates. For example, the subsample from `singleIndex.dat` over the period March 1992 through January 1993 may be produced using

```
> singleIndex.dat[timeEvent("3/1/1992","1/31/1993"),]
```

Most S-PLUS functions have methods to handle “timeSeries” objects. Some common examples are the S-PLUS functions `colMeans`, `colVars` and `colStdevs` which compute the mean, variance and standard deviation value for each column of data:

```
> colMeans(singleIndex.dat)
      MSFT      SP500
26.74513 730.3805
```

For functions that do not have methods to handle “timeSeries” objects, the extractor function `seriesData` should be used to extract the data slot of the “timeSeries” prior to applying the function:

```
> colMeans(seriesData(singleIndex.dat))
      MSFT      SP500
26.74513 730.3805
```

All of the S+FinMetrics modeling and support functions are designed to accept “timeSeries” objects in a uniform way.

2.2.2 S-PLUS “timeDate” Objects

Time and date information in S-PLUS may be stored in “timeDate” objects. The S-PLUS function `timeDate` is used to create “timeDate” objects. For example, to create a “timeDate” object for the date January 1, 2002 for the US Pacific time zone use

```
> td = timeDate("1/1/2002",in.format="%m/%d/%Y",
+ zone="Pacific")
```

The date information is specified in a character string and the optional arguments `in.format` and `zone` determine the input date format and the time zone, respectively. The input formats are single-element character vectors consisting of input fields which start with “%” and end with a letter. The default input date format may be viewed with

```
> options("time.in.format")
$time.in.format:
[1] "%m[/][.]%d[/][,]%y [%H[:%M[:%S[.%N]]][%p][[(%)3Z()]]]"
```

and examples of common date formats can be found in the S-PLUS object `format.timeDate`

```
> names(format.timeDate)
```

```

[1] "1/3/1998"
[2] "3/1/1998"
...
[32] "03 Jan 1998 14:04:32 (PST)"
> format.timeDate[[1]]$input
[1] "%m/%d/%Y"

```

The result of `timeDate` is an object of class “`timeDate`”

```

> class(td)
[1] "timeDate"
> td
[1] 1/1/02 0:00:00 AM
> slotNames(td)
[1] ".Data"          ".Data.names"    ".Data.classes"
[4] "format"         "time.zone"

```

“`timeDate`” objects have a number of slots that are used to specify and control time and date information. Full details may be seen using

```
> ?class.timeDate
```

The `.Data` slot is a list with components giving the Julian date representation of the day and time within the day. The Julian day represents the number of days since January 1, 1960 and the Julian time within the day indicates the number of milliseconds since midnight Greenwich mean time (GMT)

```

> td@.Data
[[1]]:
[1] 15341

[[2]]:
[1] 28800000

```

Since the US Pacific Time Zone is 8 hours behind GMT, the number of milliseconds since Greenwich mean time is $8 * 60 * 60 * 1000 = 28,800,000$. The output display format of the date information is specified in the `format` slot

```

> td@format
[1] "%m/%d/%02y %H:%02M:%02S %p"

```

Like input formats, output formats are single-element character vectors consisting of output fields, which start with “%” and end with a letter, and other characters that are simply printed. The above format specifies printing the date as month/day/year and then hour:minute:second and AM or PM. The integers 02 before y, M and S fix the output width to 2 characters. All supported output fields are described in the help file for

`class.timeDate` and a list of example output formats are given in the S-PLUS object `format.timeDate`. For example,

```
> names(format.timeDate)[18]
[1] "03 Jan 1998"
> format.timeDate[[18]]$output
[1] "%02d %b %Y"
```

Time Zone Issues

The time and date information stored in a “timeDate” object is aligned to the time zone specified in the `time.zone` slot

```
> td@time.zone
[1] "Pacific"
```

To modify the output format of a “timeDate” object to display time zone information simply add “%z”

```
> td@format = paste(td@format, "%z")
> td
[1] 1/1/02 0:00:00 AM Pacific
```

The object `td` is aligned to the US Pacific time zone. If the `zone` argument to `timeDate` is omitted when the “timeDate” object is created the default time zone in `options(“time.zone”)` is used². For example,

```
> options("time.zone")
$time.zone:
[1] "Pacific"
> td2 = timeDate("Mar 02, 1963 08:00 PM",
+ in.format="%m %d, %Y %H:%M %p",
+ format="%b %02d, %Y %02I:%02M %p %z")
> td2
[1] Mar 02, 1963 08:00 PM Pacific
```

Note that the above example shows that the output format of the “timeDate” object can be specified when the object is created using the argument `format`.

All of the time zone specifications supported by S-PLUS are described in the help file for `class.timeZone` and these specifications are defined relative to times and dates given in GMT. The time zone specifications include daylight savings time in various areas around the world. To see how a time zone specification affects a `timeDate` object, consider what

²On Windows platforms, the time zone specification is obtained from the Windows regional settings. The examples in this section were created on a Windows computer in the U.S. Pacific time zone. Therefore, the default time zone taken from the Windows regional settings is “Pacific”.

happens when the time zone for the object `td` is changed to US Eastern Time:

```
> td@time.zone = "Eastern"
> td
[1] 1/1/02 3:00:00 AM Eastern
> td@.Data
[[1]]:
[1] 15341

[[2]]:
[1] 28800000
```

Since US Eastern Time is three hours ahead of US Pacific Time the displayed date is moved ahead three hours. That is, midnight US Pacific Time on January 1, 2002 is the same as 3 AM US Eastern Time on January 1, 2002. Notice that changing the time zone information does not alter the Julian date information in the `.Data` slot. To align the Julian date representation to reflect the number of milliseconds from GMT on US Eastern time the millisecond information in the second component of the `.Data` slot must be adjusted directly.

If a “`timeDate`” object is created in GMT then the S-PLUS function `timeZoneConvert` may be used to re-align the millisecond offset to a specified time zone. For example,

```
> tdGMT = timeDate("1/1/2002",zone="GMT",
+ format="%m/%d/%02y %H:%02M:%02S %p %z")
> tdGMT
[1] 1/1/02 0:00:00 AM GMT
> tdGMT@.Data
[[1]]:
[1] 15341

[[2]]:
[1] 0

> tdPST = timeZoneConvert(tdGMT,"PST")
> tdPST
[1] 1/1/02 0:00:00 AM PST
> tdPST@.Data
[[1]]:
[1] 15341

[[2]]:
[1] 28800000
```

Be aware that `timeZoneConvert` is not designed to convert the millisecond offsets from one arbitrary time zone other than GMT to another arbitrary time zone.

Mathematical Operations with “timeDate” Objects

Since “timeDate” objects have a Julian date representation, certain mathematical operations like addition and subtractions of numbers may be performed on them and the result will also be a “timeDate” object. For example,

```
> td1 = timeDate("1/1/2002",in.format="%m/%d/%Y",
+ zone="GMT",format="%m/%d/%04Y %H:%02M:%02S %p %z")
> td2 = timeDate("2/1/2002",in.format="%m/%d/%Y",
+ zone="GMT",format="%m/%d/%04Y %H:%02M:%02S %p %z")
> td1
[1] 1/1/2002 0:00:00 AM GMT
> td2
[1] 2/1/2002 0:00:00 AM GMT

> as.numeric(td1)
[1] 15341
> td1 + 1
[1] 1/2/2002 0:00:00 AM GMT
> td1 + 0.5
[1] 1/1/2002 12:00:00 PM GMT
> td1 - 1
[1] 12/31/2001 0:00:00 AM GMT
> 2*td1
[1] 30682
> td1+td2
[1] 2/2/2004 0:00:00 AM GMT
```

Adding two “timeDate” objects together creates another “timeDate” object with date given by the addition of the respective Julian dates. Subtraction of two “timeDate” objects, however, produces an sv4 object of class “timeSpan”

```
> td.diff = td2 - td1
> class(td.diff)
[1] "timeSpan"
> td.diff
[1] 31d 0h 0m 0s OMS
> slotNames(td.diff)
[1] ".Data"          ".Data.names"    ".Data.classes"
[4] "format"
```

The “timeSpan” object `td.diff` gives the time difference between `td1` and `td2` - 31 days, 0 hours, 0 minutes, 0 seconds and 0 milliseconds. The Julian date information is kept in the `.Data` slot and the output format is in the `format` slot. Details about “timeSpan” objects is given in The *S-PLUS Guide to Statistics, Vol. II*, chapter 25.

2.2.3 Creating Common “timeDate” Sequences

Most historical financial time series are regularly spaced calendar-based time series; e.g. daily, monthly or annual time series. However, some financial time series are irregularly spaced. Two common examples of irregularly spaced financial time series are daily closing prices and intra-day transactions level data. There are a variety of time and date functions in S-PLUS that may be used to create regularly spaced and irregularly spaced “timeDate” sequences for essentially any kind of financial data. These functions are illustrated using the following examples³.

