Portfolio Management with R

Enrico Schumann <es@enricoschumann.net>

Package version 0.17-0
28 February 2021
2

Contents

1 Introduction
   About PMwR 5
   Inspirations 6
   Principles 6
   Other packages 8
      datetimeutils 8
      mailtools 8
      textutils 9
      tsdb 9
   Setting up R 10
   Typographical conventions 10

2 Keeping track of transactions: journals
   Overview 11
   Fields 11
   Creating and combining journals 13
   Selecting transactions 17
   Computing balances 19
      Computing positions from journals 19
      Algorithms for computing balances 22
   Aggregating and transforming journals 24

3 Computing profit and loss
   Simple cases 27
      Total profit/loss 27
      P/L over time 35

4 Computing returns
   Simple returns 39
   Holding-period returns 42
   Portfolio returns 46
   Return contribution 47
   External Cashflows 48

5 Backtesting
   Decisions 51
   Data structure 52
   Function arguments 54
      Available information within functions 54
      Function arguments 55
   Examples: A single asset 56
      A useless first example 57
      More-useful examples 59
   Examples: Several assets 68
      A simple example 68
Miscellaneous 69
Handling missing values 69
Distributing backtest computations 71
Remembering an entry price 72
Delaying signals 72
Specifying when to compute a signal and trade 73
Writing a log 74
Selecting parameters: calling bttest recursively 74
Time-varying asset universes 76

6 Rebalancing a portfolio
Usage with unnamed vectors 83
Usage with named vectors 85
Usage with positions 86
Algorithms 86
Substituting a basket by its components 89

7 Summarising portfolio time-series
Creating NAV series 91
Summarising NAV series 92
Drawdowns and Streaks 94
  Drawdown 94
  Streaks 96

8 Scaling series
Examples 97
Scaling a series: how it works 101

9 Analysing trades
Exposure 103
Splitting and rescaling 105

10 Plotting irregularly-spaced series during trading hours
An example 107
More examples 110
  Value of plot_trading_hours 110
  Adding grid lines 110
Daily data 113

11 Valuation
Prices 115

12 Other Tools
Dividend adjustments 117
Stocks splits 117
Treasuries quotes 118
Validating ISINs 118
Price tables 119
Trees 119
1 Introduction

1.1 About PMwR

This book describes how to use the PMwR package. PMwR provides a small set of reliable, efficient and convenient tools that help in processing and analysing trade and portfolio data. The package does not provide a complete application that could be used ‘as is’; rather, the package provides building blocks for creating such an application.

PMwR grew out of various pieces of software that I have written since 2008, first at the University of Geneva, later during my work at financial firms. The package is under active development and changes frequently, which is mainly because the code has been written over many years and is in need of being groomed for general use. Consequently, this book will change as frequently as the package. I am grateful for comments and suggestions. The book itself is written in Org mode. The complete tangled code is available from the book’s website.

The PMwR package is on CRAN; to install, simply type

```
install.packages("PMwR")
```

in an R session. The development version of the package is available from http://enricoschumann.net/R/packages/PMwR/. To get the devel version from within R, use the following command:

```
install.packages("PMwR",
               repos = c('http://enricoschumann.net/R',
                          getOption('repos')))
```
The package depends on several other packages, which are automatically obtained from the same repository and from cran. The package source code is also pushed to public repositories at

https://github.com/enricoschumann/PMwR

and

https://gitlab.com/enricoschumann/PMwR.

There is currently no automatic build process for Windows. Recent versions of the package (since 0.3-4) are pure R code and can be built without any prerequisites except an R installation; older versions contained C code, so you needed to have the necessary tool chain installed (typically by installing Rtools). If you have problems building or installing the package for Windows, please contact me and I will provide you with a Windows version.

1.2 Inspirations

I have benefited a lot from using and learning from various pieces of free software.

A great inspiration were and are the principles behind the tools that make working on a Unix-type operating system so productive (and so much fun). In particular, that a programme should do one thing, and that it should do this one thing thoroughly. See for instance Raymond (2003) or Gancarz (2003).

A related inspiration came from the Lua language, which I used much in the past: small is beautiful. (Lua even removed functions.)

Closer to finance, there is the Ledger project.

1.3 Principles

small The aim of PMwR is to provide a small set of tools. This comes at the price: interfaces may be more complicated, with arguments being overloaded. But with few functions, it is easier to remember a function name or to find it in the first place.
flexible and general  `PMwR` aims to be open to different types of instruments, different timestamps, etc. In this respect, I learnt from using the `zoo` package. It gives a good example (I think) how a package should work: blend well with standard data structures, be idiomatic, flexible with regards to timestamps,...

functional  Start with a quote from K&R, chapter 1: "With properly designed functions, it is possible to ignore how a job is done; knowing what is done is sufficient". `PMwR` emphasises computations: transforming simple and transparent data structures into other data structures. In R, a computation means calling a function. There are many good reasons for using functions.

- clearer code; easier to reuse; easier to maintain
- provide a clear view of what is needed for a specific computation; thus, they help with parallel/distributed computing
- easier to test functionality
- input data is not changed, which makes code more reliable
- clean workspace after function call has ended

There are more advantages, actually; such as the application of techniques such as memoisation.

Computations provided by `PMwR` do not – for developers: should not – rely on global options/settings. The exception are functions that are used interactively, which essentially means `print` methods. (In scripts or methods, you should prefer `cat`.)

matching by name  Whenever possible and intuitive, data should be matched by name, not by position. (In other languages, we would use a hash table instead of a vector.) This is most natural with vectors that store scalar information about instruments, such as prices or multipliers. In such cases, data input is preferred in the form of named vectors.

vectorisation  Functions should do vectorisation when it is beneficial in terms of speed or clarity of code. Likewise, functions should work
on matrices directly (typically columnwise) when it simplifies or speeds up things. Otherwise, applying the function (i.e. looping) should be left to the user.

An example may clarify this: drawdown (actually in the NMOF package) is internally computed through \texttt{cumsum}, so it will be fast for a single vector. But for a matrix of time series, it would need a loop, which will be left to the user. On the other hand, \texttt{returns} can be computed very efficiently for a matrix.

1.4 Other packages

Several other packages originated from \texttt{PMwR}; initially, much of their code had been part of \texttt{PMwR}.

1.4.1 datetimeutils

From the DESCRIPTION file:

Utilities for handling dates and times, such as selecting particular days of the week or month, formatting timestamps as required by RSS feeds, or converting timestamp representations of other software (such as 'MATLAB' and 'Excel') to R. The package is lightweight (no dependencies, pure R implementations) and relies only on R’s standard classes to represent dates and times (’Date’ and ’POSIXt’); it aims to provide efficient implementations, through vectorisation and the use of R’s native numeric representations of timestamps where possible.

https://github.com/enricoschumann/datetimeutils
http://enricoschumann.net/R/packages/datetimeutils/

1.4.2 mailtools

From the DESCRIPTION file:

Utilities to handle email.
1.4.3 textutils

From the DESCRIPTION file:
Utilities for handling character vectors that store human-readable text (either plain or with markup, such as HTML or \LaTeX). The package provides, in particular, functions that help with the preparation of plain-text reports, e.g. for expanding and aligning strings that form the lines of such reports. The package also provides generic functions for transforming R objects to HTML and to plain text.

https://github.com/enricoschumann/textutils
http://enricoschumann.net/R/packages/textutils

1.4.4 tsdb

From the DESCRIPTION file:
A terribly-simple data base for numeric time series, written purely in R, so no external database-software is needed. Series are stored in plain-text files (the most-portable and enduring file type) in CSV format. Timestamps are encoded using R’s native numeric representation for 'Date'/\POSIXct', which makes them fast to parse, but keeps them accessible with other software. The package provides tools for saving and updating series in this standardised format, for retrieving and joining data, for summarising files and directories, and for coercing series from and to other data types (such as 'zoo' series).

https://github.com/enricoschumann/textutils
http://enricoschumann.net/R/packages/textutils
Chapter 1. Typographical conventions

1.5 Setting up R

In this manual, all R output will be presented in English. In case you run R in a different locale, but want to receive messages in English, type this:

```
Sys.setenv(LANGUAGE = "en")
```

1.6 Typographical conventions

R code is shown in a typewriter font like this.

```
1+1
```

The results of a computation are shown as follows:

```
[1] 2
```

Note that R output is marked with a vertical line, so that it can more easily be distinguished from the input. The book’s whole code is exported into a single file. To make it easier to find particular pieces of code, some code chunks have names that are printed in the margin, such as `[code-example]` here. In the code file, the chunk will begin with a line

```
#+NAME: code-example
```
2 Keeping track of transactions: journals

2.1 Overview

The ultimate basis of many financial computations are lists of transactions. \texttt{pmwr} provides an S3 class \texttt{journal} for handling such lists. A \texttt{journal} is a list of atomic vectors, to which a class attribute is attached. (Thus, a \texttt{journal} is quite similar to a dataframe.) Such a list is created through the function \texttt{journal}. Methods should not rely on this list being sorted in any particular way: components of a journal should always be retrieved by name, never by position. (In this respect a journal differs from a dataframe, for which we can meaningfully refer to the $n$-th column.) A journal’s components, such as \texttt{amount} or \texttt{timestamp}, are called \texttt{fields} in this manual.

The simplicity of the class is intended, because it is meant for interactive analyses. The user may – and is expected to – dissect the information in a \texttt{journal} at will; such dissections include removing the class attribute.

2.2 Fields

What is actually stored in a \texttt{journal} is up to the user. A number of fields are, however, required for certain operations and so it is recommended that they be present:

\textbf{amount} The notional amount that is transacted. \texttt{amount} is, in a way, the most important property of a journal. When functions compute
something from the journal (the number of transactions, say), they will often look at amount.

**timestamp** When did the transaction take place? A numeric or character vector; should be sortable.

**price** Well, price. Well, there are many types of prices. The price specified in a journal may be used to compute profit/loss; therefore, the difference between prices should be proportional to profit/loss for the transactions. Unfortunately, there are many instruments that are not quoted in transaction prices: Options may be quoted in implied volatility, bonds in yield, etc. For such instruments, the net-asset value (NAV) should be used.

**instrument** Description of the financial instrument; typically an identifier, a.k.a. ticker or symbol. That is, a string or perhaps a number; but not a more-complex object (recall that journals are lists of atomic vectors).

**id** A transaction identifier, possibly but not necessarily unique.

**account** Description of the account.

... other fields. They must be named, as in `fees = c(1,2,1)`.

All fields except amount can be missing. Such missing values will be ‘added back’ as NA with the exception of id and account, which will be NULL. To be clear: amount could be a vector of only NA values, but it cannot be left out when the journal is created. (This will become clearer with the examples below.)

A journal may have no transactions at all in it. In such a case all fields have length zero, e.g. amount would be `numeric(0)` and so on. Such empty journals may be created by saying `journal()` or by coercing a zero-row dataframe to a journal, via a call to `as.journal`.

Transactions in a journal may be organised in hierarchies, such as

```
account => subaccount => subsubaccount => ... => instrument
```

This is useful and necessary when you have traded an instrument for different accounts, say, or as part of different strategies. Such a hierarchy

---

*Numbers have to be converted to character in some situations, for instance when used as names.*
may be completely captured in the instrument field, by concatenating account hierarchy and instrument using a separator pattern such as ::.

The result would be 'namespaced' instruments such as Pension::Equities::AMZN. Alternatively, part of the hierarchy may be stored in the account field.

2.3 Creating and combining journals

The function journal creates journal objects. See ?journal for details about the function and about methods for journal objects. At its very minimum, a journal must contain amounts of something.

```r
J <- journal(amount = c(1, 2, -2, 3))
J
```

```
 amount
 1 1
 2 2
 3 -2
 4 3
```

4 transactions

Actually, that is not true. On occasion it is useful to create an empty journal, one with no entries at all. You can do so by saying journal(), without any arguments.

```r
journal()
```

```
no transactions
```

To see the current balance, which is nothing more than the sum over all amounts, you can use position.

```r
position(J)
```

```
4
```

Suppose you wanted to note how many bottles of milk and wine you have stored in your basement. Whenever you add to your stock, you have a positive amount; whenever you retrieve bottles, you have a negative amount. Then, by keeping track of transactions, you do not have

---

\(^2\)This notation is inspired by the syntax of ledger files. See https://www.ledger-cli.org/.
Chapter 2. Creating and combining journals

to take stock (apart, perhaps, from occasional checking that you did not miss a transaction), as long as you keep track of what you put into your cellar and what you take out.

There may be some analyses you can do on flows alone: you may check your drinking habits for patterns, such as slow accumulation of wine, followed by rapid consumption; or the other way around. But typically, you will want to analyse your transactions later, and then the more information you record about them – when, what, why, at what price, etc. –, the better. Journals allow you to store such information. To show how they are used, let us switch to a financial example.

Suppose you have transacted the following trades.

```
| timestamp | account | instrument | amount | price       | note        |
|------------+---------+------------+--------+------------+-------------|
| 2017-08-01 | Pension | AMZN       | 10     | 1001.00    |             |
| 2017-08-01 | Pension | MSFT       | 220    | 73.10      |             |
| 2017-07-14 | Trading | AMZN       | 10     | 1001.50    |             |
| 2017-07-31 | Trading | AMZN       | -5     | 1014.00    | take profit |
| 2017-08-15 | Trading | AMZN       | 10     | 985.50     |             |
| 2017-10-05 | Pension | MSFT       | 70     | 74.40      |             |
```

This table is formatted in Org syntax. The function `org_journal` converts such a table into a journal.

```
org_journalt <- function(file, text, timestamp.as.Date = TRUE) {
  ans <- orgutils::readOrg(text = text)
  if (timestamp.as.Date && "timestamp" %in% colnames(ans))
    ans$timestamp <- as.Date(ans$timestamp)
  ans <- as.journal(ans)
  ans
}
```

We read the table into a journal.

```
J <- org_journalt(text = "
| timestamp | account | instrument | amount | price       | note        |
|------------+---------+------------+--------+------------+-------------|
| 2017-08-01 | Pension | AMZN       | 10     | 1001.00    |             |
| 2017-08-01 | Pension | MSFT       | 220    | 73.10      |             |
| 2017-07-14 | Trading | AMZN       | 10     | 1001.50    |             |
| 2017-07-31 | Trading | AMZN       | -5     | 1014.00    | take profit |
| 2017-08-15 | Trading | AMZN       | 10     | 985.50     |             |
| 2017-10-05 | Pension | MSFT       | 70     | 74.40      |             |
```

A print method defines how a journal is displayed. See `?print.journal` for details. In general, you can always get help for methods for generic functions by saying `?<generic_function>.journal`, e.g. `?print.journal` or `?as.data.frame.journal`.

```r
print(J, max.print = 2, exclude = c("instrument", "note"))
```

```
account   timestamp  amount  price
1  Pension  2017-08-01 10  1001.0
2  Pension  2017-08-01 220 73.1
[ .... ]
6 transactions
```

A `str` method shows the fields in the journal.

```r
str(J)
```

```
'journal': 6 transactions
  $ instrument: chr [1:6] "AMZN" "MSFT" "AMZN" "AMZN" ...
  $ account : chr [1:6] "Pension" "Pension" "Trading" ...
  $ timestamp : Date[1:6], format: "2017-08-01" ...
  $ amount : int [1:6] 10 220 10 -5 10 70
  $ price : num [1:6] 1001 73.1 1001.5 1014 985.5 ...
  $ note : chr [1:6] "" "" "" "take profit" ...
```

You may notice that the output is similar to that of a `data.frame` or `list`. That is because `J` is a list of atomic vectors, with a class attribute. Essentially, it is little more than a list of the columns of the above table.

But note that `journal` would silently have added required fields such as `price`, initialised as `NA`.

```r
str(journal(amount = 1))
```

```
'journal': 1 transaction
  $ instrument: chr NA
  $ timestamp : num NA
  $ amount : num 1
  $ price : num NA
```
In the example, the timestamps are of class `Date`. But essentially, any vector of mode character or numeric can be used, for instance `POSIXct`, or other classes. Here is an example that uses the `nanotime` package (Eddelbuettel, 2017).

```r
library("nanotime")
journal(amount = 1:3, timestamp = nanotime(Sys.time()) + 1:3)
```

<table>
<thead>
<tr>
<th>timestamp</th>
<th>amount</th>
</tr>
</thead>
<tbody>
<tr>
<td>2020-08-09T08:27:19.812951001+00:00</td>
<td>1</td>
</tr>
<tr>
<td>2020-08-09T08:27:19.812951002+00:00</td>
<td>2</td>
</tr>
<tr>
<td>2020-08-09T08:27:19.812951003+00:00</td>
<td>3</td>
</tr>
</tbody>
</table>

3 transactions

Journals can be combined with `c`.

```r
J2 <- J
J2$remark <- rep("new", length(J))
c(J, J2)
```

<table>
<thead>
<tr>
<th>instrument</th>
<th>timestamp</th>
<th>amount</th>
<th>price</th>
<th>account</th>
<th>note</th>
<th>remark</th>
</tr>
</thead>
<tbody>
<tr>
<td>AMZN</td>
<td>2017-08-01</td>
<td>10</td>
<td>1001.0</td>
<td>Pension</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MSFT</td>
<td>2017-08-01</td>
<td>220</td>
<td>73.1</td>
<td>Pension</td>
<td></td>
<td></td>
</tr>
<tr>
<td>AMZN</td>
<td>2017-07-14</td>
<td>10</td>
<td>1001.5</td>
<td>Trading</td>
<td></td>
<td></td>
</tr>
<tr>
<td>AMZN</td>
<td>2017-07-31</td>
<td>-5</td>
<td>1014.0</td>
<td>Trading</td>
<td>take profit</td>
<td></td>
</tr>
<tr>
<td>AMZN</td>
<td>2017-08-15</td>
<td>10</td>
<td>986.5</td>
<td>Trading</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MSFT</td>
<td>2017-10-05</td>
<td>70</td>
<td>74.4</td>
<td>Pension</td>
<td></td>
<td></td>
</tr>
<tr>
<td>AMZN</td>
<td>2017-08-01</td>
<td>10</td>
<td>1001.0</td>
<td>Pension</td>
<td></td>
<td>new</td>
</tr>
<tr>
<td>MSFT</td>
<td>2017-08-01</td>
<td>220</td>
<td>73.1</td>
<td>Pension</td>
<td></td>
<td>new</td>
</tr>
<tr>
<td>AMZN</td>
<td>2017-07-14</td>
<td>10</td>
<td>1001.5</td>
<td>Trading</td>
<td></td>
<td>new</td>
</tr>
<tr>
<td>AMZN</td>
<td>2017-07-31</td>
<td>-5</td>
<td>1014.0</td>
<td>Trading</td>
<td>take profit</td>
<td>new</td>
</tr>
<tr>
<td>AMZN</td>
<td>2017-08-15</td>
<td>10</td>
<td>986.5</td>
<td>Trading</td>
<td></td>
<td>new</td>
</tr>
<tr>
<td>MSFT</td>
<td>2017-10-05</td>
<td>70</td>
<td>74.4</td>
<td>Pension</td>
<td></td>
<td>new</td>
</tr>
</tbody>
</table>

12 transactions

The new combined journal will not be sorted by date. In general, a journal need not be sorted in any particular way. There is a `sort` method for journals, whose default is to sort by timestamp. We can also sort by other fields, for instance by amount.

```r
sort(c(J, J2), by = c("amount", "price"), decreasing = FALSE)
```

<table>
<thead>
<tr>
<th>instrument</th>
<th>timestamp</th>
<th>amount</th>
<th>price</th>
<th>account</th>
<th>note</th>
<th>remark</th>
</tr>
</thead>
<tbody>
<tr>
<td>AMZN</td>
<td>2017-07-31</td>
<td>-5</td>
<td>1014.0</td>
<td>Trading</td>
<td>take profit</td>
<td>&lt;NA&gt;</td>
</tr>
</tbody>
</table>
You can query the number of transactions in a journal with `length`.

### 2.4 Selecting transactions

In an interactive session, you can use `subset` to select transactions.

```r
subset(J, amount > 10)
```

<table>
<thead>
<tr>
<th>instrument</th>
<th>timestamp</th>
<th>amount</th>
<th>price</th>
<th>account</th>
<th>comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>MSFT</td>
<td>2017-08-01</td>
<td>220</td>
<td>73.1</td>
<td>Pension</td>
<td></td>
</tr>
<tr>
<td>MSFT</td>
<td>2017-10-05</td>
<td>70</td>
<td>74.4</td>
<td>Pension</td>
<td></td>
</tr>
</tbody>
</table>

2 transactions

With `subset`, you need not quote the expression that selects trades and you can directly access a journal’s fields. Because of the way `subset` evaluates its arguments, it should not be used within functions. (See the Examples section in `?journal` for what can happen then.)

More generally, to extract or change a field, use its name, either through the `$` operator or double brackets `[[...]]`.

```r
J$amount
```

---

3The behaviour of `[[` may change in the future: it may then be used to iterate over the transactions in a journal, not the fields. This would be motivated by [https://developer.r-project.org/blosxom.cgi/R-devel/NEWS/2016/03/09](https://developer.r-project.org/blosxom.cgi/R-devel/NEWS/2016/03/09) even though the commit was reversed two days later [https://developer.r-project.org/blosxom.cgi/R-devel/NEWS/2016/03/11](https://developer.r-project.org/blosxom.cgi/R-devel/NEWS/2016/03/11)
Chapter 2. Selecting transactions

You can also replace specific fields.

```
J$comment[1] <- "some note"
```

<table>
<thead>
<tr>
<th>instrument</th>
<th>timestamp</th>
<th>amount</th>
<th>price</th>
<th>account</th>
<th>comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>AMZN</td>
<td>2017-08-01</td>
<td>10</td>
<td>1001.0</td>
<td>Pension</td>
<td>a note</td>
</tr>
<tr>
<td>MSFT</td>
<td>2017-08-01</td>
<td>220</td>
<td>73.1</td>
<td>Pension</td>
<td></td>
</tr>
<tr>
<td>AMZN</td>
<td>2017-07-14</td>
<td>10</td>
<td>1001.5</td>
<td>Trading</td>
<td></td>
</tr>
<tr>
<td>AMZN</td>
<td>2017-07-31</td>
<td>-5</td>
<td>1014.0</td>
<td>Trading</td>
<td>take profit</td>
</tr>
<tr>
<td>AMZN</td>
<td>2017-08-15</td>
<td>10</td>
<td>985.5</td>
<td>Trading</td>
<td></td>
</tr>
<tr>
<td>MSFT</td>
<td>2017-10-05</td>
<td>70</td>
<td>74.4</td>
<td>Pension</td>
<td></td>
</tr>
</tbody>
</table>

6 transactions

The `[` method works with integers or logicals, returning the respective transactions.

```
J[2:3]
```

<table>
<thead>
<tr>
<th>instrument</th>
<th>timestamp</th>
<th>amount</th>
<th>price</th>
<th>account</th>
<th>comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>MSFT</td>
<td>2017-08-01</td>
<td>220</td>
<td>73.1</td>
<td>Pension</td>
<td></td>
</tr>
<tr>
<td>AMZN</td>
<td>2017-07-14</td>
<td>10</td>
<td>1001.5</td>
<td>Trading</td>
<td></td>
</tr>
</tbody>
</table>

2 transactions

```
J[J$amount < 0]
```

<table>
<thead>
<tr>
<th>instrument</th>
<th>timestamp</th>
<th>amount</th>
<th>price</th>
<th>account</th>
<th>comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>AMZN</td>
<td>2017-07-31</td>
<td>-5</td>
<td>1014</td>
<td>Trading</td>
<td>take profit</td>
</tr>
</tbody>
</table>

1 transaction

You can also pass a string, which is then interpreted as a regular expression that is matched against all character fields in the journal.

```
J["Pension"]
```

<table>
<thead>
<tr>
<th>instrument</th>
<th>timestamp</th>
<th>amount</th>
<th>price</th>
<th>account</th>
<th>comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>AMZN</td>
<td>2017-08-01</td>
<td>10</td>
<td>1001.0</td>
<td>Pension</td>
<td>a note</td>
</tr>
<tr>
<td>MSFT</td>
<td>2017-08-01</td>
<td>220</td>
<td>73.1</td>
<td>Pension</td>
<td></td>
</tr>
<tr>
<td>MSFT</td>
<td>2017-10-05</td>
<td>70</td>
<td>74.4</td>
<td>Pension</td>
<td></td>
</tr>
</tbody>
</table>

3 transactions
You can also specify the fields to match the string against.

```r
J["Pension", match.against = "instrument"]
```

no transactions

By default, case is ignored, but you can set `ignore.case` to `FALSE`.

```r
J["pension", ignore.case = FALSE]
```

no transactions

Finally, you can invert the selection with `invert`.

```r
J["Pension", invert = TRUE]
```

```
   instrument    timestamp    amount    price    account       comment
   <chr>          <Date>       <int>      <dbl>     <chr>          <chr>    
1     AMZN 2017-07-14       10  1001.5 Trading
2     AMZN 2017-07-31       -5  1014.0 Trading take profit
3     AMZN 2017-08-15       10   985.5 Trading
```

3 transactions

### 2.5 Computing balances

#### 2.5.1 Computing positions from journals

The function `position` gives the current balance of all instruments.

```r
position(J)
```

```
2017-10-05
AMZN  25
MSFT  290
```

To get the position at a specific date, use the `when` argument.

```r
position(J, when = as.Date("2017-08-10"))
```

```
2017-08-10
AMZN   15
MSFT   220
```

If you do not like such a tabular view, consider splitting the journal.
Chapter 2. Computing balances

```r
lapply(split(J, J$instrument),
    position, when = as.Date("2017-08-10"))
```

- **$AMZN**
  - 2017-08-10
    - AMZN 15

- **$MSFT**
  - 2017-08-10
    - MSFT 220

To get a time series of positions, you can use specific keywords for `when`
- `all` will print the position at all timestamps in the journal.

```r
position(J, when = "all")
```

<table>
<thead>
<tr>
<th></th>
<th>AMZN</th>
<th>MSFT</th>
</tr>
</thead>
<tbody>
<tr>
<td>2017-08-01</td>
<td>15</td>
<td>220</td>
</tr>
<tr>
<td>2017-07-14</td>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>2017-07-31</td>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>2017-08-15</td>
<td>25</td>
<td>220</td>
</tr>
<tr>
<td>2017-10-05</td>
<td>25</td>
<td>290</td>
</tr>
</tbody>
</table>

Keywords `first` and `last` give you the first and last position. (The latter is the default; so if `when` is not specified at all, the last position is computed.)

```r
endofday computes the positions at the ends of calendar days in the journal.
```

```r
endofmonth/=endofyear prints the positions at the ends of all calendar months/years between the first and the last timestamp. (The function `nth_day` in package `datetimeutils` offers more options.)
```

We are not limited to the timestamps that exist in the journal.

```r
position(J, when = seq(from = as.Date("2017-07-10"),
    to = as.Date("2017-07-20"),
    by = "1 day"))
```

<table>
<thead>
<tr>
<th></th>
<th>AMZN</th>
<th>MSFT</th>
</tr>
</thead>
<tbody>
<tr>
<td>2017-07-10</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2017-07-11</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2017-07-12</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2017-07-13</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2017-07-14</td>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>2017-07-15</td>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>2017-07-16</td>
<td>10</td>
<td>0</td>
</tr>
</tbody>
</table>
By default, position will show you positions of all instruments, even if they are zero.

```r
position(J, when = as.Date("2017-7-15"))
```

<table>
<thead>
<tr>
<th>Date</th>
<th>AMZN</th>
<th>MSFT</th>
</tr>
</thead>
<tbody>
<tr>
<td>2017-07-15</td>
<td>10</td>
<td>0</td>
</tr>
</tbody>
</table>

You can suppress such positions with `drop.zero`.

```r
position(J, when = as.Date("2017-7-15"), drop.zero = TRUE)
```

<table>
<thead>
<tr>
<th>Date</th>
<th>AMZN</th>
</tr>
</thead>
<tbody>
<tr>
<td>2017-07-15</td>
<td>10</td>
</tr>
</tbody>
</table>

`drop.zero` can also be a numeric value, in which case it is interpreted as an absolute tolerance. This is useful in cases such as this one:

```r
position(journal(instrument = "USD",
                timestamp = as.Date("2012-01-05"),
                amount = c(0.1, 0.1, 0.1, -0.3)),
                drop.zero = TRUE)
```

<table>
<thead>
<tr>
<th>Date</th>
<th>USD</th>
</tr>
</thead>
<tbody>
<tr>
<td>2012-01-05</td>
<td>2.7756e-17</td>
</tr>
</tbody>
</table>

```r
position(journal(instrument = "USD",
                timestamp = as.Date("2012-01-05"),
                amount = c(0.1, 0.1, 0.1, -0.3)),
                drop.zero = 1e-12)
```

(Note that there is no output.)

As a final example, when accounts are specified, we may also aggregate positions by account.

```r
position(J, use.account = TRUE)
```
Chapter 2. Computing balances

As described above, each instrument gets its own 'namespace'.

```r
as.data.frame(position(J, use.account = TRUE))
```

<table>
<thead>
<tr>
<th>Pension.AMZN</th>
<th>Pension.MSFT</th>
<th>Trading.AMZN</th>
</tr>
</thead>
<tbody>
<tr>
<td>2017-10-05</td>
<td>10</td>
<td>290</td>
</tr>
<tr>
<td>2017-10-05</td>
<td>15</td>
<td>15</td>
</tr>
</tbody>
</table>
```

2.5.2 Algorithms for computing balances

We have three vectors: `when`, `timestamp` and `amount`. Vectors `when` and `timestamp` are of the same type and are both sorted in increasing order; `timestamp` and `amount` have the same length. The result of the computation is a vector `position` with the same length as `when`.

```c
i, j = 0 /* i loops over when; j loops over amount/timestamp */
for (i = 0; i < length(when); i++) {
    if (i == 0)
        pos[i] = 0;
    else
        pos[i] = pos[i - 1];
    while (timestamp[j] <= when[i] && j < length(j))
        position[i] += amount[j++];
}
```

Below follow two Perl snippets that compute positions from list of trades. (Perl syntax is similar to C syntax; in particular, array indices start at 0. This makes Perl very useful to test algorithms that are later to be coded in C.)

```perl
when and timestamp sorted

use warnings;
use strict;
use v5.14;
```
my @when = (0,1,2,7);    ## when to compute position
my @timestamp = (0,0,0,2);  ## timestamps of trades
my @amount = (1,1,1,-2);   ## traded amounts

## when and timestamp sorted
my @pos = (0) x @when;  ## same length as @when
my $i = 0;
my $j = 0;

## /* i loops over when; j loops over amount/timestamp */
for ($i = 0; $i < @when; $i++) {
    if ($i == 0) {
        $pos[$i] = 0;
    } else {
        $pos[$i] = $pos[$i - 1];
    }
    while ($j < @amount && $timestamp[$j] <= $when[$i]) {
        $pos[$i] += $amount[$j];
        $j += 1;
    }
}
say "@pos";
3 3 1 1

when and timestamp unsorted

use warnings;
use strict;
use v5.14;

my @when = (0,1,2,7);    ## when to compute position
my @timestamp = (0,0,0,2);  ## timestamps of trades
my @amount = (1,1,1,-2);   ## traded amounts

my @pos = (0) x @when;  ## same length as @when
my $i = 0;
my $j = 0;
@pos = (0,0);
for ($i = 0; $i < @when; $i++) {
    for ($j = 0; $j < @timestamp; $j++) {
Chapter 2. Aggregating and transforming journals

Often the data provided by journals needs to be processed in some way. A straightforward strategy is to call `as.data.frame` on the journal and then to use one of the many functions and methods that can be used for dataframes, such as `aggregate` or `apply`.

Even without coercion to a dataframe: A journal is a list of atomic vectors and hence already very similar to a dataframe. As a consequence, many computations can also be done directly on the journal, in particular with `tapply`. An example: you have a journal `trades` and want to compute monthly turnover (two-way). If there is only one instrument or all instruments may be added without harm (typically when they are denominated in the same currency), you can use this expression:

```r
tapply(trades, INDEX = format(trades$timestamp, "%Y-%m"), FUN = function(x) sum(abs(x$amount)))
```

To break it down by instrument, just add instrument as a second grouping variable to the `INDEX` argument.

```r
tapply(trades, INDEX = list(format(trades$timestamp, "%Y-%m"), trades$instrument), FUN = function(x) sum(abs(x$amount)))
```

A special case is when a journal is to be processed into a new journal. For this, PMWR defines an aggregate method for journals:

```r
aggregate.journal(x, by, FUN, ...)
```
The method splits the journal according to the grouping argument by, which can be a list (as in the default method) or an atomic vector. The argument FUN can either be a function or list. If a function, it should receive a journal and also evaluate to a journal. (Note that this is different from R’s `aggregate.data.frame`, which calls FUN on all columns, but in turn cannot address specific columns of the data.frame.) If FUN is a list, its elements should be named functions. The names should match fields in the journal.

An example: we have a journal covering two trading days and wish to create a summary journal, which aggregates buys and sells for every day.

```r
J <- org_journal(text = "@\chunkname{aggregate\hyp{}by\hyp{}day}@ |
instrument | timestamp      | amount | price |
|------------+----------------+--------+-------|
| A          | 2013-09-02 Mon |     -3 |   102 |
| B          | 2013-09-02 Mon |     -3 |   104 |
| B          | 2013-09-02 Mon |      3 |   106 |
| B          | 2013-09-02 Mon |     -2 |   104 |
| A          | 2013-09-03 Tue |     -1 |   110 |
| A          | 2013-09-03 Tue |      1 |   104 |
| A          | 2013-09-03 Tue |      5 |   108 |
| A          | 2013-09-03 Tue |      3 |   107 |
| B          | 2013-09-03 Tue |     -4 |   102 |
| B          | 2013-09-03 Tue |      3 |   106 |
")

fun <- function(x) {
  journal(timestamp = as.Date(x$timestamp[1]),
          amount = sum(x$amount),
          price = sum(x$amount*x$price)/sum(x$amount),
          instrument = x$instrument[1L])
}

aggregate(J,
          by = list(J$instrument,
                    sign(J$amount),
                    as.Date(J$timestamp)),
          FUN = fun)
```

The results is a journal, but with at most a single buy or sell transaction per instrument per day: see the buy transaction for instrument A on September, 3.
Chapter 2. Aggregating and transforming journals

<table>
<thead>
<tr>
<th>instrument</th>
<th>timestamp</th>
<th>amount</th>
<th>price</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A 2013-09-02</td>
<td>-3</td>
<td>102.0000</td>
</tr>
<tr>
<td>2</td>
<td>B 2013-09-02</td>
<td>-5</td>
<td>104.0000</td>
</tr>
<tr>
<td>3</td>
<td>B 2013-09-02</td>
<td>3</td>
<td>106.0000</td>
</tr>
<tr>
<td>4</td>
<td>A 2013-09-03</td>
<td>-1</td>
<td>110.0000</td>
</tr>
<tr>
<td>5</td>
<td>B 2013-09-03</td>
<td>-4</td>
<td>102.0000</td>
</tr>
<tr>
<td>6</td>
<td>A 2013-09-03</td>
<td>9</td>
<td>107.2222</td>
</tr>
<tr>
<td>7</td>
<td>B 2013-09-03</td>
<td>3</td>
<td>106.0000</td>
</tr>
</tbody>
</table>

7 transactions
3 Computing profit and loss

In this chapter we will deal with computing profit and loss (P/L) measured in amounts of currency. If you are interested in computing returns, see Section Computing returns.

3.1 Simple cases

3.1.1 Total profit/loss

We buy one unit of an asset at a price of 100 euro and we sell it for 101. We have made a profit of 1 euro.

This simple case is frequent enough that we should make the required computation simple as well. The \texttt{PMwR} package provides a function \texttt{pl}, which for this case may be called as follows.

\begin{verbatim}
pl(price = c(100, 101),
    amount = c(1, -1))
\end{verbatim}

\begin{verbatim}
P/L total 1
average buy 100
average sell 101
cum. volume 2
\end{verbatim}

P/L 'total is in units of instrument;''
volume is sum of /absolute/ amounts.

Instead of a vectors \texttt{price} and \texttt{amount}, you could also have passed a journal to \texttt{pl}.
Chapter 3. Simple cases

In principle, profit/loss (P/L) is straightforward to compute. Let $x$ be a vector of the absolute amounts traded, and let $p$ be a vector of the prices at which we traded. Then P/L is just the difference between what we received when selling and what we paid when buying.

$$\sum x_i \text{sell}_i p_i \text{sell}_i - \sum x_i \text{buy}_i p_i \text{buy}_i$$  \hspace{1cm} (3.1)

This can be simplified when we impose the convention that sold amounts are negative.

$$\frac{P}{L} = - \sum_{x < 0} x_i p_i - \sum_{x > 0} x_i p_i$$  \hspace{1cm} (3.2)

$$= - \sum x_i p_i$$  \hspace{1cm} (3.3)

The function pl also expects this convention: in the code example we had $x = [1, -1]'$.

There are several ways to perform this basic (or fundamental, rather) computation. Here are some, along with some timing results.

```r
amount <- rep(c(-100, 100), 500)
price <- rep(100, length(amount))
library("rbenchmark")
benchmark(
    ## variations
    amount %*% price,
    sum(amount*price),
    crossprod(amount, price),
    t(amount * price) %*% rep(1, length(amount)), ## matrix summing
    ## settings
    columns = c("test", "elapsed", "relative"),
    order = "relative",
    replications = 50000)
```

<table>
<thead>
<tr>
<th>test</th>
<th>elapsed</th>
<th>relative</th>
</tr>
</thead>
<tbody>
<tr>
<td>amount %*% price</td>
<td>0.126</td>
<td>1.000</td>
</tr>
<tr>
<td>crossprod(amount, price)</td>
<td>0.138</td>
<td>1.095</td>
</tr>
<tr>
<td>sum(amount * price)</td>
<td>0.172</td>
<td>1.365</td>
</tr>
<tr>
<td>t(amount * price) %*% rep(1, length(amount))</td>
<td>0.440</td>
<td>3.492</td>
</tr>
</tbody>
</table>

pl uses the straightforward sum(amount * price) variant; only when very long vectors are used, it switches to crossprod.
pl also accepts an argument instrument: if it is available, pl computes and reports P/L for each instrument separately. As an example, suppose you traded shares of two German companies, Adidas and Commerzbank. We collect the transactions in a journal.

```r
J <- readOrg(text = "
| instrument  | amount | price |
|-------------+--------+-------|
| Adidas      |     50 |   100 |
| Adidas      |    -50 |   102 |
| Commerzbank |    500 |     8 |
| Commerzbank |   -500 |     7 |
")
J <- as.journal(J)
J
```

| instrument  | amount | price |
|-------------+--------+-------|
| Adidas      |     50 |   100 |
| Adidas      |    -50 |   102 |
| Commerzbank |    500 |     8 |
| Commerzbank |   -500 |     7 |

4 transactions

We now pass the journal directly to pl.

```r
pl(J)
```

**Adidas**

- P/L total: 100
- Average buy: 100
- Average sell: 102
- Cumulative volume: 100

**Commerzbank**

- P/L total: -500
- Average buy: 8
- Average sell: 7
- Cumulative volume: 1000

P/L 'total is in units of instrument;' ',

volume is sum of /absolute/ amounts.

An aside: since the shares are denominated in the same currency (euro), total profit is the same even if we had left out the instruments; however, average buying and selling prices becomes less informative.
Financial instruments differ not only in the currencies in which they are denominated. Many derivatives have multipliers, which you may also specify. Suppose you have traded FGBL (German Bund futures) and FESX (EURO STOXX 50 futures). One point of the FGBL translates into 1000 euros; for the FESX it is 10 euros.

```r
J <- readOrg(text = "
| instrument  | amount |  price |
|-------------+--------+--------|
| FGBL MAR 16 |      1 | 165.20 |
| FGBL MAR 16 |     -1 | 165.37 |
| FGBL JUN 16 |      1 | 164.12 |
| FGBL JUN 16 |     -1 | 164.13 |
| FESX JUN 16 |      5 | 2910   |
| FESX JUN 16 |     -5 | 2905   |
")
J <- as.journal(J)
futures_pl <- pl(J,
    multiplier = c("^FGBL" = 1000, "^FESX" = 10),
    multiplier.regexp = TRUE)
futures_pl
```

FESX JUN 16
  P/L total  -250
  average buy 2910
  average sell 2905
  cum. volume 10

FGBL JUN 16
  P/L total  10
  average buy 164.12
  average sell 164.13
  cum. volume 2

FGBL MAR 16
  P/L total  170
  average buy 165.2
  average sell 165.37
  cum. volume 2

P/L 'total is in units of instrument;' volume is sum of /absolute/ amounts.

Note that we used a named vector to pass the multipliers. Per default, the names of this vector need to exactly match the instruments’ names.
Setting `multiplier.regexp` to TRUE causes the names of the multiplier vector to be interpreted as (Perl-style) regular expressions.

At this point, it may be helpful to describe how we can access the results of such P/L computations (other than having them printed to the console, that is). The function `pl` always returns a list of lists – one list for each instrument.

```r
str(futures_pl)
```

```
List of 3
$ FESX JUN 16:List of 6
 ..$ pl : num -250
 ..$ realised : logi NA
 ..$ unrealised: logi NA
 ..$ buy : num 2910
 ..$ sell : num 2905
 ..$ volume : num 10
$ FGBL JUN 16:List of 6
 ..$ pl : num 10
 ..$ realised : logi NA
 ..$ unrealised: logi NA
 ..$ buy : num 164
 ..$ sell : num 164
 ..$ volume : num 2
$ FGBL MAR 16:List of 6
 ..$ pl : num 170
 ..$ realised : logi NA
 ..$ unrealised: logi NA
 ..$ buy : num 165
 ..$ sell : num 165
 ..$ volume : num 2
- attr(*, "class")= chr "pl"
- attr(*, "along.timestamp")= logi FALSE
- attr(*, "instrument")= chr [1:3] "FESX JUN 16" "FGBL JUN 16" "FGBL MAR 16"
```

Each such list contains numeric vectors: `pl`, `realised`, `unrealised`, `buy`, `sell`, `volume`. There may also be an additional vector, `timestamp`, to be described later in Section PL over time. The vectors `realised` and `unrealised` will be NA unless `along.timestamp` is not FALSE, also described in Section PL over time. Data can be extracted by standard methods.

```r
unlist(futures_pl[["FESX JUN 16"]])
```

```
<table>
<thead>
<tr>
<th>pl</th>
<th>realised</th>
<th>unrealised</th>
<th>buy</th>
<th>sell</th>
<th>volume</th>
</tr>
</thead>
<tbody>
<tr>
<td>-250</td>
<td>NA</td>
<td>NA</td>
<td>2910</td>
<td>2905</td>
<td>10</td>
</tr>
</tbody>
</table>
```
unlist(lapply(futures_pl, `[[[`, "volume"))

<table>
<thead>
<tr>
<th></th>
<th>FESX JUN 16</th>
<th>FGBL JUN 16</th>
<th>FGBL MAR 16</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>10</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

You may prefer `sapply(...)` instead of `unlist(lapply(...))`. Also, extracting the raw P/L numbers of each instrument is so common that you can say `pl(pl(...))`. So you could have written:

```r
pl(pl(J, multiplier = c("FGBL" = 1000, "FESX" = 10), multiplier.regexp = TRUE))
```

<table>
<thead>
<tr>
<th></th>
<th>FESX JUN 16</th>
<th>FGBL JUN 16</th>
<th>FGBL MAR 16</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>-250</td>
<td>10</td>
<td>170</td>
</tr>
</tbody>
</table>

It is often more convenient to have the data presented as a table, which we can create with `as.data.frame`.

```r
as.data.frame(futures_pl)
```

<table>
<thead>
<tr>
<th></th>
<th>pl</th>
<th>buy</th>
<th>sell</th>
<th>volume</th>
</tr>
</thead>
<tbody>
<tr>
<td>FESX JUN 16</td>
<td>-250</td>
<td>2910.00</td>
<td>2905.00</td>
<td>10</td>
</tr>
<tr>
<td>FGBL JUN 16</td>
<td>10</td>
<td>164.12</td>
<td>164.13</td>
<td>2</td>
</tr>
<tr>
<td>FGBL MAR 16</td>
<td>170</td>
<td>165.20</td>
<td>165.37</td>
<td>2</td>
</tr>
</tbody>
</table>

Or if you like ASCII tables, with `toOrg`.

```r
toOrg(as.data.frame(futures_pl), row.names = "instrument")
```

<table>
<thead>
<tr>
<th></th>
<th>instrument</th>
<th>pl</th>
<th>buy</th>
<th>sell</th>
<th>volume</th>
</tr>
</thead>
<tbody>
<tr>
<td>FESX JUN 16</td>
<td>-250</td>
<td>2910</td>
<td>2905</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>FGBL JUN 16</td>
<td>10</td>
<td>164.12</td>
<td>164.13</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>FGBL MAR 16</td>
<td>170</td>
<td>165.2</td>
<td>165.37</td>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>

We can also use `pl` when there are open positions. The simplest example is a journal of just one trade.