Regularly and irregularly spaced sequences may be created using the S-PLUS functions `timeCalendar`, `timeSeq` and `timeSequence`. The function `timeSeq` is the most flexible. The following examples illustrate the use of these functions for creating common “timeDate” sequences.

Annual Sequences

Creating a “timeDate” sequence for an annual time series from 1900 to 1910 may be done in a variety of ways. Perhaps, the simplest way uses the S-PLUS `timeCalendar` function:

```
> td = timeCalendar(y=1900:1910,format="%Y")
> class(td)
[1] "timeDate"
> td
[1] 1900 1901 1902 1903 1904 1905 1906 1907 1908 1909 1910
```

The `timeCalendar` function produces an object of class “timeDate”. The argument `format="%Y"` specifies the output format of the “timeDate” object as a four digit year.

Since `td` contains a sequence of dates, the Julian date information for all of the dates is available in the `.Data` slot

```
> td@.Data
[[1]]:
[1] -21914 -21549 -21184 -20819 -20454 -20088 -19723 -19358
[9] -18993 -18627 -18262
```

³To avoid problems with time zone specifications, all examples in this sections were created after setting the default time zone to GMT using `options(time.zone="GMT")`.

```
[[2]]:
[1] 0 0 0 0 0 0 0 0 0 0 0
```

An annual sequence from 1900 to 1910 may also be computed using the S-PLUS function `timeSeq`:

```
> timeSeq(from="1/1/1900", to="1/1/1910", by="years",
+ format="%Y")
[1] 1900 1901 1902 1903 1904 1905 1906 1907 1908 1909 1910
```

The argument `by="years"` specifies annual spacing between successive values in the sequence starting at 1/1/1900 and ending at 1/1/1910. The date formats for the starting and ending dates must conform to the default input format for “timeDate” objects (see `options("time.in.format")`).

Finally, an annual sequence from 1900 to 1910 may be created using the S-PLUS function `timeSequence`:

```
> tds = timeSequence("1/1/1900", "1/1/1910", by="years",
+ format="%Y")
> class(tds)
[1] "timeSequence"
> tds
from: 1900
to: 1910
by: +1yr
[1] 1900 1901 1902 ... 1910
```

`timeSequence` creates an object of class “timeSequence” which stores time and date information in a compact fashion. The “timeSequence” object may be converted to a “timeDate” object using the S-PLUS as function

```
> td = as(tds, "timeDate")
> td
[1] 1900 1901 1902 1903 1904 1905 1906 1907 1908 1909 1910
```

Quarterly Sequences

A quarterly “timeDate” sequence from 1900:I through 1902:IV may be created using `timeSeq` with the `by="quarters"` option:

```
> timeSeq(from="1/1/1900", to="10/1/1902", by="quarters",
+ format="%Y:%Q")
[1] 1900:I 1900:II 1900:III 1900:IV 1901:I 1901:II
[7] 1901:III 1901:IV 1902:I 1902:II 1902:III 1902:IV
```

The output format character `%Q` displays the quarter information. Notice that the dates are specified as the first day of the quarter.

Monthly Sequences

Now consider creating a monthly “timeDate” sequence from January 1, 1900 through March 1, 1901. This may be done using `timeCalendar`

```
> timeCalendar(m=rep(1:12,length=15),y=rep(1900:1901,each=12,
+ length=15), format="%b %Y")
[1] Jan 1900 Feb 1900 Mar 1900 Apr 1900 May 1900 Jun 1900
[7] Jul 1900 Aug 1900 Sep 1900 Oct 1900 Nov 1900 Dec 1900
[13] Jan 1901 Feb 1901 Mar 1901
```

or `timeSeq`

```
> timeSeq(from="1/1/1900",to="3/1/1901",by="months",
+ format="%b %Y")
[1] Jan 1900 Feb 1900 Mar 1900 Apr 1900 May 1900 Jun 1900
[7] Jul 1900 Aug 1900 Sep 1900 Oct 1900 Nov 1900 Dec 1900
[13] Jan 1901 Feb 1901 Mar 1901
```

To create a monthly sequence of end of month values from December 31, 1899 through February 28, 1901, subtract 1 from the above calculation:

```
> timeSeq(from="1/1/1900",to="3/1/1901",by="months",
+ format="%b %Y") - 1
[1] Dec 1899 Jan 1900 Feb 1900 Mar 1900 Apr 1900 May 1900
[7] Jun 1900 Jul 1900 Aug 1900 Sep 1900 Oct 1900 Nov 1900
[13] Dec 1900 Jan 1901 Feb 1901
```

Weekly Sequences

Weekly sequences are best created using `timeSeq` with `by="weeks"`. For example, a weekly sequence from Monday January 1, 1990 to Monday Feb 26, 1990 may be created using

```
> timeSeq(from="1/1/1990",to="3/1/1990",by="weeks",
+ format="%a %b %d, %Y")
[1] Mon Jan 1, 1990 Mon Jan 8, 1990 Mon Jan 15, 1990
[4] Mon Jan 22, 1990 Mon Jan 29, 1990 Mon Feb 5, 1990
[7] Mon Feb 12, 1990 Mon Feb 19, 1990 Mon Feb 26, 1990
```

To create a weekly sequence starting on a specific day, say Wednesday, make the starting date a Wednesday.

Daily Sequences

A regularly spaced daily sequence may be created using `timeSeq` with `by = "days"`. For an irregularly spaced daily sequence of weekdays use `timeSeq` with `by = "weekdays"`. For financial asset price data that trades on U.S. exchanges, the relevant “daily” sequence of dates is an irregularly spaced

sequence based on business days. Business days are weekdays excluding certain holidays. For example, consider creating a daily “timeDate” sequence for the month of January, 2000 for a time series of asset prices that trade on the New York stock exchange (NYSE). The NYSE is not open on weekends and on certain holidays and these dates should be omitted from the “timeDate” sequence. The S-PLUS function `holiday.NYSE` returns the New York Stock Exchange holidays for a given year, 1885-present, according to the historical and current (as of 1998) schedule, not including special-event closure days or partial-day closures. The NYSE holidays for 2000 are

```
> holiday.NYSE(2000)
[1] 1/17/2000  2/21/2000  4/21/2000  5/29/2000  7/4/2000
[6] 9/4/2000   11/23/2000 12/25/2000
```

Martin Luther King day on Monday January 17th is the only weekday holiday. A “timeDate” sequence of business days excluding the holiday 1/17/2000 may be created using

```
> timeSeq(from="1/3/2000",to="1/31/2000",by="bizdays",
+ holidays=holiday.NYSE(2000),format="%a %b %d, %Y")
[1] Mon Jan 3, 2000 Tue Jan 4, 2000 Wed Jan 5, 2000
[4] Thu Jan 6, 2000 Fri Jan 7, 2000 Mon Jan 10, 2000
[7] Tue Jan 11, 2000 Wed Jan 12, 2000 Thu Jan 13, 2000
[10] Fri Jan 14, 2000 Tue Jan 18, 2000 Wed Jan 19, 2000
[13] Thu Jan 20, 2000 Fri Jan 21, 2000 Mon Jan 24, 2000
[16] Tue Jan 25, 2000 Wed Jan 26, 2000 Thu Jan 27, 2000
[19] Fri Jan 28, 2000 Mon Jan 31, 2000
```

The argument `holidays=holiday.NYSE(2000)` in conjunction with `by = "bizdays"` instructs `timeSeq` to exclude the weekday dates associated with the NYSE holidays for 2000. Notice that the date Mon Jan 17, 2000 has been omitted from the sequence.