```r
pl(amount = 1, price = 100)
```

P/L total NA
average buy 100
average sell NaN
cum. volume 1
P/L ‘total is in units of instrument;' ‘volume is sum of /absolute/ amounts.’
sum(amount’) is not zero: specify ‘vprice to compute P/L.

There can be no P/L number since the position is not closed. But the message that is shown tells us what to do: we need to specify a price at which the open position is to be valued. This valuation price is passed as argument vprice (v as in valuation).

```r
pl(amount = 1, price = 100, vprice = 105)
```

<table>
<thead>
<tr>
<th>P/L total</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>average buy</td>
<td>100</td>
</tr>
<tr>
<td>average sell</td>
<td>105</td>
</tr>
<tr>
<td>cum. volume</td>
<td>1</td>
</tr>
</tbody>
</table>

‘P/L total’ is in units of instrument; ‘volume’ is sum of /absolute/ amounts. average sell includes ‘vprice’

Note that average sell takes into account the valuation price that we specified. But cum. volume has remained 1 since only 1 unit was actually traded.

A common task is to compute P/L over a specified period of time such as one trading day. The procedure for such a case requires three ingredients:

1. the initial position and its valuation prices,
2. the trades during the period,
3. the final position and its prices.

Suppose yesterday, at market close, we had the following positions.

```r
yesterday_position <- c("FESX JUN 16" = -20, "FGBL JUN 16" = 10)
yesterday_prices <- c("FESX JUN 16" = 2912, "FGBL JUN 16" = 164.23)
```

In an earlier version of the code, average buy/sell price did not take into account vprice. But specifying vprice creates a fiction, anyway (P/L ‘as if’ the trade was closed), and this fiction is more informative if the average buy/sell price is adjusted.
Chapter 3. Simple cases

Note that, as with the multipliers above, we use named vectors for both the position and the prices: the names indicate the instruments.

Trading just ended, and we have done the following trades.

<table>
<thead>
<tr>
<th>instrument</th>
<th>amount</th>
<th>price</th>
</tr>
</thead>
<tbody>
<tr>
<td>FGBL MAR 16</td>
<td>1</td>
<td>165.20</td>
</tr>
<tr>
<td>FGBL MAR 16</td>
<td>-1</td>
<td>165.37</td>
</tr>
<tr>
<td>FGBL JUN 16</td>
<td>1</td>
<td>164.12</td>
</tr>
<tr>
<td>FGBL JUN 16</td>
<td>-1</td>
<td>164.13</td>
</tr>
<tr>
<td>FESX JUN 16</td>
<td>5</td>
<td>2910.00</td>
</tr>
<tr>
<td>FESX JUN 16</td>
<td>-5</td>
<td>2905.00</td>
</tr>
</tbody>
</table>

6 transactions

Now we pass the three ingredients – initial position, trades during the period, and valuation prices for the final, open positions – to pl.

```r
pl(J, 
  initial.position = yesterday_position, 
  initial.price = yesterday_prices, 
  vprice = c("FESX JUN 16" = 2902, "FGBL JUN 16" = 164.60), 
  multiplier = c("FGBL" = 1000, "FESX" = 10), 
  multiplier.regexp = TRUE)
```

FESX JUN 16
- P/L total 1750
- average buy 2903.6
- average sell 2910.6
- cum. volume 10

FGBL JUN 16
- P/L total 3710
- average buy 164.22
- average sell 164.56
- cum. volume 2

FGBL MAR 16
- P/L total 170
- average buy 165.2
- average sell 165.37
- cum. volume 2

P/L 'total is in units of instrument;'
volume is sum of /absolute/ amounts.
for FESX JUN 16: average buy includes ‘slopeprice
for FGBL JUN 16: average sell includes ‘slopeprice

An aside: we could have simulated this computation by creating one journal of the initial position and another journal (with reversed amount signs) for the final position, merging all three journals and then computing P/L.

3.1.2 P/L over time

In the examples above, we computed total P/L. It is also illuminating to see how P/L evolved over time. Suppose that a stock trader bought one share at 50, one share at 90 and sold two shares at 100. These trades resulted in a profit of 60, or an average return of more than +40% (bought at an average price of 70, and sold at 100).

J <- journal(price = c(90, 50, 100),
              amount = c(1, 1, -2))
pl(J)

P/L total 60
average buy 70
average sell 100
cum. volume 4

'P/L total' is in units of instrument;
'volume' is sum of /absolute/ amounts.

That may appear like some pretty good trading. Yet suppose that the order of the trades was

buy at 90 => buy at 50 => sell at 100.

You may have noticed that the journal that we created above already has the trades ordered this way. We may not know what was traded and when, but there is clearly some information in the order of the trades and the drawdown that it implies: namely a mark-to-market loss of at least 40 before it recovered. For situations like this, the argument along.timestamp can be used.

pl(J, along.timestamp = TRUE)
Chapter 3. Simple cases

<table>
<thead>
<tr>
<th>timestamp</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>P/L total</td>
<td>0</td>
<td>-40</td>
<td>60</td>
</tr>
<tr>
<td>__ realised</td>
<td>0</td>
<td>0</td>
<td>60</td>
</tr>
<tr>
<td>__ unrealised</td>
<td>0</td>
<td>-40</td>
<td>0</td>
</tr>
<tr>
<td>average buy</td>
<td>70</td>
<td></td>
<td></td>
</tr>
<tr>
<td>average sell</td>
<td>100</td>
<td></td>
<td></td>
</tr>
<tr>
<td>cum. volume</td>
<td>1</td>
<td>2</td>
<td>4</td>
</tr>
</tbody>
</table>

'P/L total' is in units of instrument; 'volume' is sum of /absolute/ amounts.

Note that we do not provide an actual timestamp, in which case the function uses integers 1, 2, ..., \( \text{length}(J) \). With no further arguments, as here, the function computes the running position and evaluates it at every trade with the trade’s price. This may not be totally accurate because of bid–ask spreads or other transaction costs. But it provides more information than only computing the aggregate P/L for the trades.

\[
\text{str}(\text{pl}(J, \text{along.timestamp} = \text{TRUE}))
\]

List of 1

$ : List of 7

..$ timestamp : int [1:3] 1 2 3
..$ pl : num [1:3] 0 -40 60
..$ realised : num [1:3] 0 0 60
..$ unrealised: num [1:3] 0 -40 0
..$ buy : num 70
..$ sell : num 100
..$ volume : num [1:3] 1 2 4
- attr(*, "class")= chr "pl"
- attr(*, "along.timestamp")= logi TRUE
- attr(*, "instrument")= logi NA

As described above, \( \text{pl} \) object is a list of lists: one list for each instrument. But now the components \( \text{pl}, \text{realised} \) and so on actually hold time-series, with the times given by \( \text{timestamp} \).

In the previous section, we used \( \text{vprice} \) to value a final open position. We can also use it to value a position over time. See Profit/Loss for Open Positions, and this example on quant.stackexchange.com.
3.2 More-complicated cases

Unfortunately, in real life computing P/L is often more complicated:

• One asset-price unit may not translate into one currency unit: there may be multipliers a.k.a. contract factors; there are even instruments with variable multipliers, e.g. Australian government-bond futures.

• Asset positions may map into cashflows in non-obvious ways. The simple case is the delay in actual payment and delivery of an asset, which is often two or three days. The more problematic cases are derivatives with daily adjustments of margins. In such cases, one may need to model (i.e. keep track of) the actual account balances.

• Assets may be denominated in various currencies.

• Currencies themselves may be assets in the portfolio. Depending on how they are traded (cash, forwards, &c.), computing P/L may not be straightforward.

How – or, rather, to what degree – these complications are handled is, as always, up to the user. For a single instrument, computing P/L in units of the instrument is usually meaningful, though perhaps not always intuitive. But adding up the profits and losses of several assets will in general not work because of multipliers or different currencies. A simple and transparent way is then to manipulate the journal before P/L is computed (e.g., multiply notionals by their multipliers).
Chapter 3. More-complicated cases
4 Computing returns

4.1 Simple returns

The function \texttt{return} computes returns from prices. The function computes what are called simple or discrete returns:\footnote{The function never computes logarithmic returns.} let \( P_t \) be the price at point in time \( t \), then

\[
    r_t = R_t - 1 = \frac{P_t}{P_{t-1}} - 1 = \frac{P_t - P_{t-1}}{P_{t-1}}. \tag{4.1}
\]

For computing profit/loss in currency units, see Section Computing profit and (or) loss.

Typically, we transform a whole series \( P_{t_1}, P_{t_2}, P_{t_3}, \ldots \) into returns \( R_{t_2}, R_{t_3}, \ldots \), which is a one-liner in R:

\begin{verbatim}
    simple_returns <- function(x)
        x[-1L]/x[-length(x)] - 1
\end{verbatim}

(You may argue that these are two lines: yet even a one-liner, if used repeatedly, should be written as a function.)

Let us try it. \texttt{pmwr} comes with two small datasets, DAX and REXP. DAX stands for \textit{Deutscher Aktienindex} (German Equity Index), and REXP stands for \textit{Rentenindex} (Performance). Both datasets are dataframes of one column that contains the price for the day, with the timestamps stored as rownames in format YYYY-MM-DD.

\begin{verbatim}
    head(DAX)
\end{verbatim}
Chapter 4. Simple returns

We extract the prices for the first five business days of 2014 and put them into a vector $P$.

```r
P <- head(DAX[[1]], n = 5)
P
```

```
[1] 9400.04 9435.15 9428.00 9506.20 9497.84
```

Now we call `simple_returns`.

```r
simple_returns(P)
```

```
[1] 0.00373509 -0.00075780 0.00829444 -0.00087943
```

In fact, using `returns` as provided by `pmwr` would have given the same result.

```r
returns(P)
```

```
[1] 0.00373509 -0.00075780 0.00829444 -0.00087943
```

`pmwr`'s `returns` function offers more convenience than `simple_returns`. For instance, it will recognise when the input argument has several columns, such as a matrix or a dataframe. In such a case, it computes returns for each column.²

```r
returns(cbind(P, P))
```

```
P P
[1,] 0.003735 0.003735
[2,] -0.000758 -0.000758
[3,] 0.008294 0.008294
[4,] -0.000879 -0.000879
```

²See Vectorisation.
The argument pad determines how the initial observation is handled. The default, NULL, means that the first observation is dropped. It is often useful to use NA instead, since in this way the returns series keeps the same length as the original price series.

```r
data.frame(price = P, returns = returns(P, pad = NA))
```

<table>
<thead>
<tr>
<th>price</th>
<th>returns</th>
</tr>
</thead>
<tbody>
<tr>
<td>9400.0</td>
<td>NA</td>
</tr>
<tr>
<td>9435.1</td>
<td>0.00373509</td>
</tr>
<tr>
<td>9428.0</td>
<td>-0.00075780</td>
</tr>
<tr>
<td>9506.2</td>
<td>0.00829444</td>
</tr>
<tr>
<td>9497.8</td>
<td>-0.00087943</td>
</tr>
</tbody>
</table>

Setting pad to 0 can also be useful, because then it is easy to 'rebuild' the original series with `cumprod`. (But see Section Scaling series for a description of the function `scale1`, which is even more convenient.)

```r
P[1] * cumprod(1 + returns(P, pad = 0))
```

| 1 9400.04 9435.15 9428.00 9506.20 9497.84 |

`returns` also has an argument `lag`, with default 1. This can be used to compute rolling returns, such as 30-day returns, etc.

`returns` is a generic function, which goes along with some overhead. If you need to compute returns on simple data structures as in the examples above and need fast computation, then you may also use `.returns`. The function is the actual workhorse that performs the raw return calculation.

Besides having methods for numeric vectors (which includes those with a dim attribute, i.e. matrices) and dataframes, `returns` also understands `zoo` objects. So let us create two `zoo` series, `dax` and `rex`.

```r
library("zoo")
dax <- zoo(DAX[[1]], as.Date(row.names(DAX)))
rex <- zoo(REXP[[1]], as.Date(row.names(REXP)))
str(dax)
```

'zoo' series from 2014-01-02 to 2015-12-30
Data: num [1:505] 9400 9435 9428 9506 9498 ...
Index: Date[1:505], format: "2014-01-02" "2014-01-03" ...
Chapter 4. Holding-period returns

str(rex)

'zoo' series from 2014-01-02 to 2015-12-30
  Data: num [1:502] 441 441 442 442 442 ...
  Index: Date[1:502], format: "2014-01-02" "2014-01-03" ...

returns(head(dax, 5), pad = NA)

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>NA</td>
<td>0.003735</td>
<td>-0.000758</td>
<td>0.008294</td>
<td>-0.000879</td>
</tr>
</tbody>
</table>

Matrices work as well. We combine both series into a two-column matrix drax.³

```
drax <- cbind(dax, rex)
returns(head(drax, 5))
```

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>dax</td>
<td>rex</td>
</tr>
<tr>
<td>2014-01-03</td>
<td>0.003735</td>
</tr>
<tr>
<td>2014-01-06</td>
<td>-0.000758</td>
</tr>
<tr>
<td>2014-01-07</td>
<td>0.008294</td>
</tr>
<tr>
<td>2014-01-08</td>
<td>-0.000879</td>
</tr>
</tbody>
</table>

As you see, just as for a numeric matrix, the function computes the returns for each column.

In fact, zoo objects bring another piece of information – timestamps – that returns can use. (Since xts series inherit from zoo, they will work as well.)

### 4.2 Holding-period returns

When a timestamp is available, returns can compute returns for specific calendar periods. As an example, we use the daily DAX levels in 2014 and 2015 and compute monthly returns from them.

```
returns(dax, period = "month")
```

³In case you did not know: drax is not only the name of a dataset in this book, but also the name of Marvel-comics superhero and of the villain of a James Bond novel. The latter is actually German, which makes it obvious to choose the name for representing German indices. :-)}
If you prefer to not use zoo or xts, you may also pass the timestamp explicitly to returns.

```r
returns(coreda(dax), t = index(dax), period = "month")
```

Despite the way these monthly returns are printed: the result of the function call is a numeric vector (the return numbers), with additional information added through attributes. There is also a class attribute, which has value `p_returns`. The advantage of this data structure is that it is ‘natural’ to compute with the returns, e.g. to calculate means, extremes and other quantities.

```r
range(returns(dax, period = "month"))
```

Most useful, however, is probably the print method, whose results you have seen above. You may also compute monthly returns for matrices, i.e. for more than one asset. But now the print method will behave differently. The function’s assumption is that now it would be more convenient to print the returns aligned by date in a table.

```r
returns(drax, period = "month")
```
Chapter 4. Holding-period returns

2015-01-30  9.1  0.3
2015-02-27  6.6  0.1
2015-03-31  5.0  0.3
2015-04-30  -4.3 -0.5
2015-05-29  -0.4 -0.2
2015-06-30  -4.1 -0.8
2015-07-31  3.3  0.7
2015-08-31  -9.3  0.0
2015-09-30  -5.8  0.4
2015-10-30  12.3  0.4
2015-11-30  4.9  0.3
2015-12-30  -5.6 -0.6

If you rather wanted the other, one-row-per-year display, just call the function separately for each series.

\[
\text{lapply(list(DAX = dax, REXP = rex),}
\text{returns, period = "month")}
\]

\[
\begin{array}{cccccccccccccc}
\text{Jan} & \text{Feb} & \text{Mar} & \text{Apr} & \text{May} & \text{Jun} & \text{Jul} & \text{Aug} & \text{Sep} & \text{Oct} & \text{Nov} & \text{Dec} & \text{YTD} \\
2014 & -1.0 & 4.1 & -1.4 & 0.5 & 3.5 & -1.1 & -4.3 & 0.7 & 0.0 & -1.6 & 7.0 & -1.8 & 4.3 \\
2015 & 9.1 & 6.6 & 5.0 & -4.3 & -0.4 & -4.1 & 3.3 & -9.3 & -5.8 & 12.3 & 4.9 & -5.6 & 9.6 \\
\end{array}
\]

\[
\begin{array}{cccccccccccccc}
\text{Jan} & \text{Feb} & \text{Mar} & \text{Apr} & \text{May} & \text{Jun} & \text{Jul} & \text{Aug} & \text{Sep} & \text{Oct} & \text{Nov} & \text{Dec} & \text{YTD} \\
2014 & 1.8 & 0.4 & 0.1 & 0.3 & 0.9 & 0.4 & 0.4 & 1.0 & -0.1 & 0.1 & 0.4 & 1.0 & 7.1 \\
2015 & 0.3 & 0.1 & 0.3 & -0.5 & -0.2 & -0.8 & 0.7 & 0.0 & 0.4 & 0.4 & 0.3 & -0.6 & 0.5 \\
\end{array}
\]

See ?print.p_returns for more display options. For instance:

\[
\text{print(returns(dax, period = "month"),}
\text{digits = 2, year.rows = FALSE, plus = TRUE,}
\text{month.names = 1:12)}
\]

\[
\begin{array}{cccc}
\text{2014} & \text{2015} \\
1 & -1.00 & +9.06 \\
2 & +4.14 & +6.61 \\
3 & -1.40 & +4.95 \\
4 & +0.50 & -4.28 \\
5 & +3.54 & -0.35 \\
6 & -1.11 & -4.11 \\
7 & -4.33 & +3.33 \\
8 & +0.67 & -9.28 \\
9 & +0.04 & -5.84 \\
10 & -1.56 & +12.32 \\
\end{array}
\]
There are methods `toLatex` and `toHTML` for monthly returns. In Sweave documents, you need to use `results = tex` and `echo = false` in the `Sweave` chunk options:

\begin{tabular}{rrrrrrrrrrrrrr}
<<results=tex,echo=false>>= 
\begin{tabular}{l}
toLatex(returns(dax, period = "month")) 
\end{tabular}
\end{tabular}

(There is also a vignette that gives examples for `toLatex`; say `vignette("FinTeX", package = "PMwR")` to open it.)

To get annualised returns, use period `ann` (or actually any string matched by the regular expression `^ann`; case is ignored).

```r
returns(dax, period = "ann")
```

6.9% [02 Jan 2014 -- 30 Dec 2015]

Now let us try a shorter period.

```r
returns(window(dax, end = as.Date("2014-1-31")), period = "ann")
```

-1.0% [02 Jan 2014 -- 31 Jan 2014; less than one year, not annualised]

The function did not annualise: it refuses to do so if the time period is shorter than one year. (You may verify the return for January 2014 in the tables above.) To force annualising, add a !. The exclamation mark serves as a mnemonic that it is now imperative to annualise.

```r
returns(window(dax, end = as.Date("2014-1-31")), period = "ann!")
```

-11.8% [02 Jan 2014 -- 31 Jan 2014; less than one year, but annualised]

There are several more accepted values for `period`, such as `year`, `quarter`, `month-to-date (mtd)`, `year-to-date (ytd)` or `inception-to-date (total)`. 

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>+7.01</td>
<td>+4.90</td>
</tr>
<tr>
<td>12</td>
<td>-1.76</td>
<td>-5.62</td>
</tr>
<tr>
<td>YTD</td>
<td>+4.31</td>
<td>+9.56</td>
</tr>
</tbody>
</table>
Chapter 4. Portfolio returns

The help page of \texttt{returns} lists all options. Note that any such setting for \texttt{period} requires that the timestamp can be coerced to \texttt{Date}; for instance, intraday time-series with \texttt{POSIXct} timestamps would work as well.

4.3 Portfolio returns

Sometimes we may need to compute returns for a portfolio of fixed weights, given an assumption when the portfolio is rebalanced. For instance, we may want to see how a constant allocation of 10\%, 50\% and 40\% to three funds would have done, assuming that a portfolio is rebalanced once a month. If more detail is necessary, then function \texttt{btest} can be used; see Chapter Backtesting. But the simple case can be done with \texttt{returns} already. Here is an example.

```r
prices <- c(100, 102, 104, 104, 104.5, 2, 2.2, 2.4, 2.3, 2.5, 3.5, 3, 3.1, 3.2, 3.1)
dim(prices) <- c(5, 3)
prices

[,1] [,2] [,3]
[1,] 100.0  2.0  3.5
[2,] 102.0  2.2  3.0
[3,] 104.0  2.4  3.1
[4,] 104.0  2.3  3.2
[5,] 104.5  2.5  3.1
```

Now suppose we want a constant weight vector, \([0.1, 0.5, 0.4]'\), but only rebalance at times 1 and 4. (That is, we rebalance the portfolio only with the prices at timestamps 1 and 4.)

```r
returns(prices, 
weights = c(10, 50, 40)/100,
rebalance.when = c(1, 4))

[1] -0.0051429 0.0637565 -0.0128240 0.0314590
attr(,"holdings")
[,1] [,2] [,3]
[1,] 0.00100000 0.25000 0.11429
[2,] 0.00100000 0.25000 0.11429
```
The result also contains, as attributes, the imputed holdings and the single period contributions.

Argument `weights` does not have to be a vector. It can also be a matrix. In such a case, each row is interpreted as a portfolio. Instead of weights, we could also specify fixed positions. See `?returns` for different possibilities to call `returns`.

### 4.4 Return contribution

Let \( w(t, i) \) be the weight of portfolio segment \( i \) at the beginning of period \( t \), and let \( r(t, i) \) be the return of segment \( i \) over period \( t \). Then the portfolio return over period \( t \), \( r_p(t) \), is a weighted sum of the \( N \) segment returns.

\[
    r_p(t) = \sum_{i=1}^{N} r(t, i) w(t, i) .
\]

(4.2)

When the weights sum to unity, we may also write

\[
    1 + r_p(t) = \sum_{i=1}^{N} (1 + r(t, i)) w(t, i)
\]

(4.3)

or, defining \( 1 + r \equiv R \),

\[
    R_p(t) = \sum_{i=1}^{N} R(t, i) w(t, i) .
\]

(4.4)

The total return contribution of segment \( i \) over time equals

\[
    \sum_{i=1}^{T-1} \left( R(t, i) w(t, i) \prod_{s=i+1}^{T} R_p(s) - 1 \right) + r(T, i) w(T, i) .
\]

(4.5)
Chapter 4. External Cashflows

In this way, a segment's return contribution in one period is reinvested in the overall portfolio in succeeding periods.

The calculation is provided in the function `rc` ('return contribution').

```r
weights <- rbind(c( 0.25, 0.75), ## the assets' weights
c( 0.40, 0.60), ## during three periods
c( 0.25, 0.75))

R <- rbind(c( 1 , 0), ## the assets' returns
    c( 2.5, -1.0), ## during these periods
    c(-2 , 0.5))/100

rc(R, weights, segment = c("equities", "bonds"))
```

```
$period_contributions
   timestamp equities  bonds  total
      1        0.0025 0.00000  0.00250
      2        0.0100-0.00600  0.00400
      3       -0.0050 0.00375 -0.00125

$total_contributions
  equities  bonds  total
   0.00749 -0.00224  0.00525
```

4.5 Returns when there are external cashflows

External cashflows (or transfers of positions) can be handled just like dividends. The following table shows the values and cashflows of a hypothetical portfolio.

<table>
<thead>
<tr>
<th>timestamp</th>
<th>value</th>
<th>cashflow</th>
<th>value with cashflow</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>+100</td>
<td>100</td>
</tr>
<tr>
<td>2</td>
<td>101</td>
<td>0</td>
<td>101</td>
</tr>
<tr>
<td>3</td>
<td>104</td>
<td>0</td>
<td>104</td>
</tr>
<tr>
<td>4</td>
<td>103</td>
<td>+100</td>
<td>203</td>
</tr>
<tr>
<td>5</td>
<td>204</td>
<td>-200</td>
<td>4</td>
</tr>
</tbody>
</table>

A total-return series, based on column value with cashflow but excluding column cashflow, can be computed with `div_adjust`. 

```r
```
cf <- c(100, 100, -200)
t <- c(1, 4, 5)
x <- c(100, 101, 104, 203, 4)
div_adjust(x, t, div = -cf, backward = FALSE)

[1] 100.000 101.000 104.000 103.000 103.507

More conveniently, the function `unit_prices` helps to compute so-called time-weighted returns of a portfolio when there are in- and outflows. (The term time-weighted returns is actually a misnomer, as returns are not weighted at all. They are only time-weighted if time-periods are of equal length.) We repeat the previous example.

NAV <- data.frame(timestamp = 1:5,
               NAV = x)

cf <- data.frame(timestamp = t,
              cashflow = cf)

unit_prices(NAV, cf)

<table>
<thead>
<tr>
<th>timestamp</th>
<th>NAV</th>
<th>price</th>
<th>units</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>100.000</td>
<td>1.0000000</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>101.000</td>
<td>1.0000000</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>104.000</td>
<td>1.0000000</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>203.000</td>
<td>1.9708738</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>103.507</td>
<td>0.0386446</td>
</tr>
</tbody>
</table>

The function returns a dataframe: to compute returns, use the `price` column.
Chapter 4. External Cashflows
5 Backtesting

This chapter explains how to test trading strategies with the `btest` function. A recent tutorial is available from SSRN.

5.1 Decisions

At a given instant in time (in actual life, ‘now’), a trader needs to answer the following questions:

1. Do I want to compute a new target portfolio, yes or no? If yes, go ahead and compute the new target portfolio.

2. Given the target portfolio and the actual portfolio, do I want to rebalance (i.e. close the gap between the actual portfolio and the target portfolio)? If yes, rebalance.

If such a decision is not just hypothetical, then the answer to the second question may lead to a number of orders sent to a broker. Note that many traders do not think in terms of stock (i.e. balances) as we did here; rather, they think in terms of flow (i.e. orders). Both approaches are equivalent, but the described one makes it easier to handle missed trades and synchronise accounts.

During a backtest, we will simulate the decisions of the trader. How precisely we simulate depends on the trading strategy. The `btest` function is meant as a helper function to simulate these decisions. The logic for the decisions described above must be coded in the functions `do.signal`, `signal` and `do.rebalance`. 
Implementing \texttt{btest} required a number of decision too: (i) what to model (i.e., how to simulate the trader), and (ii) how to code it. As an example for point (i): how precisely do we want to model the order process (e.g., use limit orders?, allow partial fills?) Example for (ii): the backbone of \texttt{btest} is a loop that runs through the data. Loops are slow in R when compared with compiled languages,\footnote{Unqualified statements such as ‘R is slow’ or ‘loops are slow’ are not useful. In the current version of \texttt{pmwR}, the overhead of running a backtest with \texttt{btest} with 500 assets and 5000 observations (and rebalancing each of those assets at each of those 5000 timestamps) is less than five seconds; if only few assets are touched, overhead is less than one second.} so should we vectorise instead? Vectorisation is indeed often possible, namely if trading is not path-dependent. If we have already a list of trades, we can efficiently transform them into a profit-and-loss in R without relying on an explicit loop (see Section Computing profit and (or) loss). Yet, one advantage of looping is that the trade logic is more similar to actual trading; we may even be able to reuse some code in live trading.

Altogether, the aim for \texttt{btest} is to stick to the functional paradigm as much as possible. Functions receive arguments and evaluate to results; but they do not change their arguments, nor do they assign or change other variables ‘outside’ their environment, nor do the results depend on some variable outside the function. This creates a problem, namely how to keep track of state. If we know what variables need to be persistent, we could pass them to the function and always have them returned. But we would like to be more flexible, so we can pass an environment; examples are below. To make that clear: functional programming should not be seen as a yes-or-no decision; it is a matter of degree. And more of the functional approach can help already.

### 5.2 Data structure

All computations of \texttt{btest} will be based on one or several price series of length $T$. Internally, these prices are stored in numeric matrices.

Prices are passed as argument \texttt{prices}. For a single asset, this must be a matrix of prices with four columns: open, high, low and close.

For $n$ assets, you need to pass a list of length four: \texttt{prices[[1]]} must be a matrix with $n$ columns containing the open prices for the assets;
prices[[2]] is a matrix with the high prices, and so on. For instance, with two assets, you need four matrices with two columns each:

<table>
<thead>
<tr>
<th>open</th>
<th>high</th>
<th>low</th>
<th>close</th>
</tr>
</thead>
<tbody>
<tr>
<td>+---+</td>
<td>+---+</td>
<td>+---+</td>
<td>+---+</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

If only close prices are used, then for a single asset, use either a matrix of one column or a numeric vector. For multiple assets a list of length one must be passed, containing a matrix of close prices. For example, with 100 close prices of 5 assets, the prices should be arranged in a matrix \( p \) of size 100 times 5; and \( \text{prices} = \text{list}(p) \).

The \( \text{btest} \) function runs from \( b+1 \) to \( T \). The variable \( b \) is the burn-in and it needs to be a positive integer. When we take decisions that are based on past data, we will lose at least one data point. In rare cases \( b \) may be zero.

Here is an important default: at time \( t= \), we can use information up to time \( t-1 \). Suppose that \( t \) were 4. We may use all information up to time 3, and trade at the open in period 4:

<table>
<thead>
<tr>
<th>t</th>
<th>time</th>
<th>open</th>
<th>high</th>
<th>low</th>
<th>close</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>HH:MM:SS</td>
<td></td>
<td>⌊</td>
<td></td>
<td>⌊</td>
</tr>
<tr>
<td>2</td>
<td>HH:MM:SS</td>
<td></td>
<td>⌊</td>
<td></td>
<td>⌊</td>
</tr>
<tr>
<td>3</td>
<td>HH:MM:SS</td>
<td></td>
<td>⌊</td>
<td></td>
<td>⌊</td>
</tr>
<tr>
<td>4</td>
<td>HH:MM:SS</td>
<td></td>
<td>⌊</td>
<td></td>
<td>⌊</td>
</tr>
<tr>
<td>5</td>
<td>HH:MM:SS</td>
<td></td>
<td>⌊</td>
<td></td>
<td>⌊</td>
</tr>
</tbody>
</table>

We could also trade at the close:

<table>
<thead>
<tr>
<th>t</th>
<th>time</th>
<th>open</th>
<th>high</th>
<th>low</th>
<th>close</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>HH:MM:SS</td>
<td></td>
<td>⌊</td>
<td></td>
<td>⌊</td>
</tr>
<tr>
<td>2</td>
<td>HH:MM:SS</td>
<td></td>
<td>⌊</td>
<td></td>
<td>⌊</td>
</tr>
<tr>
<td>3</td>
<td>HH:MM:SS</td>
<td></td>
<td>⌊</td>
<td></td>
<td>⌊</td>
</tr>
<tr>
<td>4</td>
<td>HH:MM:SS</td>
<td></td>
<td>⌊</td>
<td></td>
<td>⌊</td>
</tr>
<tr>
<td>5</td>
<td>HH:MM:SS</td>
<td></td>
<td>⌊</td>
<td></td>
<td>⌊</td>
</tr>
</tbody>
</table>

No, we cannot trade at the high or low. (Some people like the idea, as a robustness check, to always buy at the high, sell at the low. Robustness
checks – forcing a bit of bad luck into the simulation – are a good idea, notably bad executions. High/low ranges can inform such checks, but using these ranges does not go far enough, and is more of a good story than a meaningful test.)

5.3 Function arguments

5.3.1 Available information within functions

`btest` expects as arguments a number of functions, such as `signal`; see the next section for a complete list. The default is to specify no arguments to these functions, because they can all access the following ‘objects’. These objects actually are, with the exception of `Globals`, themselves functions that can access certain data. These functions can only read; there are no replacement functions. The exception is `Globals`, which is an environment, and which can explicitly be used for writing (i.e. storing data).

- **Open** open prices
- **High** high prices
- **Low** low prices
- **Close** close prices
- **Wealth** the total wealth (cash plus positions) at a given point in time
- **Cash** cash (in accounting currency)
- **Time** current time (an integer)
- **Timestamp** the timestamp when that is specified (i.e. when the argument `timestamp` is supplied); if not, it defaults to `Time`
- **Portfolio** the current portfolio
- **SuggestedPortfolio** the currently-suggested portfolio
- **Globals** an environment (not a function)
All functions take as their first argument a `lag`, which defaults to 1. So to get the most recent close price, say

```r
Close()
```

which is the same as `Close(lag = 1)`. 

The `lag` can be a vector, too: the expression

```r
Close(Time():1)
```

for instance will return all available close prices. So in period 11, say, you want close prices for lags 10, 9, ..., 1. Hence, to receive prices in their correct order, the lag sequence must always be in reverse order.

If you find it awkward to specify the lag in this reverse order, you may use the argument `n` instead, which specifies to retrieve the last `n` data points. So the above `Close(Time():1)` is equivalent to

```r
Close(n = Time())
```

and saying

```r
Close(n = 10)
```

will get you the last ten closing prices in their actual temporal order.

### 5.3.2 Function arguments

**signal** The function `signal` uses information until and including `t-1` and returns the suggested portfolio (a vector) to be held at `t`. This position should be in units of the instruments; if you prefer to work with weights, then you should set `convert.weights` to `TRUE`. Then, the value returned by `signal` will be interpreted as weights and will be automatically converted to position sizes.

**do.signal** `do.signal` uses information until and including `t-1` and must return `TRUE` or `FALSE` to indicate whether a signal (i.e. new suggested position) should be computed. This is useful when the signal computation is costly and only be done at specific points in time. If the function is not specified, it defaults to `function()` `TRUE`. Instead of a function, this may also be
56

Chapter 5. Examples: A single asset

- a vector of integers, which then indicate the points in time when to compute a position; or
- a vector of logical values, which then indicate the points in time when to compute a position; or
- a vector that inherits from the class of timestamp (e.g. Date); or
- one of the keywords firstofmonth, lastofmonth, firstofuqarter or lastofmonth. In this case, timestamp must inherit from Date or be coercible to Date. (Options can easily be specified with function nth_day in package datetimeutils.)

`do.rebalance` just like `do.signal`, but refers to the actual trading. If the function is not specified, it defaults to `function() TRUE`. Note that rebalancing can typically not take place at a higher frequency than implied by `signal`. That is because calling `signal` leads to a position, and when this position does not change (i.e. `signal` was not called), there is actually no need to rebalance. So `do.rebalance` is normally used when rebalancing should be done less often than signal computation, e.g. when the decision whether to trade or not is conditional on something.

`print.info` The function is called at the end of an iteration. Whatever it returns will be ignored since it is called for its side effect: print information to the screen, into a file or into some other connection.

`cashflow` The function is called at the end of each iteration; its value is added to the `cash`. The function provides a clean way to, for instance, add accrued interest to or subtract fees from a strategy.

### 5.4 Examples: A single asset

It is best to describe the `btest` function through a number of simple examples.
5.4.1 A useless first example

I really like simple examples. Suppose we have a single instrument, and we use only close prices. The trading rule is to buy, and then to hold forever. All we need is the time series of the prices and the signal function. As an instrument we use the EURO STOXX 50 future with expiry September 2015.

```r
timestamp <- structure(c(16679L, 16680L, 16681L, 16682L, 16685L, 16686L, 16687L, 16688L, 16689L, 16692L, 16693L), class = "Date")
prices <- c(3182, 3205, 3272, 3185, 3201, 3236, 3272, 3224, 3194, 3188, 3213)
data.frame(timestamp, prices)
```

<table>
<thead>
<tr>
<th>timestamp</th>
<th>prices</th>
</tr>
</thead>
<tbody>
<tr>
<td>2015-09-01</td>
<td>3182</td>
</tr>
<tr>
<td>2015-09-02</td>
<td>3205</td>
</tr>
<tr>
<td>2015-09-03</td>
<td>3272</td>
</tr>
<tr>
<td>2015-09-04</td>
<td>3185</td>
</tr>
<tr>
<td>2015-09-07</td>
<td>3201</td>
</tr>
<tr>
<td>2015-09-08</td>
<td>3236</td>
</tr>
<tr>
<td>2015-09-09</td>
<td>3272</td>
</tr>
<tr>
<td>2015-09-10</td>
<td>3224</td>
</tr>
<tr>
<td>2015-09-11</td>
<td>3194</td>
</tr>
<tr>
<td>2015-09-14</td>
<td>3188</td>
</tr>
<tr>
<td>2015-09-15</td>
<td>3213</td>
</tr>
</tbody>
</table>

The signal function is very simple indeed.

```r
signal <- function()
1
```

signal must be written so that it returns the suggested position in units of the asset. In this first example, the suggested position always is 1 unit. It is only a suggested portfolio because we can specify rules whether or not to trade. Examples follow below.

To test this strategy, we call `btest`. The initial cash is zero per default, so initial wealth is also zero in this case. We can change it through the argument `initial.cash`.

```r
(solution <- btest(prices = prices, signal = signal))
```
Chapter 5. Examples: A single asset

initial wealth 0 => final wealth 8

The function returns a list with a number of components, but they are not printed. Instead, a simple print method displays some information about the results. In this case, it tells us that the total equity of the strategy increased from 0 to 8.

We arrange more details into a data.frame. `suggest` is the suggested position; `position` is the actual position.

```r
trade_details <- function(solution, prices) {
  data.frame(price = prices,
             suggest = solution$suggested.position,
             position = unname(solution$position),
             wealth = solution$wealth,
             cash = solution$cash)
}
```

```r
trade_details(unclass(solution), prices)
```

<table>
<thead>
<tr>
<th>price</th>
<th>suggest</th>
<th>position</th>
<th>wealth</th>
<th>cash</th>
</tr>
</thead>
<tbody>
<tr>
<td>3182</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3205</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>-3205</td>
</tr>
<tr>
<td>3272</td>
<td>1</td>
<td>1</td>
<td>67</td>
<td>-3205</td>
</tr>
<tr>
<td>3185</td>
<td>1</td>
<td>1</td>
<td>-20</td>
<td>-3205</td>
</tr>
<tr>
<td>3201</td>
<td>1</td>
<td>1</td>
<td>-4</td>
<td>-3205</td>
</tr>
<tr>
<td>3236</td>
<td>1</td>
<td>1</td>
<td>31</td>
<td>-3205</td>
</tr>
<tr>
<td>3272</td>
<td>1</td>
<td>1</td>
<td>67</td>
<td>-3205</td>
</tr>
<tr>
<td>3224</td>
<td>1</td>
<td>1</td>
<td>19</td>
<td>-3205</td>
</tr>
<tr>
<td>3194</td>
<td>1</td>
<td>1</td>
<td>-11</td>
<td>-3205</td>
</tr>
<tr>
<td>3188</td>
<td>1</td>
<td>1</td>
<td>-17</td>
<td>-3205</td>
</tr>
<tr>
<td>3213</td>
<td>1</td>
<td>1</td>
<td>8</td>
<td>-3205</td>
</tr>
</tbody>
</table>

We bought in the second period because the default setting for the burnin `b` is 1. Thus, we lose one observation. In this particular case here, we do not rely in any way on the past; hence, we set `b` to zero. With this setting, we buy at the first price and hold until the end of the data.

```r
solution <- btest(prices = prices, signal = signal, b = 0)
```

```r
trade_details(solution, prices)
```

<table>
<thead>
<tr>
<th>price</th>
<th>suggest</th>
<th>position</th>
<th>wealth</th>
<th>cash</th>
</tr>
</thead>
<tbody>
<tr>
<td>3182</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>-3182</td>
</tr>
<tr>
<td>3205</td>
<td>1</td>
<td>1</td>
<td>23</td>
<td>-3182</td>
</tr>
<tr>
<td>3272</td>
<td>1</td>
<td>1</td>
<td>90</td>
<td>-3182</td>
</tr>
</tbody>
</table>
If you prefer the trades only, i.e. not the position series, the solution also contains a journal. (See Keeping track of transactions: journals for more on journals.)

```
journal(solution)
```

```
instrument  timestamp  amount  price
1  asset 1  1  1  3182
```

1 transaction

To make the journal more informative, we can pass `timestamp` and `instrument` information when we call `btest`.

```
journal(btest(prices = prices, signal = signal, b = 0,
    timestamp = timestamp, ## defined above ,
    ## together with prices
    instrument = "FESX SEP 2015"))
```

```
instrument  timestamp  amount  price
1  FESX SEP 2015  2015-09-01  1  3182
```

1 transaction

Before we go to the next examples, a final remark, on data frequency. I have used daily data here, but any other frequency, also intraday data, is fine. `btest` will not care of what frequency your data are or whether your data are regularly spaced; it will only loop over the observations that it is given. It is your own responsibility to write `signal` (and other functions) in such a way that they encode a meaningful trade logic.

### 5.4.2 More-useful examples

Now we make our strategy slightly more selective. The trading rule is to have a position of 1 unit of the asset whenever the last observed price is
below 3200 and to have no position when the price is above 3200. The signal function could look like this.

```r
signal <- function() {
  if (Close() < 3200)
    1
  else
    0
}
```

If you like to write clever code, you may as well have written this:

```r
signal <- function()
  Close() < 3200
```

The logical value of the comparison `Close() < 3200` would be converted to either 0 or 1. But the more verbose version above is clearer.²

We call `btest` and check the results.

```r
solution <- btest(prices = prices, signal = signal)
trade_details(solution, prices)
```

<table>
<thead>
<tr>
<th></th>
<th>price</th>
<th>suggest</th>
<th>position</th>
<th>wealth</th>
<th>cash</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3182</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>3205</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>-3205</td>
</tr>
<tr>
<td>3</td>
<td>3272</td>
<td>0</td>
<td>0</td>
<td>67</td>
<td>67</td>
</tr>
<tr>
<td>4</td>
<td>3185</td>
<td>0</td>
<td>0</td>
<td>67</td>
<td>67</td>
</tr>
<tr>
<td>5</td>
<td>3201</td>
<td>1</td>
<td>1</td>
<td>67</td>
<td>-3134</td>
</tr>
<tr>
<td>6</td>
<td>3236</td>
<td>0</td>
<td>0</td>
<td>102</td>
<td>102</td>
</tr>
<tr>
<td>7</td>
<td>3272</td>
<td>0</td>
<td>0</td>
<td>102</td>
<td>102</td>
</tr>
<tr>
<td>8</td>
<td>3224</td>
<td>0</td>
<td>0</td>
<td>102</td>
<td>102</td>
</tr>
<tr>
<td>9</td>
<td>3194</td>
<td>0</td>
<td>0</td>
<td>102</td>
<td>102</td>
</tr>
<tr>
<td>10</td>
<td>3188</td>
<td>1</td>
<td>1</td>
<td>102</td>
<td>-3086</td>
</tr>
<tr>
<td>11</td>
<td>3213</td>
<td>1</td>
<td>1</td>
<td>127</td>
<td>-3086</td>
</tr>
</tbody>
</table>

(Yes, this strategy works better than the simple buy-and-hold, but I hope you agree that this is only because of luck.)

The argument `initial.position` specifies the initial position; default is no position. Suppose we had already held one unit of the asset.

²Remember what Brian Kernighan said: *Everyone knows that debugging is twice as hard as writing a program in the first place. So if you’re as clever as you can be when you write it, how will you ever debug it?*
solution <- btest(prices = prices, signal = signal, initial.position = 1)

Then the results would have looked as follows.

trade_details(solution, prices)

<table>
<thead>
<tr>
<th>price</th>
<th>suggest</th>
<th>position</th>
<th>wealth</th>
<th>cash</th>
</tr>
</thead>
<tbody>
<tr>
<td>3182</td>
<td>1</td>
<td>1</td>
<td>3182</td>
<td>0</td>
</tr>
<tr>
<td>3205</td>
<td>1</td>
<td>1</td>
<td>3205</td>
<td>0</td>
</tr>
<tr>
<td>3272</td>
<td>0</td>
<td>0</td>
<td>3272</td>
<td>3272</td>
</tr>
<tr>
<td>3185</td>
<td>0</td>
<td>0</td>
<td>3272</td>
<td>3272</td>
</tr>
<tr>
<td>3201</td>
<td>1</td>
<td>1</td>
<td>3272</td>
<td>71</td>
</tr>
<tr>
<td>3236</td>
<td>0</td>
<td>0</td>
<td>3307</td>
<td>3307</td>
</tr>
<tr>
<td>3272</td>
<td>0</td>
<td>0</td>
<td>3307</td>
<td>3307</td>
</tr>
<tr>
<td>3224</td>
<td>0</td>
<td>0</td>
<td>3307</td>
<td>3307</td>
</tr>
<tr>
<td>3194</td>
<td>0</td>
<td>0</td>
<td>3307</td>
<td>3307</td>
</tr>
<tr>
<td>3188</td>
<td>1</td>
<td>1</td>
<td>3307</td>
<td>119</td>
</tr>
<tr>
<td>3213</td>
<td>1</td>
<td>1</td>
<td>3332</td>
<td>119</td>
</tr>
</tbody>
</table>

In the example above, we use the close price, but we do not access the data directly. A function Close is defined by btest and passed as an argument to signal. Note that we do not add it as a formal argument to signal since this is done automatically. In fact, doing it manually would trigger an error message:

```r
signal <- function(Close = NULL) ## ERROR: argument name
## 'Close' not allowed
```

```r
Error in btest(prices = prices, signal = signal) :
'Close' cannot be used as an argument name for 'signal'
```

Similarly, we have functions Open, High and Low; see Section 5.3 above for all functions.

Suppose we wanted to add a variable: a threshold that tells us when to buy. This would need to be an argument to signal; it would also need to be passed with the ... argument of btest.