Intra-day Irregularly Spaced Sequences

Sequences of irregularly spaced intra-day dates may be created using the function `timeCalendar`. For example, consider creating a sequence of hourly observations only during the hypothetical trading hours from 9:00 AM to 3:00 PM from Monday January 3, 2000 through Tuesday January 4, 2000. Such a sequence may be created using `timeCalendar` as follows

```
> timeCalendar(h=rep(9:15,2),d=rep(3:4,each=7),
+ y=2000,format="%a %b %d, %Y %02I:%02M %p")
[1] Mon Jan 3, 2000 09:00 AM Mon Jan 3, 2000 10:00 AM
[3] Mon Jan 3, 2000 11:00 AM Mon Jan 3, 2000 12:00 PM
[5] Mon Jan 3, 2000 01:00 PM Mon Jan 3, 2000 02:00 PM
[7] Mon Jan 3, 2000 03:00 PM Tue Jan 4, 2000 09:00 AM
[9] Tue Jan 4, 2000 10:00 AM Tue Jan 4, 2000 11:00 AM
```

```
[11] Tue Jan 4, 2000 12:00 PM Tue Jan 4, 2000 01:00 PM
[13] Tue Jan 4, 2000 02:00 PM Tue Jan 4, 2000 03:00 PM
```

In a similar fashion, a sequence of minute observations from 9:00 AM to 3:00 PM on Monday January 3, 2000 and Tuesday January 4, 2000 may be created using

```
> timeCalendar(min=rep(rep(0:59,6),2),
+ h=rep(9:14,each=60,length=360*2),
+ d=rep(3:4,each=360,length=360*2),
+ y=2000,format="%a %b %d, %Y %02I:%02M %p")
[1] Mon Jan 3, 2000 09:00 AM Mon Jan 3, 2000 09:01 AM
[3] Mon Jan 3, 2000 09:02 AM Mon Jan 3, 2000 09:03 AM
...
[359] Mon Jan 3, 2000 02:58 PM Mon Jan 3, 2000 02:59 PM
[361] Tue Jan 4, 2000 09:00 AM Tue Jan 4, 2000 09:01 AM
...
[719] Tue Jan 4, 2000 02:58 PM Tue Jan 4, 2000 02:59 PM
```

2.2.4 Miscellaneous Time and Date Functions

In addition to the time and date functions discussed so far, S-PLUS has a number of miscellaneous time and date functions. In addition **S+FinMetrics** provides a few time and date functions. These are summarized in Table 2.1.

2.2.5 Creating “timeSeries” Objects

S-PLUS “timeSeries” objects are created with the `timeSeries` function. Typically a “timeSeries” is created from some existing data in a data frame or matrix and a “timeDate” object. For example,

```
> my.df = data.frame(x=abs(rnorm(10,mean=5)),
+ y=abs(rnorm(10,mean=10)))
> my.td = timeCalendar(y=1990:1999,format="%Y")
> my.ts = timeSeries(data=my.df,pos=my.td)
> my.ts
Positions      x      y
1990      4.250 11.087
1991      5.290 11.590
1992      5.594 11.848
1993      5.138 10.426
1994      5.205  9.678
1995      4.804 11.120
1996      5.726 11.616
1997      6.124  9.781
1998      3.981 10.725
```

S-PLUS function	Description
month.day.year	Converts calendar dates to Julian dates
julian	Converts Julian dates to calendar dates
quarters	Create an ordered factor corresponding to quarters
months	Create an ordered factor corresponding to months
days	Create an ordered factor corresponding to days
weekdays	Create an ordered factor corresponding to weekdays
years	Create an ordered factor corresponding to years
yeardays	Extract year day from date
hours	Extract hour from date
minutes	Extract minutes from date
seconds	Extract seconds from date
hms	Create data frame containing hours, minutes and seconds
mdy	Create data frame containing month, day and year
wdydy	Create data frame containing weekday, year day and year
leap.year	Determines if year number corresponds to a leap year
holidays	Generate a collection of holidays
holiday.fixed	Generate holidays that occur on fixed dates
holiday.weekday.number	Generate holidays that occur on weekdays
S+FinMetrics function	Description
days.count	Count number of days between two dates
is.weekday	Tests if date is a weekday
is.weekend	Tests if date is a weekend
is.bizday	Tests if date is a business day
imm.dates	Create International Monetary Market dates

TABLE 2.1. Miscellaneous time and date functions

```
1999      6.006 10.341
```

Information about the “timeSeries” object may be added to the `title`, `documentation` and `units` slots:

```
> my.ts@title = "My timeSeries"
> my.ts@documentation = c("Simulated annual price data using ",
+ "the S-PLUS function rnorm")
> my.ts@units = c("US dollars", "US dollars")
```

The `title` and `units` information is utilized in certain plot functions.

Creating “timeSeries” Objects from Time Series in Data Frames

Very often time series data that are in data frames have a date variable with a formatted date string. The S-PLUS function `timeDate` has a variety of input formats that may be used to convert such date strings into “timeDate” objects. For example, the `S+FinMetrics` data frame `yhoo.df` contains daily high, low, open and close prices as well as volume information for Yahoo stock for the month of February 2002

```
> yhoo.df[1:2,]
      Date  Open High  Low Close  Volume
1 1-Feb-02 17.26 17.3 16.35 16.68 6930100
2 4-Feb-02 16.55 16.6 15.60 15.75 8913700
```

The variable `Date` is a character vector containing the date strings. A “timeDate” sequence created from the date strings in `Date` is

```
> td = timeDate(yhoo.df[,1],in.format="%d-%m-%y",
+ format="%a %b %d, %Y")
> td[1:2]
[1] Fri Feb 1, 2002 Mon Feb 4, 2002
```

A “timeSeries” object containing the data from `yhoo.df` is created using

```
> yhoo.ts = timeSeries(pos=td,data=yhoo.df[,-1])
> yhoo.ts[1:2,]
      Positions  Open High  Low Close  Volume
Fri Feb 1, 2002 17.26 17.3 16.35 16.68 6930100
Mon Feb 4, 2002 16.55 16.6 15.60 15.75 8913700
```

High frequency data, however, is often recorded using nonstandard time formats. For example, consider the transactions level data for the month of December 1999 for 3M stock in the `S+FinMetrics` data frame `highFreq3m.df`

```
> highFreq3m.df[1:2,]
      trade.day trade.time trade.price
1          1      34412      94.688
2          1      34414      94.688
```

The variable `trade.day` contains the integer trading day of the month, the variable `trade.time` contains the integer trade time recorded as the number of seconds from midnight and the variable `trade.price` contains the transaction price in dollars. A “timeDate” sequence may be easily created from the trade day and trade time information as follows

```
> td = timeDate(julian=(highFreq3M.df$trade.day-1),
+ ms=highFreq3M.df$trade.time*1000,
+ in.origin=c(month=12,day=1,year=1999),zone="GMT")
> td[1:2]
[1] 12/1/99 9:33:32 AM 12/1/99 9:33:34 AM
```

The function `timeDate` can create a “timeDate” sequence using Julian date and millisecond information. The argument `julian` takes an integer vector containing the number of days since the date specified in the argument `in.origin`, and the argument `ms` takes an integer vector containing the number of milliseconds since midnight. In the above example, `in.origin` is specified as December 1, 1999 and the optional argument `zone` is used to set the time zone to GMT. A “timeSeries” object containing the high frequency data in `highFreq3M.df` is created using

```
> hf3M.ts = timeSeries(pos=td,data=highFreq3M.df)
```

2.2.6 Aggregating and Disaggregating Time Series

Often a regularly spaced financial time series of a given frequency may need to be aggregated to a coarser frequency or disaggregated to a finer frequency. In addition, aggregation and disaggregation may involve flow or stock variables. The S-PLUS functions `aggregateSeries` and `align` may be used for such purposes. To enhance and extend the disaggregation functionality in S-PLUS the S+FinMetrics function `disaggregate` is introduced.

Aggregating Time Series

Given a monthly “timeSeries” of end of month prices over a number of years, suppose one would like to create an annual time series consisting of the end of month December prices. Such a series may be easily constructed by subsetting using the S-PLUS function `months`:

```
> dec.vals = "Dec"==months(positions(singleIndex.dat))
> annual.p = singleIndex.dat[dec.vals,]
> annual.p
Positions      MSFT  SP500
Dec 1990       2.090  330.2
Dec 1991       4.635  417.1
Dec 1992       5.336  435.7
Dec 1993       5.039  466.4
```

Dec 1994	7.641	459.3
Dec 1995	10.969	615.9
Dec 1996	20.656	740.7
Dec 1997	32.313	970.4
Dec 1998	69.344	1229.2
Dec 1999	116.750	1469.3
Dec 2000	43.375	1320.3

Another way to create the above annual time series is to use the S-PLUS `aggregateSeries` function with a user-written function to pick off December values. One such function, based on the S-PLUS function `hloc` used to compute high, low, open and close values, is

```
pickClose = function(x)
{
# return closing values of a vector
  if(length(dim(x))) x = as.vector(as.matrix(x))
  len = length(x)
  if(!len)
    as(NA, class(x))
  else x[len]
}
```

The annual data is then constructed using `aggregateSeries` with optional arguments `FUN=pickClose` and `by="years"`

```
> annual.p = aggregateSeries(singleIndex.dat,
+ FUN=pickClose,by="years")
> positions(annual.p)%format = "%Y"
> annual.p
Positions      MSFT  SP500
1990           2.090  330.2
1991           4.635  417.1
1992           5.336  435.7
1993           5.039  466.4
1994           7.641  459.3
1995          10.969  615.9
1996          20.656  740.7
1997          32.313  970.4
1998          69.344 1229.2
1999         116.750 1469.3
2000          43.375 1320.3
2001          61.063 1366.0
```

The function `aggregateSeries` passes to the function `pickClose` data from `singleIndex.dat` in blocks of year's length. The function `pickClose`

simply picks off the last value for the year. Since `singleIndex.dat` only has data for January 2, 2001, the 2001 value for `annual.p` is this value.