```r
signal <- function(threshold) {
  if (Close() < threshold)
    1
  else
    0
}
```
solution <- btest(prices = prices,  
               signal = signal,  
               threshold = 3190)

trade_details(solution, prices)

<table>
<thead>
<tr>
<th>price</th>
<th>suggest</th>
<th>position</th>
<th>wealth</th>
<th>cash</th>
</tr>
</thead>
<tbody>
<tr>
<td>3182</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3205</td>
<td>1</td>
<td>1</td>
<td>0 -3205</td>
<td></td>
</tr>
<tr>
<td>3272</td>
<td>0</td>
<td>0</td>
<td>67</td>
<td>67</td>
</tr>
<tr>
<td>3185</td>
<td>0</td>
<td>0</td>
<td>67</td>
<td>67</td>
</tr>
<tr>
<td>3201</td>
<td>1</td>
<td>1</td>
<td>67 -3134</td>
<td></td>
</tr>
<tr>
<td>3236</td>
<td>0</td>
<td>0</td>
<td>102</td>
<td>102</td>
</tr>
<tr>
<td>3272</td>
<td>0</td>
<td>0</td>
<td>102</td>
<td>102</td>
</tr>
<tr>
<td>3194</td>
<td>0</td>
<td>0</td>
<td>102</td>
<td>102</td>
</tr>
<tr>
<td>3188</td>
<td>0</td>
<td>0</td>
<td>102</td>
<td>102</td>
</tr>
<tr>
<td>3213</td>
<td>1</td>
<td>1</td>
<td>102 -3111</td>
<td></td>
</tr>
</tbody>
</table>

So far we have treated Close as a function without arguments, but actually it has an argument lag that defaults to 1. Suppose the rule were to buy if the last close is below the second-to-last close. signal could look like this.

signal <- function() {  
  if (Close(1L) < Close(2L))  
    1  
  else  
    0  
}

We could also have written (Close() < Close(2L)). In any case, the rule uses the close prices of yesterday and of the day before yesterday, so we need to increase b.

trade_details(btest(prices = prices, signal = signal, b = 2), prices)

<table>
<thead>
<tr>
<th>price</th>
<th>suggest</th>
<th>position</th>
<th>wealth</th>
<th>cash</th>
</tr>
</thead>
<tbody>
<tr>
<td>3182</td>
<td>0</td>
<td>NA</td>
<td>NA</td>
<td>0</td>
</tr>
<tr>
<td>3205</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3272</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3185</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3201</td>
<td>1</td>
<td>1</td>
<td>0 -3201</td>
<td></td>
</tr>
</tbody>
</table>
If we want to trade a different size, we have `signal` return the desired value.

```r
signal <- function()
  if (Close() < 3200)
    2 else 0

trade_details(btest(prices = prices, signal = signal), prices)
```

```
<table>
<thead>
<tr>
<th>price</th>
<th>suggest</th>
<th>position</th>
<th>wealth</th>
<th>cash</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3182</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>3205</td>
<td>2</td>
<td>2</td>
<td>-6410</td>
</tr>
<tr>
<td>3</td>
<td>3272</td>
<td>0</td>
<td>0</td>
<td>134</td>
</tr>
<tr>
<td>4</td>
<td>3185</td>
<td>0</td>
<td>0</td>
<td>134</td>
</tr>
<tr>
<td>5</td>
<td>3201</td>
<td>2</td>
<td>2</td>
<td>-6268</td>
</tr>
<tr>
<td>6</td>
<td>3236</td>
<td>0</td>
<td>0</td>
<td>204</td>
</tr>
<tr>
<td>7</td>
<td>3272</td>
<td>0</td>
<td>0</td>
<td>204</td>
</tr>
<tr>
<td>8</td>
<td>3224</td>
<td>0</td>
<td>0</td>
<td>204</td>
</tr>
<tr>
<td>9</td>
<td>3194</td>
<td>0</td>
<td>0</td>
<td>204</td>
</tr>
<tr>
<td>10</td>
<td>3188</td>
<td>2</td>
<td>2</td>
<td>-6172</td>
</tr>
<tr>
<td>11</td>
<td>3213</td>
<td>2</td>
<td>2</td>
<td>-6172</td>
</tr>
</tbody>
</table>
```

A often-used way to specify a trading strategy is to map past prices into +1, 0 or -1 for long, flat or short. A signal is often only given at a specified point (like in 'buy one unit now'). Example: suppose the third day is a Thursday, and our rule says 'buy after Thursday'.

```r
signal <- function()
  if (Time() == 3L)
    1 else 0

trade_details(btest(prices = prices, signal = signal), prices)
```

```
<table>
<thead>
<tr>
<th>price</th>
<th>suggest</th>
<th>position</th>
<th>wealth</th>
<th>cash</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3182</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>3205</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>3272</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>3185</td>
<td>1</td>
<td>1</td>
<td>-3185</td>
</tr>
</tbody>
</table>
```
Chapter 5. Examples: A single asset

But this is not what we wanted. If the rule is to buy and then keep the
long position, we should have written it like this.

```r
signal <- function()
  if (Time() == 3L)
    1 else Portfolio()
```

The function `Portfolio` evaluates to last period’s portfolio. Like `Close`,
its first argument sets the time lag, which defaults to 1.

```r
trade_details(btest(prices = prices, signal = signal), prices)
```

We may also prefer to specify `signal` so that it evaluates to a weight;
for instance, after a portfolio optimisation. In such a case, you need to
set `convert.weights` to `TRUE`. (Make sure to have a meaningful initial
wealth: 5 percent of nothing is nothing.)

```r
signal <- function()
  0.05

solution <- btest(prices = prices,
  signal = signal,
  initial.cash = 100,
  convert.weights = TRUE)
trade_details(solution, prices)
```
Note that until now we — potentially — rebalanced in every period. If you
do not want that, we need to specify `do.rebalance`.

```r
do.rebalance <- function() {
  if (sum(abs(SuggestedPortfolio(0) - Portfolio())) > 1e-3)
    TRUE
  else
    FALSE
}
```

```r
solution <- btest(prices = prices,
                  signal = signal,
                  initial.cash = 100,
                  do.rebalance = do.rebalance,
                  convert.weights = TRUE)
```

```r
trade_details(solution, prices)
```

<table>
<thead>
<tr>
<th>price</th>
<th>suggest</th>
<th>position</th>
<th>wealth</th>
<th>cash</th>
</tr>
</thead>
<tbody>
<tr>
<td>3182</td>
<td>0.00000</td>
<td>0.00000</td>
<td>100.000</td>
<td>100.00000</td>
</tr>
<tr>
<td>3205</td>
<td>0.03142</td>
<td>0.03142</td>
<td>100.000</td>
<td>-0.72282</td>
</tr>
<tr>
<td>3272</td>
<td>0.03120</td>
<td>0.03120</td>
<td>102.106</td>
<td>-0.72282</td>
</tr>
<tr>
<td>3185</td>
<td>0.03120</td>
<td>0.03120</td>
<td>99.371</td>
<td>-0.72282</td>
</tr>
<tr>
<td>3201</td>
<td>0.03120</td>
<td>0.03120</td>
<td>99.874</td>
<td>-0.72282</td>
</tr>
<tr>
<td>3236</td>
<td>0.03120</td>
<td>0.03120</td>
<td>100.974</td>
<td>-0.72282</td>
</tr>
<tr>
<td>3272</td>
<td>0.03120</td>
<td>0.03120</td>
<td>102.106</td>
<td>-0.72282</td>
</tr>
<tr>
<td>3224</td>
<td>0.03120</td>
<td>0.03120</td>
<td>100.597</td>
<td>-0.72282</td>
</tr>
<tr>
<td>3194</td>
<td>0.03120</td>
<td>0.03120</td>
<td>99.654</td>
<td>-0.72282</td>
</tr>
<tr>
<td>3188</td>
<td>0.03120</td>
<td>0.03120</td>
<td>99.466</td>
<td>-0.72282</td>
</tr>
<tr>
<td>3213</td>
<td>0.03120</td>
<td>0.03120</td>
<td>100.251</td>
<td>-0.72282</td>
</tr>
</tbody>
</table>
Chapter 5. Examples: A single asset

do_rebalance is called after signal. Hence, the suggested position is known and the lag should be zero (‘SuggestedPortfolio(0’)).

The `tol` argument works similarly: it instructs `btest` to only rebalance when the maximum absolute suggested change in any single position is greater than `tol`. Default is `0.00001`, which practically means always rebalance.

```r	solution <- btest(prices = prices, 
  signal = signal, 
  initial.cash = 100, 
  tol = 2e-5, 
  convert.weights = TRUE)
trade_details(solution, prices)
```

<table>
<thead>
<tr>
<th>prices</th>
<th>sp</th>
<th>asset.1</th>
<th>wealth</th>
<th>cash</th>
</tr>
</thead>
<tbody>
<tr>
<td>1  3182 0.00000 0.00000 100 100.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2  3205 0.00157 0.00157 100 95.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3  3272 0.00156 0.00157 100 95.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4  3185 0.00153 0.00153 100 95.1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5  3201 0.00157 0.00157 100 95.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6  3236 0.00156 0.00157 100 95.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7  3272 0.00155 0.00155 100 95.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8  3224 0.00153 0.00155 100 95.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9  3194 0.00155 0.00155 100 95.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10 3188 0.00157 0.00155 100 95.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11 3213 0.00157 0.00157 100 95.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Passing environments

To keep information persistent, we can use environments. As an example, we store (and update) the most recent entry price.

```r
notepad <- new.env() 
notepad$entry <- numeric(length(prices))
signal <- function(threshold, notepad) {
  notepad$entry[Time(0L)] <- notepad$entry[Time(1L)] 
  if (Close() < threshold) {
    if (Portfolio() < 1) 
      notepad$entry[Time(0L)] <- Close(0L)
    1 
  } else {
```
solution <- btest(prices = prices,
signal = signal,
threshold = 3200,
notepad = notepad)

cbind(trade_details(solution, prices), entry = notepad$entry)

<table>
<thead>
<tr>
<th>price</th>
<th>suggest</th>
<th>position</th>
<th>wealth</th>
<th>cash</th>
<th>entry</th>
</tr>
</thead>
<tbody>
<tr>
<td>3182</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3205</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>-3205</td>
<td>3205</td>
</tr>
<tr>
<td>3272</td>
<td>0</td>
<td>0</td>
<td>67</td>
<td>67</td>
<td>3205</td>
</tr>
<tr>
<td>3185</td>
<td>0</td>
<td>0</td>
<td>67</td>
<td>67</td>
<td>3205</td>
</tr>
<tr>
<td>3201</td>
<td>1</td>
<td>1</td>
<td>67</td>
<td>-3134</td>
<td>3201</td>
</tr>
<tr>
<td>3236</td>
<td>0</td>
<td>0</td>
<td>102</td>
<td>102</td>
<td>3201</td>
</tr>
<tr>
<td>3272</td>
<td>0</td>
<td>0</td>
<td>102</td>
<td>102</td>
<td>3201</td>
</tr>
<tr>
<td>3224</td>
<td>0</td>
<td>0</td>
<td>102</td>
<td>102</td>
<td>3201</td>
</tr>
<tr>
<td>3194</td>
<td>0</td>
<td>0</td>
<td>102</td>
<td>102</td>
<td>3201</td>
</tr>
<tr>
<td>3188</td>
<td>1</td>
<td>1</td>
<td>102</td>
<td>-3086</td>
<td>3188</td>
</tr>
<tr>
<td>3213</td>
<td>1</td>
<td>1</td>
<td>127</td>
<td>-3086</td>
<td>3188</td>
</tr>
</tbody>
</table>

Let us check.

subset(journal(solution), amount > 0)

btest provides an environment Globals for exactly such purposes.

signal <- function(threshold) {
  Globals$entry[Time(0L)] <- Globals$entry[Time(1L)]
  if (Close() < threshold) {
    if (Portfolio() < 1)
      Globals$entry[Time(0L)] <- Close(0L)
      1
  } else {
    0
  }
}

solution <- btest(prices = prices,
signal = signal,
threshold = 3200,
include.data = TRUE)
cbind(trade_details(solution, prices),
entry = solution$Globals$entry)

<table>
<thead>
<tr>
<th></th>
<th>price</th>
<th>suggest</th>
<th>position</th>
<th>wealth</th>
<th>cash</th>
<th>entry</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3182</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>NA</td>
</tr>
<tr>
<td>2</td>
<td>3205</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>-3205</td>
<td>3205</td>
</tr>
<tr>
<td>3</td>
<td>3272</td>
<td>0</td>
<td>0</td>
<td>67</td>
<td>67</td>
<td>3205</td>
</tr>
<tr>
<td>4</td>
<td>3185</td>
<td>0</td>
<td>0</td>
<td>67</td>
<td>67</td>
<td>3205</td>
</tr>
<tr>
<td>5</td>
<td>3201</td>
<td>1</td>
<td>1</td>
<td>67</td>
<td>-3134</td>
<td>3201</td>
</tr>
<tr>
<td>6</td>
<td>3236</td>
<td>0</td>
<td>0</td>
<td>102</td>
<td>102</td>
<td>3201</td>
</tr>
<tr>
<td>7</td>
<td>3272</td>
<td>0</td>
<td>0</td>
<td>102</td>
<td>102</td>
<td>3201</td>
</tr>
<tr>
<td>8</td>
<td>3224</td>
<td>0</td>
<td>0</td>
<td>102</td>
<td>102</td>
<td>3201</td>
</tr>
<tr>
<td>9</td>
<td>3194</td>
<td>0</td>
<td>0</td>
<td>102</td>
<td>102</td>
<td>3201</td>
</tr>
<tr>
<td>10</td>
<td>3188</td>
<td>1</td>
<td>1</td>
<td>102</td>
<td>-3086</td>
<td>3188</td>
</tr>
<tr>
<td>11</td>
<td>3213</td>
<td>1</td>
<td>1</td>
<td>127</td>
<td>-3086</td>
<td>3188</td>
</tr>
</tbody>
</table>

### 5.5 Examples: Several assets

It does not really make a difference whether `btest` is called with a single or with several instruments. The pattern in `signal` is still to call `Close()` and friends to obtain data, but now these functions will return matrices with more than one column. For instance, when you have 5 assets, then `Close(n = 250)` would return a matrix of size $250 \times 5$. When `signal` has finished its computations, it is now expected to return a vector of positions or weights. In the example with 5 assets, it should return a vector of length 5.

#### 5.5.1 A simple example

```r
prices1 <- c(100, 98, 98, 97, 96, 98, 97, 98, 99, 101)
prices2 <- c(100, 99, 100, 102, 101, 100, 96, 97, 95, 82)
prices <- cbind(A = prices1, B = prices2)

signal <- function()
  if (Close()[1L] > Close()[2L])
    c(2, 0) else c(0, 1)

(solution <- btest(prices = list(prices),
                   signal = signal,
                   b=2))
```
trade_details <- function(solution, prices)
data.frame(price = prices,
suggest = solution$suggested.position,
position = solution$position, ## do not unname
wealth = solution$wealth,
cash = solution$cash)

trade_details(solution, prices)

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>100</td>
<td>0</td>
<td>0</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>2</td>
<td>98</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>98</td>
<td>100</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>97</td>
<td>102</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>96</td>
<td>101</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>98</td>
<td>100</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>97</td>
<td>96</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>98</td>
<td>97</td>
<td>2</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>9</td>
<td>99</td>
<td>95</td>
<td>2</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>10</td>
<td>101</td>
<td>82</td>
<td>2</td>
<td>0</td>
<td>2</td>
</tr>
</tbody>
</table>

journal(solution)

<table>
<thead>
<tr>
<th>instrument</th>
<th>timestamp</th>
<th>amount</th>
<th>price</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>3</td>
<td>1</td>
<td>100</td>
</tr>
<tr>
<td>A</td>
<td>8</td>
<td>2</td>
<td>98</td>
</tr>
<tr>
<td>B</td>
<td>8</td>
<td>-1</td>
<td>97</td>
</tr>
</tbody>
</table>

3 transactions

5.6 Miscellaneous

There is more than one way to accomplish a certain task.

5.6.1 Handling missing values

Missing values are everywhere financial time-series. (I do hope the irony
is not wasted.) There are holidays on some exchanges but not on others;
stocks and bonds become halted or delisted; derivatives and bonds come
to life and expire; or databases may simply lack prices for certain dates
because of technical glitches.
Backtesting requires careful data preparation, which includes handling missing values. Unfortunately, there is no foolproof way to handle `NA` values, which is why `btest` does not have an `na.rm` argument. (In an experimental branch, there is an argument `allow.na`; but that branch will probably never be merged into the master branch.)

The strategies to handle `NA` values depend on the situation. If only initial price observations are missing, you can set the burnin `b` so that those values are skipped.

Copying forward the last available price is often acceptable for liquid assets when positions need to be valued, or perhaps when indicators such as moving averages are computed. But the backtest should in general not trade on such prices. This is most obvious for illiquid instruments such as options: the last trade may be some time back, and in the meantime market makers may have substantially moved bid/ask.

A more-general approach is to keep track of available assets. It is useful here to keep in mind `btest`’s general mechanism for computing trades: it compares suggested positions with actual ones, and considers the differences to be trades. However, if both suggested and actual positions are zero, nothing needs to be done for those assets. And `btest` takes this literally: those assets will not be included in computations such as the valuation of the portfolio. So a generic strategy is to check whether asset prices exists when the suggested portfolio is computed and to set the positions of missing values to zero.

Two examples follow.

```r
prices <- 11:15
prices[4:5] <- NA

signal <- function() {
  if (Time(0) <= 2)
    1 else 0
}
bt <- btest(prices, signal)
position(bt)
journal(bt)
```

```
[,1]
[1,] 0
[2,] 1
[3,] 0
```
2 transactions

prices <- cbind(11:15, NA)

signal <- function()
  c(1, 0)

bt <- btest(list(prices), signal)
positional(bt)
journal(bt)

5.6.2 Distributing backtest computations

btest offers functionality to run several backtests, or variations of backtests, automatically. Since such computations are independent from each other, they can also be distributed. btest relies on the parallel package to do this.

A simple example.

library("PMwR")
signal <- function(h) {
  if (Close() > h)
    1
  else
    0
}
Chapter 5. Miscellaneous

btest(1:10, signal = signal, h = 1)

You may now specify variations for each argument.

btest(1:10, signal = signal, variations = list(h = 1:5))

The default is to process these variations through a loop.

```r
signal <- function(h) {
  Sys.sleep(0.1)
  if (Close() > h)
    1
  else
    0
}
```

```r
system.time(btest(1:10, signal = signal, variations = list(h = 1:5)))
```

```r
system.time(btest(1:10, signal = signal, variations = list(h = 1:5),
  variations.settings = list(method = "parallel", cores = 5)))
```

```r
signal1 <- function()
  1

signal2 <- function()
  2
```

```r
btest(1:10, variations = list(signal = list(signal1, signal2)))
```

### 5.6.3 Remembering an entry price

In `signal`, assign the current price (with lag 0) to `Globals`. (That is easiest because `do.rebalance` may not be defined.)

### 5.6.4 Delaying signals

In real life, the information that is used for computing signals may arrive later expected. (This can happen on any time-scale, i.e. for high and low frequency trading.) Or our signal is current, but for some reason we cannot execute a trade.

The first case can usually be handled in `signal`, e.g. by using larger lags. One way to incorporate the second case is to add a random variable to `do.rebalance`:
if (runif(1) > prob_of_delay)
    TRUE else FALSE

If TRUE, rebalancing will take place.

5.6.5 Specifying when to compute a signal and trade

btest takes two functions, do.signal and do.rebalance, that tell the algorithm when to compute a new portfolio and when to rebalance. There are different ways to specify these points in time: as a function that returns TRUE or FALSE (most general), but also as integers, logicals or actual timestamps (e.g. dates).

Supplying particular timestamps is useful when you know you want to trade on a specific calendar day, say. That is OK because you know in advance when this calendar day is going to be. But be careful when you use other information to specify when to trade. The following examples are not equivalent:

btest(prices = prices,
      signal = signal,
      do.signal = prices > 3600)

btest(prices = prices,
      signal = signal,
      do.signal = function() Close() > 3600)

Loosely speaking, both variations compute a signal and trade only when prices is above 3600. But in the first version, there will be no time lag: if the prices exceeds 3600 at time $t_i$, we will trade at $t_i$. In the second example, Close() comes with a default lag of 1: if the price exceeds 3600 at $t_i$, we will trade at $t_{i+1}$, which is the more realistic case.

When timestamp is of a type that can be coerced to Date, you can also use the keywords such as firstofmonth or lastofmonth:

btest(prices = prices,
      signal = signal,
      do.signal = "firstofmonth")
Chapter 5. Miscellaneous

5.6.6 Writing a log

Specify the function `print.info`. The function is called at the very end of an iteration, so it is best to use no time lag. An example

```r
print.info <- function() {
  cat("Time",
      sprintf("%2d", Time(0L)), "...
      sprintf("%3d", Wealth(0L)), "\n"
  flush.console()
}
```

And since `cat` has a `file` argument, you can have it write such information into a logfile.

5.6.7 Selecting parameters: calling `btest` recursively

Suppose you have a strategy that depends on a parameter vector \( \theta \). For a given \( \theta \), the signal for the strategy would look like this.

```r
signal <- function(theta) {
  ## compute position as a function of theta
}
```

Now suppose we do not know \( \theta \). We might want to test several values, and then keep the best one. For this, we need to call `btest` recursively: at a point in time \( t \), the strategy simulates the results for various values for \( \theta \) and chooses the best \( \theta \), according to some criterion \( f \).

A useful idiom is this:

```r
signal <- function(theta) {
  if (not defined theta) {
    - run btest with theta_1, ... theta_n, select best theta
    - theta = argmin_theta f(btest(theta_i))
  }
  compute position as a function of theta
}
```

`btest` will first be invoked without \( \theta \) (or `NULL`). When the function calls `signal`, \( \theta \) is not defined and `signal` will call `btest` with a specified \( \theta \).
Let us look at an actual example.

```r
require("tseries")
require("zoo")
require("runStats")

## tmp <- get.hist.quote("^GSPC",
## start = "2010-01-01",
## end = "2013-12-31", quote = "Close")

signal <- function(Data) {
  if (is.na(Data$N)) {
    message(Timestamp(0))
    price <- Close(n = 500)
    Ns <- c(10,20)
    Data1 <- list(N = 10, hist = 200)
    res1 <- btest(price, signal, Data = Data1, b = 200)
    Data2 <- list(N = 20, hist = 200)
    res2 <- btest(price, signal, Data = Data2, b = 200)
    message("N 10 : ", round(tail(res1$wealth, 1), 2))
    message("N 20 : " , round(tail(res2$wealth, 1), 2))
    N <- if (tail(res1$wealth, 1) > tail(res2$wealth, 1))
    10
    else
    20
    message("N is ---> ", N, "\n")
  } else {
    N <- Data$N
  }

  ##
  price <- Close(n = Data$hist)
  MA <- runStats("mean", price, N = N)
  pos <- 0
  if (Close() > tail(MA, 1))
    pos <- 1
  pos
}
```
Chapter 5. Miscellaneous

Data <- list(N = NA, hist = 200)
res <- btest(tmp$Close, signal,
Data = Data,
b = 500,
initial.cash = 100,
convert.weights = TRUE,
timestamp = index(tmp))

par(mfrow = c(2,1))
plot(index(tmp), res$wealth, type = "s")
plot(tmp)

5.6.8 Time-varying asset universes

We'll see how to deal with a particular case of missing values: when certain assets are available only at certain times. We first get some data: time-series of industry portfolios from Kenneth French’s website at https://mba.tuck.dartmouth.edu/pages/faculty/ken.french/ The dataset comprises 30 series of daily data, and we use a subset that starts in January 1990.

library("NMOF")
library("zoo")
P <- French(dest.dir = tempdir(),
"30_Industry_Portfolios_daily_CSV.zip",
price.series = TRUE,
na.rm = TRUE)

P <- zoo(P, as.Date(row.names(P)))
P <- window(P, start = as.Date("1990-1-1"))
str(P)

zoo series from 1990-01-02 to 2021-01-29
 Data: num [1:7831, 1:30] 807 802 796 790 790 ...
 - attr(*, "dimnames")=List of 2
 ..$ : chr [1:7831] "1990-01-02" "1990-01-03" ...
 ..$ : chr [1:30] "Food" "Beer" "Smoke" "Games" ...
Index: Date[1:7831], format: "1990-01-02" ...

Actually, the data are complete: there are no missing values.

any(is.na(P))
So let us make them incomplete: in series 16 to 30, we remove all data before January 2000.

\[
\text{window}(P[, 16:30], \text{end} = \text{as.Date("1999-12-31")}) \leftarrow \text{NA}
\]

The key feature of \texttt{btest} to handle such data is this: if an asset is not selected (i.e. has a zero position), it is not required for valuing the position, and so it can be missing. Suppose we wanted to simulate a 50/50 investment in only the first two series (which, we know, are complete). With \texttt{btest}, we could do it as follows.

\[
\begin{align*}
\text{library("PMwR")}
\end{align*}
\]

\[
\begin{align*}
\text{bt} \leftarrow \text{btest}(\text{prices} = \text{list(coredata}(P)), \\
\text{timestamp} = \text{index}(P), \\
\text{signal} = \text{function}() \{ \\
\text{w} \leftarrow \text{numeric(ncol(Close()))} \\
\text{w}[1:2] \leftarrow \text{c}(0.5, 0.5) \\
\text{w}
\}, \\
\text{do.signal} = \text{"lastofquarter"}, \\
\text{convert.weights} = \text{TRUE}, \\
\text{initial.cash} = 100)
\end{align*}
\]

\[
\begin{align*}
\text{head(journal}(bt), n = 10, \text{by} = \text{FALSE})
\end{align*}
\]

As you can see, the function does not complain. If you check the journal, you’ll find that all transactions have been in Food and Beer, the first two industries.

<table>
<thead>
<tr>
<th>instrument</th>
<th>timestamp</th>
<th>amount</th>
<th>price</th>
</tr>
</thead>
<tbody>
<tr>
<td>Food</td>
<td>1990-03-30</td>
<td>0.0660393017</td>
<td>758.8663</td>
</tr>
<tr>
<td>Beer</td>
<td>1990-03-30</td>
<td>0.0335054119</td>
<td>1481.5517</td>
</tr>
<tr>
<td>Food</td>
<td>1990-06-29</td>
<td>0.0026905121</td>
<td>842.7345</td>
</tr>
<tr>
<td>Beer</td>
<td>1990-06-29</td>
<td>-0.0011305346</td>
<td>1775.1047</td>
</tr>
<tr>
<td>Food</td>
<td>1990-09-28</td>
<td>-0.0014597110</td>
<td>774.9077</td>
</tr>
<tr>
<td>Beer</td>
<td>1990-09-28</td>
<td>0.0007077629</td>
<td>1575.3859</td>
</tr>
<tr>
<td>Food</td>
<td>1990-12-31</td>
<td>0.0008250171</td>
<td>881.3539</td>
</tr>
<tr>
<td>Beer</td>
<td>1990-12-31</td>
<td>-0.0003957095</td>
<td>1824.9844</td>
</tr>
<tr>
<td>Food</td>
<td>1991-03-28</td>
<td>-0.0004125792</td>
<td>1079.7564</td>
</tr>
<tr>
<td>Beer</td>
<td>1991-03-28</td>
<td>0.0001984854</td>
<td>2237.6230</td>
</tr>
</tbody>
</table>

10 transactions
Chapter 5. Miscellaneous

Now we can start the actual example. The aim in this exercise is to compute a minimum-variance portfolio over all available assets. We begin by defining when certain assets were available, and placing this information in a data-frame active.

```r
active <- data.frame(instrument = colnames(P),
                      start = c(rep(as.Date("1990-1-1"), 15),
                                rep(as.Date("2001-1-1"), 15)),
                      end = tail(index(P), 1))
```

<table>
<thead>
<tr>
<th>instrument</th>
<th>start</th>
<th>end</th>
</tr>
</thead>
<tbody>
<tr>
<td>Food</td>
<td>1990-01-01</td>
<td>2021-01-29</td>
</tr>
<tr>
<td>Beer</td>
<td>1990-01-01</td>
<td>2021-01-29</td>
</tr>
<tr>
<td>Smoke</td>
<td>1990-01-01</td>
<td>2021-01-29</td>
</tr>
<tr>
<td>Games</td>
<td>1990-01-01</td>
<td>2021-01-29</td>
</tr>
<tr>
<td>Books</td>
<td>1990-01-01</td>
<td>2021-01-29</td>
</tr>
<tr>
<td>Hshld</td>
<td>1990-01-01</td>
<td>2021-01-29</td>
</tr>
<tr>
<td>Clths</td>
<td>1990-01-01</td>
<td>2021-01-29</td>
</tr>
<tr>
<td>Hlth</td>
<td>1990-01-01</td>
<td>2021-01-29</td>
</tr>
<tr>
<td>Chems</td>
<td>1990-01-01</td>
<td>2021-01-29</td>
</tr>
<tr>
<td>Txtls</td>
<td>1990-01-01</td>
<td>2021-01-29</td>
</tr>
<tr>
<td>Cnstr</td>
<td>1990-01-01</td>
<td>2021-01-29</td>
</tr>
<tr>
<td>Steel</td>
<td>1990-01-01</td>
<td>2021-01-29</td>
</tr>
<tr>
<td>FabPr</td>
<td>1990-01-01</td>
<td>2021-01-29</td>
</tr>
<tr>
<td>ElcEq</td>
<td>1990-01-01</td>
<td>2021-01-29</td>
</tr>
<tr>
<td>Autos</td>
<td>1990-01-01</td>
<td>2021-01-29</td>
</tr>
<tr>
<td>Carry</td>
<td>2001-01-01</td>
<td>2021-01-29</td>
</tr>
<tr>
<td>Mines</td>
<td>2001-01-01</td>
<td>2021-01-29</td>
</tr>
<tr>
<td>Coal</td>
<td>2001-01-01</td>
<td>2021-01-29</td>
</tr>
<tr>
<td>Oil</td>
<td>2001-01-01</td>
<td>2021-01-29</td>
</tr>
<tr>
<td>Util</td>
<td>2001-01-01</td>
<td>2021-01-29</td>
</tr>
<tr>
<td>Telcm</td>
<td>2001-01-01</td>
<td>2021-01-29</td>
</tr>
<tr>
<td>Servs</td>
<td>2001-01-01</td>
<td>2021-01-29</td>
</tr>
<tr>
<td>BusEq</td>
<td>2001-01-01</td>
<td>2021-01-29</td>
</tr>
<tr>
<td>Paper</td>
<td>2001-01-01</td>
<td>2021-01-29</td>
</tr>
<tr>
<td>Trans</td>
<td>2001-01-01</td>
<td>2021-01-29</td>
</tr>
<tr>
<td>Whls1</td>
<td>2001-01-01</td>
<td>2021-01-29</td>
</tr>
<tr>
<td>Rtail</td>
<td>2001-01-01</td>
<td>2021-01-29</td>
</tr>
<tr>
<td>Meals</td>
<td>2001-01-01</td>
<td>2021-01-29</td>
</tr>
<tr>
<td>Meals</td>
<td>2001-01-01</td>
<td>2021-01-29</td>
</tr>
<tr>
<td>Fin</td>
<td>2001-01-01</td>
<td>2021-01-29</td>
</tr>
<tr>
<td>Other</td>
<td>2001-01-01</td>
<td>2021-01-29</td>
</tr>
</tbody>
</table>

Note that we did set `start` to 2001, not 2000. You’ll see shortly, why.

Now for the signal function. It receives `active` as an argument.
mv <- function(active) {
## find those assets that are active
## ==> 'j' is a logical vector that
## indicates the active assets
j <- Timestamp() >= active["start"] &
    Timestamp() <= active["end"]

## get last 260 prices of active assets and compute
## variance-covariance matrix
P.j <- Close(n = 260)[, j]
R.j <- returns(P.j)
S <- cov(R.j)

## compute minimum-variance weights
w.j <- NMOF::minvar(S, wmin = 0, wmax = 0.10)

## create a zero-vector with length equal to number
## of total assets and assign the weights at
## appropriate positions
w <- numeric(length(j))
w[j] <- w.j
w
}

Now you see why we used 2001 as the start date for series 16 to 30: we’ll use one year of historical data to compute the variance-covariance matrix. (Note that there are better ways to come up with forecasts of the variance-covariance matrix, e.g. methods that apply shrinkage. But the purpose of this note is to show how to handle missing values in bttest, not to discuss empirical methods.)

We call bttest.

bt.mv <- bttest(prices = list(coredata(P)),
    timestamp = index(P),
    signal = mv,
    do.signal = "lastofquarter",
    convert.weights = TRUE,
    initial.cash = 100,
    active = active,
    b = 260)
Chapter 5. Miscellaneous

bt.mv

<table>
<thead>
<tr>
<th>instrument</th>
<th>timestamp</th>
<th>amount</th>
<th>price</th>
</tr>
</thead>
<tbody>
<tr>
<td>Oil</td>
<td>2001-03-30</td>
<td>0.0104934366</td>
<td>2656.871</td>
</tr>
<tr>
<td>Oil</td>
<td>2001-06-29</td>
<td>-0.0003607878</td>
<td>2709.119</td>
</tr>
<tr>
<td>Oil</td>
<td>2001-09-28</td>
<td>0.0011873853</td>
<td>2383.685</td>
</tr>
<tr>
<td>Oil</td>
<td>2001-12-31</td>
<td>-0.0043576713</td>
<td>2549.018</td>
</tr>
<tr>
<td>Oil</td>
<td>2002-03-28</td>
<td>-0.0037902744</td>
<td>2807.207</td>
</tr>
</tbody>
</table>

5 transactions

As expected, the first trades occur only in 2001.

A final remark: we would not have needed to prepare active upfront. Instead, we could have checked for missing values in the signal function.

```r
mv_with_NA_check <- function() {
  ## fetch data and check for missing values
  P <- Close(n = 260)
  j <- !apply(P, 2, anyNA)
  
  ## get last 250 prices of active assets and compute
  ## variance--covariance matrix
  P.j <- P[, j]
  R.j <- returns(P.j)
  S <- cov(R.j)
  
  ## compute minimum-variance weights
  w.j <- NMOF::minvar(S, wmin = 0, wmax = 0.10)
  
  ## create a zero-vector with length equal to number
  ## of total assets and assign the weights at
  ## appropriate positions
  w <- numeric(length(j))
  w[j] <- w.j
  w
}
```
bt.mv2 <- btest(prices = list(coredata(P)),
    timestamp = index(P),
    signal = mv_with_NA_check,
    do.signal = "lastofquarter",
    convert.weights = TRUE,
    initial.cash = 100,
    b = 260)

bt.mv2
head(journal(bt.mv)["Oil"], 5)

<table>
<thead>
<tr>
<th>initial wealth 100 =&gt; final wealth 1779.84</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total return 1679.8%</td>
</tr>
</tbody>
</table>

We get the same results. But defining an explicit list is more, well, explicit. Which is often a good thing when analysing data; notably, because it sets an expectation that those active time-series don’t have missing values.
Chapter 5. Miscellaneous
6 Rebalancing a portfolio

In principle, rebalancing a portfolio is straightforward. You have a current position $x_0$, and a target position $x_1$. You take the elementwise difference between these vectors, and you obtain the orders that you need to have executed.

Software can help here in two ways. First, it can make the process of order generation faster, simpler and more reliable, by computing and creating orders, and submitting them automatically. Second, software can help to determine how to rebalance. Institutional investors spend large amounts of time on rebalancing, because here they can measure improvements (notably, saved transaction costs).

6.1 Usage with unnamed vectors

The function `rebalance` computes the transactions necessary for moving from one portfolio to another. The default setting is that the current portfolio is in currency units; the target portfolio in weights.

To compute the required order sizes, we also need the current prices of the assets. When `current`, `target` and `price` are unnamed, the assets’ positions in the vectors need to match.

Suppose we have three stocks A, B and C with prices 1, 2 and 3. The main use case is a situation like this: you hold 50, 30 and 20 shares of these three stocks. However, suppose you have a target weight of 50%, 30% and 20%.

```r
prices <- 1:3
```
Chapter 6. Usage with unnamed vectors

current <- c(50, 30, 20)
target <- c(0.5, 0.3, 0.2)
rebalance(current, target, prices, match.names = FALSE)

<table>
<thead>
<tr>
<th>price</th>
<th>current value %</th>
<th>target value %</th>
<th>order</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>50 29.4</td>
<td>85 50.0</td>
<td>35</td>
</tr>
<tr>
<td>2</td>
<td>30 35.3</td>
<td>26 30.6</td>
<td>-4</td>
</tr>
<tr>
<td>3</td>
<td>20 35.3</td>
<td>11 19.4</td>
<td>-9</td>
</tr>
</tbody>
</table>


Or perhaps you prefer an equal weight for every asset. Note that target now is a single number.

rebalance(current, target = 1/length(current),
           price = prices, match.names = FALSE)

<table>
<thead>
<tr>
<th>price</th>
<th>current value %</th>
<th>target value %</th>
<th>order</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>50 29.4</td>
<td>57 33.5</td>
<td>7</td>
</tr>
<tr>
<td>2</td>
<td>30 35.3</td>
<td>28 32.9</td>
<td>-2</td>
</tr>
<tr>
<td>3</td>
<td>20 35.3</td>
<td>19 33.5</td>
<td>-1</td>
</tr>
</tbody>
</table>


Note that the target weights cannot be reached exactly because the function rounds to integers.

rebalance also supports a number of special cases. Suppose you want to go into cash and close every position.

1 rebalance(current = current, target = 0,
           price = prices, match.names = FALSE)

<table>
<thead>
<tr>
<th>price</th>
<th>current value %</th>
<th>target value %</th>
<th>order</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>50 29.4</td>
<td>0 0.0</td>
<td>-50</td>
</tr>
<tr>
<td>2</td>
<td>30 35.3</td>
<td>0 0.0</td>
<td>-30</td>
</tr>
<tr>
<td>3</td>
<td>20 35.3</td>
<td>0 0.0</td>
<td>-20</td>
</tr>
</tbody>
</table>


Suppose we have no current position and want to give equal weight to each stock. Note first that we need to specify a notional. Also, rebalance now assumes that you want to invest in every stock for which a price is supplied.
rebalance(current = 0, target = 1/3, notional = 100, 
price = prices, match.names = FALSE)

<table>
<thead>
<tr>
<th>price</th>
<th>current</th>
<th>value %</th>
<th>target value %</th>
<th>order</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1</td>
<td>0</td>
<td>0 0.0</td>
<td>33 33.0</td>
<td>33</td>
</tr>
<tr>
<td>2 2</td>
<td>0</td>
<td>0 0.0</td>
<td>17 34.0</td>
<td>17</td>
</tr>
<tr>
<td>3 3</td>
<td>0</td>
<td>0 0.0</td>
<td>11 33.0</td>
<td>11</td>
</tr>
</tbody>
</table>

Notional: 100. Target net amount : 100. Turnover (2-way): 100.

6.2 Usage with named vectors

More usefully, rebalance can also use the names of the vectors current, target and price. The argument match.names must be set to TRUE for this (which is the default, actually).

prices <- c(1,1,1,1)
names(prices) <- letters[1:4]
current <- c(a = 0, b = 10)
target <- c(a = 0, d = 0.5)
rebalance(current, target, prices)

<table>
<thead>
<tr>
<th>price</th>
<th>current</th>
<th>value %</th>
<th>target value %</th>
<th>order</th>
</tr>
</thead>
<tbody>
<tr>
<td>b 1</td>
<td>10</td>
<td>10 100.0</td>
<td>0 0.0</td>
<td>-10</td>
</tr>
<tr>
<td>d 1</td>
<td>0</td>
<td>0 0.0</td>
<td>5 50.0</td>
<td>5</td>
</tr>
</tbody>
</table>


To also show all instruments, set the argument drop.zero to FALSE.

print(rebalance(current, target, prices), drop.zero = FALSE)

<table>
<thead>
<tr>
<th>price</th>
<th>current</th>
<th>value %</th>
<th>target value %</th>
<th>order</th>
</tr>
</thead>
<tbody>
<tr>
<td>a 1</td>
<td>0</td>
<td>0 0.0</td>
<td>0 0.0</td>
<td>0</td>
</tr>
<tr>
<td>b 1</td>
<td>10</td>
<td>10 100.0</td>
<td>0 0.0</td>
<td>-10</td>
</tr>
<tr>
<td>d 1</td>
<td>0</td>
<td>0 0.0</td>
<td>5 50.0</td>
<td>5</td>
</tr>
</tbody>
</table>

6.3 Usage with positions

In Section Keeping track of transactions: journals we used the function `position` to compute balances from transactions. The function may also be directly used to set up a position.

```r
position(amount = 1)
```

```r
1
```

```r
position(amount = c(1, 2, 3), instrument = letters[1:3])
```

```r
a 1
b 2
c 3
```

Note that with more than one instrument, these instruments need to be named. Otherwise, position would aggregate the positions (as it does for a journal). As a short-cut, you can also pass a named vector. (See [http://enricoschumann.net/notes/computing-positions.html](http://enricoschumann.net/notes/computing-positions.html) for more details.)

```r
position(amount = c(a = 1, b = 2, c = 3))
```

```r
a 1
b 2
c 3
```

Such positions can now be passed as arguments `current` and `target` into function `rebalance`.

6.4 Algorithms

Whenever you need to round positions, you may prefer to do an actual optimisation. The ideal place for this optimisation is the original objective function, not in `rebalance`. And the differences, if there are any at all, are typically small. But here is an example.

```r
n <- 10
target <- runif(n)
target <- target/sum(target)
```
price <- sample(10:200, n, replace = TRUE)
s <- sample(c(1,5,10,100), n, replace = TRUE,
prob = c(0.4,0.4,0.1,0.1))
data.frame(price = price, lot.size = s)

<table>
<thead>
<tr>
<th>price</th>
<th>lot.size</th>
</tr>
</thead>
<tbody>
<tr>
<td>178</td>
<td>5</td>
</tr>
<tr>
<td>37</td>
<td>5</td>
</tr>
<tr>
<td>62</td>
<td>5</td>
</tr>
<tr>
<td>93</td>
<td>1</td>
</tr>
<tr>
<td>81</td>
<td>5</td>
</tr>
<tr>
<td>111</td>
<td>5</td>
</tr>
<tr>
<td>146</td>
<td>5</td>
</tr>
<tr>
<td>154</td>
<td>5</td>
</tr>
<tr>
<td>187</td>
<td>1</td>
</tr>
<tr>
<td>138</td>
<td>1</td>
</tr>
</tbody>
</table>

Now suppose we have only a limited budget available.

budget <- 10000
x <- rebalance(0, target, notional = budget,
price = price, match.names = FALSE)
x

<table>
<thead>
<tr>
<th>price</th>
<th>current value</th>
<th>%</th>
<th>target value</th>
<th>%</th>
<th>order</th>
</tr>
</thead>
<tbody>
<tr>
<td>178</td>
<td>0</td>
<td>0.0</td>
<td>712</td>
<td>7.1</td>
<td>4</td>
</tr>
<tr>
<td>37</td>
<td>0</td>
<td>0.0</td>
<td>1480</td>
<td>14.8</td>
<td>40</td>
</tr>
<tr>
<td>62</td>
<td>0</td>
<td>0.0</td>
<td>1240</td>
<td>12.4</td>
<td>20</td>
</tr>
<tr>
<td>93</td>
<td>0</td>
<td>0.0</td>
<td>1488</td>
<td>14.9</td>
<td>16</td>
</tr>
<tr>
<td>81</td>
<td>0</td>
<td>0.0</td>
<td>1053</td>
<td>10.5</td>
<td>13</td>
</tr>
<tr>
<td>111</td>
<td>0</td>
<td>0.0</td>
<td>666</td>
<td>6.7</td>
<td>6</td>
</tr>
<tr>
<td>146</td>
<td>0</td>
<td>0.0</td>
<td>584</td>
<td>5.8</td>
<td>4</td>
</tr>
<tr>
<td>154</td>
<td>0</td>
<td>0.0</td>
<td>924</td>
<td>9.2</td>
<td>6</td>
</tr>
<tr>
<td>187</td>
<td>0</td>
<td>0.0</td>
<td>935</td>
<td>9.3</td>
<td>5</td>
</tr>
<tr>
<td>138</td>
<td>0</td>
<td>0.0</td>
<td>966</td>
<td>9.7</td>
<td>7</td>
</tr>
</tbody>
</table>


Now we use TAopt, from the NMOF package, to find the optimal integer portfolio.

require("NMOF")
ediff <- function(x) {
  tmp <- x*price/budget - target
  sum(tmp*tmp)
}

Now suppose we have only a limited budget available.

budget <- 10000
x <- rebalance(0, target, notional = budget,
price = price, match.names = FALSE)
x

<table>
<thead>
<tr>
<th>price</th>
<th>current value</th>
<th>%</th>
<th>target value</th>
<th>%</th>
<th>order</th>
</tr>
</thead>
<tbody>
<tr>
<td>178</td>
<td>0</td>
<td>0.0</td>
<td>712</td>
<td>7.1</td>
<td>4</td>
</tr>
<tr>
<td>37</td>
<td>0</td>
<td>0.0</td>
<td>1480</td>
<td>14.8</td>
<td>40</td>
</tr>
<tr>
<td>62</td>
<td>0</td>
<td>0.0</td>
<td>1240</td>
<td>12.4</td>
<td>20</td>
</tr>
<tr>
<td>93</td>
<td>0</td>
<td>0.0</td>
<td>1488</td>
<td>14.9</td>
<td>16</td>
</tr>
<tr>
<td>81</td>
<td>0</td>
<td>0.0</td>
<td>1053</td>
<td>10.5</td>
<td>13</td>
</tr>
<tr>
<td>111</td>
<td>0</td>
<td>0.0</td>
<td>666</td>
<td>6.7</td>
<td>6</td>
</tr>
<tr>
<td>146</td>
<td>0</td>
<td>0.0</td>
<td>584</td>
<td>5.8</td>
<td>4</td>
</tr>
<tr>
<td>154</td>
<td>0</td>
<td>0.0</td>
<td>924</td>
<td>9.2</td>
<td>6</td>
</tr>
<tr>
<td>187</td>
<td>0</td>
<td>0.0</td>
<td>935</td>
<td>9.3</td>
<td>5</td>
</tr>
<tr>
<td>138</td>
<td>0</td>
<td>0.0</td>
<td>966</td>
<td>9.7</td>
<td>7</td>
</tr>
</tbody>
</table>


Now we use TAopt, from the NMOF package, to find the optimal integer portfolio.

require("NMOF")
ediff <- function(x) {
  tmp <- x*price/budget - target
  sum(tmp*tmp)
}
neighbour <- function(x) {
  i <- sample.int(length(x), size = 1L)
  x[i] <- x[i] + if (runif(1) > 0.5) - s[i] else s[i]
  x
}

sol <- TAopt(ediff,
  algo = list(x0 = numeric(length(price)),
              neighbour = neighbour,
              q = 0.1,
              nS = 1000,
              printBar = FALSE))

Threshold Accepting.

Computing thresholds ... OK.
Estimated remaining running time: 0.23 secs.

Running Threshold Accepting...
Initial solution: 0.109341
 Finished.
Best solution overall: 0.001108741

df <- data.frame(TA = sol$xbest, rounded = s*round(x$target/s))
df[apply(df, 1, function(i) any(i != 0)), ]

<table>
<thead>
<tr>
<th>TA</th>
<th>rounded</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>40</td>
</tr>
<tr>
<td>3</td>
<td>20</td>
</tr>
<tr>
<td>4</td>
<td>16</td>
</tr>
<tr>
<td>5</td>
<td>15</td>
</tr>
<tr>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>7</td>
<td>5</td>
</tr>
<tr>
<td>8</td>
<td>5</td>
</tr>
<tr>
<td>9</td>
<td>5</td>
</tr>
<tr>
<td>10</td>
<td>7</td>
</tr>
</tbody>
</table>

The difference.

ediff(sol$xbest) - ediff(s*round(x$target/s))

[1] 0
6.5 Substituting a basket by its components

If you run tests with baskets of instruments or whole strategies, you often need to substitute the components of the basket for overall basket. PMwR provides a function `replace_weight` that helps with this task. (It is also helpful if you have hierarchies of benchmarks or want to do a 'lookthrough' through a subportfolio within your portfolio.)

Suppose we have this weight vector:

\[
\mathbf{w} \leftarrow \begin{pmatrix}
\text{basket}_1 &=& 0.3, \\
\text{basket}_2 &=& 0.5, \\
\text{basket}_3 &=& 0.2
\end{pmatrix}
\]

We also know what the first two baskets represent.

\[
\mathbf{b}_1 \leftarrow \begin{pmatrix}
a &=& 0.5, \\
b &=& 0.2, \\
c &=& 0.3
\end{pmatrix}
\]
\[
\mathbf{b}_2 \leftarrow \begin{pmatrix}
d &=& 0.1, \\
e &=& 0.2, \\
a &=& 0.7
\end{pmatrix}
\]

Now we can call `replace_weight`.