The method described above may also be used to construct end-of-month closing price data from a “timeSeries” of daily closing price data. For example, the commands to create end of month closing prices from daily closing prices for Microsoft, taken from the `S+FinMetrics` “timeSeries” `DowJones30`, using `aggregateSeries` with `FUN = pickClose` and `by = "months"` are

```
> msft.daily.p = DowJones30[, "MSFT"]
> msft.daily.p@title = "Daily closing price on Microsoft"
> msft.daily.p@units = "Dollar price"
> msft.monthly.p = aggregateSeries(msft.daily.p, FUN=pickClose,
+ by="months", adj=0.99)
> msft.monthly.p[1:12]
Positions MSFT
1/31/1991 2.726
2/28/1991 2.882
3/31/1991 2.948
4/30/1991 2.750
5/31/1991 3.049
6/30/1991 2.838
7/31/1991 3.063
8/31/1991 3.552
9/30/1991 3.708
10/31/1991 3.912
11/30/1991 4.052
12/31/1991 4.635
```

The option `adj=0.99` adjusts the positions of the monthly data to the end of the month. Notice that the end of month dates are not necessarily the last trading days of the month.

The monthly closing price data may be extracted from the daily closing price data by clever use of subscripting⁴. One way to do this is

```
> end.month.idx =
+ which(diff(as.numeric(months(positions(msft.daily.p)))) != 0)
> msft.monthly.p = msft.daily.p[end.month.idx]
> msft.monthly.p[1:12]
Positions MSFT
1/31/1991 2.726
2/28/1991 2.882
3/28/1991 2.948
4/30/1991 2.750
```

⁴This method was suggested by Steve McKinney.

```

5/31/1991 3.049
6/28/1991 2.838
7/31/1991 3.063
8/30/1991 3.552
9/30/1991 3.708
10/31/1991 3.912
11/29/1991 4.052
12/31/1991 4.635

```

A common aggregation operation with financial price data is to construct a *volume weighted average price* (vwap). This may be easily accomplished with `aggregateSeries` and a user-specified function to compute the vwap. For example, consider the daily open, high, low and close prices and volume on Microsoft stock from October 2, 2000 through August 31, 2001 in the S+FinMetrics "timeSeries" `msft.dat`.

```

> smpl = (positions(msft.dat) >= timeDate("10/1/2000") &
+ positions(msft.dat) <= timeDate("8/31/2001"))
> msft.dat[smpl,]
Positions Open High Low Close Volume
10/2/2000 60.50 60.81 58.25 59.13 29281200
...
8/31/2001 56.85 58.06 56.30 57.05 28950400

```

A function that can be used to aggregate open, high, low and close prices, volume and compute the open and close vwap is

```

vol.wtd.avg.price = function(x) {
  VolumeSum = as.double(sum(x[, "Volume"]))
  nrowx = numRows(x)
  return(data.frame(Open = x[1, "Open"],
    High = max(x[, "High"]),
    Low = min(x[, "Low"]),
    Close = x[nrowx, "Close"],
    vwap.Open = sum(x[, "Open"] * x[, "Volume"])/VolumeSum,
    wap.Close = sum(x[, "Close"] * x[, "Volume"])/VolumeSum,
    Volume = VolumeSum))
}

```

Using `aggregateSeries` and the function `vol.wtd.avg.price` one can compute the monthly open, high, low, close prices, volume, and open and close vwap

```

> msft.vwap.dat = aggregateSeries(x = msft.dat[smpl,],
+ by = "months", FUN = vol.wtd.avg.price,
+ together = T)
> positions(msft.vwap.dat)@format="%b %Y"
> msft.vwap.dat[, -7]

```

Positions	Open	High	Low	Close	vwap.Open	vwap.Close
Oct 2000	60.50	70.13	48.44	68.88	59.10	59.48
Nov 2000	68.50	72.38	57.00	57.38	68.35	67.59
...						
Aug 2001	66.80	67.54	56.30	57.05	62.99	62.59

Disaggregating Time Series

Consider the problem of creating a daily “timeSeries” of inflation adjusted (real) prices on Microsoft stock over the period January 2, 1991 through January 2, 2001. To do this the daily nominal prices must be divided by a measure of the overall price level; e.g. the consumer price level (CPI). The daily nominal stock price data is in the “timeSeries” `msft.daily.p` created earlier and the CPI data is in the `S+FinMetrics` “timeSeries” `CPI.dat`. The CPI data, however, is only available monthly.

```
> start(CPI.dat)
[1] Jan 1913
> end(CPI.dat)
[1] Nov 2001
```

and represents the average overall price level during the month but is recorded at the end of the month. The CPI data from December 1990 through January 2001 is extracted using

```
> smpl = (positions(CPI.dat) >= timeDate("12/1/1990")
+ & positions(CPI.dat) <= timeDate("2/1/2001"))
> cpi = CPI.dat[smpl,]
> cpi[1:3]
Positions    CPI
Dec 1990    134.3
Jan 1991    134.8
Feb 1991    134.9
```

To compute real daily prices on Microsoft stock, the monthly CPI data in the “timeSeries” object `cpi` must be *disaggregated* to daily data. This disaggregation may be done in a number of ways. For example, the CPI for every day during the month of January, 1991 may be defined as the monthly CPI value for December, 1990 or the monthly CPI value for January, 1991. Alternatively, the daily values for January 1991 may be computed by linearly interpolating between the December, 1990 and January, 1991 values. The S-PLUS function `align` may be used to do each of these disaggregations.

The `align` function aligns a “timeSeries” object to a given set of positions and has options for the creation of values for positions in which the “timeSeries” does not have values. For example, the disaggregated

CPI using the previous month's value for the current month's daily data is constructed using

```
> cpi.daily.before =
+ align(cpi,positions(msft.daily.p),how="before")
> cpi.daily.before[c(1:3,21:23)]
Positions    CPI
1/2/1991 134.3
1/3/1991 134.3
1/4/1991 134.3
1/30/1991 134.3
1/31/1991 134.8
2/1/1991 134.8
```

The new positions to align the CPI values are the daily positions of the “timeSeries” `msft.daily.p`, and the argument `how="before"` specifies that the previous month's CPI data is to be used for the current month's daily CPI values. Similarly, the disaggregated CPI using the next month's value for the current month's daily data is constructed using

```
> cpi.daily.after =
+ align(cpi,positions(msft.daily.p),how="after")
> cpi.daily.after[c(1:3,21:23)]
Positions    CPI
1/2/1991 134.8
1/3/1991 134.8
1/4/1991 134.8
1/30/1991 134.8
1/31/1991 134.8
2/1/1991 134.9
```

Finally, the disaggregated daily CPI using linear interpolation between the monthly values is constructed using

```
> cpi.daily.interp = align(cpi,positions(msft.daily.p),
+ how="interp")
> cpi.daily.interp[c(1:3,21:23)]
Positions    CPI
1/2/1991 134.3
1/3/1991 134.3
1/4/1991 134.4
1/30/1991 134.8
1/31/1991 134.8
2/1/1991 134.8
```

The daily real prices on Microsoft stock using the interpolated daily CPI values are then

```
> msft.daily.rp = (msft.daily.p/cpi.daily.interp)*100
```

Disaggregating Time Series using the **S+FinMetrics** `disaggregate` Function

With economic and financial time series, it is sometimes necessary to distribute a flow variable or time average a stock variable that is observed at a low frequency to a higher frequency. For example, a variable of interest may only be observed on an annual basis and quarterly or monthly values are desired such that their sum is equal to the annual observation or their average is equal to the annual observation. The **S+FinMetrics** function `disaggregate` performs such disaggregations using two methods. The first method is based on cubic spline interpolation and is appropriate if the only information is on the series being disaggregated. The second method utilizes a generalized least squares (gls) fitting method due to Chow and Lin (1971) and is appropriate if information is available on one or more related series that are observed at the desired disaggregated frequency. The arguments expected by `disaggregate` are

```
> args(disaggregate)
function(data, k, method = "spline", how = "sum", x = NULL,
+ out.positions = NULL, ...)
```

where `data` is a vector, matrix or “timeSeries” of low frequency data, `k` is the number of disaggregation periods, `method` determines the disaggregation method (spline or gls), `how` specifies if the disaggregated values sum to the aggregated values or are equal on average to the disaggregated values, `x` represents any related observed data at the disaggregated frequency and `out.positions` represents a “timeDate” sequence for the resulting output.