```r
replace_weight(w, 
  basket_1 = b1, 
  basket_2 = b2)
```

```
basket_1::a  basket_1::b  basket_1::c  
0.15 0.06 0.09

basket_2::d  basket_2::e  basket_2::a  
0.05 0.10 0.35

basket_3  
0.20
```

If the names of the baskets or of the things in the baskets have spaces or other characters that cause trouble, quote them.

```r
replace_weight(c("basket 1" = 0.3, 
  "basket 2" = 0.7), 
  "basket 1" = b1, 
  "basket 2" = b2)
```
Chapter 6. Substituting a basket by its components
7 Summarising portfolio time-series

Strategies or portfolios are often analysed purely through their price (a.k.a. NAV or equity) series: more-detailed data may not be available (e.g. for a fund); or it may simply be more convenient to abstract from the positions to the NAV level.

To handle such series, *pmwr* provides an S3 class `NAVseries`. (I will write NAV series for the actual data series and `NAVseries` for the specific implementation.) An NAV series is nothing more than a time-series: a vector of NAVs, together with a vector of timestamps. Then why not simply use an existing time-series class, such as `zoo`? One reason is clarity. A `zoo` or `xts` object is much more general than an NAV series: it may represent more than one series; or it may represent, for instance, returns. An NAV series promises to represent NAVs (i.e. levels, not changes in levels) of a single series, nothing else. Furthermore, defining our own class allows us to define specific methods where appropriate; at the same time we may piggyback on existing time-series methods by defining methods for coercion, e.g. `as.zoo.NAVseries` or `as.xts.NAVseries`.

7.1 Creating NAV series

*pmwr* comes with a dataset called `DAX`, which stands for *Deutscher Aktienindex* (German Equity Index). The dataset is a data-frame of one column that contains the price for the day, with the timestamps stored as rownames in format `YYYY-MM-DD`.

```
str(DAX)
```
Chapter 7. Summarising NAV series

A concise summary is printed.

There is also a generic function `as.NAVseries`, which can be used to coerce for instance `zoo` series to `NAVseries`.

If you have run a backtest (see Chapter Backtesting), you can extract the NAV series by saying

```r
as.NAVseries(btest(....))
```

### 7.2 Summarising NAV series

Calling `summary` on an NAV series should provide various useful statistics of the series. You may notice that the summary does not provide too many statistics. That is on purpose: it has been said that the purpose of statistics is to reduce many numbers to only few (not the other way around). Thus, `summary.NAVseries` will confine itself to few statistics that can be computed in reasonably robust way. But that I mean that a
A statistic is useful for many types of NAV series. Numbers that are provided should be formatted and presented in a way that is inline with (good) industry practice and makes sense statistically: returns, for instance, will only be annualised when the NAV series spans more than one calendar year; volatility will always be annualised, and be computed from monthly data (if possible); all numbers are rounded to a meaningful precision (Ehrenberg, 1981).

```
summary(dax)
```

<table>
<thead>
<tr>
<th>DAX</th>
<th>02 Jan 2014 ==&gt; 30 Dec 2015 (505 data points, 0 NAs)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>9400.04 10743</td>
</tr>
<tr>
<td>High</td>
<td>12374.73 (10 Apr 2015)</td>
</tr>
<tr>
<td>Low</td>
<td>8571.95 (15 Oct 2014)</td>
</tr>
<tr>
<td>Return (%)</td>
<td>6.9 (annualised)</td>
</tr>
<tr>
<td>Max.drawdown (%)</td>
<td>23.8</td>
</tr>
<tr>
<td>_ peak</td>
<td>12374.73 (10 Apr 2015)</td>
</tr>
<tr>
<td>_ trough</td>
<td>9427.64 (24 Sep 2015)</td>
</tr>
<tr>
<td>_ recovery</td>
<td>(NA)</td>
</tr>
<tr>
<td>_ underwater now (%)</td>
<td>13.2</td>
</tr>
<tr>
<td>Volatility (%)</td>
<td>18.0 (annualised)</td>
</tr>
<tr>
<td>_ upside</td>
<td>14.4</td>
</tr>
<tr>
<td>_ downside</td>
<td>10.4</td>
</tr>
</tbody>
</table>

```
Monthly returns
```

```
<table>
<thead>
<tr>
<th>Jan</th>
<th>Feb</th>
<th>Mar</th>
<th>Apr</th>
<th>May</th>
<th>Jun</th>
<th>Jul</th>
<th>Aug</th>
<th>Sep</th>
<th>Oct</th>
<th>Nov</th>
<th>Dec</th>
<th>YTD</th>
</tr>
</thead>
<tbody>
<tr>
<td>2014</td>
<td>-1.0</td>
<td>4.1</td>
<td>-1.4</td>
<td>0.5</td>
<td>3.5</td>
<td>-1.1</td>
<td>-4.3</td>
<td>0.7</td>
<td>0.0</td>
<td>-1.6</td>
<td>7.0</td>
<td>-1.8</td>
</tr>
<tr>
<td>2015</td>
<td>9.1</td>
<td>6.6</td>
<td>5.0</td>
<td>-4.3</td>
<td>-0.4</td>
<td>-4.1</td>
<td>3.3</td>
<td>-9.3</td>
<td>-5.8</td>
<td>12.3</td>
<td>4.9</td>
<td>-5.6</td>
</tr>
</tbody>
</table>
```

For summaries of NAV series, a method for \texttt{toLatex} can be used to fill

\footnote{An example of a statistic that cannot be computed in a robust way is a beta coefficient – or, in fact, any other multivariate statistic. Such numbers will depend on a chosen benchmark, and what is more, they typically need careful checking because of asynchronous time-series: A mutual fund’s NAV that is published with a delay may show little correlation with a stock-market index (when matched on time-stamps), but that is only because of the ‘attenuation bias’, which skews the correlation towards zero.}
Chapter 7. Drawdowns and Streaks

PMwR provided two functions that may provide insights into NAV series: drawdowns and streaks. A vignette provides some examples.

vignette("Drawdowns_streaks", package = "PMwR")

7.3 Drawdowns and Streaks

Let $v$ be a time series of portfolio values, with observations at $t = 0, 1, 2 \ldots T$. Then the drawdown $D$ of this series at time $t$ is defined as

$$D_t = v_{\max} - v_t$$

(7.1)

where $v_{\max}$ is the running maximum, that is, $v_{\max} = \max\{v_t | t' \in [0, t]\}$.

$D$ is a whole vector of length $T + 1$; though most of the time people care most about its maximum value.

Other functions may be computed to capture the information in the drawdown vector, for example, the mean time underwater (i.e., the average time elapsed between two consecutive values in $D$ that are sufficiently close to zero), or the correlation between a portfolio’s drawdown and the drawdown of an alternative asset, like an index. The definition above gives $D$ in currency terms. A percentage drawdown is often preferred, obtained by using the logarithm of $v$, or by dividing $D_t$ by $v_{\max}$.

In pseudocode, drawdown can be computed as follows.

1: set $high = P_1$
2: set $maxdown = 0$
3: for $t = 2 : \text{length}(P)$ do
4:     if $P_t > high$ then
5:         $high = P_t$
6:     else
7:         compute $underwater = 1 - P_t/\text{high}$
8:     if $underwater > maxdown$ then
9:         $maxdown = underwater$
The function drawdowns relies on the function drawdown in the NMOF package.

\[
\text{NMOF::drawdown}
\]

\[
\begin{align*}
\text{function} & \quad (v, \text{relative} = \text{TRUE}, \text{summary} = \text{TRUE}) \\
& \{ \\
& \quad \text{cv} <- \text{cummax}(v) \\
& \quad \text{rd} <- \text{cv} - v \\
& \quad \text{if} (\text{relative}) \\
& \quad \quad \text{rd} <- \text{rd}/\text{cv} \\
& \quad \text{if} (\text{summary}) \{ \\
& \quad \quad \text{troughTime} <- \text{which.max}(\text{rd}) \\
& \quad \quad \text{peakTime} <- \text{which.max}(v[\text{seq_len}(\text{troughTime})]) \\
& \quad \quad \text{list}(\text{maximum} = \text{max}(\text{rd}), \\
& \quad \quad \quad \text{high} = v[\text{peakTime}], \\
& \quad \quad \quad \text{high.position} = \text{peakTime}, \\
& \quad \quad \quad \text{low} = v[\text{troughTime}], \\
& \quad \quad \quad \text{low.position} = \text{troughTime}) \\
& \quad \} \quad \text{else rd} \\
& \}
\end{align*}
\]

Let us look at the five worst drawdowns.

\[
\text{dd} <- \text{drawdowns(dax)} \\
\text{dd}[\text{order}(\text{dd}\$\text{max}, \text{decreasing} = \text{TRUE})[1:5], ]
\]

\[
\begin{array}{cccccc}
\text{peak} & \text{trough} & \text{recover} & \text{max} \\
18 & 2015-04-10 & 2015-09-24 & <\text{NA}> & 0.238154 \\
9 & 2014-07-03 & 2014-10-15 & 2014-12-05 & 0.145320 \\
10 & 2014-12-05 & 2014-12-15 & 2015-01-16 & 0.074661 \\
4 & 2014-01-17 & 2014-03-13 & 2014-05-13 & 0.074430 \\
17 & 2015-03-16 & 2015-03-26 & 2015-04-10 & 0.026631 \\
\end{array}
\]

TODO Splitting a time-series into drawdowns.
7.3.2 Streaks

The function `streaks` returns a data-frame of uninterrupted up and down movements.

```
streaks(dax, up = 0.1, down = -0.1, initial.state = "up")
```

<table>
<thead>
<tr>
<th>start</th>
<th>end</th>
<th>state</th>
<th>return</th>
</tr>
</thead>
<tbody>
<tr>
<td>2014-01-02</td>
<td>2014-07-03</td>
<td>up</td>
<td>0.066956</td>
</tr>
<tr>
<td>2014-07-03</td>
<td>2014-10-15</td>
<td>down</td>
<td>-0.145320</td>
</tr>
<tr>
<td>2014-10-15</td>
<td>2015-04-10</td>
<td>up</td>
<td>0.443631</td>
</tr>
<tr>
<td>2015-04-10</td>
<td>2015-09-24</td>
<td>down</td>
<td>-0.238154</td>
</tr>
<tr>
<td>2015-09-24</td>
<td>2015-11-30</td>
<td>up</td>
<td>0.207325</td>
</tr>
<tr>
<td>2015-11-30</td>
<td>2015-12-30</td>
<td>down</td>
<td>-0.056159</td>
</tr>
</tbody>
</table>
8 Scaling series

Visual comparisons of time-series are ubiquitous in finance.\(^1\) The function `scale1` helps with scaling the levels of time-series so that it becomes easier to compare them. It is a generic function; and PMwR provides methods for numeric vectors/matrices, and for zoo and NAVseries objects.

8.1 Examples

To explain what the function does, we use two very short time-series: the values of the DAX, the German stock-market index, and the REXP, a German government-bond index, from 2 January and 8 January 2014 (just 5 trading days). We also combine them into a matrix `drax`.

```r
dax <- DAX[1:5, ]
rexp <- REXP[1:5, ]
drax <- cbind(dax, rexp)
```

Calling `scale1` on `dax` is equivalent to dividing the whole series by its first element.

```r
scale1(dax) == dax/dax[1]
```

```
[1] TRUE TRUE TRUE TRUE TRUE TRUE
```

Lest you skip the rest of the chapter: `scale1` comes with several additional features.

---

\(^1\)Transformating or scaling data are a key element of exploratory data analysis in general. See Tukey's EDA (1977). TODO: find H. Simon reference on scaling (taking reciprocal value). See also Schumann (2013).
Chapter 8. Examples

It is common, too, to scale to a level of 100. We either multiply the whole series by 100, or use the \texttt{level} argument.

\begin{verbatim}
scale1(dax, level = 100)
[1] 100.00 100.37 100.30 101.13 101.04
attr("scale1_origin")
[1] 1
\end{verbatim}

(The \texttt{scale1_origin} attribute will be explained shortly.)

If we give a matrix to \texttt{scale1}, the function scales each column separately.

\begin{verbatim}
scale1(drax, level = 100)

 dax   rexp
[1,] 100.00 100.00
[2,] 100.37 100.06
[3,] 100.30 100.23
[4,] 101.13 100.29
[5,] 101.04 100.28
attr("scale1_origin")
[1] 1
\end{verbatim}

\texttt{scale1} is a generic function; it works, for instance, with \texttt{zoo} objects.

\begin{verbatim}
library("zoo")
drax.zoo <- zoo(drax, as.Date(row.names(DAX)[1:5]))
scale1(drax.zoo, level = 100)

 dax   rexp
2014-01-02 100.00 100.00
2014-01-03 100.37 100.06
2014-01-06 100.30 100.23
2014-01-07 101.13 100.29
2014-01-08 101.04 100.28
attr("scale1_origin")
[1] 2014-01-02
\end{verbatim}

\begin{verbatim}
plot(scale1(drax.zoo, level = 100),
plot.type = "single",
xlab = "",
ylab = "",
col = c("darkblue", "darkgreen"))
\end{verbatim}

The argument \texttt{when} defines the origin.
Figure 8.1: Scaled series: Both dax and rexp now start at 100.

```
scale1(drax, when = 3, level = 100)

   dax    rexp
[1,] 99.703 99.769
[2,] 100.076 99.830
[3,] 100.000 100.000
[4,] 100.829 100.062
[5,] 100.741 100.049
attr(,"scale1_origin")
[1] 3
```

This origin is attached to the scaled series as an attribute `scale1_origin`. This is useful if you want mark the start of the scaled series; for instance, in a plot with `abline`.

With a `zoo` object, `when` should be compatible with the class of the object’s index.

```
scale1(drax.zoo, when = as.Date("2014-01-07"), level = 100)

   dax    rexp
2014-01-02 98.883 99.707
2014-01-03 99.253 99.768
2014-01-06 99.177 99.938
2014-01-07 100.000 100.000
2014-01-08 99.912 99.987
attr(,"scale1_origin")
```
when also understands the keyword first.complete, which is actually the default, and the keywords first and last. first.complete is useful when some series have leading missing values.

```r
drax[1:2, 1] <- NA
```

```r
drax

 dax  rexp
[1,] NA 440.53
[2,] NA 440.79
[3,] 9428.0 441.55
[4,] 9506.2 441.82
[5,] 9497.8 441.76
```

```r
scale1(drax, level = 100)   ## 'first.complete' is the default
```

```r
drax

 dax  rexp
[1,] NA 99.769
[2,] NA 99.830
[3,] 100.00 100.000
[4,] 100.83 100.062
[5,] 100.74 100.049
attr(,"scale1_origin")
[1] 3
```

When the argument centre is TRUE, the mean return is subtracted from the returns.

```r
scale1(drax.zoo, centre = TRUE)
```

```r
drax

 dax  rexp
2014-01-02 1.00000 1.00000
2014-01-03 1.00114 0.99991
2014-01-06 0.99779 1.00091
2014-01-07 1.00348 1.00083
2014-01-08 1.00000 1.00000
attr(,"scale1_origin")
[1] 2014-01-02
```

The default is to subtract the geometric mean: the series will have a growth rate of zero; it will end where it started.

The argument scale takes a standard deviation and scales the returns to that standard deviation.

```r
apply(returns(scale1(drax.zoo, scale = 0.02)), 2, sd)
```
This may create fairer comparisons, for instance, between fund prices that exhibit very different volatilities. It can also help to visualise correlation.

It should be stressed that `centre` and `scale` change *returns*, but `scale1` expects and returns *levels* (not returns).

The `zoo` method has a further argument that affects returns: `inflate`. To illustrate its use, let us create a constant series.

```r
z <- zoo(100, 
  seq(from = as.Date("2015-1-1"), 
  to = as.Date("2016-1-1"), 
  by = "1 day"))
head(z)
tail(z)
```

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td></td>
</tr>
</tbody>
</table>
```

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2015-12-27</td>
<td>2015-12-28</td>
<td>2015-12-29</td>
<td>2015-12-30</td>
<td>2015-12-31</td>
<td>2016-01-01</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td></td>
</tr>
</tbody>
</table>
```

`inflate` should be a numeric value: the annual growth rate that is added to the time-series's return (or that is subtracted from it, if negative).

```r
head(scale1(z, inflate = 0.02))
tail(scale1(z, inflate = 0.02))
```

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0000</td>
<td>1.0001</td>
<td>1.0001</td>
<td>1.0002</td>
<td>1.0002</td>
<td>1.0003</td>
<td></td>
</tr>
</tbody>
</table>
```

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2015-12-27</td>
<td>2015-12-28</td>
<td>2015-12-29</td>
<td>2015-12-30</td>
<td>2015-12-31</td>
<td>2016-01-01</td>
<td></td>
</tr>
<tr>
<td>1.0197</td>
<td>1.0198</td>
<td>1.0198</td>
<td>1.0199</td>
<td>1.0199</td>
<td>1.0200</td>
<td></td>
</tr>
</tbody>
</table>
```

### 8.2 Scaling a series: how it works

The previous section provided examples of scaling series. In this section, we are going to see how `scale1` does its computations.
Chapter 8. Scaling a series: how it works

First, a series $P$ passed to `scale1` is transformed into returns, $R$. The `scale` argument allows you to set a desired volatility for the series’s returns, defined as their standard deviation. The computation uses the fact that multiplying a random variable by a number $b$ changes its variance to $b^2$ times its original variance. Hence, `scale1` divides the returns by the actual standard deviation and then multiplies them by the desired one (i.e. the value passed via the `scale` argument).

Changing total return (or, equivalently, average return) is slightly more complicated. Suppose we want to scale the total return of the series $P$ such that it equals some target return $r_*$. Start with writing the total return as the product of single-period returns.

$$\frac{P_1}{P_0} \frac{P_2}{P_1} \cdots \frac{P_T}{P_{T-1}} = \frac{P_T}{P_0} = (1 + r_1)(1 + r_2)(1 + r_3)\cdots = \prod_{t=1}^{T} 1 + r_t \quad (8.1)$$

There clearly is an infinity of possible adjustments that would do the trick. We might, for instance, change only $P_0$ or $P_T$ so that the desired return is achieved.

But that is probably not what we want. A reasonable requirement is that the scaling touches as few other statistical properties as possible. Adding a constant $z$ to the return in every period does that: it does not change the volatility of the returns; neither does it affect linear or rank correlation of the returns with some other series. To compute $z$, we need to solve the following equation:

$$(1 + r_1 + z)(1 + r_2 + z)(1 + r_3 + z)\cdots = 1 + r_* \quad (8.2)$$

Alternatively, we may use logs.

$$\sum_i \log(1 + r_i + z) = \log(1 + r_*) \quad (8.3)$$

This is an application for root-finding (see chapter 11 of Gilli, Maringer, and Schumann, 2019), for which `scale1` uses `uniroot`. 
9 Analysing trades

For some strategies or trading approaches, we may prefer to analyse trades, not equity series. (A case in point are intraday strategies, which have no exposure over night.) That is, we do not evaluate the strategy’s performance at pre-defined, usually equally-spaced points in time, but rather split the trading history into separate trades.

9.1 Exposure

We have the following trades and times.

amount <- c(1, 3, -3, 1, -3, 1)
time <- c(0, 1, 3, 4, 7, 12)

The holding period (duration) of these trades can be computed so:

data.frame(position = cumsum(amount)[length(amount)],
            from = time[-length(time)],
            to = time[-1L],
            duration = diff(time))

<table>
<thead>
<tr>
<th>position</th>
<th>from</th>
<th>to</th>
<th>duration</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>4</td>
<td>7</td>
</tr>
<tr>
<td>5</td>
<td>-1</td>
<td>7</td>
<td>5</td>
</tr>
</tbody>
</table>

We can plot the exposure. See Figure 9.1.

plot(c(time[1], time), cumsum(c(0, amount)),
     type = "s", xlab = "time", ylab = "position")
Thus, we have had a position from time 0 to time 12 (hours into the trading day, say), but its size varied. The function `tw_exposure` (time-weighted exposure) computes the average absolute exposure.