To illustrate the use of `disaggregate`, consider the problem of disaggregating the annual dividend on the S&P 500 index to a monthly dividend. Since the annual dividend is a flow variable, the sum of the monthly dividends should equal the annual dividend. The annual S&P 500 dividend information over the period 1871 - 2000 is in the **S+FinMetrics** “timeSeries” `shiller.annual`. The disaggregated monthly dividend values such that their sum is equal to the annual values is created using

```
> monthly.dates = timeSeq(from="1/1/1871",to="12/31/2000",
+ by="months",format="%b %Y")
> div.monthly =
+ disaggregate(shiller.annual[, "dividend"],12,
+ out.positions=monthly.dates)
> div.monthly[1:12]
Positions dividend
Jan 1871  0.02999
Feb 1871  0.01867
Mar 1871  0.01916
Apr 1871  0.01963
May 1871  0.02009
```

```

Jun 1871 0.02054
Jul 1871 0.02097
Aug 1871 0.02140
Sep 1871 0.02181
Oct 1871 0.02220
Nov 1871 0.02259
Dec 1871 0.02296
> sum(div.monthly[1:12])
[1] 0.26
> shiller.annual[1,"dividend"]
Positions dividend
1871      0.26

```

For the S&P 500 index, the index price is available in the **S+FinMetrics** monthly “timeSeries” **shiller.dat**. This information may be utilized in the disaggregation of the annual dividend using the **gls** method as follows

```

> smpl = positions(shiller.dat) <= timeDate("12/31/2000")
> price.monthly = as.matrix(seriesData(shiller.dat[smpl,"price"]))
> div2.monthly =
+ disaggregate(shiller.annual[, "dividend"], 12,
+ method="gls", x=price.monthly, out.positions=monthly.dates)
> div2.monthly[1:12]
Positions dividend
Jan 1871 0.006177
Feb 1871 0.010632
Mar 1871 0.014610
Apr 1871 0.018104
May 1871 0.021104
Jun 1871 0.023569
Jul 1871 0.025530
Aug 1871 0.027043
Sep 1871 0.028063
Oct 1871 0.028508
Nov 1871 0.028548
Dec 1871 0.028111
> sum(div2.monthly[1:12])
[1] 0.26
> shiller.annual[1,"dividend"]
Positions dividend
1871      0.26

```

2.2.7 Merging Time Series

Often one would like to combine several “timeSeries” objects into a single “timeSeries” object. The S-PLUS functions **c**, **concat** and **cbind**

do not operate on “timeSeries” objects. Instead, the S-PLUS function `seriesMerge` is used to combine or merge a collection of “timeSeries”. To illustrate, consider creating a new “timeSeries” object consisting of the S+FinMetrics “timeSeries” `CPI.dat` and `IP.dat` containing monthly observations on the U.S. consumer price index and U.S. industrial production index, respectively:

```
> CPI.dat
Positions    CPI
Jan 1913     9.80
Feb 1913     9.80
...
Nov 2001    177.60
> IP.dat
Positions    IP
Jan 1919     7.628
Feb 1919     7.291
...
Nov 2001    137.139
```

Notice that the start date for `CPI.dat` is earlier than the start date for `IP.dat`, but the end dates are the same. A new “timeSeries” containing both `CPI.dat` and `IP.dat` with positions aligned to those for `IP.dat` using `seriesMerge` is

```
> IP.CPI.dat = seriesMerge(IP.dat,CPI.dat,
+ pos=positions(IP.dat))
> IP.CPI.dat[1:2,]
Positions    IP  CPI
Jan 1919    7.628 16.5
Feb 1919    7.291 16.2
```

To create a “timeSeries” with positions given by the union of the positions for `CPI.dat` and `IP.dat` set `pos="union"` in the call to `seriesMerge`. Since `IP.dat` does not have observations for the dates January 1913 through December 1918, NA values for IP for these dates will be inserted in the new “timeSeries”.

2.2.8 Dealing with Missing Values Using the S+FinMetrics Function *interpNA*

Occasionally, time series data contain missing or incorrect data values. One approach often used to fill-in missing values is interpolation⁵. The S-PLUS

⁵More sophisticated imputation methods for dealing with missing values are available in the library S+MISSINGDATA which is included with S-PLUS.

`align` function may be used for this purpose. The `S+FinMetrics` function `interpNA` performs similar missing value interpolation as `align` but is easier to use and is more flexible. The arguments expected by `interpNA` are

```
> args(interpNA)
function(x, method = "spline")
```

where `x` is a rectangular object and `method` sets the interpolation method. Valid interpolation methods are “before”, “after”, “nearest”, “linear” and (cubic) “spline”. To illustrate the use of `interpNA`, note that the closing price for the Dow Jones Industrial Average in the S-PLUS “timeSeries” `djia` has a missing value on January 18, 1990:

```
> djia.close = djia[positions(djia) >= timeDate("1/1/1990"),
+ "close"]
> djia.close[10:12,]
  Positions  close
01/17/1990 2659.1
01/18/1990    NA
01/19/1990 2677.9
```

To replace the missing value with an interpolated value based on a cubic spline use

```
> djia.close = interpNA(djia.close)
> djia.close[10:12,]
  Positions      1
01/17/1990 2659.1
01/18/1990 2678.7
01/19/1990 2677.9
```

2.3 Time Series Manipulation in S-PLUS

There are several types of common manipulations and transformations that often need to be performed before a financial time series is to be analyzed. The most important transformations are the creation of lagged and differenced variables and the creation of returns from asset prices. The following sections describe how these operations may be performed in S-PLUS.

2.3.1 *Creating Lags and Differences*

Three common operations on time series data are the creation of lags, leads, and differences. The S-PLUS function `shift` may be used to create leads and lags, and the generic function `diff` may be used to create differences. However, these functions do not operate on “timeSeries” objects in the

most convenient way. Consequently, the **S+FinMetrics** module contains the functions **tslag** and **diff.timeSeries** for creating lags/leads and differences.

Creating Lags and Leads Using the **S+FinMetrics** Function **tslag**

The **S+FinMetrics** function **tslag** creates a specified number of lag/leads of a rectangular data object. The arguments expected by **tslag** are

```
> args(tslag)
function(x, k = 1, trim = F)
```

where **x** is any rectangular object, **k** specifies the number of lags to be created (negative values create leads) and **trim** determines if NA values are to be trimmed from the result. For example, consider the “**timeSeries**” **singleIndex.dat** containing monthly prices on Microsoft and the S&P 500 index. The first five values are

```
> singleIndex.dat[1:5,]
Positions MSFT SP500
Jan 1990  1.285 329.1
Feb 1990  1.371 331.9
Mar 1990  1.538 339.9
Apr 1990  1.611 330.8
May 1990  2.028 361.2
```

The “**timeSeries**” of lagged values using **tslag** are

```
> tslag(singleIndex.dat[1:5,])
Positions MSFT.lag1 SP500.lag1
Jan 1990      NA      NA
Feb 1990  1.285    329.1
Mar 1990  1.371    331.9
Apr 1990  1.538    339.9
May 1990  1.611    330.8
```

Notice that **tslag** creates a “**timeSeries**” containing the lagged prices on Microsoft and the S&P 500 index. The variable names are adjusted to indicate the type of lag created and since **trim=F**, NA values are inserted for the first observations. To create a “**timeSeries**” without NA values in the first position, use **tslag** with **trim=T**:

```
> tslag(singleIndex.dat[1:5,],trim=T)
Positions MSFT.lag1 SP500.lag1
Feb 1990  1.285    329.1
Mar 1990  1.371    331.9
Apr 1990  1.538    339.9
May 1990  1.611    330.8
```

Leads are created by setting **k** equal to a negative number:

```
> tslag(singleIndex.dat[1:5,],k=-1)
Positions MSFT.lead1 SP500.lead1
Jan 1990  1.371      331.9
Feb 1990  1.538      339.9
Mar 1990  1.611      330.8
Apr 1990  2.028      361.2
May 1990   NA        NA
```

To create a “timeSeries” with multiple lagged values, simply specify the lags to create in the call to `tslag`. For example, specifying `k=c(1,3)` creates the first and third lag

```
> tslag(singleIndex.dat[1:5,],k=c(1,3))
Positions MSFT.lag1 SP500.lag1 MSFT.lag3 SP500.lag3
Jan 1990   NA        NA        NA        NA
Feb 1990  1.285      329.1      NA        NA
Mar 1990  1.371      331.9      NA        NA
Apr 1990  1.538      339.9      1.285      329.1
May 1990  1.611      330.8      1.371      331.9
```

Similarly, specifying `k=-1:1` creates

```
> tslag(singleIndex.dat[1:5,],k=-1:1)
Positions MSFT.lead1 SP500.lead1 MSFT.lag0 SP500.lag0
Jan 1990  1.371      331.9      1.285      329.1
Feb 1990  1.538      339.9      1.371      331.9
Mar 1990  1.611      330.8      1.538      339.9
Apr 1990  2.028      361.2      1.611      330.8
May 1990   NA        NA        2.028      361.2
MSFT.lag1 SP500.lag1
NA        NA
1.285      329.1
1.371      331.9
1.538      339.9
1.611      330.8
```

Creating Differences Using the S+FinMetrics Function `diff.timeSeries`

The S+FinMetrics function `diff.timeSeries` is a method function for the generic S-PLUS function `diff` for objects of class “timeSeries” and creates a specified number of differences of a “timeSeries” object. The arguments expected by `diff.timeSeries` are

```
> args(diff.timeSeries)
function(x, lag = 1, differences = 1, trim = T, pad = NA)
```

where `x` represents a “timeSeries” object, `lag` specifies the number of lagged periods used in the difference, `differences` specifies the number

of times to difference the series, `trim` determines if the resulting series is to have NA values removed and trimmed and `pad` specifies the value to be padded to the series in the positions where the differencing operation exceeds the start or the end positions. For example, consider again the “timeSeries” `singleIndex.dat` containing monthly prices on Microsoft and the S&P 500 index. Let P_t denote the price at time t . To create the first difference $\Delta P_t = P_t - P_{t-1}$ use `diff` with `lag=1`:

```
> diff(singleIndex.dat[1:5,],lag=1,trim=F)
Positions  MSFT  SP500
Jan 1990      NA    NA
Feb 1990  0.0868  2.81
Mar 1990  0.1667  8.05
Apr 1990  0.0729 -9.14
May 1990  0.4167 30.43
```

To create the difference $P_t - P_{t-2}$ and pad the result with zeros instead of NAs use `diff` with `lag=2` and `pad=0`:

```
> diff(singleIndex.dat[1:5,],lag=2,trim=F,pad=0)
Positions  MSFT  SP500
Jan 1990  0.0000  0.00
Feb 1990  0.0000  0.00
Mar 1990  0.2535 10.86
Apr 1990  0.2396 -1.09
May 1990  0.4896 21.29
```

To create the 2^{nd} difference $\Delta^2 P_t = \Delta(P_t - P_{t-1}) = P_t - 2P_{t-1} + P_{t-2}$ use `diff` with `lag=1` and `diff=2`:

```
> diff(singleIndex.dat[1:5,],lag=1,diff=2,trim=F)
Positions  MSFT  SP500
Jan 1990      NA    NA
Feb 1990      NA    NA
Mar 1990  0.0799  5.24
Apr 1990 -0.0938 -17.19
May 1990  0.3438 39.57
```

Unlike `tslag`, `diff.timeSeries` does not rename the variables to indicate the differencing operation performed. Additionally, `diff.timeSeries` will not accept a vector of values for the arguments `lag` and `differences`.

2.3.2 Return Definitions

Simple Returns

Let P_t denote the price at time t of an asset that pays no dividends and let P_{t-1} denote the price at time $t - 1$. Then the *simple net return* on an

investment in the asset between times $t - 1$ and t is defined as

$$R_t = \frac{P_t - P_{t-1}}{P_{t-1}} = \% \Delta P_t. \quad (2.1)$$

Writing $\frac{P_t - P_{t-1}}{P_{t-1}} = \frac{P_t}{P_{t-1}} - 1$, we can define the *simple gross return* as

$$1 + R_t = \frac{P_t}{P_{t-1}}. \quad (2.2)$$

Unless otherwise stated, references to returns mean net returns.

The simple two-period return on an investment in an asset between times $t - 2$ and t is defined as

$$\begin{aligned} R_t(2) &= \frac{P_t - P_{t-2}}{P_{t-2}} = \frac{P_t}{P_{t-2}} - 1 \\ &= \frac{P_t}{P_{t-1}} \cdot \frac{P_{t-1}}{P_{t-2}} - 1 \\ &= (1 + R_t)(1 + R_{t-1}) - 1. \end{aligned}$$

Then the simple two-period gross return becomes

$$1 + R_t(2) = (1 + R_t)(1 + R_{t-1}) = 1 + R_{t-1} + R_t + R_{t-1}R_t,$$

which is a *geometric* (multiplicative) sum of the two simple one-period gross returns and not the simple sum of the one period returns. If, however, R_{t-1} and R_t are small then $R_{t-1}R_t \approx 0$ and $1 + R_t(2) \approx 1 + R_{t-1} + R_t$ so that $R_t(2) \approx R_{t-1} + R_t$.

In general, the k -period gross return is defined as the geometric average of k one period gross returns

$$\begin{aligned} 1 + R_t(k) &= (1 + R_t)(1 + R_{t-1}) \cdots (1 + R_{t-k+1}) \\ &= \prod_{j=0}^{k-1} (1 + R_{t-j}) \end{aligned} \quad (2.3)$$

and the k -period net return is

$$R_t(k) = \prod_{j=0}^{k-1} (1 + R_{t-j}) - 1. \quad (2.4)$$

Continuously Compounded Returns

Let R_t denote the simple one period return on an investment. The *continuously compounded one period return*, r_t , is defined as

$$r_t = \ln(1 + R_t) = \ln\left(\frac{P_t}{P_{t-1}}\right) \quad (2.5)$$

where $\ln(\cdot)$ is the natural log function. To see why r_t is called the continuously compounded return, take exponentials of both sides of (2.5) to give

$$e^{r_t} = 1 + R_t = \frac{P_t}{P_{t-1}}.$$

Rearranging gives

$$P_t = P_{t-1}e^{r_t},$$

so that r_t is the continuously compounded growth rate in prices between periods $t-1$ and t . This is to be contrasted with R_t which is the simple growth rate in prices between periods $t-1$ and t without any compounding. Since $\ln\left(\frac{x}{y}\right) = \ln(x) - \ln(y)$ it follows that

$$\begin{aligned} r_t &= \ln\left(\frac{P_t}{P_{t-1}}\right) \\ &= \ln(P_t) - \ln(P_{t-1}) \\ &= p_t - p_{t-1} \end{aligned}$$

where $p_t = \ln(P_t)$. Hence, the continuously compounded one period return, r_t , can be computed simply by taking the first difference of the natural logarithms of prices between periods $t-1$ and t .

Given a one period continuously compounded return r_t , it is straightforward to solve back for the corresponding simple net return R_t :

$$R_t = e^{r_t} - 1$$

Hence, nothing is lost by considering continuously compounded returns instead of simple returns.

The computation of multi-period continuously compounded returns is considerably easier than the computation of multi-period simple returns. To illustrate, consider the two period continuously compounded return defined as

$$r_t(2) = \ln(1 + R_t(2)) = \ln\left(\frac{P_t}{P_{t-2}}\right) = p_t - p_{t-2}.$$

Taking exponentials of both sides shows that

$$P_t = P_{t-2}e^{r_t(2)}$$

so that $r_t(2)$ is the continuously compounded growth rate of prices between periods $t-2$ and t . Using $\frac{P_t}{P_{t-2}} = \frac{P_t}{P_{t-1}} \cdot \frac{P_{t-1}}{P_{t-2}}$ and the fact that $\ln(x \cdot y) = \ln(x) + \ln(y)$ it follows that

$$\begin{aligned} r_t(2) &= \ln\left(\frac{P_t}{P_{t-1}} \cdot \frac{P_{t-1}}{P_{t-2}}\right) \\ &= \ln\left(\frac{P_t}{P_{t-1}}\right) + \ln\left(\frac{P_{t-1}}{P_{t-2}}\right) \\ &= r_t + r_{t-1}. \end{aligned}$$

Hence the continuously compounded two period return is just the sum of the two continuously compounded one period returns.

The continuously compounded k -period return is defined as

$$r_t(k) = \ln(1 + R_t(k)) = \ln\left(\frac{P_t}{P_{t-k}}\right) = p_t - p_{t-k}. \quad (2.6)$$

Using similar manipulations to the ones used for the continuously compounded two period return the continuously compounded k -period return may be expressed as the sum of k continuously compounded one period returns:

$$r_t(k) = \sum_{j=0}^{k-1} r_{t-j}. \quad (2.7)$$

The additivity of continuously compounded returns to form multiperiod returns is an important property for statistical modeling purposes.