```
tw_exposure(amount, time)
```

1.75

To give a simpler example: suppose we bought at the open of a trading day and sold at noon. The average exposure for the day is thus half a contract.

```
amount <- c(1, -1, 0)
time <- c(0, 0.5, 1)
tw_exposure(amount, time)
```

0.5

If we bought at the open, went short at noon, and closed the position at the end of the day, the average exposure would be one contract, since absolute position size is relevant.

```
amount <- c(1, -2, 1)
time <- c(0, 0.5, 1)
tw_exposure(amount, time)
```

1

Whether absolute exposure is used is controlled by an argument `abs.value`. Setting it to FALSE can be useful to detect long or short biases.
9.2 Splitting and rescaling

We have the following trades.

```r
timestamp <- 1:3
amount <- c(-1, 2, -1)
price <- c(100, 99, 101)
```

Calling `split_trades` will return a list of two single trades. Each single trade, in turn, is a list with components `amount`, `price` and `timestamp`.

```r
split_trades(amount = amount,
            price = price,
            timestamp = timestamp,
            aggregate = FALSE)
```

```
[[1]]
[[1]]$amount
 [1] -1  1
[[1]]$price
 [1] 100  99
[[1]]$timestamp
 [1] 1  2

[[2]]
[[2]]$amount
 [1]  1 -1
[[2]]$price
 [1]  99 101
[[2]]$timestamp
 [1] 2  3
```

Note that the second transaction (buy 2 @ 99) has been split up: buying one contract closes the first trade; the other contract opens the second
Chapter 9. Splitting and rescaling

trade. This splitting is useful in its own right: there are accounting systems around that cannot handle a trade that switches a position directly from long to short, or vice versa. Instead, the trade needs first be closed (i.e. the net position becomes zero).

With argument `aggregate` set to `TRUE`, the function reconstructs the total series, but with those trades splitted that change the position’s sign.

```r
split_trades(amount, price, timestamp, aggregate = TRUE)
```

```r
$amount
[1] -1 1 1 -1
```

```r
$price
[1] 100 99 99 101
```

```r
$timestamp
[1] 1 2 2 3
```

Another example. We have the following trades and impose a limit that the maximum absolute exposure for the trader should only be 2.

```r
timestamp <- 1:6
amount <- c(-1,-1,-1,1,1,1)
price <- c(100,99,98,98,99,100)
limit(amount, price, timestamp, lim = 2)
```

```r
$amount
[1] -1 -1 1 1
```

```r
$price
[1] 100 99 99 100
```

```r
$timestamp
[1] 1 2 5 6
```

Scaling the trades.

```r
scale_to_unity(amount)
```

```r
[1] -0.333 -0.333 -0.333 0.333 0.333 0.333
```

Closing the trade at once.

```r
close_on_first(amount)
```

```r
[1] -1 -1 -1 3 0 0
```
10 Plotting irregularly-spaced series during trading hours

10.1 An example

We have the following sample of prices of the Bund future contract, traded at the Eurex in Germany.

<table>
<thead>
<tr>
<th>times</th>
<th>prices</th>
</tr>
</thead>
<tbody>
<tr>
<td>2012-10-18 20:00:09</td>
<td>139.82</td>
</tr>
<tr>
<td>2012-10-18 20:01:11</td>
<td>139.82</td>
</tr>
<tr>
<td>2012-10-18 20:01:59</td>
<td>139.8</td>
</tr>
<tr>
<td>2012-10-18 20:01:29</td>
<td>139.81</td>
</tr>
<tr>
<td>2012-10-18 20:16:49</td>
<td>139.77</td>
</tr>
<tr>
<td>2012-10-18 20:50:49</td>
<td>139.85</td>
</tr>
<tr>
<td>2012-10-18 21:23:19</td>
<td>139.76</td>
</tr>
<tr>
<td>2012-10-18 21:41:39</td>
<td>139.76</td>
</tr>
<tr>
<td>2012-10-18 21:59:59</td>
<td>139.77</td>
</tr>
<tr>
<td>2012-10-19 09:16:10</td>
<td>139.8</td>
</tr>
<tr>
<td>2012-10-19 09:49:31</td>
<td>139.86</td>
</tr>
<tr>
<td>2012-10-19 21:12:49</td>
<td>140.46</td>
</tr>
<tr>
<td>2012-10-19 21:42:31</td>
<td>140.39</td>
</tr>
<tr>
<td>2012-10-22 08:45:15</td>
<td>140.14</td>
</tr>
<tr>
<td>2012-10-22 09:05:33</td>
<td>140.15</td>
</tr>
</tbody>
</table>

Note that I have left the time zone to the operating system. Since my computer is typically located in the time zone that the tz database
Chapter 10. An example

Figure 10.1: Price plot on a standard x-axis.

(http://www.iana.org/time-zones) calls 'Europe/Berlin', the first time should be 2012-10-18 20:00:09. If, for instance, your computer is in 'America/Chicago' instead and you run the above code, the first time would be 2012-10-18 13:00:09. Which is right: it is the correct time, only translated into Chicago local time.

A plot of price against time looks like this.

plot(times, prices, type = "s")

Such a plot is fine for many purposes. But the contract for which we have prices is only traded from Monday to Friday, not on weekends, and it is traded only from 08:00 to 22:00 Europe/Berlin time. So the plot should omit those times at which no trading takes place. This is what the function plot_trading_hours does.

```r
tmp <- plot_trading_hours(x = prices, t = times,
interval = "1 sec",
labels = "day",
fromHHMMSS = "080000",
toHHMMSS = "220000",
type = "s")
```

What we need for such a plot is a function that maps actual time to a point on the x-scale, while the y-scale stays unchanged. If we were talking only about days, not times, we needed something like this:
Figure 10.2: Price plot on a standard x-axis.

<table>
<thead>
<tr>
<th>day</th>
<th>x-position</th>
<th>mapped x-position</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thursday</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Friday</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Saturday</td>
<td>3</td>
<td>&lt;removed&gt;</td>
</tr>
<tr>
<td>Sunday</td>
<td>4</td>
<td>&lt;removed&gt;</td>
</tr>
<tr>
<td>Monday</td>
<td>5</td>
<td>3</td>
</tr>
</tbody>
</table>

This mapping is what plot_trading_hours provides. And not much more: the design goal of the function is to make it as much as possible an ordinary plot; or more specifically, to make it as similar as possible to the plot function. Indeed, plot_trading_hours calls plot with a small number of default settings:

```
list(type = "l", xaxt = "n", xlab = "", ylab = ")
```

These settings can all be overridden through the ... argument, which is passed to plot. Note that we already set s as the plot’s type in the last code chunk. The only required setting is suppressing the x-axis with setting xaxt to 'n', because plot_trading_hours will create its own x-axis via a call to axis(1, ...). In case you wish to use your own axis specification, either set do.plotAxis to FALSE or pass settings to axis through the list axis1.par.
10.2 More examples

10.2.1 Value of plot_trading_hours

Like plot, plot_trading_hours is typically called for its side effect: creating a plot. But it also returns useful information (invisibly, unless called with do.plot = FALSE).

```r
str(tmp)
```

```r
List of 6
$ t : int [1:15] 1 63 81 111 1001 3041 4991 6091 7191 ... 
$ x : num [1:15] 140 140 140 140 140 ... 
$ axis.pos : num [1:2] 7193 57594 
$ timegrid : POSIXct[1:61527], format: "2012-10-18 20:00:09" ... 
$ map : function (t)
```

This information can be used to add elements to plots. An example follows.

10.2.2 Adding grid lines

We can add grid lines with abline. The y-axis poses no special problem. The positions of the x-axis ticks are returned from plot_trading_hours.

```r
tmp <- plot_trading_hours(x = prices, t = times, 
interval = "1 sec", 
labels = "day", 
fromHHMMSS="080000", 
toHHMMSS = "220000", 
type = "s")
abline(h = axTicks(2), v = tmp$axis.pos, 
col = "lightgrey", lty = "dotted")
```

If we want to add to a specific time, say 19 October, 13:10:23, we can use the function map that the call to plot_trading_hours returns. We first create the specific time with, for example, ISOdatetime or strptime.

```r
## Again, I do not specify a time zone since time zones 
## depend on the operating system. To reproduce the
```
## example, you may use this representation:
```r
mytime <- structure(1350645023,
  class = c("POSIXct", "POSIXt"),
  tzone = "")
mytime <- ISOdatetime(2012, 10, 19, 13, 10, 23)
mytime
```

```r
```

Now we use `map` to translate this time into the appropriate $x$-position.

```r
tmp <- plot_trading_hours(x = prices, t = times,
  interval = "1 sec", labels = "day",
  fromHHM MSS="080000",
  toHHMSS = "220000",
  type = "s")
abline(h = axTicks(2), v = tmp$axis.pos,
  col = "lightgrey", lty = "dotted")
abline(v = tmp$map(mytime)$t, col = "red")
```

The function `map` returns a list with two components, $t$ and $i$x.

```r
tmp$map(mytime)
```

```r
$t
[1] 25816

$ix
[1] 1
```
The first component is the appropriate position on the x-axis; since it is a time it is called \( t \). The second component gives the subscripts to values that should actually be plotted. As an example, suppose that we wish to plot points at several prices at 21:00:00 for several days.

```r
## moretimes <- structure(c(1350586800, 1350673200, 1350759600),
##                         class = c("POSIXct", "POSIXt"), tzone = ")

moretimes <- ISOdatetime(2012, 10, 18:20, 21, 00, 00)
values <- seq(140, 140.20, length.out = length(moretimes))
data.frame(times = moretimes,
            weekday = format(moretimes, "%A"),
            values)

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2012-10-18 21:00:00</td>
<td>Thursday 140.0</td>
</tr>
<tr>
<td>2</td>
<td>2012-10-19 21:00:00</td>
<td>Friday 140.1</td>
</tr>
<tr>
<td>3</td>
<td>2012-10-20 21:00:00</td>
<td>Saturday 140.2</td>
</tr>
</tbody>
</table>
```

But 20 October 2012 is a Saturday, and so it does not appear in the plot.
The values that should be plotted can conveniently be found by using `ix`.

```r
values[tmp$map(moretimes)$ix]
```

```
[1] 140.0 140.1
```

### 10.3 Daily data

`plot_trading_hours` also handles data on a daily frequency. The function will assume such a case if the timestamp is of class `Date`; it will then choose sensible defaults for the time-axis. In effect, the function will remove weekends and, if specified, holidays.

As an example, consider the first 10 observations of the DAX dataset.

```r
x <- DAX[1:10,]
t <- as.Date(row.names(DAX)[1:10])
data.frame(t, x, weekday = weekdays(t))
```

If we plot these data, there will be a value plotted for 2014-01-04, as marked by the vertical line.

```r
plot(t, x, type = "l")
abline(v = as.Date("2014-01-04"))
```

That is despite the fact that this is a Saturday.
Chapter 10. Daily data

Figure 10.6:
daily.

```
format(as.Date("2014-01-04"), "%A")
```

Saturday

This is not a bug: it is the default behaviour of plot. Saturday’s value results from an interpolation. `plot_trading_hours` instead ignores such nonexisting dates.

```
plot_trading_hours(x, t)
```
11 Valuation

Computing the value of a position is, in principle, straightforward: multiply the prices of assets by the numbers of contracts you hold and sum the resulting values.

This immediately leads to three questions:

1. What is the price?
2. What is a contract?
3. Are we allowed to sum?

11.1 Prices

Valuing an instrument can mean using either a market price or a theoretical price. In the discussion that follows, I will assume that we already have prices (or net-present values).
12 Other Tools

12.1 Dividend adjustments

The function div_adjust corrects price series for dividends. It is meant as a low-level function and is implemented to work on numeric vectors. Consider a hypothetical price series $x$, which goes ex-dividend at time 3.

```r
x <- c(9.777, 10.04, 9.207, 9.406)
div <- 0.7
t <- 3
```

The default for div_adjust is to match the final price.

```r
div_adjust(x, t, div)
```

```
```

If you prefer a correction that matches the first price, set argument backward to FALSE.

```r
div_adjust(x, t, div, backward = FALSE)
```

```
```

12.2 Stocks splits

The function split_adjust handles stock splits. It is implemented to work on numeric vectors.

```r
x <- c(9.777, 10.04, 9.207, 9.406)
div <- 0.7
t <- 3
```
Chapter 12. Validating ISINs

12.3 Treasuries quotes

US treasury bonds are often quoted in 1/32nds of points. For instance, the price 110'030 would mean 110+3/32.

The function `quote32` lets you parse and ‘pretty-print’ such prices.

```r
quote32(c("110-235", "110-237"))
```

```
[1] 110-23+ 110-23¾
```

Internally, `quote32` will store the prices as numeric values: the fractions are only used for printing.

```r
as.numeric(quote32(c("110-235", "110-237")))
```

```
[1] 110.73 110.74
```

```r
dput(quote32(c("110-235", "110-237")))
```

```
structure(c(110.734375, 110.7421875),
  handle = c(110, 110),
  ticks = c(23, 23),
  fraction = c(2, 3),
  class = "quote32")
```

12.4 Validating ISINs

An ISIN, which stands for International Securities Identification Number, uniquely\(^1\) identifies a security.

```r
is_valid_ISIN(c("DE0007236101", "DE0007236102"))
```

```
[1] TRUE FALSE
```

\(^1\)More or less uniquely, depending on the definition of security. ISINs are widely used for equities, in particular in European markets. But note that ISINs do not contain information about the trading venue. Also, for derivatives, an ISIN may not uniquely identify the contract: for instance, Euro-Bund Futures, traded at the Eurex, have a single ISIN for all maturities.
12.5 Price tables

A pricetable is a matrix of prices, with some added functionality for subsetting.

12.6 Trees

To normal people, a tree consists of a trunk, branches and leaves. To people who do graph theory, a tree is a connected graph with only one path between any two nodes.

Trees are useful to represent hierarchies – just think of a file tree.
Chapter 12. Trees
13 FAQ/ FRC (Frequently-required computations)

I have a list of timestamped trades and I need to compute P/L between two points in time, for instance between yesterday’s evening close and now (intraday).

Call the points in time $t_0$ and $t_1$. The easiest case is if there were no positions at both $t_0$ and $t_1$. In that case, create a journal of your trades, and call pl.

If there were positions, you will need the valuation prices for all instruments with positions at both points in time. Then, you can use pl; see arguments initial.position and vprice.

Alternatively, you would arrive at the P/L as follows:

1. Compute the position at $t_0$ and make it a journal $J_0$. The prices need to be the valuation prices. (That is, pretend you opened the position at their valuation prices at $t_0$.)

2. Take all transactions at $t > t_0$ and $t \leq t_1$ and put them into a journal $J$.

3. Compute the position at $t_1$ make it a journal $J_1$, and multiply all amounts by -1. The prices need to be the valuation prices. (That is, pretend you closed the position at their valuation prices at $t_1$.)

4. Combine $J_0$, $J$, and $J_1$ and compute the P/L.
Chapter 13. FAQ/FRC (Frequently-required computations)

How can I compute portfolio returns when I don’t have prices, but only returns of the assets?

Compute artificial prices; e.g. using something like

\[ \text{cumprod}(c(1, 1 + r)) \]

Then use \texttt{return}s.

I have a portfolio with constant weights. How to compute its returns when it is rebalanced at specific times?

Compute artificial prices, and then use \texttt{return}s: see arguments \texttt{weights} and \texttt{rebalance.when}. See Section Portfolio returns.

I have a journal of trades and want to compute the number of trades per day?

Use \texttt{tapply}. If the timestamp inherits from class \texttt{Date} or can be coerced to \texttt{Date}, the following line will compute the number of trades per day.

\[ \text{tapply}(J, \text{as.Date}(J\$timestamp), \text{length}) \]

I have a list of trades: instrument, side (buy/sell), quantity, when and at what price. How to compute the profit and loss for each?

See \texttt{pl}.

I have a list of trades in an instrument and want to plot these trades against the price of the traded instrument.
Use pl; in particular, pass the prices with vprice.

I have a signal series (+1, 0, 0, +1, ...), and need to transform it into a profit-and-loss series.

If these are positions, pass the signals to btest and access them with signals[Time()].

I need to determine the month-to-date profit-and-loss.

1. compute position on last day of last month
2. make journal from position (add prices)
3. combine with journal since month start
4. use ~pl on all instruments

btest: I want to print my current P/L in every period.

Use print.info.

btest: I invest in assets that pay accrued interest.

Directly work with the dirty prices. If the signals depend on clean prices, pass them as extra information and access them with clean_price[Time()]. Alternatively, work with the clean prices, and use cashflow to add the accrued interest to the cash account.

btest: Can I rebalance more frequently than I compute a signal?
Chapter 13. FAQ/ FRC (Frequently-required computations)

You can, but it does not make sense in the standard setup. That is, no rebalancing will take place, even if you instruct btest to do so. The reason is that a signal computes a suggested position (in units of the instrument); once this position has been built up, no more trading is required. This is even true when using weights: The argument convert.weights is a convenience that converts weights into a suggested position; btest does not store these weights, only the suggested position.
14 Appendix: Classes and data structures

The following classes are implicitly defined (i.e. they are S3 classes):

**journal** keeps transactions. Internally, an object of class `journal` is named list of atomic vectors.

**position** the numerical positions of different accounts/instruments at specific points in time. Always stored in a numeric matrix with attributes `timestamp` and `instrument`; points in time are in rows, instruments in columns.

**period returns** numeric vector (potentially a matrix) with attributes `timestamp` and `period`. The class is called `p_returns`.

**instrument** term sheet (description etc); it does know nothing about market data – not yet implemented

**cashflow** internal – not yet implemented

**NAVseries** store a time-series of net asset values

**pricetable** a matrix of NAVs (or prices); each column corresponds to one asset. Additional attributes `instrument` and `timestamp`. Often, pricetables will be created corresponding to positions.
Chapter 14. Appendix: Classes and data structures
15 Appendix: Notes for developers

15.1 Methods for returns

Methods are responsible for 'stripping' the input down to x and t, calling 'returns.default' or some other method, and then to re-assemble the original class's structure. When period is not specified, methods should keep timestamp information for themselves and not pass it on. That is, returns.default should only ever receive a timestamp when period is specified.
Chapter 15. Methods for returns
16 Appendix: R and package versions used

R version 4.0.3 (2020-10-10)
Platform: x86_64-pc-linux-gnu (64-bit)
Running under: Ubuntu 20.10

Matrix products: default
BLAS: /usr/lib/x86_64-linux-gnu/openblas-openmp/libblas.so.3
LAPACK: /usr/lib/x86_64-linux-gnu/openblas-openmp/libopenblas-r0.3.10.so

locale:
[1] LC_CTYPE=en_US.UTF-8  LC_NUMERIC=C
[3] LC_TIME=en_GB.UTF-8  LC_COLLATE=en_US.UTF-8
[5] LC_MONETARY=en_GB.UTF-8  LC_MESSAGES=en_US.UTF-8
[7] LC_PAPER=en_GB.UTF-8  LC_NAME=C
[9] LC_ADDRESS=C  LC_TELEPHONE=C

attached base packages:
[1] stats  graphics  grDevices  utils  datasets  methods
[7] base

other attached packages:
[1] zoo_1.8-8  PMwR_0.16-0

loaded via a namespace (and not attached):
[1] datetimeutils_0.4-1  compiler_4.0.3  tools_4.0.3
[4] parallel_4.0.3  fastmatch_1.1-0  NMOF_2.3-0
[7] grid_4.0.3  orgutils_0.5-0  textutils_0.2-1
[10] lattice_0.20-41
Chapter 16. Appendix: R and package versions used
Bibliography


Index

_returns (function), 41
aggregate_.journal (method), 24
annualised returns, 45
as.data.frame_.journal (method), 24
as.journal (function), 12
as.NAVseries (function), 92
backtesting
delaying signals, 72
frequency, 59
btest (R function in package PMwR), 51
btest
frequency, 59
burn-in, 53
C, 22
datetimeutils (R package), 8, 56
DAX, 39
DAX (dataset), 97
distributed computing, 7
div_adjust (R function), 48
drawdown, 94
drawdown (R function in package NMOF), 8
Emacs, 5
French, Kenneth, 76
functional programming, 7
GitHub, 6
GitLab, 6
inflate (argument to scale1), 101
is_valid_.ISIN (function), 118
ISIN, 118
journal
aggregating journals, 24
backtest, 59
combining journals, 16
comparison with dataframe, 11
concatenating journals, 16
definition, 11
empty journals, 12, 13
number of transactions, 17
print journals (print method), 15
sorting journals, 16
splitting, 19
subsetting, 17
journal (function), 13
lookthrough, 89
mailtools (R package), 8
nanotime (R package), 16
NAV series
summary, 92
NAVseries (function), 92
NMOF (R package), 87
NMOF (R package), 87
nth_day (R function in package datetimeutils), 20, 56

Org-mode, 5
org_journal (R function), 14
overnight gap, 108

Packages, see R packages
Perl, 22
p1 (function), 27
plot_trading_hours (function), 108

position
compute from journal, 19
directly declare a position, 86
position (function), 19
profit/loss
over specific period of time, 33–35
with open trades, 32

quote32 (function), 118

R packages
NMOF, 87
datetimeutils, 8, 56
mailtools, 8
nanotime, 16
textutils, 9
tsd, 9
xts, 42
zoo, 7, 41

rebalance
a portfolio, 83
during backtest, 56
rebalance (function), 83
replace_weight (function), 89
returns
MTD, 46

YTD, 46
annualised, 45
for calendar period, 42
monthly, 42
print method, 43
rolling returns, 41
specifying a lag, 41
time-weighted, 48, 49
when position is rebalanced periodically, 46
with external cashflows, 48

returns (function), 41
returns (function), 39
REXP, 39
REXP (dataset), 97

scale1 (function), 97
streaks in time-series, 96
Sweave
preparing returns tables, 45

TAopt (R function in package NMOF), 87
tapply, 24
textutils (R package), 9
time-weighted exposure, 103
time-weighted returns, 48, 49
timezones, 108, 110
toLatex (function)

summary.NAV method, 93
toOrg (function), 32
trading hours, 108
tsd (R package), 9
tz database, 108

uniroot (function), 102

xts (R package), 42
zoo (R package), 7, 41
zoo
Chapter 16. Appendix: R and package versions used

NAV series, 91
returns calculation, 41