2.3.3 Computing Asset Returns Using the *S+FinMetrics* Function *getReturns*

Given a data set with asset prices the **S+FinMetrics** function `getReturns` may be used to compute discrete and continuously compounded returns. The arguments to `getReturns` are

```
> args(getReturns)
function(x, type = "continuous", percentage = F, trim = T)
```

where `x` is any rectangular data object and `type` specifies the type of returns to compute (discrete or continuously compounded). To illustrate, the **S+FinMetrics** “timeSeries” `singleIndex.dat` contains monthly closing prices on Microsoft stock and the S&P 500 index, adjusted for stock splits and dividends, over the period January 1990 through January 2001.

```
> colIds(singleIndex.dat)
[1] "MSFT" "SP500"
> singleIndex.dat[1:3,]
Positions MSFT SP500
Jan 1990  1.2847 329.08
Feb 1990  1.3715 331.89
Mar 1990  1.5382 339.94
```

A “timeSeries” of simple one-period discrete returns expressed as percentages is computed as

```
> ret.d = getReturns(singleIndex.dat, type="discrete",
+ percentage=T)
> ret.d[1:3,]
Positions MSFT SP500
```

```
Feb 1990    6.756  0.8539
Mar 1990   12.155  2.4255
Apr 1990    4.739 -2.6887
```

By default the first observation in the “timeSeries” is trimmed. To retain the first (NA) observation use the optional argument `trim=F`

```
> ret.d = getReturns(singleIndex.dat,type="discrete",trim=F)
> ret.d[1:3,]
Positions      MSFT      SP500
Jan 1990         NA         NA
Feb 1990  0.067564  0.008539
Mar 1990  0.121546  0.024255
```

Continuously compounded returns are created by specifying the optional argument `type="continuous"`

```
> ret.cc = getReturns(singleIndex.dat,type="continuous")
> ret.cc[1:3,]
Positions      MSFT      SP500
Feb 1990  0.065380  0.0085027
Mar 1990  0.114708  0.0239655
Apr 1990  0.046304 -0.0272552
```

Multiperiod returns may be computed from a “timeSeries” of one period returns using the S-PLUS function `aggregateSeries`. Multiperiod returns may be either overlapping or non-overlapping. For example, consider computing a monthly “timeSeries” of overlapping annual continuously compounded returns from the monthly continuously compounded returns in the “timeSeries” `ret.cc` using `aggregateSeries`:

```
> ret12.cc = aggregateSeries(ret.cc,moving=12,FUN=sum)
> ret12.cc[1:3,]
Positions      MSFT      SP500
Feb 1990  0.75220  0.044137
Mar 1990  0.74254  0.100749
Apr 1990  0.65048  0.098743
> colSums(seriesData(ret.cc[1:12,]))
      MSFT      SP500
0.7522 0.044137
```

The argument `moving=12` and `FUN=sum` tells `aggregateSeries` to compute a moving sum of twelve returns. Hence, the annual return reported for Feb 1990 is the sum of the twelve monthly returns from February 1990 through January 1991. Non-overlapping annual returns are computed from the monthly returns using `aggregateSeries` with the option `by="years"`

```
> ret12.cc = aggregateSeries(ret.cc,by="years",FUN=sum)
> ret12.cc[1:3,]
```

```

Positions    MSFT      SP500
Jan 1990    0.48678 0.0034582
Jan 1991    0.79641 0.2335429
Jan 1992    0.14074 0.0436749
> colSums(seriesData(ret.cc[1:11,]))
      MSFT      SP500
0.48678 0.0034582

```

The “timeSeries” `ret12.cc` is now an annual series of non-overlapping annual returns. Notice that the annual return for January 1990 is computed using only the eleven returns from February 1990 through December 1990.

Multiperiod discrete returns (2.4) may be computed using the function `aggregateSeries` with `FUN=prod`. For example, a monthly “timeSeries” of overlapping annual discrete returns is computed as

```

> ret12.d = aggregateSeries((1+ret.d),moving=12,FUN=prod)-1
> ret12.d[1:3,]
Positions    MSFT      SP500
Feb 1990    1.12166 0.045126
Mar 1990    1.10128 0.105999
Apr 1990    0.91646 0.103783
> prod(seriesData(1+ret.d[1:12,1]))-1
[1] 1.1217

```

Notice that 1 is added to the return data and 1 is subtracted from the result in order to compute (2.4) properly. Non-overlapping multiperiod discrete returns may be computed using

```

> ret12.d = aggregateSeries((1+ret.d),by="years",FUN=prod)-1
> ret12.d[1:3,]
Positions    MSFT      SP500
Jan 1990      NA      NA
Jan 1991    1.2176 0.26307
Jan 1992    0.1511 0.04464

```

2.4 Visualizing Time Series in S-PLUS

Time series data in “timeSeries” objects may be visualized by using the S-PLUS generic `plot` function, the S-PLUS `trellisPlot` function, or by using the `S+FinMetrics` plotting functions based on Trellis graphics.

2.4.1 Plotting “timeSeries” Using the S-PLUS Generic *plot* Function

The S-PLUS generic `plot` function has a method function, `plot.timeSeries`, for plotting “timeSeries” objects. To illustrate, consider the monthly clos-



FIGURE 2.1. Monthly closing prices on Microsoft stock created using `plot.timeSeries`.

ing prices of Microsoft stock over the period January 1990 to January 2001 in the “timeSeries” object `msft.p` created earlier:

```
> msft.p@title
[1] "Monthly closing price on Microsoft"
> msft.p@units
[1] "US dollar price"
```

Figure 2.1 shows the output produced by the generic `plot` function

```
> plot(msft.p)
```

Notice how the information in the `title` and `units` slots is utilized in the plot. To eliminate the horizontal and vertical grid lines specify `reference.grid=F` in the call to `plot`. To show the price data on a logarithmic scale specify `log.axes="y"` in the call to `plot`.

Multiple series (on the same scale) may also be plotted together on the same plot using `plot`⁶. For example, the prices for Microsoft and the S&P 500 index in the “timeSeries” `singleIndex.dat` may be plotted together using

⁶To create a scatterplot of two “timeSeries” use the extractor function `seriesData` possibly in conjunction with the coercion function `as.matrix` on the “timeSeries” objects in the call to `plot`. Alternatively, the `S+FinMetrics` function `rvfPlot` may be used.

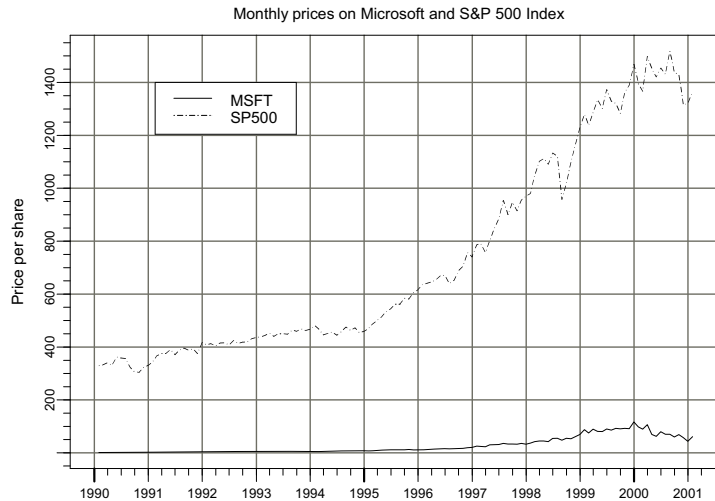


FIGURE 2.2. Monthly closing prices on Microsoft and the S&P 500 index created using `plot.timeSeries`.

```
> plot(singleIndex.dat, plot.args=list(lty=c(1,3)))
> legend(0.1,1400, legend=colIds(singleIndex.dat), lty=c(1,3))
```

The plot is illustrated in Figure 2.2. Notice how the line types are specified as a list argument to the optional argument `plot.args`. In the placement of the legend, the x-axis units are treated as values in the unit interval.

Multipanel plots may be created by specifying the plot layout using the S-PLUS function `par`. Figure 2.3 shows a two panel plot of the price data in `singleIndex.dat` produced using

```
> par(mfrow=c(2,1))
> plot(singleIndex.dat[, "MSFT"],
+ main="Monthly price on Microsoft")
> plot(singleIndex.dat[, "SP500"],
+ main="Monthly price on S&P 500 index")
```

Two specialized plot types for financial data can be made with the function `plot.timeSeries`. The first is a high/low/open/close (hloc) plot and the second is a stackbar plot. These plots are made by setting `plot.type = "hloc"` or `plot.type = "stackbar"` in the call to `plot.timeSeries`. For a hloc plot, the “timeSeries” to be plotted must have hloc information or such information must be created using `aggregateSeries` with the S-PLUS function `hloc`. Stackbar plots are generally used for plotting asset volume

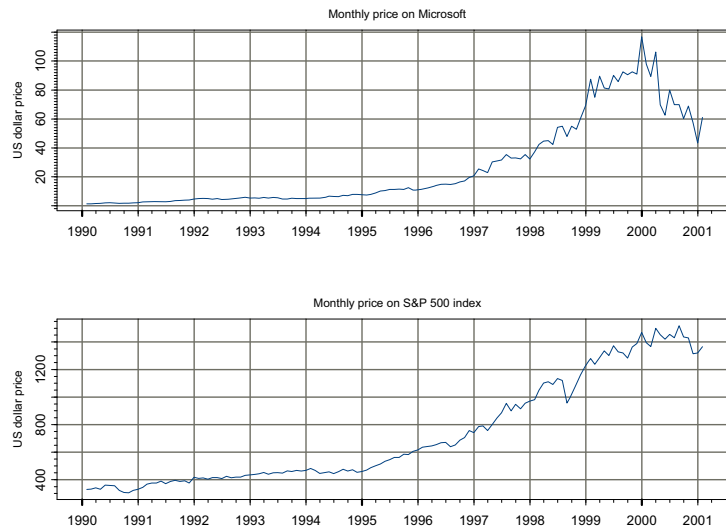


FIGURE 2.3. Two panel plot created using `par(mfrow=c(2,1))` in conjunction with `plot.timeSeries`.

information. To illustrate these plot types, consider the monthly data from the Dow Jones Industrial Averages in the S-PLUS “timeSeries” `djia`:

```
> colIds(djia)
[1] "open" "high" "low" "close" "volume"
```

Figure 2.4 gives a multipanel plot showing high, low, open, close and volume information created by

```
> smpl = (positions(djia) >= timeDate("9/1/1987") &
+ positions(djia) <= timeDate("11/30/1987"))
> par(mfrow=c(2,1))
> plot(djia[smpl,1:4],plot.type="hloc")
> plot(djia[smpl,5],plot.type="stackbar")
```

Lines may be added to an existing time series plot using the S-PLUS function `lines.render` and stackbar information may be added using the S-PLUS function `stackbar.render`. See chapter 26 in the *S-PLUS Guide to Statistics Vol. II* for details on using these functions.

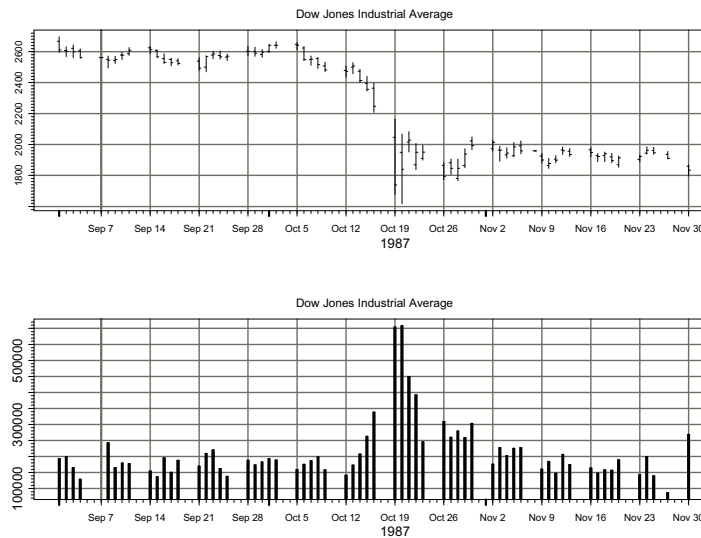


FIGURE 2.4. Monthly high, low, open, close and volume information for the Dow Jones Industrial Average using `plot.timeSeries` with `type="hloc"` and `type="stackbar"`.

Function	Description
<code>seriesPlot</code>	Trellis time series plot
<code>histPlot</code>	Trellis histogram plot
<code>qqPlot</code>	Trellis qq-plot for various distributions

TABLE 2.2. **S+FinMetrics** Trellis plotting functions

2.4.2 Plotting “timeSeries” Using the *S+FinMetrics* Trellis Plotting Functions

S+FinMetrics provides several specialized Trellis-based plotting functions for “timeSeries” objects. These functions extend the **S-PLUS** function `TrellisPlot.timeSeries` and are summarized in Table 2.2.

All of the functions in the table can create multi-panel plots with text labels in the panel strips. For the following examples, monthly return data on six stocks from the **S+FinMetrics** “timeSeries” `DowJones30` will be used. This data is created using

```
> DJ.ret = getReturns(DowJones30[,1:6], percentage=T)
> colIds(DJ.ret)
[1] "AA" "AXP" "T" "BA" "CAT" "C"
```



FIGURE 2.5. Multi-panel time plot created using the **S+FinMetrics** function **seriesPlot**.

The function **seriesPlot** may be used to create single panel or multi-panel time plots. To create the multi-panel time plot of the six Dow Jones 30 assets shown in Figure 2.5 use

```
> seriesPlot(DJ.ret,one.plot=F,strip.text=colIds(DJ.ret),
+ main="Monthly returns on six Dow Jones 30 stocks")
```

Notice that each time plot has a different scale.

The function **histPlot** may be used to create either a single panel histogram of one data series or a multi-panel plot of histograms for multiple series. The multi-panel plot in Figure 2.6 is created using

```
> histPlot(DJ.ret,strip.text=colIds(DJ.ret),
+ main="Histograms of returns on six Dow Jones 30 stocks")
```

Notice that each histogram uses the same bins.

Single panel or multi-panel Trellis-based qq-plots using Gaussian, Student-t, and double exponential distributions may be created using the function **qqPlot**. To illustrate, consider computing qq-plots for the six Dow Jones 30 assets using six Student-t reference distributions with degrees of freedom equal to 5, 6, 7, 8, 9 and 10. These qq-plots, shown in Figure 2.7, are created using

```
> s.text = paste(colIds(DJ.ret),5:10,sep=" ", "df")
```

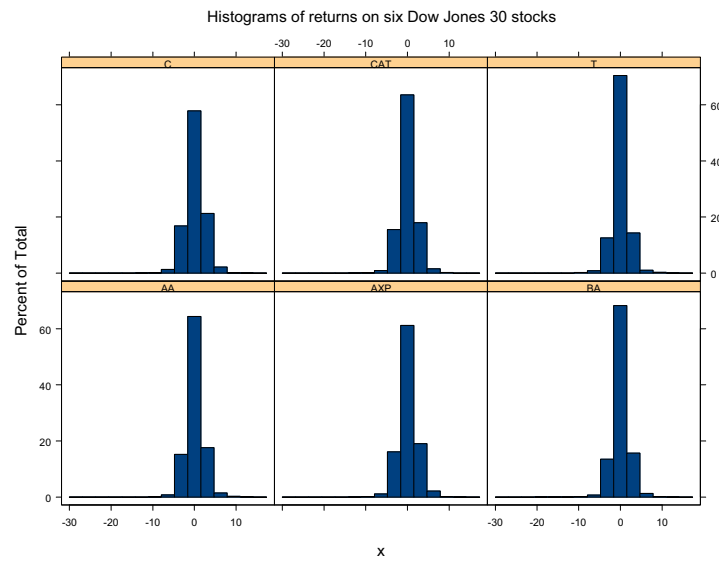


FIGURE 2.6. Multi-panel histogram plot created using the **S+FinMetrics** function `histPlot`.

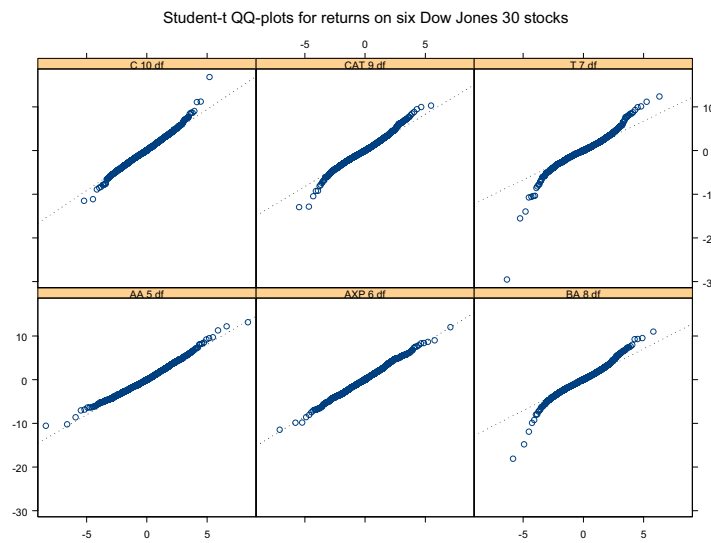


FIGURE 2.7. Multi-panel qq-plots created using the **S+FinMetrics** function `qqPlot`.

```
> qqPlot(DJ.ret,strip.text=s.text,  
+ distribution="t",dof=c(5,6,7,8,9,10), id.n=FALSE,  
+ main="Student-t QQ-plots for returns on six Dow Jones 30 stocks")
```

Notice how the degree of freedom for each Student-t distribution along with the asset name is indicated in the strip text. The optional argument `id.n=FALSE` suppresses the identification of outliers on the qq-plots.

2.5 References

CHOW, G., AND LIN, A. (1971). “Best Linear Unbiased Interpolation, Distribution, and Extrapolation of Time Series by Related Series,” *Review of Economics & Statistics*, 53, 372-375.

Modeling Financial Time Series with S-PLUS®

Zivot, E.; Wang, J.

2006, XXII, 998 p. 270 illus., Softcover

ISBN: 978-0-387-27965-7